Karlsruhe Institute of Technology

Performanzabschätzung von parallelen Programmen durch symbolische Ausführung

Masterarbeit
von

Janis Estelmann

An der Fakultät für Informatik
Institut für Programmstrukturen
und Datenorganisation (IPD)

Erstgutachter:	Prof. Dr. Walter F. Tichy
Zweitgutachter:	Prof. Dr. Ralf H. Reussner
Betreuender Mitarbeiter:	M.Sc. Marc Aurel Kiefer

Bearbeitungszeit: 01.05.2018 – 31.10.2018

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Die Regeln zur Sicherung guter wissenschaftlicher Praxis im Karlsruher Institut für Technologie (KIT) habe ich befolgt.

Karlsruhe, 31.10.2018

. . . .

(**Janis Estelmann**)

Kurzfassung

Die Steigerung der Leistungsfähigkeit aktueller Prozessorgenerationen wird durch das Hinzufügen von zusätzlichen Rechenkernen erreicht. Durch die hieraus gegebene Möglichkeit der nebenläufigen Ausführung von unabhängigen Teilstücken einer Anwendung kann die Ausführungszeit signifikant verringert werden. Damit eine Anwendung von Parallelisierung profitieren kann, müssen hierfür passende Stellen im Quellcode vom Entwickler erkannt und parallelisiert werden.

Durch eine Bestimmung der Ausführungskosten können teure Pfade erkannt und analysiert werden. Da sich die Laufzeiten einer Anwendung durch unterschiedliche Eingabeparameter ändern, müssen diese entsprechend gewählt werden. Um passende Eingaben zu erzeugen, kann das Prinzip der symbolischen Ausführung verwendet werden. Hierbei wird der Kontrollfluss analysiert und die Variablenbelegung so angepasst, dass eine hohe Abdeckung ermöglicht wird. Ein großes Problem der symbolischen Ausführung stellt die Pfad Explosion dar. Die fortschreitende Teilung von Pfaden an Verzweigungspunkten führt zu einem exponentiellen Wachstum der Anzahl an Pfaden. Ab einem bestimmten Punkt kann deren Anzahl zu groß werden, als dass sie die Testanwendung sinnvoll verwalten könnte.

Das Ziel dieser Arbeit ist es, einen Entwickler dabei zu unterstützen, in einer bereits vorhandenen Anwendung Schleifen mit Parallelisierungspotential im Quellcode zu finden. Hierzu soll das auf der Compiler-Infrastruktur LLVM aufbauende Test-Programm KLEE erweitert werden. Die primäre Aufgabe von KLEE ist es, durch symbolische Ausführung Ausführungspfade durch Anwendungen zu bestimmen und deren Wege auf vorgegebene Fehlerfälle zu untersuchen.

Mit Hilfe der symbolischen Ausführung sollen Variablenbelegungen bestimmt werden, die zu hohen Ausführungskosten bei einem Pfad führen. Durch die Analyse der Pfade auf Hot-Spots, also Bereiche, die besonders hohe Kosten verursachen, wird es einem Entwickler ermöglicht, gezielt diese auf ihr Parallelisierungspotential zu untersuchen.

Um dem Problem der Pfad Explosion entgegen zu wirken, muss die Implementierung sinnvoll zwischen Ausführungspfaden wählen können. Es sollen nur die Pfade weiter analysiert werden, bei denen absehbar ist, dass sie zu interessanten Problemstellen führen.

Inhaltsverzeichnis

1. Einleitung 1
- 1.1. Zielsetzung der Arbeit . . . 2
- 1.2. Gliederung der Arbeit . . . 3

2. Grundlagen 5
- 2.1. Parallele Programmausführung . . . 5
- 2.2. Profiling . . . 5
- 2.3. Aufrufgraph, Kontrollflussgraph . . . 6
- 2.4. Symbolische Ausführung . . . 6
  - 2.4.1. Beispiel . . . 8
  - 2.4.2. Pfad Explosion . . . 9
- 2.5. SMT Solver . . . 9
- 2.6. LLVM . . . 10
  - 2.6.1. LLVM Datenstrukturen . . . 11
  - 2.6.2. LLVM-Zwischensprache . . . 12
    - 2.6.2.1. Beispiel . . . 12
  - 2.6.3. LLVM-Pässe . . . 13
- 2.7. KLEE . . . 14
  - 2.7.1. KQuery . . . 14
  - 2.7.2. Solver Chain . . . 16
  - 2.7.3. Datenverarbeitung . . . 17
  - 2.7.4. Beispiel . . . 18
  - 2.7.5. Einschränkungen von KLEE . . . 19
    - 2.7.5.1. Pfad Explosion . . . 19
    - 2.7.5.2. Solver . . . 19
    - 2.7.5.3. Externe Funktionen . . . 20
  - 2.7.6. Weitere Anwendungsgebiete . . . 20
    - 2.7.6.1. KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment . . . 20
    - 2.7.6.2. Analyzing Protocol Implementations for Interoperability . 20
    - 2.7.6.3. Server-side Verification of Client Behavior in Online Games 20
    - 2.7.6.4. GKLEE: Concolic Verification and Test Generation for GPUs 21
    - 2.7.6.5. KLOVER: Automatic Test Generation for C and C++ Programs, Using Symbolic Execution . . . 21

3. Verwandte Arbeiten 23
- 3.1. Implementation of Path Profiling in the Low-Level Virtual-Machine (LLVM) Compiler Infrastructure . . . 23
- 3.2. ParaSCAN: A Static Profiler to Help Parallelization . . . 23
- 3.3. Input-Sensitive Profiling . . . 24
  - 3.3.1. Multithreaded Input-Sensitive Profiling . . . 24
- 3.4. Generating Performance Distributions via Probabilistic Symbolic Execution 24

3.5. WISE: Automated Test Generation for Worst-Case Complexity 25
3.6. Symbolic Complexity Analysis Using Context-Preserving Histories 25
3.7. Directed symbolic execution 26
3.8. Rapid Techniques for Performance Estimation of Processors 26

4. Analyse **29**
4.1. Definition der Problemstellung 29
4.2. Grundidee 30
4.3. Voraussetzungen 31
4.3.1. Control Flow Graph 31
4.3.2. Schleifen-Erkennung 31
4.3.2.1. Ablauf der Schleifen-Erkennung 32
4.3.3. Kostenmodell 33
4.4. Definition der Ziele 33
4.4.1. Z1: Interprocedural Control-Flow Graph aufbauen 33
4.4.2. Z2: Pfad Explosion einschränken 33
4.4.3. Z3: Schleifenerkennung 33
4.4.4. Z4: Hot-Path Analyse 33
4.4.5. Z5: Ausgabe von Pfadinformationen 33
4.4.6. Z6: Performanzsteigerung durch Entfernen von Funktionalität 33

5. Entwurf **35**
5.1. Ablauf des Verfahrens 35
5.2. Z1: Interprocedural Control-Flow Graph aufbauen 37
5.3. Z2: Pfad Explosion einschränken 37
5.4. Z3: Schleifenerkennung 37
5.5. Z4: Hot-Path Analyse 37
5.6. Z5: Ausgabe von Pfadinformationen 38
5.7. Z6: Performanzsteigerung durch Entfernen von Funktionalität 38

6. Implementierung **39**
6.1. Verwendete Werkzeuge 39
6.2. Verwendete LLVM-Pässe 39
6.2.1. LoopSimplify 39
6.2.2. LoopInfo 40
6.2.3. TargetTransformInfo 40
6.2.4. UnifyFunctionExitNodes 40
6.3. Verwendete Klassen 41
6.4. Umsetzung der Ziele 43
6.4.1. Z1: Interprocedural Control-Flow Graph aufbauen 43
6.4.2. Z2: Pfad Explosion einschränken 44
6.4.2.1. Schattenstrukturen 44
6.4.2.2. Vorgabe der Ausführungspfade 45
6.4.3. Z3: Schleifenerkennung 46
6.4.4. Z4: Hot-Path Analyse 46
6.4.5. Z5: Ausgabe von Pfadinformationen 47
6.4.6. Z6: Performanzsteigerung durch Entfernen von Funktionalität 47

7. Evaluation **49**
7.1. Z1: Interprocedural Control-Flow Graph aufbauen 49
7.2. Z2: Pfad Explosion einschränken 52
7.3. Z3, Z4, Z5: Zusammenfasung 53
7.4. Z6: Performanzsteigerung durch Entfernen von Funktionalität 56

8. Zusammenfassung und Ausblick **57**

Literaturverzeichnis **61**

Anhang **65**

A. KLEE Quelltextänderungen 65

B. Quelltextausschnitte 65

Abbildungsverzeichnis

2.1. Veranschaulichung der Aufteilung der Daten auf mehrere Prozessoren. . . . 6
2.2. Darstellung der Ausführungspfade durch die Funktion. 9
2.3. Darstellung von Ausführungspfaden einer Anwendung. 10
2.4. Darstellung des modularen Aufbaus der LLVM-Infrastruktur. 11
2.5. Die Solver Chain von KLEE. 17

4.1. Übersicht über den erweiterten Prozessablauf von KLEE. 30
4.2. Beispiel für eine Schleife vor und nach der Vereinfachung der Struktur durch den *loop-simplify* Pass. 32

6.1. UML Klassendiagramm des Interprocedural Control-Flow Graph. 43
6.2. Schematische Darstellung des Ablaufs zur Erstellung des Interprocedural Control-Flow Graphs. 44
6.3. UML Klassendiagramm der Schattenstrukturen. 45
6.4. UML Klassendiagramm der *KCallPathManager* Klasse. 46

1. Einleitung

Bei aktuellen Prozessorgenerationen wird die Leistungsfähigkeit in der Regel durch das Hinzufügen von zusätzlichen Rechenkernen gesteigert. Dies ermöglicht es, ein Programm in mehrere Teilstücke aufzuteilen und diese nebenläufig auf den einzelnen Kernen auszuführen. Die Dauer der Ausführung wird somit verringert, da im Gegensatz zu einer sequentiellen Bearbeitung nicht auf die Beendigung eines Teilstücks gewartet werden muss. Von dem Leistungszuwachs profitieren Anwendungen allerdings nur, wenn die damit verbundenen Parallelisierungsmöglichkeiten erkannt und ausgenutzt werden. Für den Entwickler eines parallel ausführbaren Programms kann es schwierig sein, die ihm zur Verfügung stehenden Möglichkeiten gewinnbringend einzusetzen. Üblicherweise ist beim Entwickeln nicht einfach ersichtlich, an welchen Stellen eine Parallelisierung von Nutzen ist oder sich eventuell sogar negativ auf die Ausführungskosten auswirkt.

In der Praxis eignen sich Schleifen gut zur Parallelisierung. Schleifen sind Strukturen im Quellcode, bei denen ein Anweisungsblock bzw. der Schleifenkörper wiederholt ausgeführt wird, bis ein Abbruchkriterium erreicht wurde. Wird beispielsweise in einer Schleife 100-mal eine Berechnung durchgeführt, so kann die Ausführungszeit der Schleife auf einem Prozessor mit vier Kernen um bis zu 75 Prozent reduziert werden. Da auf den Kernen vier Berechnungen parallel durchgeführt werden können, muss die Schleife somit nur noch 25-mal durchlaufen werden.

Um die Leistungsfähigkeit eines Programms in Bezug auf dessen Ausführungskosten bestimmen zu können, müssen diese zur Laufzeit mit einem Profiler gemessen werden. Problematisch ist hierbei, dass das Verhalten des Programms im Normalfall unterschiedlich bezüglich wechselnder Eingabeparameter ist. Die Eingaben, die ein Programm entgegennimmt, können einen signifikanten Einfluss auf die Dauer der Ausführung haben. Beispielsweise wird zur Bestimmung der Frage, ob eine kleine Zahl eine Primzahl ist, wesentlich weniger Zeit benötigt, als für die selbe Bestimmung einer großen Zahl, da bei kleinen Zahlen weniger Überprüfungen durchgeführt werden müssen. Auch werden oft nicht alle möglichen Ausführungszweige eines Programms durchlaufen und somit nur ein Teil des vorhandenen Quellcodes bei der Kostenbestimmung berücksichtigt.

Damit es dem Entwickler ermöglicht wird, alle Ausführungszweige zu testen, kann das Prinzip der symbolischen Ausführung verwendet werden. Bei der symbolischen Ausführung wird der Code der Anwendung schrittweise analysiert und Werte für die verwendeten Variablen bestimmt, sodass jeder Pfad im Programm erreichbar ist. Unter einem Ausführungspfad versteht man hierbei die Reihenfolge von durchlaufenen Funktionen und

Quellcodebereichen, bis das Ende des Programms erreicht wurde. Themen rund um die symbolische Ausführung werden seit 1976 [Kin76] wissenschaftlich untersucht und beständig weiterentwickelt[1 2]. Mit Hilfe dieser Technik wird es ermöglicht, Eingaben generieren zu lassen, mit denen sich die Ausführungskosten des Programms bezüglich der unterschiedlichen Programmpfade bestimmen lassen.

Das Wissen um die Ausführungskosten bei unterschiedlichen Eingaben ist beispielsweise für die Verhinderung eines Denial of Service Angriffs hilfreich. Bei dieser Art von Angriff wird ein System mit Eingaben aufgerufen, die zu einer hohen Auslastung und damit zu einer Ressourcenknappheit führen. Auf diese Weise ist das System vollständig damit beschäftigt, die schädliche Aufgabe auszuführen, sodass es nicht mehr für weitere Anfragen genutzt werden kann. Das Ziel des Angreifers ist es hierbei, selbst einen möglichst geringen Aufwand bei großem Nutzen zu erzielen. Bezogen auf das Thema dieser Arbeit wäre es demnach wünschenswert, bereits im Vorfeld Eingaben zu bestimmen, die zu hohen Ausführungskosten führen, um frühzeitig Gegenmaßnahmen einleiten zu können.

Ein großes Problem der symbolischen Ausführung stellt die sog. Pfad Explosion dar. Trifft ein Pfad währen der Ausführung auf eine Verzweigung im Kontrollfluss muss er in zwei unterschiedliche Pfade aufgetrennt werden. Jeder Pfad folgt anschließend einem unterschiedlichen Weg der Verzweigung. Auf diese Weise wird ein exponentielles Wachstum der Anzahl der Pfade ausgelöst. Dieses Wachstum führt dazu, dass es nicht immer möglich ist, alle Pfade einer Anwendung zu untersuchen, da dies das Testsystem bezüglich des Speicherverbrauchs bzw. der Rechenzeit überlasten würde.

1.1. Zielsetzung der Arbeit

Das Ziel dieser Arbeit ist es, einen Entwickler dabei zu unterstützen in einer bereits vorhandenen Anwendung Schleifen im Quellcode zu finden die Parallelisierungspotential besitzen.

Als Grundlage soll das im Jahr 2008 vorgestellte Test-Programm KLEE ([CDE+08], siehe Kapitel 2.7) verwendet werden. KLEE baut auf der Compiler-Infrastruktur LLVM ([LA04], siehe Kapitel 2.6) auf und ermöglicht das Generieren von Testfällen für Anwendungen. Dazu nutzt KLEE die symbolische Ausführung um mögliche Pfade zu finden und auf vorgegebene Fehlerfälle wie beispielsweise eine mögliche Division durch Null zu prüfen.

Im Rahmen dieser Arbeit soll die bisherige Funktionalität von KLEE erweitert werden. Die symbolische Ausführung soll nicht nur für die Bestimmung von Variablen genutzt werden, die einen möglichst hohen Anteil an Ausführungspfaden abdecken. Stattdessen sollen in Abhängigkeit der gewählten Variablen die Ausführungskosten der gefundenen Pfade berechnet werden. Hierzu muss der Quellcode von KLEE angepasst werden, damit dieser als Profiler zusätzlich Informationen zu den benötigten Ausführungskosten des aktuellen Pfades generiert. Um den auszuführenden Programm-Instruktionen Ausführungskosten zuweisen zu können, muss ein Kostenmodell verwendet werden. Dies ermöglicht es Pfade zu bestimmen, die hohe Kosten zur Laufzeit verursachen.

Um einen Entwickler bei der Auswertung der Informationen zu den Ausführungskosten eines Pfades zu unterstützen, sollen die Abschnitte eines Pfades identifiziert werden, die für einen Großteil der verursachten Kosten verantwortlich sind. Diese sog. Hot-Spots werden üblicherweise Schleifen sein, da dort Arbeit wiederholt ausgeführt wird. Durch die Identifizierung solcher teuren Schleifen, kann geprüft werden, ob sich diese parallelisieren lassen.

[1] Grafische Übersicht über wichtige Meilensteine zur symbolischen Ausführung: `https://github.com/enzet/symbolic-execution` (abgerufen am 19. September 2018)

[2] Tabellarische Übersicht zu wissenschaftlichen Veröffentlichungen zu symbolischer Ausführung: `https://github.com/saswatanand/symexbib` (abgerufen am 19. September 2018)

Damit die Auswertung der einzelnen Pfade in annehmbarer Laufzeit zu einem Ergebnis führt, muss das Problem der Pfad Explosion eingeschränkt werden. Hierzu soll untersucht werden, welche Möglichkeiten zur Einschränkung zur Verfügung stehen und wie man diese einsetzen könnte. Da der Einsatzzweck von KLEE primär die Fehlersuche in Anwendungen ist, sollen außerdem bestimmte Funktionalitäten eingeschränkt werden, um eine Steigerung der Performanz zu erzielen. Die Fehlersuche wird im Kontext dieser Arbeit nicht benötigt und die Deaktivierung der hierfür nötigen Funktionen wirkt sich positiv auf die Laufzeit von KLEE aus.

1.2. Gliederung der Arbeit

In Kapitel 2 werden die *Grundlagen* rund um symbolische Ausführung, LLVM und KLEE erläutert um das Verständnis der weiteren Kapitel zu erleichtern. Es wird dabei auch auf Techniken eingegangen, die während der Arbeit verwendet werden.

Verwandte Arbeiten werden in Kapitel 3 vorgestellt. Sie bieten einen Überblick über den aktuellen Stand der Forschung zu ähnlich gelagerten Themen.

Die *Analyse* der Anforderungen an die Implementierung dieser Arbeit erfolgt in Kapitel 4. Es werden die Grundidee sowie die hierzu benötigten Voraussetzungen und Eigenschaften beschrieben. Abschließend folgt eine Definition der zu erfüllenden Ziele.

Der *Entwurf* des kompletten Systems in Kapitel 5 beschreibt dessen Aufbau. Es wird der grobe Rahmen vorgegeben, wie die Lösung der Probleme zu implementieren ist.

Das Kapitel 6 widmet sich der *Implementierung* der im vorherigen Kapitel beschriebenen Ziele. Es liefert detaillierte Informationen, die die Implementierung nachvollziehbar machen.

Die *Evaluation* in Kapitel 7 testet die vorgestellte Implementierung und prüft, ob die definierten Ziele erreicht wurden.

Abschließend folgt mit Kapitel 8 die *Zusammenfassung* der vorherigen Kapitel. Außerdem wird ein *Ausblick* geboten, wie die erstellte Implementierung erweitert werden könnte.

2. Grundlagen

In diesem Kapitel werden die Grundlagen und Begriffe erklärt, die zum Verständnis der folgenden Kapitel wichtig sind.

2.1. Parallele Programmausführung

Unter paralleler Programmausführung versteht man das nebenläufige Ausführen von Instruktionen auf mehreren Prozessoren bzw. Prozessorkernen. Im Gegensatz dazu kann bei der sequentiellen Programmausführung immer nur Befehl verarbeitet werden. Die Verteilung von Aufgaben auf mehrere Prozessoren hat den Vorteil, dass diese Aufgaben schneller beendet werden können, da sie nicht nacheinander, sondern gleichzeitig bearbeitet werden.

Wenn beispielsweise eine Liste mit n natürlichen Zahlen auf das Vorhandensein von ungeraden Zahlen geprüft werden soll, muss der Prozessor im sequentiellen Fall jede der n Zahlen untersuchen und benötigt für diese Arbeit n Zeit. Stehen für die selbe Aufgabe allerdings m Prozessoren zur Verfügung, lässt sich die Zahlen-Liste in m gleichmäßige Bereiche aufspalten. Diese kleineren Teilaufgaben werden nun auf die m verfügbaren Prozessoren verteilt. Jeder der Prozessoren muss nun nur noch $\frac{n}{m}$ Zahlen untersuchen und benötigt daher auch nur noch $\frac{n}{m}$ Zeit. Da alle Prozessoren ihre Teilaufgaben unabhängig voneinander parallel bearbeiten können, ist auch die Gesamtzeit zur Bearbeitung der großen Aufgabe auf $\frac{n}{m}$ gesunken.

In Abbildung 2.1 ist das Verfahren für $n = 10$ und $m = 5$ dargestellt.

2.2. Profiling

Profiling ist in der Softwaretechnik ein Verfahren, mit dem es Entwicklern ermöglicht wird, Performanceprobleme in Anwendungen aufzudecken. Das Profiling kann zum Beispiel Fragen beantworten, wie oft eine Funktion aufgerufen wurde oder wie lange die Ausführung von Funktionen gedauert hat. Viele Profiler unterstützen beispielsweise das Anzeigen des *Hot-Paths* eines Programms. Unter dem Hot-Path versteht man die Ausführungsreihenfolge von Funktionen, für deren Ausführung die meiste Zeit in einem Programm benötigt wird. Die Daten, die während der Programmausführung durch den Profiler gesammelt werden, können anschließend vom Entwickler ausgewertet werden. Aus den gewonnenen Informationen kann bestimmt werden, bei welchen Funktionen eine Optimierung, z.B. durch parallele Programmausführung, sinnvoll ist, um die Ausführungsdauer der Anwendung positiv zu beeinflussen.

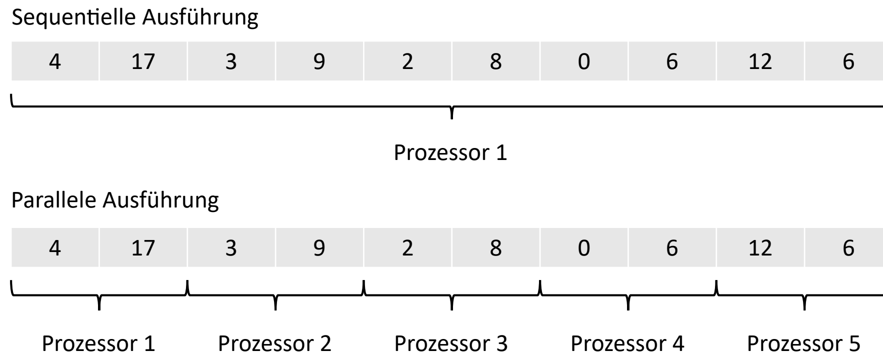


Abbildung 2.1.: Veranschaulichung der Aufteilung der Daten auf mehrere Prozessoren.

Beim Profiling wird zwischen zwei Arten, *dynamischem* und *statischem* Profiling, unterschieden. Beim dynamischen Profiling werden Informationen gesammelt, während das Programm ausgeführt wird. Dadurch, dass in diesem Fall nicht nur der Code der Anwendung ausgeführt werden muss, ergibt sich ein Overhead im Vergleich zur Anwendungsausführung ohne Profiler, da das Sammeln der Informationen auch Ressourcen verbraucht. Außerdem kann der dynamische Profiler nur Informationen über Funktionen sammeln, die auch wirklich ausgeführt werden. Werden manche Funktionen nur in Abhängigkeit von Eingabeparametern der Anwendung verwendet, bleiben diese vor der Analyse verborgen.

Statisches Profiling erfolgt hingegen ohne Programmausführung lediglich durch die Analyse des Programms, beispielsweise beim Kompilieren. Dies führt dazu, dass der Profiler lediglich Schätzungen abgeben kann, wie lange die Ausführungszeit einer Funktion sein könnte. Diese Informationen können während der Übersetzung durch den Compiler genutzt werden, um Entscheidungen für Optimierungen zu treffen (z.B. das Speichern von Werten in Registern statt im Hauptspeicher).

2.3. Aufrufgraph, Kontrollflussgraph

Unter einem Aufrufgraph (CG, von english „Call Graph“) versteht man einen gerichteten Graphen, der beschreibt wie sich Funktionen innerhalb einer Anwendung gegenseitig aufrufen. Die Knoten des Graphen entsprechen hierbei den einzelnen Funktionen. Besteht zwischen zwei Knoten eine Kante, entspricht dies einem Funktionsaufruf. Sind im Graphen Zyklen vorhanden, entspricht dies einem rekursiven Funktionsaufruf.

Unter einem Kontrollflussgraphen (CFG, von englisch „Control Flow Graph“) versteht man einen gerichteten Graphen, der den Kontrollfluss innerhalb einer Funktion beschreibt. Ein Knoten des Graphen entspricht einem Block von Instruktionen, die sequentiell ausgeführt werden. Durch einen Verzweigungspunkt werden im Graph Kanten zwischen Knoten erzeugt. Mit Hilfe eine Zyklensuche ist es möglich Schleifen innerhalb der Logik zu erkennen.

Werden beide Arten von Graphen miteinander kombiniert, spricht man von einem *Interprocedural Control-Flow Graph* (ICFG) [LR91]. Hierbei handelt es sich um einen Kontrollflussgraphen, der nicht nur auf eine Funktion beschränkt ist, sondern alle Funktionen der Anwendung umfasst.

2.4. Symbolische Ausführung

Symbolische Ausführung ist ein Software-Analyseverfahren, hinter dem die Idee steckt, dass ein Programm ausgeführt wird, ohne dass dessen Eingabeparameter mit konkreten

Werten vorbelegt werden. Als Ergebnis der symbolischen Ausführung wird eine mathematische Funktion ausgegeben, in der die Eingabeparameter als Variablen repräsentiert sind. Durch sog. Solver wird es ermöglicht, zu gewünschten Ausgabewerten passende Eingabewerte bzw. Bereiche von Werten zu ermitteln, die diese Ausgabe erzeugen.

Das Verfahren kann im Bereich von Software-Tests eingesetzt werden, um geeignete Testfälle zu generieren. Software-Tests sind Sammlungen von Funktionsaufrufen, die das Verhalten einer Funktion testen. Dazu werden bestimmte Eingabeparameter vorgegeben und nach der Funktionsausführung auf das gewünschte Ergebnis hin überprüft. Mit Software-Tests kann eine höhere Software-Qualität erzielt werden, da bei fortschreitender Entwicklung sichergestellt werden kann, dass vorhandene Funktionen bei Codeänderungen weiterhin das korrekte Ergebnis liefern. Da die Software-Tests für jede zu testende Funktion manuell erzeugt werden müssen, ist das Testen von Software mit einem hohen Aufwand verbunden. Ein großes Problem stellt dabei die gewünschte hohe Abdeckung aller möglichen Ausführungspfade einer Funktion mit Tests dar. An dieser Stelle kann die symbolische Ausführung behilflich sein, da mit ihrer Hilfe passende Testfälle für Funktionen erzeugt werden können.

Durch symbolische Ausführung ist es möglich Ausführungspfade durch Funktionen zu bestimmen. Da Funktionen selten linear ablaufen, sondern durch *if else* Anweisungen bedingte Bereiche besitzen, existieren mehrere gültige Pfade. Die symbolische Ausführung erzeugt für jeden vorhandenen Verzweigungspunkt eine weitere Bedingung, die diesen Pfad abdeckt. Ein Solver (siehe Kapitel 2.5) der korrekt alle Bedingungen eines Pfads auflösen kann, wird Eingabeparameter generieren, durch die das Programm exakt diesem Ausführungspfad folgt.

Bezogen auf die Software-Tests ist es somit möglich alle Ausführungspfade einer Funktion, sowie die hierfür nötigen Eingabeparameter zu bestimmen. Mit diesen Informationen lassen sich Tests erstellen, die eine Funktion mit vollständiger Abdeckung testen können.

Falls der Kontrollfluss einer Funktion nicht von einer symbolischen Eingabe abhängt, wird für diese nur ein einziger Pfad gefunden werden. Der Grund hierfür ist, dass alle vorhandenen Verzweigungspunkte konstant vorbelegt sind und der Kontrollfluss somit immer auf den gleichen Pfad geleitet wird. Im anderen Fall wird der Kontrollfluss aufgespalten und jeder Pfad einzeln weiter betrachtet.

Während der symbolischen Ausführung wird von dem Analyse-Programm eine Liste mit Ausführungszuständen verwaltet. Jeder Zustand repräsentiert einen spezifischen Ausführungspfad durch die zu testende Anwendung. In einem Zustand werden während der Ausführung die aktuell auszuführende Instruktion sowie die Belegung des verwendeten Speichers verwaltet. Außerdem enthält der Zustand eine Liste von Bedingungen, die Verzweigungspunkte erfüllen müssen, um zum aktuellen Pfad zu führen (engl. *path condition*).

Die Path Condition hat die Form eines booleschen Ausdrucks, der eine Verknüpfung der Verzweigungs-Bedingungen ist. An jedem Verzweigungspunkt der Anwendung wird eine weitere Bedingung zur Path Condition hinzugefügt. Mithilfe eines Solvers kann der Ausdruck ausgewertet werden, um konkrete Werte für die symbolischen Variablen zu berechnen. Der Solver kann hierbei auch festellen, ob die Bedingungen erfüllbar sind. Beispiel: Der Code `if (x < y && x > y)` ergibt den booleschen Ausdruck $(x < y) \wedge (x > y)$. Es ist nicht möglich eine Belegung für x und y als Ganzzahl anzugeben, die den Ausdruck erfüllt. Der dahinterliegende Pfad kann somit nicht ausgeführt werden.

Trifft die symbolische Ausführung auf einen Verzweigungspunkt, der von einer symbolischen Variable V abhängt, so prüft der Solver, ob es die bisherige Path Condition zulässt, dass V wahr und falsch sein kann. Ist dies der Fall, muss der aktuelle Zustand aufgespalten (engl. *fork*) werden. Hierbei muss der für Variablen verwendete Speicher kopiert und die Path Condition für beide möglichen Zweige angepasst werden. Für den *wahr*-Zweig wird

die Path Condition (PC) zu $PC = PC \wedge V$. Beim *falsch*-Zweig wird dementsprechend die verneinte Bedingung hinzugefügt: $PC = PC \wedge \neg V$. Anschließend werden beide Zustände unabhängig voneinander auf dem jeweiligen Pfad ausgeführt. Der Vorgang wird solange wiederholt, bis ein Zustand das Ende seines Pfades erreicht hat. Mit dem Solver können nun konkrete Werte für die in der Path Condition beschriebenen symbolischen Variablen bestimmt werden. Mit diesen Werten als Eingabe wird die ausgeführte Anwendung jenen Pfad des Zustands durchlaufen.

2.4.1. Beispiel

Quelltextausschnitt 2.1: Beispielcode zur Demonstration der symbolischen Ausführung.

```
1  int isOddOrGreaterTen(int x)
2  {
3      if (x % 2 == 1)
4      {
5          return 1;
6      }
7
8      if (x > 10)
9      {
10         return 1;
11     }
12
13     return 0;
14 }
```

Die in Listing 2.1 dargestellte Funktion soll zeigen, wie die symbolische Ausführung Ausführungspfade bestimmt. Die Funktion `isOddOrGreaterTen` erhält als Eingabeparameter eine Ganzzahl und gibt wiederum eine Ganzzahl aus. Falls die Eingabe ungerade oder größer als 10 ist, wird jeweils 1 zurückgegeben, andernfalls 0. Wie in Abbildung 2.2 zu sehen, spaltet sich der Ausführungspfad beim ersten if auf. Falls die Eingabezahl ungerade war, endet der Pfad bei diesem if mit der Ausgabe von 1. Falls die Zahl nicht ungerade ist, wird das zweite if betrachtet. Ist die Zahl größer als 10 endet auch hier ein Pfad mit der Ausgabe von 1. Der letzte Pfad wird genommen, wenn die Eingabe weder ungerade noch größer als 10 ist. Es gibt also insgesamt drei mögliche Ausführungspfade in dieser Funktion.

Bei der symbolischen Ausführung startet der Ausgangszustand (*state1*) mit einer leeren Path Condition. Am ersten Verzweigungspunkt in Zeile 3 wird geprüft, ob die Bedingung mit der aktuellen Patch Condition erfüllbar ist. Da diese leer ist, ist die Erfüllbarkeit für die Bedingung als auch ihre Negation gegeben. Aus diesem Grund muss sich der Zustand an dieser Stelle in die Zustände *state1* und *state2* aufspalten und die Ausführung anschließend mit diesen fortgeführt werden. Die Path Conditions beider Zustände lauten zu diesem Zeitpunkt:

state1 $PC = x\%2 = 1$

state2 $PC = \neg(x\%2 = 1)$

state1 trifft in Zeile 5 auf einen Rücksprungpunkt der Funktion und ist hiermit am Ende des Ausführungspfades angekommen. *state2* gelangt in Zeile 8 an einen weiteren Verzweigungspunkt. Auch diese Bedingung sowie ihre Negation ist mit der aktuellen Path Condition erfüllbar. Aus *state2* wird ein weiterer Zustand state3 abgespalten und die Path Conditions mit der neuen Bedingung erweitert:

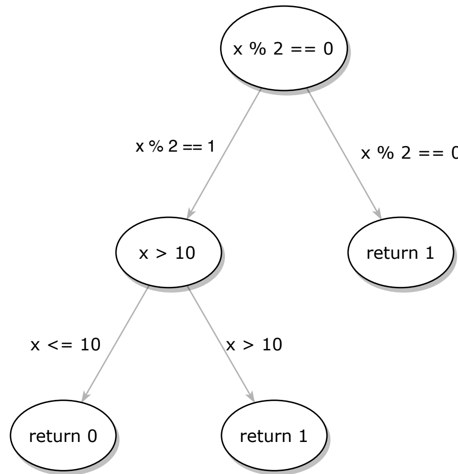


Abbildung 2.2.: Darstellung der Ausführungspfade durch die Funktion.

state1: $PC = x\%2 = 1$

state2: $PC = \neg(x\%2 = 1) \wedge x > 10$

state3: $PC = \neg(x\%2 = 1) \wedge \neg(x > 10)$

Die Ausführungspfade der Zustände *state2* und *state3* enden jeweils bei der nächsten möglichen Instruktion in Zeile 10 und 13. Da keiner der Zustände aktiv ist, wird die symbolische Ausführung beendet. Aus den Path Conditions können konkrete Werte für die Variable x bestimmt werden, die alle Bedingungen erfüllen und zum Ausführungspfad des jeweiligen Zustandes führen:

state1: $PC = x\%2 = 1 \rightarrow x = 1$

state2: $PC = \neg(x\%2 = 1) \wedge x > 10 \rightarrow x = 12$

state3: $PC = \neg(x\%2 = 1) \wedge \neg(x > 10) \rightarrow x = 0$

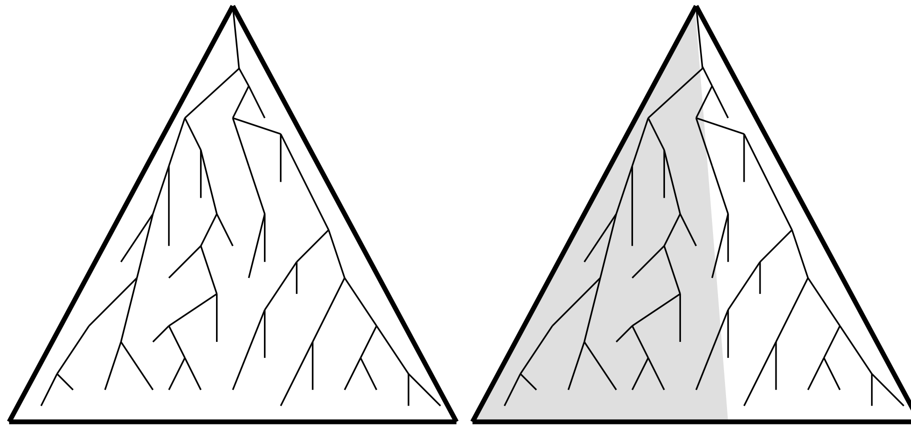
2.4.2. Pfad Explosion

Ein großes Problem bei der Nutzung von symbolischer Ausführung ist die sog. Pfad Explosion (engl. *path explosion*) [BCD+18]. Hierunter wird das exponentielle Wachstum der Anzahl an möglichen Ausführungspfaden einer Anwendung verstanden. Da an jedem Verzweigungspunkt der aktuelle Pfad aufgespalten werden muss, kann dies dazu führen, dass die Ausführung aufgrund von Zeit- und Speichermangel nicht vollendet werden kann.

Da sich das Problem nicht lösen lässt, wird versucht die Auswirkungen zu reduzieren. Dies kann beispielsweise durch das Zusammenfassen ähnlicher Pfade geschehen [KKBC12]. Außerdem können Heuristiken eingesetzt werden, die versuchen, die Anzahl an zu untersuchenden Pfaden zu begrenzen [MPFH11]. Dazu werden Regeln aufgestellt, die bestimmen, ob es sich lohnt einen Pfad weiter zu verfolgen. Da nicht jeder Ausführungspfad untersucht wird, führen alle Lösungen zu einer Verringerung der Testabdeckung.

2.5. SMT Solver

Während der symbolischen Ausführung einer Anwendung ist es an verschiedensten Stellen notwendig, die Erfüllbarkeit der Path Condition durch einen *Solver* (englisch für Löser) [ES03, MMZ+01] zu überprüfen. In der Aussagenlogik kann die Erfüllbarkeit einer



(a) Pfad Explosion durch fortschreitende Aufteilung.

(b) Einschränkung der Pfade durch Heuristik.

Abbildung 2.3.: Darstellung von Ausführungspfaden einer Anwendung.

Formel durch einen SAT Solver (von englisch *satisfiability*) entschieden werden. Da bei symbolischer Ausführung allerdings Formeln mit Prädikatenlogik erster Stufe auftreten, die Theorien über Zahlen und Datenstrukturen wie Listen und Arrays verwenden, ist ein SMT Solver (von english *satisfiability modulo theories*) nötig [DMB11].

Der Solver wird während der symbolischen Ausführung von Verzweigungspunkten verwendet, um die Path Condition auf Erfüllbarkeit zu prüfen. Auf diese Weise können Pfade von der weiteren Untersuchung ausgeschlossen werden. Beispielsweise würde es zu fälschlicherweise erkannten „Division durch Null"-Fehlern führen, wenn die Path Condition angibt, dass eine Variable ungleich 0 sein muss und das Testprogramm trotzdem eine Division mit dieser Variable durchführt. Auch wird verhindert, dass Pfade überprüft werden, die vom aktuellen Zustand aus nicht erreicht werden können. Abschließend wird der Solver verwendet, um konkrete Werte für die verwendeten symbolischen Variablen zu erzeugen, die als Eingaben für die Anwendung genutzt werden können.

Die Anfragen, die an einen Solver gestellt werden, können in einem menschenlesbaren Format dargestellt werden. Auf diese Weise ist es möglich, die während der symbolischen Ausführung erzeugten Anfragen händisch zu kontrollieren und nachzuvollziehen.

2.6. LLVM

LLVM (Low Level Virtual Machine) ist eine Infrastruktur [LA04], die genutzt werden kann, um Code beliebiger Sprachen zu übersetzen, zu optimieren und für beliebige Architekturen ausführbar zu machen. Zu diesem Zweck kann LLVM als Compiler Framework verwendet werden. Dazu ist es notwendig einen Compiler (Frontend) bereitzustellen, der Programmcode übersetzt und LLVM in Form einer assembler-ähnlichen Zwischensprache (LLVM-Bitcode, auch LLVM-IR (Intermediate Representation)) zur Verfügung stellt. LLVM optimiert diesen Zwischencode und leitet diesen an das Backend weiter. Dieses wiederum erzeugt aus dem allgemeinen Zwischencode ausführbare Programme für beliebige Architekturen.

Das Ziel von LLVM ist, dass es einfach und modular möglich sein soll, die einzelnen Bereiche unabhängig voneinander zu erweitern. Beispielsweise genügt es ein Frontend für eine bisher nicht unterstützte Programmiersprache zu entwickeln, um diese für verschiedene Architekturen übersetzen zu können. Da die Quell-Programmiersprache bedeutungslos für die Zwischensprache ist, können dort Optimierungen unabhängig von der Sprache durchgeführt werden. Der modulare Aufbau von LLVM ist in Abbildung 2.4 dargestellt.

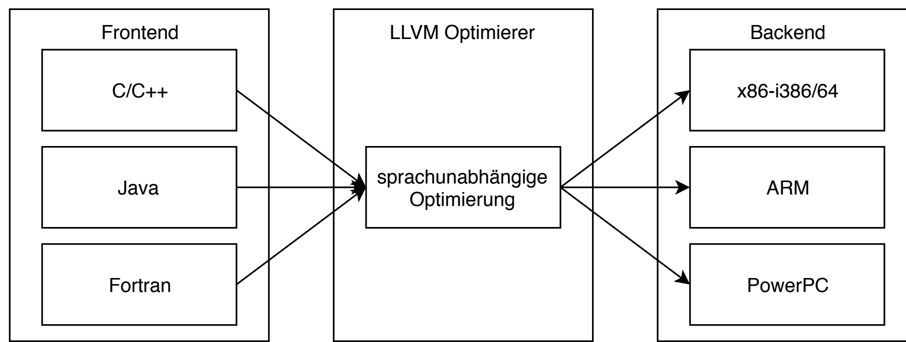


Abbildung 2.4.: Darstellung des modularen Aufbaus der LLVM-Infrastruktur.

- Im Frontend-Block sind alle Compiler enthalten, die einen Quelltext in einer bestimmten Programmiersprache verarbeiten können. Die Ausgabe der Compiler sind Instruktionen in der Zwischensprache LLVM-IR, die der LLVM-Optimierer verarbeiten kann. Um mit LLVM eine neue Programmiersprache zu unterstützen, muss lediglich ein Compiler für diese erstellt werden.
- Im LLVM Optimierer wird die Zwischensprache verarbeitet und eventuell Optimierungen oder Analysen unterzogen. Jede Optimierung oder Analyse wird in Form eines Passes bereitgestellt. Jeder Pass ist für sich eine abgeschlossene Einheit. Sollen mehrere Pässe ausgeführt werden, werden diese in Form einer Pipeline hintereinander gereiht.
- Durch das Backend wird der optimierte Code aus der Zwischensprache auf konkrete Instruktionen einer unterstützten Zielplattform abgebildet. Hierbei kann es nötig sein, nicht verfügbare Funktionalität durch Umschreiben des Codes zu emulieren. Hierbei könnte beispielsweise eine Vektoroperation durch eine Schleife ersetzt werden.

2.6.1. LLVM Datenstrukturen

Die folgende Auflistung gibt einen Überblick über die einzelnen Elemente, aus denen die LLVM-Zwischensprache logisch aufgebaut ist.

- **Module**[1]: Ein LLVM-Modul ist der übergeordnete Container, der alle übrigen Datentypen der Zwischensprache verwaltet. Er repräsentiert eine abgeschlossene Anwendung mit ihren Funktionen und Daten. Dazu enthält ein Modul unter anderem eine Liste aller globalen Variablen und allen definierten Funktionen. Außerdem ist hinterlegt, von welchen anderen Modulen (beispielsweise Laufzeitbibliotheken) dieses Modul abhängig ist.
- **Function**[2]: Eine LLVM-Funktion repräsentiert die im Quelltext definierte Funktion. Sie ist weiter in einzelne unabhängige Blöcke aufgeteilt.
- **BasicBlock**[3]: Ein LLVM-BasicBlock ist ein abgeschlossener Bereich einer Funktion, der als Container für die auszuführenden Instruktionen fungiert. Die einzelnen Blöcke innerhalb einer Funktion sind miteinander verknüpft, falls es möglich ist von einem Block zu einem anderen Block zu springen. Jeder Block entspricht hierbei einem Knoten im Kontrollflussgraphen (siehe Kapitel 2.3). Die Instruktionen innerhalb eines Blockes werden sequentiell in der Reihenfolge ihrer Definition ausgeführt.

[1] `http://llvm.org/doxygen/classllvm_1_1Module.html`
[2] `http://llvm.org/doxygen/classllvm_1_1Function.html`
[3] `http://llvm.org/doxygen/classllvm_1_1BasicBlock.html`

Ein BasicBlock ist „well-formed", wenn er aus Nicht-Terminal-Instruktionen gefolgt von einer Terminal-Instruktion besteht. Dass ein Block „well-formed" ist, wird durch LLVM vor der Verarbeitung geprüft.

- **Instruction**[4]: Eine LLVM-Instruktion beschreibt einen ausführbaren Befehl, wie beispielsweise eine Addition. Für die Analyse des Ausführungspfades sind vor allem die Terminal-Instruktionen von Bedeutung. Von diesen Instruktionen wird die Ausführung von einem BasicBlock zu mindestens einem anderen BasicBlock umgeleitet. Ein *if* im Quelltext entspricht einer Branch-Instruction, die zum jeweiligen *wahr*- bzw. *falsch*-BasicBlock leitet.

2.6.2. LLVM-Zwischensprache

LLVM-Bitcode ist eine Zwischensprache, in die beliebige Programmiersprachen übersetzt werden können. Die Zwischensprache kann in assembler-ähnlicher Form menschenlesbar gemacht werden.

LLVM-Bitcode ist in Static Single Assignment (SSA) Form notiert, bei der jede Variable nur ein einziges Mal zugewiesen wird. Diese Form eignet sich besser für Optimierungen. Im folgenden Beispielcode

```
1  x = 0;
2  x = 1;
3  y = x;
```

sieht man leicht, dass die erste Zuweisung unnötig ist und entfernt werden kann. Für einen Optimierer ist dieser Zusammenhang allerdings nicht so leicht erkennbar. Wird der Code wiefolgt in die SSA-Form umgewandelt

```
1  x1 = 0;
2  x2 = 1;
3  y1 = x2;
```

kann auch der Optimierer schnell die unnötige Zuweisung erkennen.

2.6.2.1. Beispiel

Abschließend soll der bereits in Abschnitt 2.4.1 vorgestellte Beispielcode 2.1 in LLVM-Bitcode gezeigt werden, wie er vom Compiler *Clang* erzeugt wurde. Um das Verständnis zu erleichtern, wurde jede Zeile kommentiert.

Quelltextausschnitt 2.2: Die Function *isOddOrGreaterTen* in LLVM-Bitcode mit eingefügten Kommentaren.

```
1  define i32 @isOddOrGreaterTen(i32 %x) {
2    %1 = alloca i32, align 4       // Variable für Ergebnis
3    %2 = alloca i32, align 4       // Variable für Parameter
4    store i32 %x, i32* %2          // Parameter x in %2
       kopieren
5    %3 = load i32* %2              // Wert von %2 in %3
       laden
6    %4 = srem i32 %3, 2            // Modulus mit 2
       berechnen
7    %5 = icmp eq i32 %4, 1         // %4 mit 1 vergleichen
```

[4] http://llvm.org/doxygen/classllvm_1_1Instruction.html

```
8    br i1 %5, label %6, label %7    // Sprung zu %6 oder %7
9
10 ; <label>:6
11   store i32 1, i32* %1            // 1 als Ergebnis
        speichern
12   br label %12                    // zu %12 springen
13
14 ; <label>:7
15   %8 = load i32* %2               // Wert von %2 in %8
        laden
16   %9 = icmp sgt i32 %8, 10        // %8 mit 10 vergleichen
17   br i1 %9, label %10, label %11 // Sprung zu %10 oder %11
18
19 ; <label>:10
20   store i32 1, i32* %1            // 1 als Ergebnis
        speichern
21   br label %12                    // zu %12 springen
22
23 ; <label>:11
24   store i32 0, i32* %1            // 1 als Ergebnis
        speichern
25   br label %12                    // zu %12 springen
26
27 ; <label>:12
28   %13 = load i32* %1              // Wert von %1 in %13
        laden
29   ret i32 %13                     // %13 zurückgeben
30 }
```

2.6.3. LLVM-Pässe

Pässe werden in LLVM zu Optimierungs- und Analysezwecken eingesetzt [LLV08]. Sie sind in verschiedene Geltungsbereiche aufgeteilt, die jeweils angeben, auf welche Daten ein Pass Zugriff hat. Nur in dem ihm zugeordneten Bereich ist es einem Pass erlaubt, Änderungen oder Optimierungen durchzuführen. Wenn durch einen Pass Daten modifiziert wurden, können bereits ausgeführte Pässe erneut durchlaufen werden, da sich für diese eventuell die Ausgangslage der Daten geändert hat. Ein Beispiel hierfür könnte ein Pass sein, der nicht erreichbaren Code entfernt. Durch eine Änderung eines anderen Passes könnten Funktionen wegoptimiert worden sein, die im vorherigen Durchlauf noch verwendet wurden.

- ModulePass[5]: Dieser Pass erhält als Eingabe das komplette Modul. Aus diesem Grund hat er Zugriff auf alle Daten und Funktionen.
- FunctionPass[6]: Dieser Pass wird einzeln mit allen Funktionen eines Moduls aufgerufen. Der Zugriffsbereich beschränkt sich auf alle Daten, die diese Funktion betreffen. Da hiermit sichergestellt ist, dass keine Daten außerhalb der Funktion geändert werden, kann LLVM diese Art von Pässen parallel auf mehreren Funktionen gleichzeitig ausführen.
- LoopPass[7]: Bei diesem Pass ist die Eingabe jede zuvor erkannte Schleife innerhalb einer Funktion. Bei geschachtelten Schleifen wird der Pass ausgehend von der inneren

[5] http://llvm.org/doxygen/classllvm_1_1ModulePass.html
[6] http://llvm.org/doxygen/classllvm_1_1FunctionPass.html
[7] http://llvm.org/doxygen/classllvm_1_1LoopPass.html

Schleife aufgerufen. Eine Schleife ist in diesem Kontext als eine Liste von BasicBlocks definiert, aus denen die Schleife besteht.

- BasicBlockPass[8]: Dieser Pass wird für jeden BasicBlock innerhalb einer Funktion aufgerufen. Ein BasicBlockPass ist die feingranularste Art von Pässen, da ein Pass auf Instruktionsebene keinen Sinn ergibt. Bei diesem stünde dem Pass keinerlei Informationen zum Kontext zur Verfügung.

Um die Wiederverwendbarkeit zu verbessern und Funktionalitäten unter den Pässen zu teilen, ist es möglich, dass Pässe Abhängigkeiten zu anderen Pässen angeben. LLVM sorgt vor der Ausführung eines Passes dafür, dass alle registrierten Abhängigkeiten geladen und ausgeführt wurden. Der Pass kann damit auf die von diesen Pässen erzeugten Analysen zugreifen. Ein Pass der beispielsweise Schleifen optimieren möchte, muss dazu keine Schleifenerkennung implementieren. Er registriert lediglich einen Pass, der diese Aufgabe übernimmt, als Abhängigkeit und kann während der Ausführung auf dessen Ergebnisse zugreifen.

2.7. KLEE

KLEE [CDE+08] ist eine symbolische virtuelle Maschine, die auf der LLVM-Infrastruktur aufbaut. Die Hauptfunktion von KLEE ist das Erzeugen von Software-Testfällen durch symbolische Ausführung. Es greift dabei auf die Ausführungspfad-Analyse zurück und erzeugt durch einen Solver Variablenbelegungen, die in den Testfällen eine hohe Abdeckung erzielen. KLEE gehört zur Gruppe der *Dynamic Symbolic Execution*-Programme, die sowohl konkreten Code als auch symbolische Ausführung unterstützen.

Neben der Erzeugung von Testfällen kann KLEE außerdem Programme auf häufig auftretende Fehler wie Division durch Null oder Speicherzugriffsfehler testen. Spezielle Operationen wie z.B. die Division sind als gefährliche Operationen gekennzeichnet, bei denen diese zusätzlichen Tests ausgeführt werden. Falls ein solcher Fehler gefunden wird, wird dieser gesondert mit den zu ihm führenden Eingabeparametern aufgelistet.

Die Hauptbestandteile von KLEE sind der *Executor* (engl. für Ausführer) und die von ihm verwendete *Solver Chain* (engl. für Löser-Kette).

- Der Executor führt die Instruktionen der zu testenden Anwendung aus und verwaltet die hierbei gefunden Pfade.
- Die Solver Chain ist eine Verkettung von Solvern, die unterschiedliche Funktionen wie Caches und Logging bieten.

2.7.1. KQuery

KLEE verwendet eine eigene Sprache *KQuery* [KLE08] um die Anfragen an die Solver zu beschreiben. Diese kann während der Ausführung in einem menschenlesbaren Format zur Kontrolle ausgegeben werden. Mit dem Programm *kleaver* lassen sich Anfragen aus dem KQuery Format einlesen und ausführen.

In folgendem Listing ist die Ausgabe der KQuery-Anfragen von KLEE dargestellt, die bei der Analyse der *isOddOrGreaterTen* Funktion erzeugt werden.

```
1 # Query 0 -- Type: Validity, Instructions: 19
2 array x[4] : w32 -> w8 = symbolic
3 (query [] (Eq 1
```

[8] http://llvm.org/doxygen/classllvm_1_1BasicBlockPass.html

```
4                 (SRem w32 (ReadLSB w32 0 x)
5                           2)))
6  #   OK -- Elapsed: 2.283704e-02
7  #   Validity: 0
8
9  # Query 1 -- Type: Validity, Instructions: 25
10 array x[4] : w32 -> w8 = symbolic
11 (query [(Eq false
12              (Eq 1
13                  (SRem w32 N0:(ReadLSB w32 0 x)
14                            2)))]
15         (Slt 10 N0))
16 #   OK -- Elapsed: 2.431798e-02
17 #   Validity: 0
18
19 # Query 2 -- Type: InitialValues, Instructions: 28
20 array x[4] : w32 -> w8 = symbolic
21 (query [(Eq 1
22              (SRem w32 (ReadLSB w32 0 x)
23                        2))]
24         false []
25         [x])
26 #   OK -- Elapsed: 3.600121e-05
27 #   Solvable: true
28 #     x = [1,0,0,0]
29
30 # Query 3 -- Type: InitialValues, Instructions: 33
31 array x[4] : w32 -> w8 = symbolic
32 (query [(Eq false
33              (Eq 1
34                  (SRem w32 N0:(ReadLSB w32 0 x)
35                            2)))
36          (Eq false (Slt 10 N0))]
37         false []
38         [x])
39 #   OK -- Elapsed: 6.103516e-05
40 #   Solvable: true
41 #     x = [0,0,0,0]
42
43 # Query 4 -- Type: InitialValues, Instructions: 37
44 array x[4] : w32 -> w8 = symbolic
45 (query [(Eq false
46              (Eq 1
47                  (SRem w32 N0:(ReadLSB w32 0 x)
48                            2)))
49          (Slt 10 N0)]
50         false []
51         [x])
52 #   OK -- Elapsed: 4.100800e-05
53 #   Solvable: true
54 #     x = [12,0,0,0]
```

An den einzelnen Abfragen kann man erkennen, dass sie aufeinander aufbauen und mit den jeweiligen Bedingungen wie $x > 10$ erweitert werden. Außerdem sind in den Zeilen 28, 41 und 54 konkrete Werte für die symbolische Variable x angegeben, die die entsprechenden Bedingungen erfüllen.

2.7.2. Solver Chain

Die Solver Chain von KLEE ist in mehrere unabhängige Schichten aufgeteilt, die Anfragen selbst bearbeiten oder an die nächste Schicht weiterleiten können. Die einzelnen Schichten bilden dabei eine Kette, bei der die einzelnen Glieder je nach Einstellung aktiviert oder deaktiviert werden können. Es wurde diese Art von Architektur gewählt, da sie es ermöglicht die Anfragen bereits in Schichten zu bearbeiten, die weniger teuer in der Ausführung sind. Auch können Anfragen transformiert oder optimiert werden, damit tiefere Schichten diese besser verarbeiten können.

Die Abbildung 2.5 zeigt die schematische Darstellung der Solver Chain in KLEE.

- **Logging Solver**: Die Aufgabe der Logging Schicht ist es, die Anfragen zu protokollieren und auszugeben (siehe Kapitel 2.7.1). Die Schicht ist doppelt vorhanden, da die Anfragen durch die Verarbeitung in den folgenden Schichten geändert oder bereits verarbeitet werden können. Auf diese Weise ist es möglich sowohl alle Anfragen, als auch nur diese zu protokollieren, die von den eigentlichen Solvern gelöst werden müssen.
- **Validating Solver**: Der Validating Solver ist in den Standardeinstellungen deaktiviert und dient der Fehlersuche in KLEE selbst. Seine Aufgabe ist es, eine Anfrage an mehrere Core Solver zu delegieren und anschließend zu überprüfen, ob beide Solver ein identisches Ergebnis berechnet haben.
- **Independent Solver**: Der Independent Solver untersucht die Anfrage auf Abhängigkeiten und versucht die Anfrage in mehrere kleinere Anfragen aufzuteilen. Der Vorteil von kleineren Anfragen besteht darin, dass diese einfacher zu lösen sind und in Verbindung mit Caching Schicht zu einer besseren Trefferquote führen.
- **Caching Solver**: Der Caching Solver enthält einen Zwischenspeicher, in dem zu Anfragen die berechneten Ergebnisse gespeichert werden. Wird eine Anfrage erneut ausgeführt, kann das passende Ergebnis aus dem Cache geladen werden, ohne dass es nochmals berechnet werden muss.
- **Core Solver**: Die tiefste Schicht enthält die eigentlichen SMT Solver, die die Anfragen lösen müssen. KLEE unterstützt drei verschiedene Solver, die per Programmeinstellung ausgewählt werden.
  - **STP**[9]: Der *Simple Theorem Prover* ist der Standard-Solver von KLEE. Zu Beginn der Entwicklung von KLEE war ausschließlich STP als Solver verfügbar, sodass er am besten integriert und getestet ist. STP wurde auf Basis von EXE entwickelt [CGP+08].
  - **Z3**[10]: Der Z3 Solver wurde von Microsoft Research entwickelt und kann in verschiedenen Programmiersprachen und Laufzeitumgebungen wie C/C++, Java und .NET genutzt werden [DMB08].
  - **MetaSMT**[11]: MetaSMT ist ein Framework, das als Abstraktionsschicht für unterschiedliche Solver dient [HFF+11, RSW+14]. Auf diese Weise können Solver verwendet werden, für die KLEE keine direkte Unterstützung bietet.

[9] `https://stp.github.io/`
[10] `https://github.com/Z3Prover/z3`
[11] `http://www.informatik.uni-bremen.de/agra/eng/metasmt.php`

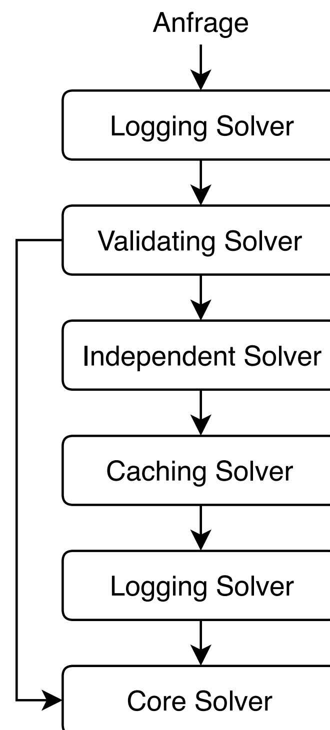


Abbildung 2.5.: Die Solver Chain von KLEE.

2.7.3. Datenverarbeitung

Durch den Funktionsaufruf von `klee_make_symbolic` im Quellcode wird KLEE mitgeteilt, welche Variablen als symbolische Variablen verwendet werden sollen. Nicht als symbolisch gekennzeichnete Variablen werden konventionell bei der Ausführung behandelt. Mit diesen Variablen wird keine gesonderte Pfadanalyse durchgeführt und es werden auch keine Testfälle erzeugt. Falls keine Codeänderungen durchgeführt werden können, ermöglicht es KLEE außerdem die Parameter der Standardeingabe symbolisch zu machen.

Als Eingabe benötigt KLEE Programmcode in Form von LLVM-Bitcode (siehe 2.6.2), da es keine Tests auf nativen Anwendungen durchführen kann. Der einfacher zu verarbeitende Zwischencode durchläuft anschließend die symbolische Ausführungspfadanalyse und es werden zu den Pfaden passende Testfälle erzeugt. Als Ausgabe erzeugt KLEE mehrere Dateien, die Informationen über die Ausführung enthalten.

info
In der *info* Datei sind allgemeine Informationen über die Ausführung enthalten. Es wird angezeigt, wie lange der komplette Vorgang gedauert hat, wie viele Pfade dabei insgesamt gefunden wurden. Außerdem wird die Anzahl der Instruktionen, die auf allen Pfaden ausgeführt wurden, angegeben.

run.stats
Neben der *info* Datei enthält die Datei *run.stats* viele weitergehende statistische Informationen. Der Inhalt der Datei kann mit der Anwendung *klee-stats* angezeigt werden. Zu den angezeigten Informationen gehört unter anderem wie viele Instruktionen verarbeitet wurden und welche Abdeckung im Code dabei in Prozent erreicht wurde. Außerdem wird der Speicherverbrauch während der Ausführung und die Anzahl von an den Solver übergebenen Abfragen angezeigt.

warnings.txt, **errors.txt**

Die Dateien *warnings.txt* und *errors.txt* enthalten jeweils von KLEE während der Ausführung ausgegebene Warnungen bzw. Fehler.

all-queries.kquery, **solver-queries.kquery**
Die beiden Dateien *all-queries.kquery* und *solver-queries.kquery* enthalten die durch die symbolische Ausführung erzeugten Abfragen für den Solver. In *all-queries.kquery* werden alle Anfragen protokolliert, die an die Solver Chain (siehe Kapitel 2.7.2) gestellt werden. In *solver-queries.kquery* sind wiederum nur die Anfragen enthalten, die tatsächlich an einen SMT Solver weitergeileitet wurden. Alle Anfragen sind menschenlesbar im Format KQuery (siehe Kapitel 2.7.1) gespeichert.

test<N>.ktest
Pro gefundenem Pfad wird eine *test<N>.ktest* Datei angelegt, die mit der Anwendung *ktest-tool* ausgewertet werden kann. Die Datei enthält Daten über die symbolischen Variablen und deren Belegung mit konkreten Werten, die zu diesem Pfad geführt haben. Ein Beispiel für den Inhalt einer solchen Test-Datei zeigt Abschnitt 2.7.4.

2.7.4. Beispiel

Um den in 2.1 gezeigten Beispielcode mit KLEE zu testen, muss die Funktion in ein gültiges Programm eingebettet werden. Dazu muss eine *main*-Funktion erzeugt und in dieser der Parameter als symbolische Variable der Funktion *isOddOrGreaterTen* übergeben werden. Das vollständige Programm ist in Beispielcode 2.3 zu sehen.

Quelltextausschnitt 2.3: Vollständiger Inhalt der Datei *isOddOrGreaterTen.c* zur Verwendung mit KLEE.

```
1  #include <klee/klee.h>
2
3  int isOddOrGreaterTen(int x)
4  {
5     if (x % 2 == 1)
6     {
7        return 1;
8     }
9
10    if (x > 10)
11    {
12       return 1;
13    }
14
15    return 0;
16 }
17
18 int main()
19 {
20    int x;
21    klee_make_symbolic(&x, sizeof(x), "x");
22    return isOddOrGreaterTen(x);
23 }
```

Mit der Befehlszeile

```
1 clang -emit-llvm -c -g isOddOrGreaterTen.c
2 klee isOddOrGreaterTen.bc
```

wird das Programm mit dem Compiler *Clang* übersetzt. Durch den Parameter *emit-llvm* wird der Compiler angewiesen, den Quellcode in LLVM-Bitcode zu übersetzen. Mit diesem als Eingabe wird KLEE aufgerufen. KLEE erzeugt bei der Ausführung die folgende Konsolenausgabe.

```
1 #: klee isOddOrGreaterTen.bc
2 KLEE: output directory is "isOddOrGreaterTen/klee-out-1"
3 KLEE: Using STP solver backend
4
5 KLEE: done: total instructions = 37
6 KLEE: done: completed paths = 3
7 KLEE: done: generated tests = 3
```

Es wurden wie erwartet drei Pfade durch die Funktion gefunden. Pro Pfad wurde eine *.ktest* Datei erzeugt, die Informationen über die gewählte Variablenbelegung enthält.

```
1 #: ktest-tool --write-ints klee-out-1/test000001.ktest
2 ktest file : 'klee-out-1/test000001.ktest'
3 args       : ['isOddOrGreaterTen.bc']
4 num objects: 1
5 object    0: name: 'x'
6 object    0: size: 4
7 object    0: data: 1
8
9 #: ktest-tool --write-ints klee-out-1/test000002.ktest
10 ...
11 object    0: data: 0
12
13 #: ktest-tool --write-ints klee-out-1/test000003.ktest
14 ...
15 object    0: data: 12
```

Als Werte mit hoher Abdeckung wurden demnach die Zahlen 1, 0 und 12 berechnet. Würde man den Code neu übersetzen und die Funktion dabei mit allen drei Zahlen aufrufen, würde jeder Ausführungspfad der Funktion durchlaufen.

2.7.5. Einschränkungen von KLEE

Die meisten Einschränkungen von KLEE rühren vom allgemeinen Konzept der symbolischen Ausführung her.

2.7.5.1. Pfad Explosion

Ein großes Problem bei der symbolischen Ausführung stellt die Pfad Explosion dar (siehe Kapitel 2.4.2). Die Anzahl der Pfade wächst exponentiell an jedem Verzweigungspunkt. Dies kann bereits ab kleinen Programmen dazu führen, dass die Menge der zu durchlaufenden Pfade zu groß wird, als dass sie sinnvoll verarbeitet werden können. KLEE nutzt daher verschiedene Heuristiken um identische oder ähnliche Pfade zu finden und zusammenzufassen.

2.7.5.2. Solver

Ein weiteres Problem ist der Solver, der zur Path Condition Variablenbelegungen erzeugen muss (siehe Kapitel ch:Grundlagen:sec:Solver). Wenn die Liste der Bedingungen zu komplex wird, ist es eventuell nicht möglich in annehmbarer Zeit eine Lösung zu berechnen.

Um den Solver möglichst selten nutzen zu müssen, versucht KLEE die Pfad-Bedingungen falls möglich zu vereinfachen. Auch werden Ergebnisse des Solvers zwischengespeichert, um den Aufruf in einem identischen Fall nicht wiederholen zu müssen.

2.7.5.3. Externe Funktionen

Da KLEE native Programme nicht verarbeiten kann, ist es notwendig, das zu testende Programm in LLVM-Bitcode umzuwandeln, da nur dieser auf der virtuellen Maschine von KLEE ausgeführt werden kann. Allerdings bestehen Anwendungen üblicherweise nicht nur aus dem vorliegenden Quellcode, sondern es sind noch weitere externe Bibliotheken eingebunden. Auch alle verwendeten Bibliotheken müssen daher als LLVM-Bitcode vorliegen, um von der Anwendung genutzte Bibliotheksfunktionen testen zu können. Für die weit verbreitete POSIX-C-Bibliothek wird von KLEE eine angepasste Version der uClibc-Bibliothek angeboten. Alle dort enthaltenen Funktionen können von KLEE untersucht werden.

Falls es nicht möglich ist LLVM-Bitcode zu erzeugen, weil zu einer Bibliothek der Quellcode nicht verfügbar ist, kann die Funktion von KLEE nicht untersucht werden. Beim Aufruf der Funktion wird von KLEE die Warnung „KLEE: WARNING ONCE: calling external: ...“ ausgegeben. Sind für den Funktionsaufruf Zugriffe auf symbolische Variablen nötig, werden zu diesen durch den Solver konkrete Werte berechnet und im Speicher für die Funktion verfügbar gemacht. Anschließend wird die Funktion außerhalb der Kontrolle von KLEE ausgeführt. Falls durch die Funktion Speicher verändert wurde, werden diese Änderungen an den von KLEE verwalteten Speicherobjekten nachgezogen.

2.7.6. Weitere Anwendungsgebiete

Seit der Vorstellung von KLEE im Jahr 2008 [CDE+08] wurde eine Vielzahl von Projekten aufbauend auf KLEE entwickelt.

2.7.6.1. KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment

KleeNet [SLA+10] wurde an der RWTH-Aachen entwickelt und dient dem Testen von Übertragungsprotokollen bei Sensornetzwerken. Das Ziel war es Fehler zu entdecken, die durch seltene oder nicht vorhersehbare Ereignisse ausgelöst werden. Zu den Ereignissen zählen beispielsweise Paketverluste oder der Ausfall eines Knotens. Da diese Ereignisse oft nicht leicht zu modellieren sind, ist es schwer für diese Testfälle zu erzeugen. Um dennoch Testfälle erstellen zu können, wurde KLEE dahingehend angepasst, dass Netzwerkprotokolle mit symbolischen Eingaben verwendet werden können und damit umfassender zu testen sind.

2.7.6.2. Analyzing Protocol Implementations for Interoperability

PIC [PFK+15] ist ein auf KLEE aufbauendes Testprogramm für unterschiedliche Protokollimplementierungen. Durch die symbolische Ausführung wird getestet, ob die verschiedenen Implementierungen eines Protokolls korrekt miteinander interagieren.

2.7.6.3. Server-side Verification of Client Behavior in Online Games

In diesem Projekt [BCR11] wurde eine Methode entwickelt, mit der in Online-Spielen serverseitig überprüft werden kann, ob sich ein Client an die Spielregeln hält. Ein Problem von Online-Spielen ist das Schummeln (auch Cheating), das dem Spieler Vorteile gegenüber den anderen Spielern bringt. Da der Spieler die volle Kontrolle über den Client besitzt, kann nur auf der Serverseite effektiv überprüft werden, ob ein Client manipuliert wurde.

Mit KLEE wurde durch symbolische Ausführung untersucht, ob Eingaben, die von einem Client zum Server gesendet wurden, manipuliert sind. Dazu wird geprüft, ob der Wert der Eingabe über einen im Clientcode zu erreichenden Ausführungspfad erzeugt werden konnte. Führt kein Pfad zu diesem Wert, war die Eingabe sehr wahrscheinlich manipuliert.

2.7.6.4. GKLEE: Concolic Verification and Test Generation for GPUs

GKLEE („GPU + KLEE“) [LLS+12] ist eine Erweiterung zu KLEE, die es ermöglicht Anwendungen zu testen, die die CUDA[12] Bibliothek von NVIDIA verwenden, um rechenintensive Programmteile auf die Grafikkarte auszulagern. GKLEE analysiert hierbei sowohl die eigentliche Anwendung, als auch die Programm-Kernels, die in den Recheneinheiten der Grafikkarte ausgeführt werden.

2.7.6.5. KLOVER: Automatic Test Generation for C and C++ Programs, Using Symbolic Execution

KLEE kann aktuell nur Anwendungen untersuchen, die in der Programmiersprache C geschrieben sind. Mit KLOVER [YLK+17] wurde eine Erweiterung für KLEE geschaffen, die es ermöglicht automatisch Testfälle für Programme zu erzeugen, die in C++ entwickelt wurden. Dazu musste Unterstützung für C++ Features wie Ausnahme-Behandlung und objektorientierte Entwicklung implementiert werden. Außerdem wurde ähnlich der uClibc-Bibliothek, die uClib++ für KLEE angepasst, um Programmen die Verwendung der Standardbibliothek zu ermöglichen.

[12] https://www.nvidia.de/object/cuda-parallel-computing-de.html

3. Verwandte Arbeiten

Dieses Kapitel widmet sich der Vorstellung von verwandten Forschungsarbeiten, die ebenfalls Themen rund um Performanzabschätzung von parallelen Programmen untersucht haben. Bei manchen Arbeiten wurde zudem auch symbolische Ausführung eingesetzt, um mögliche Eingabeparameter zu bestimmen.

3.1. Implementation of Path Profiling in the Low-Level Virtual-Machine (LLVM) Compiler Infrastructure

In dieser Arbeit [Pre10] von Adam Preus wurde ein Compiler-Pass für LLVM entwickelt, der *Naïve Edge Profiling* kombiniert mit *Path Profiling* als *Combined Profiling* zur Verfügung stellt. Mit den daraus gewonnenen Informationen kann es einem Compiler ermöglicht werden, bei einem weiteren Übersetzungsvorgang effizienteren Code zu generieren. Bei jedem Durchlauf des Programms mit unterschiedlichen Eingabeparametern werden weitere Informationen gesammelt. Da keine symbolische Ausführung verwendet wird um alle möglichen Pfade zu bestimmen, stehen dem Compiler allerdings nicht zu jedem Pfad Optimierungsinformationen zur Verfügung.

3.2. ParaSCAN: A Static Profiler to Help Parallelization

ParaSCAN [URS14] von Ganesha Upadhyaya, Tyler Sondag und Hridesh Rajan ist ein statischer Profiler für in Java geschriebene Anwendungen. Neuartig am verwendeten Ansatz ist, dass für das statische Profilen Informationen von verschiedenen Analysen herangezogen werden. Dazu werden die wahrscheinliche Ausführungshäufigkeit eines Pfades und dessen Gewichtung kombiniert.

Es wird angenommen, dass der größtmögliche Nutzen für Parallelisierung in der Untersuchung der Average Case Exceution Paths liegt. Diese sind definiert als die Pfade, die im Durchschnitt oft durchlaufen werden und dabei einen großen Anteil der Arbeit eines Programms verrichten. Bisher wurden diese Pfade durch Informationen gewonnen, bei denen ein Programm wiederholt mit verschiedenen Eingaben ausgeführt und dabei die verwendeten Pfade aufgezeichnet wurden. Mit ParaSCAN ist es möglich diese Pfade auch allein durch statische Analyse zu finden.

Um dies zu erreichen, müssen zwei Probleme gelöst werden:

1. Statische Bestimmung der Ausführungshäufigkeit eines Pfades

2. Statische Bestimmung des Gewichts (beispielsweise die Ausführungsdauer, Anzahl von Prozessorzyklen, ...) eines Pfades

Um die Ausführungshäufigkeit eines Pfades statisch bestimmen zu können, wurde die Profile-Methode von Wu und Larus [WL94] angewendet, die wiederum die Heuristiken von Ball und Larus [BL96] verwendet. Um statisch das Gewicht eines Pfades bestimmen zu können, wird ein Java-Interpreter verwendet. Dieser wandelt für einige Basis-Instruktionen den Java-Bytecode in nativen Maschinencode um. Für diesen wird wiederum die Anzahl an Prozessorzyklen bestimmt. Die Laufzeit der übrigen Instruktionen wurde geschätzt, indem untersucht wurde, aus welchen Basis-Instruktionen sich diese zusammensetzen. Auf diese Weise ergibt sich kein realistisches Kostenmodell, das allerdings für die Anforderungen auch nicht benötigt wird. Wichtig ist nur bestimmen zu können, dass ein Pfad schwerer als ein anderer Pfad ist.

3.3. Input-Sensitive Profiling

In dieser Arbeit [CDF12] von Emilio Coppa, Camil Demetrescu und Irene Finocchi wurde ein Profiler entwickelt, der zu jeder im Testlauf ausgeführten Funktion die Ausführungskosten pro Eingabegröße bestimmt. Dies kann genutzt werden, um Performanz-Flaschenhälse bzgl. der auftretenden Arbeitslast zu finden.

Die Eingabegröße wird durch die *read memory size*-Metrik (RMS) bestimmt. Diese ist definiert als die Anzahl der Variablen, die von einer Funktion zum ersten Mal gelesen werden und damit gleichbedeutend zu Größe der Eingabe ist. Wird eine Variable innerhalb der Funktion beschrieben und anschließend erneut gelesen, wird dieser Lesezugriff nicht erneut gezählt. Beispielsweise hat eine Funktion, die zwei Zahlen addiert eine RMS von 2, da die zwei Eingabeparameter gelesen werden müssen. Eine Funktion die eine Liste und eine Zahl mit der Anzahl der Elemente der Liste als Eingabe erhält um diese zu addieren, hat eine RMS von $n + 2$ (Lesen der Listen- und Anzahl-Variable plus Lesen von n Listen-Einträgen).

Für jeden Funktionsaufruf können nun pro Eingabegröße die minimalen und maximalen Ausführungskosten bestimmt werden. Aus einer Reihe von Messungen mit unterschiedlichen Eingabegrößen kann anschließend abgelesen werden, welche Eingabegrößen zu einer niedrigen bzw. hohen Ausführungszeit führen.

3.3.1. Multithreaded Input-Sensitive Profiling

Diese Arbeit [CDFM13] erweitert das bisherige Input-Sensitive Profiling (siehe Kapitel 3.3) um auf parallelen Anwendungen einsetzbar zu sein. Die bisherige Herangehensweise ignorierte Thread-Kommunikation (z.B. der Austausch von Daten über eine gemeinsame Variable) innerhalb einer Funktion. Eine Folge davon ist, dass in diesem Fall die *read memory size* falsch bestimmt wird. Es wurde daher eine angepasste Metrik *threaded read memory size* eingeführt. Bei dieser wird auch ein erneutes Lesen einer Variable in einer Funktion mitgezählt, falls diese Variable seit dem vorherigen Lesevorgang durch einen anderen Thread beschrieben wurde.

3.4. Generating Performance Distributions via Probabilistic Symbolic Execution

In dieser Arbeit [CLL16] von Bihuan Chen, Yang Liu und Wei Le wurde das Performanz-Analyse Framework PerfPlotter in und für Java entwickelt. Es kann verwendet werden, um den Best- bzw. Worst-Case und die Verteilung der Laufzeit des getesteten Programms bzgl. eines Verwendungsprofils zu bestimmen.

In einem Verwendungsprofil wird angegeben, mit welcher Wahrscheinlichkeit die Eingabevariablen bestimmte Werte annehmen. Dazu wird für jede Eingabevariable ein Intervall angegeben und zu diesem die Wahrscheinlichkeit des Auftretens. Über die angegebenen Wahrscheinlichkeiten kann bei der Pfadsuche bestimmt werden, welche Abzweigungen unter Verwendung des gewählten Profils am wahrscheinlichsten genommen werden. Auf diese Weise ist es möglich, die Anzahl der zu prüfenden Pfade zu verringern und das Problem der kombinatorischen Explosion zu umgehen.

3.5. WISE: Automated Test Generation for Worst-Case Complexity

WISE (Worst-case Inputs from Symbolic Execution) [BJS09] wurde von Jacob Burnim, Sudeep Juvekar und Koushik Sen in Java entwickelt und dient dem Erzeugen von Testfällen mit Eingabedaten, die den Worst-Case bzgl. der Ausführungsgeschwindigkeit provozieren.

Um das Problem der kombinatorischen Explosion durch zu große Eingabedaten zu umgehen und trotzdem sinnvolle Testfälle erzeugen zu können, wurde ein dreistufiges Verfahren entwickelt.

Im ersten Schritt wird durch WISE eine symbolische Ausführung mit kleinen Eingabedaten durchgeführt. Auf diese Weise werden zwar nicht alle möglichen Pfade gefunden, aber der Algorithmus erfährt zu jeder Eingabe den schwersten Pfad.

Im zweiten Schritt wird aus den gelernten schwersten Pfaden ein Generator erzeugt, mit dem sich neue Pfade bestimmen lassen. Da der Generator nur mit schweren Pfaden angelernt wurde, führen auch seine Ausgaben zu schweren Pfaden. Dazu wird zu jeder Abzweigung im Pfad bestimmt, ob sie *frei* oder *vorbestimmt* ist. Bei einer freien Abzweigung dürfen beide Pfade gewählt werden. Bei einer vorbestimmten Abzweigung muss hingegen der bestimmte Pfad weiterverfolgt werden. Auf diese Weise können Teile des Ausführungsbaums ignoriert werden. Dies führt dazu, dass die Anzahl zu testender Pfade minimiert wird.

Im dritten und letzten Schritt können nun durch symbolische Ausführung für die vom Generator ausgegebenen Pfade Eingabedaten bestimmt werden. Da der symbolischen Ausführung durch den Generator vorgegeben wird, wie manche Abzweigungen zu nehmen sind, wird hierbei von der geführten symbolischen Ausführung gesprochen.

3.6. Symbolic Complexity Analysis Using Context-Preserving Histories

Die Arbeit [LKP17] von Kasper Luckow, Rody Kersten und Corina Păsăreanu greift die Grundidee von WISE (siehe Kapitel 3.5) auf. Erneut wird auf der Grundlage von symbolischer Ausführung mit kleinen Eingabedaten ein Regelwerk für Pfade erstellt, die zu langen Ausführungszeiten mit großen Eingabedaten führen. Umgesetzt wurde das Projekt als eine Erweiterung der *Symbolic PathFinder*[1] Anwendung zur Nutzung mit in Java geschriebenen Programmen.

Erweitert wurde das Regelwerk dahingehend, dass auch der Verlauf der bisherigen Entscheidungen für einen Pfad berücksichtigt wird. Wurde beispielsweise bei einer Liste mit 5 Einträgen für jedes Element ausschließlich dem letzten eine bestimmte Abzweigung gewählt, so wird dies auch bei einer Liste mit 6 Einträgen umgesetzt.

[1] `https://github.com/SymbolicPathFinder/jpf-symbc`

Zusätzlich ist das Regelwerk kontextabhängig. Dies ermöglicht unterschiedliche Entscheidungen für eine Funktion, je nachdem, in welchem Kontext diese aufgerufen wurde. Dabei wird davon ausgegangen, dass die Funktion aus einem Kontext heraus immer mit ähnlichen Eingabeparametern aufgerufen wird. Da sich dieses Verhalten von Kontext zu Kontext unterscheiden kann, wird der Verlauf dementsprechend neu bestimmt.

3.7. Directed symbolic execution

In der Arbeit „Directed symbolic execution“ [MPFH11] wird ein anderer umgekehrter Ansatz wie in den bisher vorgestellten Arbeiten verwendet. Es wird das Problem behandelt, dass durch die symbolische Ausführung von Anwendungen Pfade zu bestimmten Zeilen im Quelltext gefunden werden sollen. Das Problem tritt häufig bei der Fehlersuche während der Entwicklung auf, da beim Auftreten eines Fehlers im Normalfall nur die Quelltextzeile bekannt ist, aber nicht der Ausführungspfad mit konkreten Werten von Variablen und Parametern. Es ist somit nicht einfach möglich den Fehler zu reproduzieren, um ihn anschließend lösen zu können.

Zur Lösung des Problems werden zwei unterschiedliche Strategien für eine *geführte* symbolische Ausführung vorgestellt:

- **SDSE**: Die *shortest-distance symbolic execution* (SDSE, englisch für „Kürzeste Distanz Symbolische Ausführung“) verwendet als Metrik die Distanz des aktuellen Pfades zum gewünschten Ziel innerhalb des *Interprocedural Control-Flow Graph* (siehe Kapitel 2.3). Dazu wird an jedem Verzweigungspunkt überprüft, welche Weg schneller zum Ziel führt und dieser gewählt. Als „schneller“ ist hierbei der kürzeste Pfad im Graphen definiert. Um nicht an jedem Verzweigungspunkt den kürzesten Pfad berechnen zu müssen, werden diese einmalig zu Beginn geprüft.
- **CCBSE**: Die *call-chain-backward symbolic execution* (CCBSE, englisch für „Rückwärtige Funktionsaufrufe Symbolische Ausführung“) verwendet symbolische Ausführung ausgehend vom Ziel, um mögliche Aufrufer der Funktion zu bestimmen. Wird beispielsweise die Zeile l in Funktion f gesucht, startet die symbolische Ausführung zu Beginn der Funktion f um die Pfade zu l zu finden. Anschließend werden alle Aufrufer von f bestimmt. In jeder dieser Funktionen f_i wird wiederum die symbolische Ausführung verwendet, um die Pfade vom Beginn von f_i zum Aufruf von f zu bestimmen. Dieser Prozess wird solange wiederholt, bis ein Pfad vom Beginn der Anwendung zu l gefunden wurde.

3.8. Rapid Techniques for Performance Estimation of Processors

In der Arbeit „Rapid Techniques for Performance Estimation of Processors“ [RSW⁺10] wird ein Framework zur Einschätzung der Performance einer Anwendung auf unterschiedlichen Prozessoren vorgestellt. Als Eingabe wird die Anwendung in LLVM Zwischensprache benötigt. Aufgrund dessen kann auf die Erstellung von prozessorspezifischen Compilern und aufwendige Simulationen von Befehlssätzen verzichtet werden. Da nicht bekannt ist, wie der LLVM-Bitcode in spezifische Instruktionen für einen Prozessor abgebildet wird, muss der letztlich kompilierte Code approximiert werden. Dazu werden die LLVM Instruktionen in drei Kategorien eingeteilt:

1. **Simple Instructions**: Die Instruktionen haben eine äquivalente Abbildung in der Zielarchitektur. Ein Beispiel hierfür wäre die Addition *add*.

2. **Compound Instructions**: Die Instruktionen haben keine äquivalente Abbildung in der Zielarchitektur. Sie müssen in mehrere Simple Instructions aufgeteilt werden, die auf der Zielarchitektur verfügbar sind.

3. **Functions Calls**: Instruktionen, die eine Funktion aufrufen.

Um zu entscheiden, wie eine Instruktion für die Zielarchitektur compiliert würde, werden alle Instruktionen der Anwendung in die drei Kategorien eingeteilt. Da Simple Instructions äquivalente Abbildungen besitzen, erzeugen sie lediglich eine Instruktion. Instruktionen der Kategorie Compound Instructions werden compiliert, um zu bestimmen, auf welche Instruktionen sie abgebildet werden. Hieraus wurden für alle Instruktionen der Kategorie Gewichtungsformeln für die Zielarchitektur erstellt. Für Functions Calls wurde der gleiche Prozess wie bei Compound Instructions durchlaufen. Um zu bestimmen, wie oft eine Instruktion in der Anwendung ausgeführt wird, wurde der Profiler von LLVM verwendet. Aufgrund der nun verfügbaren Übersetzungen von LLVM Instruktionen in Instruktionen der Zielarchitektur und der Anzahl an ausgeführten LLVM Instruktionen kann das Framework schätzen, wie performant die Anwendung ausgeführt werden wird.

4. Analyse

Dieses Kapitel beschäftigt sich mit der Analyse der Anforderungen, die diese Arbeit erfüllen soll. In den Kapiteln 4.1 und 4.4 werden die zu erfüllenden Ziele der Arbeit definiert. Die Kapitel 4.2 und 4.3 zeigen die Grundidee des gewählten Lösungsansatzes und welche vorhandenen Elemente von KLEE bzw. LLVM für die Erfüllung der Ziele verwendet werden können.

4.1. Definition der Problemstellung

Im Rahmen dieser Arbeit soll die bisherige Funktionalität von KLEE erweitert werden. Durch die symbolische Ausführung sollen nicht nur Belegungen für Variablen gefunden werden, die einen möglichst hohen Anteil an Ausführungspfaden abdecken, sondern außerdem die Ausführungskosten der gewählten Pfade, in Abhängigkeit der gewählten Variablen, berechnet werden. Hierzu muss der Quellcode von KLEE angepasst werden, damit dieser als Profiler zusätzlich Informationen zu benötigten Ausführungskosten des aktuellen Pfades generiert.

Um dies zu erreichen, muss ein Modell zur Kostenberechnung von Programminstruktionen verwendet werden. Das Modell muss hierzu entscheiden können, wie der LLVM-Zwischencode in nativen Assemblercode übersetzt werden kann. Dies ist nötig, da eine Instruktion in der LLVM Zwischensprache nicht zwangsläufig einer Instruktion im Assemblercode entspricht (siehe Kapitel 3.8). Um eine möglichst allgemeingültige Sprache anzubieten, verwendet LLVM Abstrahierungen, die nicht von jeder konkreten Zielplattform unterstützt werden. Ein Beispiel hierfür ist die *switch* Instruktion[1]. Während diese in der LLVM Zwischensprache einer Instruktion entspricht, wird sie üblicherweise in eine *if else*-Kette übersetzt oder durch eine Lookup-Tabelle implementiert.

Mithilfe der Informationen der zu erwartenden Instruktionskosten ist es anschließend möglich Bereiche im Quelltext der Anwendung zu finden, die im Vergleich zum Rest lange dauern. Ein Grund für diese Hotspots werden voraussichtlich Schleifen sein, die aufgrund der aktuell gewählten symbolischen Variablen sehr oft durchlaufen werden. Da Schleifen bevorzugte Ziele zur Parallelisierung sind, kann es durch die Hotspot-Analyse ermöglicht werden, zu bestimmen, ob eine Schleife im Bezug auf die gewählten Parameter Parallelisierungspotential besitzt. Da die Struktur der LLVM-Zwischensprache „flach“ ist und es keine speziellen Instruktionen gibt, die Schleifen anzeigen, sind diese nicht direkt sichtbar. Aus

[1] `https://llvm.org/docs/LangRef.html#switch-instruction`

diesem Grund muss KLEE Schleifen erkennen und die hierbei verwendeten symbolischen Variablen bestimmen können.

Mithilfe dieser Informationen kann in Abhängigkeit der generierten Eingaben der best- und worst-case eines Programms in Bezug zu den Ausführungskosten bestimmt werden. Dies ermöglicht eine Kategorisierung der Parameter bezüglich ihres Einflusses auf die erzeugten Kosten. Als Ausgabe soll die erweiterte Version von KLEE eine Übersicht über die Ausführungskosten mit den unterschiedlichen Parametern liefern. Aus der Übersicht soll außerdem hervorgehen, welche Stellen im Programm für eine hohe Kostenentwicklung verantwortlich sind. Mithilfe des erzeugten Ergebnisses kann es dem Entwickler ermöglicht werden, die Funktionen zu verbessern, bei denen Parallelisierungspotential aufgrund von hohen Ausführungskosten besteht.

Ein weiteres Problem ist die hohe Laufzeit von KLEE aufgrund des Pfad Explosion Problems (siehe Kapitel 2.4.2). Es muss für KLEE eine Möglichkeit geschaffen werden, dass die Auswirkungen der Pfad Explosion reduziert werden. Außerdem bietet die bisherige Funktionsweise von KLEE weiteres Optimierungspotential. Da KLEE ursprünglich Fehler in den Ausführungspfaden finden soll, gibt es Funktionen, die beispielsweise bei Divisionen prüfen, ob durch Null geteilt wird. Da diese Funktionalität für diese Arbeit nicht benötigt wird und sollen sie deaktiviert werden.

4.2. Grundidee

Um die in Kapitel 4.1 dargestellten Probleme lösen zu können, soll KLEE an mehreren Stellen erweitert werden. Hierzu soll eine Pfadheuristik auf Grundlage einer geführten symbolischen Ausführung verwendet werden, die gezielt Pfade zu Funktionen bevorzugt, die Schleifen enthalten (siehe auch [MPFH11]). Auf dem Weg dorthin sollen die Ausführungskosten von einzelnen Instruktionen mit einem Kostenmodell bestimmt werden. Abschließend soll KLEE aus den gewonnenen Informationen den Hot-Path und Informationen über die „teuren“ Schleifen sowie die ihnen zugeordneten symbolischen Variablen ausgeben.

Die Abbildung 4.1 gibt einen groben Überblick über den geplanten Prozessablauf.

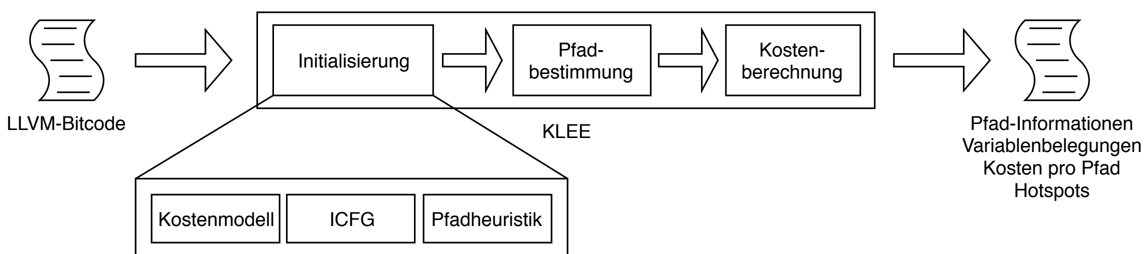


Abbildung 4.1.: Übersicht über den erweiterten Prozessablauf von KLEE.

Während der *Initialisierung* sollen möglichst viele Vorberechnungen durchgeführt werden, damit deren Ergebnisse zu späteren Zeitpunkten ohne weiteren Aufwand direkt genutzt werden können. Zu den Aufgaben der Initialisierung gehören:

- **Kostenmodell**: Das Kostenmodell und damit einhergehend die Ausführungskosten für einzelne Instruktionen können einmalig zu Beginn berechnet werden, da sich diese während der Ausführung nicht ändern. Jeder Pfad, der eine einzelne Instruktion ausführt, erzeugt hierbei dieselben Kosten. Um die Ausführungskosten eines Pfades zu bestimmen, müssen die Kosten der ausgeführten Instruktionen auf einem Pfad zusammengezählt werden.

- **Interprocedural Control-Flow Graph**: Als Grundlage für die Pfadheuristik muss ein Interprocedural Control-Flow Graph (siehe Kapitel 2.3) aufgebaut werden. Über diesen ist es möglich Pfade zu bestimmten Stellen im Programm zu bestimmen.
- **Pfadheuristik**: Aufgrund der Berechnung des Interprocedural Control-Flow Graph können zulässige Pfade durch die zu testende Anwendung bestimmt werden, die den Kontrollfluss gezielt in die Richtung von Funktionen lenken, die Schleifen enthalten.

Bei der *Pfadbestimmung* wird KLEE durch die zuvor erstellte Pfadheuristik ausgebremst, um das Problem der Pfad Explosion zu umgehen. Es werden nicht mehr alle möglichen Pfade der Anwendung durchlaufen, sondern nur noch gezielt die Pfade, die zu Schleifen in Funktionen führen.

Die *Kostenberechnung* der Pfade muss durch das Aufsummieren der durch das Kostenmodell bekannten Ausführungskosten der verwendeten Instruktionen ausgeführt werden. Hierbei ist es notwendig die Kosten für einzelne Codeblöcke nach Pfaden getrennt zu berechnen. Werden Pfade aufgespalten, müssen auch die Kosten auf beide Pfade übertragen werden.

Schlussendlich soll KLEE eine Ausgabe erzeugen, die Informationen zu den einzelnen Pfad nachvollziehbar darstellt. Zu den Informationen zählt:

- der Ausführungspfad, der anzeigt, welche Funktionen durchlaufen wurden
- die Ausführungskosten der Instruktionen auf diesem Pfad
- eine Übersicht, welche Funktionen den Großteil der Kosten verursachen

4.3. Voraussetzungen

In diesem Kapitel werden Bausteine vorgestellt, die KLEE bzw. LLVM bereits bieten und die als Teil dieser Arbeit Verwendung finden sollen.

4.3.1. Control Flow Graph

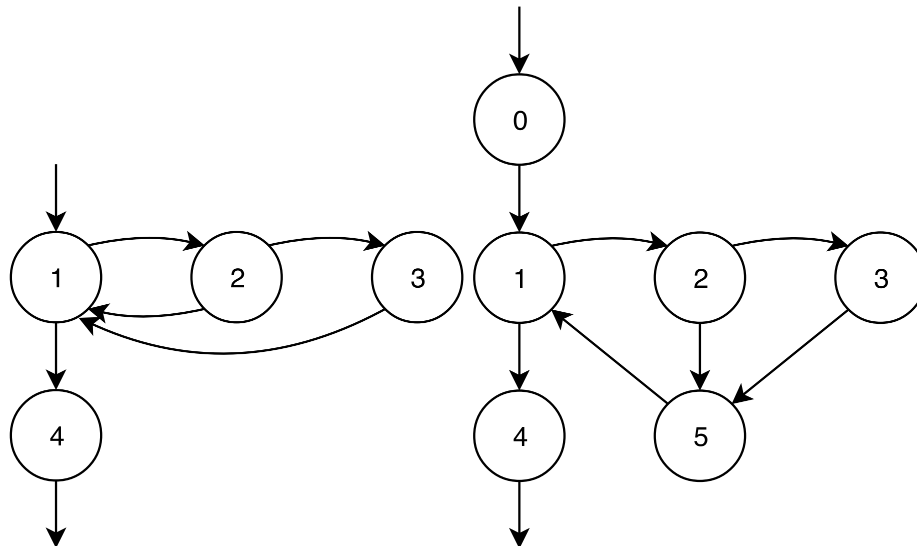
LLVM bietet Hilfs-Funktionen[2], mit denen der Kontrollfluss innerhalb einer Funktion bestimmt werden kann. Für diese Arbeit ist die vorhandene Funktionalität allerdings nicht ausreichend, da die Funktionen „state-less“ sind, also keinen Graph aufbauen, sondern nur Fragen wie „Ist BasicBlock A von BasicBlock B erreichbar?“. Da hierbei Zwischenergebnisse nicht gespeichert werden, muss bei jedem Funktionsaufruf die komplette Prüfung erneut durchlaufen werden. Innerhalb von LLVM wird eine Graphansicht des Kontrollflusses nur für Debugging-Zwecke eingesetzt.

4.3.2. Schleifen-Erkennung

LLVM bietet eine Klasse[3], mit der Schleifen innerhalb einer Funktion gefunden werden können. Die Klasse ermöglicht es beispielsweise denjenigen BasicBlock zu ermitteln, der den Beginn der Schleife definiert (Knoten 0 in der Darstellung 4.2(a)). Außerdem werden alle BasicBlocks ermittelt, die zu einer Schleife gehören. Um die Struktur von Schleifen zu vereinfachen, kann der *loop-simplify* Pass verwendet werden. Die Funktionsweise dieses Passes wird in Kapitel 6.2.1 ausführlich beschrieben.

Abbildung 4.2 zeigt die Graphstruktur einer Schleife vor und nach der Vereinfachung durch den *loop-simplify* Pass. LLVM definiert für eine Schleife die folgenden Begriffe:

[2] `http://llvm.org/doxygen/Analysis_2CFG_8h_source.html`
[3] `http://llvm.org/doxygen/classllvm_1_1LoopInfo.html`



(a) Graphstruktur einer Schleife ohne Vereinfachung.

(b) Graphstruktur der Schleife nach Anwendung des *loop-simplify* Passes.

Abbildung 4.2.: Beispiel für eine Schleife vor und nach der Vereinfachung der Struktur durch den *loop-simplify* Pass.

- **Pre-Header**: Einziger Vorgänger des Header-Blocks der Schleife (0)
- **Header**: Beginn-Block der Schleife (1)
- **Body**: Blöcke innerhalb der Schleife (1, 2, 3)
- **Latch**: Block, der vor dem Start einer Iteration ausgeführt wird (5)
- **Exiting**: Blöcke mit Nachfolgern außerhalb der Schleife (1)
- **Exit**: Blöcke mit Vorgängern innerhalb der Schleife (4)

4.3.2.1. Ablauf der Schleifen-Erkennung

Um eine Schleife in der LLVM Zwischensprache erkennen zu können, muss zunächst der Kontrollflussgraph aufgebaut werden. Innerhalb des Graphen sind Schleifen als Strukturen sichtbar, die einen Zyklus bilden. Hierzu sind ein Verzweigungspunkt und eine Rückkante nötig. Über die Rückkante ist es im Kontrollfluss möglich, erneut den Knoten mit dem Verzweigungspunkt zu erreichen. In Abbildung 4.2(a) enthält der Knoten 1 den Verzweigungspunkt. Rückkanten sind die Kanten von Knoten 2 und 3 zu Knoten 1.

Unter einer natürlichen Schleife versteht man die Menge von Knoten, die durch die Rückkante definiert sind. Sei die Rückkante gegeben von Knoten $y \rightarrow x$, so entspricht x dem Kopf der Schleife. Es darf keine weitere Kante geben, die nicht über x auf einen Knoten in der Menge zeigt. Die Schleife besteht dann aus x und allen Knoten die y ohne den Weg über x erreichen können [Al 07].

Um zu entscheiden, welcher Knoten der Kopf einer natürlich Schleife ist, wird eine Dominator-Analyse angewandt [LT79]. Der Knoten mit dem Schleifenkopf dominiert die anderen Knoten der Schleife, wenn alle möglichen Pfade zu einem Knoten durch den Schleifenkopf-Knoten laufen.

4.3.3. Kostenmodell

LLVM bietet eine Klasse[4] mit deren Hilfe generische Ausführungskosten für eine Instruktion bestimmt werden können. Die Klasse ermöglicht es außerdem statt den generischen Informationen plattformspezifische Eigenschaften zu berücksichtigen. Dazu werden über die Klassen, die im Backend für die Codeerzeugung zuständig sind, Zusatzinformationen abgerufen. Auf diese Weise kann exakt bestimmt werden, wie LLVM Instruktionen in native Instruktionen übersetzt werden. Eine weitergehende Beschreibung der Funktionsweise folgt in Kapitel 6.2.3.

4.4. Definition der Ziele

Nachfolgend werden die Ziele Z1 bis Z6 definiert, die mit der Implementierung dieser Arbeit erreicht werden sollen.

4.4.1. Z1: Interprocedural Control-Flow Graph aufbauen

Als Grundlage für die Pfadheuristik muss ein Interprocedural Control-Flow Graph (siehe Kapitel 2.3) aufgebaut werden. Dieser soll genutzt werden, um gültige Pfade zu Funktionen mit Schleifen zu finden.

4.4.2. Z2: Pfad Explosion einschränken

Das Problem der Pfad Explosion (siehe Kapitel 2.4.2) soll eingeschränkt werden. Hierzu soll eine Heuristik entwickelt werden, die auf Grundlage von Z1 zu untersuchende Pfade vorgibt.

4.4.3. Z3: Schleifenerkennung

Während der symbolischen Ausführung soll festgestellt werden, wenn ein Pfad eine Schleife betritt.

4.4.4. Z4: Hot-Path Analyse

Unter den durch die symbolische Ausführung gefundenen Pfaden, soll der Pfad mit den höchsten Ausführungskosten bestimmt werden. Innerhalb des Pfades sollen die Teile herausgestellt werden, die für die hohen Kosten verantwortlich sind.

4.4.5. Z5: Ausgabe von Pfadinformationen

Die ausgewerteten Informationen sollen als Ergebnis in eine Datei ausgegeben werden.

4.4.6. Z6: Performanzsteigerung durch Entfernen von Funktionalität

Für die Arbeit nicht benötigte Funktionen von KLEE sollen für eine bessere Performanz deaktiviert werden.

[4] `http://llvm.org/doxygen/classllvm_1_1TargetTransformInfo.html`

5. Entwurf

In diesem Kapitel wird beschrieben, wie die in Kapitel 4 analysierten Anforderungen, entworfen werden können. Der Abschnitt 5.1 beschreibt die allgemeinen Abläufe des Verfahrens. In den darauffolgenden Abschnitten wird auf jedes der gestellten Ziele gesondert eingegangen.

5.1. Ablauf des Verfahrens

Bevor KLEE mit der symbolischen Ausführung beginnt, werden nach dem Laden der LLVM Zwischencode-Datei zunächst einige Optimierungen und Transformationen durchgeführt. Hierzu wird das Pass System genutzt, um die bereits durch LLVM zur Verfügung gestellten Funktionen verwenden zu können. Einige der für Transformationen genutzten Pässe sind für die Optimierungen durch das Teilziel Z6 interessant. Dazu zählen beispielsweise:

- **DivCheckPass**[1]: Der Pass durchläuft alle Instruktionen der Anwendung und prüft, ob eine Division ausgeführt werden soll. Wird eine Division gefunden, wird vor dieser dynamisch ein Aufruf der Funktion *klee_div_zero_check*[2] eingefügt. Die Funktion wird mit dem Divisor aufgerufen und prüft, ob dieser Null ist. Falls das der Fall ist, wird automatisch ein Fehler ausgelöst, der den aktuellen Pfad in einen Fehlerzustand versetzt.
- **OvershiftCheckPass**[3]: Der Pass durchläuft alle Instruktionen der Anwendung und prüft, ob ein Bitshift ausgeführt werden soll. Wird ein Bitshift gefunden, wird vor dieser dynamisch ein Aufruf der Funktion *klee_overshift_check*[4] eingefügt. Die Funktion prüft, ob die durchgeführte Bitverschiebung innerhalb des Wertebereichs des verwendeten Datentyps stattfindet. Falls das nicht der Fall ist, wird automatisch ein Fehler ausgelöst, der den aktuellen Pfad in einen Fehlerzustand versetzt.

Außerdem werden Pässe ausgeführt, die den Kontrollfluss der Anwendung verändern:

- **LowerSwitchPass**[5]: Der Pass durchläuft alle Instruktionen der Anwendung und prüft, ob ein *switch* ausgeführt werden soll. Wird eine solche Instruktion gefunden,

[1] `https://github.com/klee/klee/blob/master/lib/Module/Passes.h#L95`
[2] `https://github.com/klee/klee/blob/master/runtime/Intrinsic/klee_div_zero_check.c`
[3] `https://github.com/klee/klee/blob/master/lib/Module/Passes.h#L117`
[4] `https://github.com/klee/klee/blob/master/runtime/Intrinsic/klee_overshift_check.c`
[5] `https://github.com/klee/klee/blob/master/lib/Module/Passes.h#L125`

wird der Switch durch eine Kette von *if else* Anweisungen ersetzt, in denen alle behandelten Fälle überprüft werden.

- **SimpleInliner**[6]: Der Pass prüft auf „kleine“ Funktionen und fügt deren Instruktionen direkt an der aufrufenden Stelle im Code ein. Auf diese Weise kann ein Funktionsaufruf eingespart werden.
- **GlobalDCELegacyPass**[7]: Der Pass entfernt „toten“ Code. Dazu zählen beispielsweise Funktionen, die nie aufgerufen werden und aus diesem Grund entfernt werden können. Dieser Pass wird mehrfach aufgerufen, um direkt auf Änderungen durch einen vorhergehenden Pass zu reagieren.
- **PromotePass**[8]: Der Pass prüft, ob es wirklich nötig ist für Variablen Speicher anzufordern oder ob die Werte direkt in Registern gehalten werden können. Auf diese Weise kann ein großer Teil der Speicherverwaltung eingespart werden. Dazu wird geprüft, ob Speicheradressen nur lokal genutzt werden. Ist dies der Fall, ersetzt der Pass die *Store*[9] Instruktion, sodass der Wert direkt in ein Register geschrieben wird. Alle Zugriffe auf die Speicheradresse werden anschließend durch einen Zugriff auf das Register ersetzt. Der Pass ist wichtig für viele weitere Optimierungs- und Analysepässe, da er die SSA-Form der LLVM-Zwischensprache (siehe Kapitel 2.6.2) sicherstellt.

Damit sich die von dieser Arbeit beigesteuerten Implementierungen in das Ökosystem eingliedern, sollen die Aufgaben der Initialisierungsphase durch das Erstellen von neuen Analysepässen erfüllt werden. Auf diese Weise wird es ermöglicht, auf Daten von bereits vorhandenen Pässen zuzugreifen, da diese nur innerhalb eines Passes abgerufen werden können.

Für die Initialisierung sind zwei Pässe notwendig. Der erste Pass dient dem Aufbauen des Interprocedural Control-Flow Graph (siehe Kapitel 2.3). Hierzu muss ein Pass auf Modulebene verwendet werden, da dieser Zugriff auf alle Funktionen des Moduls benötigt (siehe Kapitel 2.6.3). Der zweite Pass wird verwendet, um während der Ausführung benötigte Informationen einmalig zu Beginn zu sammeln. Da dieser Pass nur Zugriff auf einzelne Funktionen benötigt, ist keine Implementierung als ModulePass notwendig. Zu den Abhängigkeiten des Passes zählen der *LoopInfo*[10]- und der *TargetTransformInfo*[11]-Pass. Ersterer bietet die Erkennung von Schleifen (siehe Kapitel 4.3.2) und zweiterer bietet das Kostenmodell für Instruktionen (siehe Kapitel 4.3.3).

Nach der Initialisierung werden von KLEE sog. Schatten-Strukturen aufgebaut. Dazu werden die von LLVM verwendeten Strukturen wie *llvm::Function* und *llvm::Instruction* um weitere Eigenschaften erweitert, die währen der Ausführung verwendet werden. Um die in den Pässen gesammelten Informationen speichern zu können, sollen neue Schatten-Strukturen hinzugefügt werden. Diese sollen Daten zu Schleifen und BasicBlocks enthalten.

Vor dem Starten der symbolischen Ausführung muss die Heuristik zur Beschränkung der zu wählenden Pfade aufgebaut werden. Hierzu sollen mit den Informationen aus dem Interprocedural Control-Flow Graph Pfade zu Funktionen bestimmt werden, die Schleifen enthalten. Während der Ausführung werden die gewählten Pfade auf die durch die Heuristik vorgegebenen beschränkt.

[6] `http://llvm.org/doxygen/InlineSimple_8cpp_source.html`
[7] `http://llvm.org/doxygen/GlobalDCE_8cpp_source.html`
[8] `http://llvm.org/doxygen/Mem2Reg_8cpp_source.html`
[9] `https://llvm.org/docs/LangRef.html#i-store`
[10] `http://llvm.org/doxygen/classllvm_1_1LoopInfo.html`
[11] `http://llvm.org/doxygen/classllvm_1_1TargetTransformInfo.html`

Während der Ausführung werden pro Pfad die Ausführungskosten der Instruktionen sowie die Aufruf-Häufigkeiten von Funktionen und BasicBlocks bestimmt. Außerdem wird vermerkt, welche Schleifen durch einen Pfad ausgeführt wurden. Aus den gesammelten Informationen wird am Schluss die Ausgabe gebildet, in der der teuerste Pfad sowie die zu ihm führenden Variablenbelegungen dargestellt sind.

5.2. Z1: Interprocedural Control-Flow Graph aufbauen

Der Interprocedural Control-Flow Graph (siehe Kapitel 2.3) dient als Grundlage für die mit Z2 bestimmte Pfadheuristik und muss daher vor dieser erstellt werden. Um den Graphen bilden zu können, müssen die einzelnen BasicBlocks der Funktionen untersucht werden. Diese enthalten jeweils Informationen zu ihren direkten Nachfolgern im Kontrollfluss. Falls am Ende eines Blockes ein Verzweigungspunkt enthalten ist, hat der Block mindestens zwei Nachfolger, ansonsten üblicherweise nur einen einzigen. Dass Verzweigungspunkte nur am Ende eines Blockes auftreten können, wird durch LLVM sichergestellt (siehe Kapitel 2.6.1). Da die Nachfolger eines Knotens nur auf dessen Eltern-Funktion beschränkt sind, müssen zusätzlich noch die indirekten Nachfolger bestimmt werden. Dies sind BasicBlocks, die in anderen Funktionen liegen und durch einen Funktionsaufruf zu erreichen sind. Um diese bestimmten zu können, muss jeder Block auf Funktionsaufrufe untersucht werden. Wird ein Funktionsaufruf gefunden, muss zu diesem die aufzurufende Funktion bestimmt werden. Der Eingangsknoten der Funktion ist somit ein indirekter Nachfolger des aufrufenden BasicBlocks.

5.3. Z2: Pfad Explosion einschränken

Um die Pfad Explosion einschränken zu können, soll eine geführte symbolische Ausführung vergleichbar mit der verwandten Arbeit „Directed symbolic execution“ verwendet werden (siehe Kapitel 3.7). Dazu wird der mit Z1 erstellte Interprocedural Control-Flow Graph verwendet um gültige Pfade vorzugeben. In einem ersten Schritt müssen alle Funktionen identifiziert werden, die Schleifen enthalten. Anschließend werden alle Pfade rückläufig von den Eingangsblöcken der Schleifen bis zum Beginn der zu testenden Anwendung bestimmt. Bei Verzweigungspunkten während der symbolischen Ausführung sollen bevorzugt die Pfade gewählt werden, die in Richtung einer Schleife führen. Um zu verhindern, dass der Ausführungspfad innerhalb von Schleifen zu anderen Schleifen gelenkt wird, muss die Heuristik mit dem Erreichen einer Schleife beendet werden. Ab diesem Zeitpunkt sind wieder alle möglichen Pfade gleichberechtigt.

5.4. Z3: Schleifenerkennung

Damit während der symbolischen Ausführung festgestellt werden kann, dass ein Pfad eine Schleife betreten hat, müssen alle Schleifen zuvor erkannt und zur Überprüfung zwischengespeichert werden. Sobald der Kontrollfluss einen neuen BasicBlock betritt, kann überprüft werden, ob dieser der Kopf einer Schleife ist. Dass es nur einen solchen BasicBlock gibt, ist durch den *loop-simplify* Pass sichergestellt (siehe Kapitel 4.3.2 und 6.2.1).

5.5. Z4: Hot-Path Analyse

Um den Hot-Path bestimmen zu können, müssen zuvor die Ausführungskosten aller Instruktionen aufsummiert werden, die durch diesen Pfad ausgeführt wurden. Hierzu muss zu jedem Pfad ein Zähler verwaltet werden, der die Kosten enthält. Um die Hotspots bestimmen zu können, muss außerdem für jeden BasicBlock gezählt werden, wie oft dieser betreten wurde.

5.6. Z5: Ausgabe von Pfadinformationen

Die während der Ausführung gesammelten Informationen werden abschließend in eine Datei gespeichert. Hierbei wird der gewählte Pfad, dessen Ausführungskosten und eine Übersicht der Hotspots ausgegeben. Zusätzliche Informationen sind eine Auflistung der durchlaufenen Schleifen, symbolischen Variablen sowie deren konkrete Werte, die zum jeweiligen Pfad geführt haben.

5.7. Z6: Performanzsteigerung durch Entfernen von Funktionalität

Da der eigentliche Zweck von KLEE die Fehlersuche ist, können alle Funktionen deaktiviert werden, die ausschließlich hierfür verwendet werden. Da zum Beispiel bei Divisionen geprüft werden muss, ob der Divisor Null ist, muss hierfür der Solver einen konkreten Wert zur verwendeten symbolischen Variable berechnen. Der Aufruf des Solver kann mit einem hohen Arbeitsaufwand verbunden sein und somit führt das Entfernen dieser Funktionalität zu einer Performanzsteigerung.

6. Implementierung

Dieses Kapitel befasst sich mit der Implementierung der Klassen und Funktionen, die im vorherigen Kapitel entworfen wurden. In den Kapiteln 6.1, 6.2 und 6.3 werden verwendete Werkzeuge und Funktionalitäten vorgestellt. Abschließend wird in Kapitel 6.4 auf die Umsetzung der Ziele eingegangen.

6.1. Verwendete Werkzeuge

Da die Implementierung eine Erweiterung von KLEE darstellt, wird dessen CMake Build-System in Verbindung mit dem Clang-Compiler zum Übersetzen verwendet. Mit dem im Januar 2014 veröffentlichten LLVM 3.4 kommt allerdings eine alte Version von LLVM zum Einsatz. Bisher wurde von den Entwicklern noch kein Sprung auf die aktuelle Version 7.0 in Aussicht gestellt. Neben den von KLEE und LLVM bereitgestellten Klassen wurde außerdem die C++-Standardbibliothek verwendet.

Der für LLVM bevorzugte Compiler ist Clang, da er selbst auf LLVM aufbaut. Im Zusammenhang mit dieser Arbeit wird Clang verwendet, um Anwendungen in den LLVM Zwischencode zu übersetzen. Hierzu muss der Compileraufruf mit den Parametern *-S -emit-llvm* erweitert werden. Um beispielsweise die Datei *code.c* in die Zwischensprache zu kompilieren, muss der Befehl

```
#: clang -S -emit-llvm code.c
```

ausgeführt werden. Dies erzeugt als Ausgabe die Datei *code.bc*, die alle benötigten Daten enthält, um als *llvm::Module* geladen zu werden (siehe Kapitel 2.6.1).

6.2. Verwendete LLVM-Pässe

Nachfolgend sind alle LLVM-Pässe aufgelistet, die während der Initialisierungsphase verwendet werden. Sie werden genutzt, um die Struktur der Anwendung zu transformieren und um Analysen des Quellcodes durchzuführen.

6.2.1. LoopSimplify

Der LoopSimplify Pass[1] transformiert alle natürlichen Schleifen einer Anwendung in ihre kanonische Form. Dies ermöglicht eine einfachere Analyse und Transformation durch weitere Pässe. Um die kanonische Form zu erzeugen, werden drei Schritte ausgeführt:

[1] https://llvm.org/docs/Passes.html#loop-simplify-canonicalize-natural-loops

1. Es wird ein sog. pre-header Block vor dem Schleifenkopf angelegt. Dies hat zur Folge, dass der Schleifenkopf-Block nur noch eingehende Kanten vom pre-header Block und von Rückkanten aufweist.
2. Es wird ein sog. exit Block als Nachfolger des Schleifenkopfs angelegt. Die einzige eingehende Kante kommt vom Schleifenkopf und zeigt damit klar ein Verlassen der Schleife an.
3. Die Anzahl der Rückkanten wird auf eine einzelne begrenzt. Dazu wird ein neuer sog. latch Block eingefügt, auf den alle bisherigen Rückkanten verweisen. Er leitet den Kontrollfluss anschließend wieder an den Schleifenkopf weiter. Diese Änderung ermöglicht es, an einer zentralen Stelle zu prüfen, ob die aktuelle Schleifeniteration beendet wurde.

Ein Beispiel für die Transformation einer natürlichen Schleife in ihre kanonische zeigt Abbildung 4.2.

6.2.2. LoopInfo

Der LoopInfo Pass dient dem Auffinden von Schleifen (siehe Kapitel 4.3.2.1). Da Schleifen in der LLVM Zwischensprache nicht speziell gekennzeichnet sind, müssen sie unter Zuhilfenahme des Kontrollflussgraphen durch eine Dominator-Analyse gefunden werden [LT79]. Das hierbei erzeugte *llvm::Loop* Objekt enthält Informationen über die in der Schleife enthaltenen BasicBlocks, sowie die Ein- und Austrittsstellen der Schleife.

6.2.3. TargetTransformInfo

Die TargetTransformInfo Klasse ist im eigentlichen Sinne kein Pass, da sie keinerlei Funktionalität ausführt. Da die Ausführung der Pässe als isolierter Schritt durchgeführt wird, sollen Pässe nur auf die Daten von anderen Pässen zugreifen und nicht etwa Funktionen des Frontends verwenden. Da es für viele Optimierungspässe allerdings sehr hilfreich ist, wenn sie beispielsweise abfragen können, wie viele Bits in einem nativen Register zur Verfügung stehen werden, wird ein Interface zwischen den Pässen und dem Backend benötigt. Diese Aufgabe wird von der TargetTransformInfo Klasse übernommen. Sie besitzt mit dem BasicTTI Pass eine Basisimplementierung, die eine Verwendung ohne die Angabe einer spezifischen Plattform ermöglicht. Da die so gebotenen Informationen nur generische Werte sind, entfaltet die Klasse ihre volle Nützlichkeit erst durch eine architekturspezifische Implementierung. In LLVM 3.4 stehen unter anderem Implementierungen für x86, x86_64, ARM und Mips zur Verfügung. Jede im Backend verfügbare Plattform kann eine Erweiterung der TargetTransformInfo Klasse anbieten, um verbesserte Optimierungen zu ermöglichen.

Zu den Hauptfunktionen der Klasse zählen Funktionen, die zu erwartende Ausführungskosten für Instruktionen approximieren [LLV12]. Die Ergebnisse der Kostenberechnung besitzen keine Einheit und repräsentieren den Durchsatz einer Maschine unter der Annahme, dass Speicheranfragen direkt über den Cache beantwortet werden können und alle Verzweigungen korrekt hervorgesagt werden. Die berechneten Kosten eignen sich daher zum direkten Vergleich von Codeblöcken bzw. im Kontext dieser Arbeit von Ausführungspfaden.

6.2.4. UnifyFunctionExitNodes

Der UnifyFunctionExitNodes Pass prüft, ob es innerhalb einer Funktion mehrere Exitblöcke gibt und fasst diese zusammen. Ein Exitblock einer Funktion enthält üblicherweise ein

return Statement und leitet die Ausführung wieder zum Aufrufer zurück. Werden mehrere Exitblöcke erkannt, wird ein neuer *llvm::BasicBlock* erstellt und jedem der bisherigen Ausgänge als Nachfolger zugewiesen. Auf diese Weise wird erreicht, dass Funktionen nur über je einen Block betreten und verlassen werden können.

6.3. Verwendete Klassen

Nachfolgend sind alle Klassen aufgeführt, die für die Implementierung benötigt werden. Bei jeder Klasse wird kurz auf ihren Zweck eingegangen.

- **llvm::Module**[2]
  Die *llvm::Module* Klasse enthält alle Informationen zu einer Übersetzungseinheit. Dazu zählen beispielsweise verfügbare Funktionen oder globale Variablen (siehe Kapitel 2.6.1). Sie implementiert das Iterator-Pattern[3], mit dem sich alle Funktionen als *llvm::Function* Instanzen abrufen lassen.
- **llvm::Function**[4]
  Die *llvm::Function* Klasse enthält alle Informationen zu einer Funktion. Dazu zählen beispielsweise Angaben zu den benötigten Parametern, Rückgabewerten und den enthaltenen *llvm::BasicBlock*s (siehe Kapitel 2.6.1). Sie implementiert das Iterator-Pattern, mit dem sich alle enthaltenen Blöcke als *llvm::BasicBlock* Instanzen abrufen lassen.
- **llvm::BasicBlock**[5]
  Die *llvm::BasicBlock* Klasse enthält alle sequentiell auszuführenden Instruktionen des Abschnitts (siehe Kapitel 2.6.1). Jeder Block enthält als letzte Instruktion einen Terminator, der Auskunft über den Nachfolger des Blocks gibt. Die Klasse implementiert das Iterator-Pattern, mit dem sich alle enthaltenen Instruktionen als *llvm::Instruction* Instanzen abrufen lassen.
- **llvm::Instruction**[6]
  Die *llvm::Instruction* Klasse dient als Basis-Klasse für alle anderen konkreten Instruktionen (siehe Kapitel 2.6.1). Beispielsweise ist eine Addition als Instanz der *llvm::BinaryOperator*-Klasse[7] implementiert.
- **llvm::PassManager**[8]
  Der *llvm::PassManager* verwaltet eine Liste von Pässen, die auf einem Element (Module, Funktion, BasicBlock) ausgeführt werden. Die Klasse ist selbst auch als Pass implementiert, sodass sie als Gruppierung für Pässe eingesetzt werden kann.
- **llvm::ModulePass**[9]
  Die *llvm::ModulePass* Klasse ist die Basis-Klasse für die Implementierung von Pässen dient, die als Eingabe ein *llvm::Module* erwarten.
- **llvm::FunctionPass**[10]
  Die *llvm::FunctionPass* Klasse ist die Basis-Klasse für die Implementierung von Pässen dient, die als Eingabe eine *llvm::Function* erwarten.

[2] `http://llvm.org/doxygen/classllvm_1_1Module.html`
[3] Das Iterator-Pattern wird verwendet, um sequentiell Elemente verfügbar zu machen, ohne dass dazu Wissen über die Struktur der Auflistung nötig ist.
[4] `http://llvm.org/doxygen/classllvm_1_1Function.html`
[5] `http://llvm.org/doxygen/classllvm_1_1BasicBlock.html`
[6] `http://llvm.org/doxygen/classllvm_1_1Instruction.html`
[7] `http://llvm.org/doxygen/classllvm_1_1BinaryOperator.html`
[8] `http://llvm.org/doxygen/classllvm_1_1PassManager.html`
[9] `http://llvm.org/doxygen/classllvm_1_1ModulePass.html`
[10] `http://llvm.org/doxygen/classllvm_1_1FunctionPass.html`

- **llvm::LoopInfo**[11]
  Die *llvm::LoopInfo* Klasse dient dem Auffinden von Schleifen im Programmcode (siehe Kapitel 6.2.2 und 4.3.2.1). Auf sie kann von einem anderen Pass zugegriffen werden, um die gefundenen *llvm::Loop* Objekte auszuwerten.

- **llvm::Loop**[12]
  Die *llvm::Loop* Klasse enthält Informationen zu einer Schleife. Mit den Funktionen *getHeader()*, *getExitBlock()* und *getBlocks()* lassen sich der Schleifenkopf, der Exit-Block sowie alle in der Schleife verwendeten *llvm::BasicBlock*s abrufen. Die Funktion *getSubLoops()* gibt Zugriff auf eventuell enthaltene Schleifen innerhalb der Schleife.

- **llvm::CallSite**[13]
  Die *llvm::CallSite* Klasse ermöglicht es auf eine einheitliche Art und Weise Informationen über Instruktionen abzurufen, die für einen Funktionsaufruf zuständig sind. Sie wird verwendet, um das Aufrufziel zu bestimmen. Die Funktion *getCalledFunction()* liefert hierzu als Ergebnis eine *llvm::Function*.

- **llvm::TargetTransformInfo**[14]
  Die *llvm::TargetTransformInfo* Klasse bietet anderen Pässen Informationen aus dem LLVM-Backend. Sie ermöglicht es beispielsweise die Ausführungskosten von *llvm::Instruction*s auf einer konkreten Zielarchitektur abzufragen.

- **klee::Executor**[15]
  Die *klee::Executor* Klasse implementiert die komplette Logik von KLEE. Zu ihren Aufgaben gehört das Ausführen von Instruktionen sowie die hiermit verbundene Verwaltung der *klee::ExecutionState*s.

- **klee::ExecutionState**[16]
  Eine Instanz der *klee::ExecutionState* Klasse repräsentiert einen Pfad durch die Anwendung. Sie ist unter anderem dafür zuständig, den verwendeten Speicher und die gefundenen Pfad-Bedingungen zu verwalten. Auch ist in ihr die aktuell vom Pfad auszuführende Instruktion gespeichert.

- **klee::Expr**[17]
  Die *klee::Expr* Klasse repräsentiert einen symbolischen Ausdruck. Sie ist die Basis-Klasse von der konkrete Implementierungen wie beispielsweise *klee::BinaryExpr* erben. Instanzen von *klee::Expr* definieren für einen *klee::ExecutionState* welche Bedingungen erfüllt werden müssen, um den aktuellen Pfad zu durchlaufen.

- **klee::Array**[18]
  Ein *klee::Array* repräsentiert eine symbolische Variable. Die *klee_make_symbolic* Funktion[19] erzeugt ein neues *klee::Array* mit angegebener Größe und Namen.

- **klee::KModule**[20]
  Die *klee::KModule* Klasse ist ein Wrapper um ein *llvm::Module*. Sie enthält Listen der anderen Schattenstrukturen sowie Referenzen auf die von KLEE dynamisch hinzugefügten Hilfsfunktionen (siehe Kapitel 5.1).

[11] http://llvm.org/doxygen/classllvm_1_1LoopInfo.html
[12] http://llvm.org/doxygen/classllvm_1_1Loop.html
[13] http://llvm.org/doxygen/classllvm_1_1CallSite.html
[14] http://llvm.org/doxygen/classllvm_1_1TargetTransformInfo.html
[15] https://github.com/klee/klee/blob/master/lib/Core/Executor.h#L83
[16] https://github.com/klee/klee/blob/master/include/klee/ExecutionState.h#L66
[17] https://github.com/klee/klee/blob/master/include/klee/Expr.h#L90
[18] https://github.com/klee/klee/blob/master/include/klee/Expr.h#L481
[19] https://github.com/klee/klee/blob/master/include/klee/klee.h#L37
[20] https://github.com/klee/klee/blob/master/include/klee/Internal/Module/KModule.h#L83

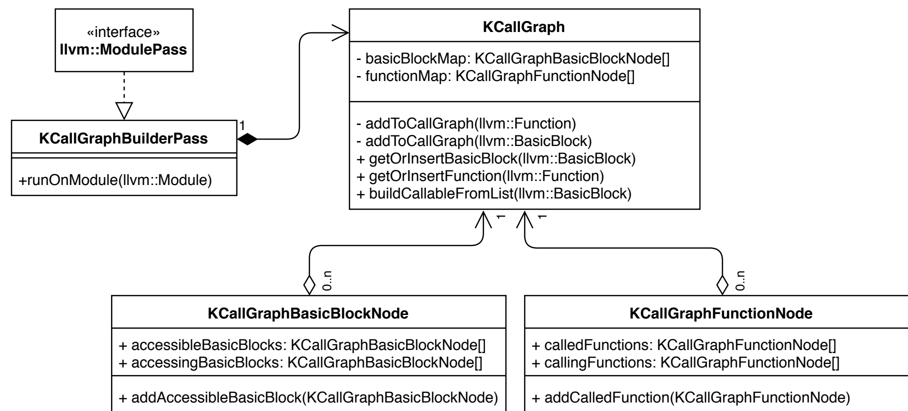


Abbildung 6.1.: UML Klassendiagramm des Interprocedural Control-Flow Graph.

- **klee::KFunction**[21]
  Die *klee::KFunction* Klasse ist als Schattenstruktur ein Wrapper um eine *llvm::Function*. Sie dient hauptsächlich als Container für die zugehörigen *klee::KInstruction*s.
- **klee::KInstruction**[22]
  Die *klee::KInstruction* Klasse definiert einen Wrapper um eine *llvm::Instruction* als Schattenstruktur. In ihr sind beispielsweise Debug-Informationen wie die zugehörige Quelltext-Zeilennummer vermerkt.

6.4. Umsetzung der Ziele

Die folgenden Kapitel erläutern die Implementierung der einzelnen Ziele.

6.4.1. Z1: Interprocedural Control-Flow Graph aufbauen

Der Interprocedural Control-Flow Graph wurde in der Klasse *KCallGraph* implementiert. Er bietet die Möglichkeit sowohl Informationen über Funktionen als auch über BasicBlocks abzurufen. Die Klasse vereint dementsprechend die Funktionen eines Call Graphs und die eines Interprocedural Control-Flow Graph in sich. Aufgebaut wird die Struktur über die *KCallGraphBuilderPass* Klasse, die als *llvm::ModulePass* implementiert ist. Die Knoten des Call Graphs entsprechen der *KCallGraphFunctionNode* Klasse, die eine *llvm::Function* kapselt. Die Knoten des Interprocedural Control-Flow Graphs entsprechen der *KCall-GraphBasicBlockNode* Klasse. Beide Knotentypen enthalten sowohl eine Liste der ein- als auch der ausgehenden Kanten. In Abbildung 6.1 sind die Beziehungen der Klassen untereinander dargestellt.

Die *processModule* Function erwartet als Parameter ein *llvm::Module*. Für jede der im Modul definierten Funktionen wird die Überladung der *addToCallGraph* Funktion aufgerufen, die eine *llvm::Function* als Parameter erwartet. Diese prüft, ob es bereits eine zur Funktion zugehörige *KCallGraphFunctionNode* Instanz gibt. Falls nicht, wird diese erstellt. Anschließend werden alle Instruktionen der Funktion nach einem Funktionsaufruf durchsucht. Ist ein Funktionsaufruf vorhanden, wird mit Hilfe der *llvm::CallSite* Klasse das Ziel des Funktionsaufrufs bestimmt. Die *getCalledFunction* liefert die passende *llvm::Function*, die anschließend als *KCallGraphFunctionNode* zum Knoten der aktuell untersuchten Funktion als Aufrufziel hinzugefügt wird.

[21]`https://github.com/klee/klee/blob/master/include/klee/Internal/Module/KModule.h#L42`
[22]`https://github.com/klee/klee/blob/master/include/klee/Internal/Module/KInstruction.h#L33`

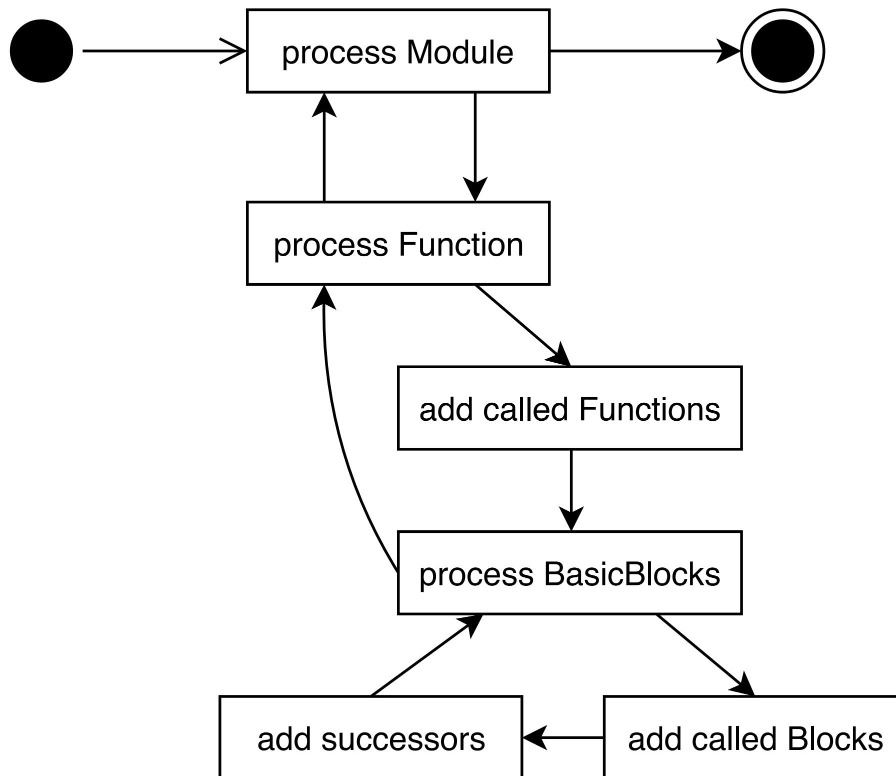


Abbildung 6.2.: Schematische Darstellung des Ablaufs zur Erstellung des Interprocedural Control-Flow Graphs.

In einem zweiten Schritt werden alle *llvm::BasicBlocks* der Funktion zu *KCallGraphBasicBlockNode*s umgewandelt. Dazu wird für jeden *llvm::BasicBlock* die *addToCallGraph* Funktion aufgerufen, die einen *llvm::BasicBlock* als Parameter erwartet. Jeder Block wird anschließend auf Funktionsaufrufe untersucht und wie im ersten Schritt wird das Ziel des Funktionsaufrufs bestimmt. Der Unterschied liegt beim zweiten Schritt darin, dass statt der eigentlichen Funktion der Eingangsblock der Funktion als Aufrufziel zum Knoten hinzugefügt wird. Abschließend wird der Terminator des Blocks untersucht und alle gefundenen Nachfolger in die Liste der Aufrufziele des Knotens eingefügt.

Dieser Vorgang wird wiederholt, bis alle Funktionen und ihre BasicBlocks abgearbeitet wurden. Die Abbildung 6.2 zeigt den schematischen Ablauf des Aufbaus des Interprocedural Control-Flow Graphs.

Gestartet wird der Vorgang in der *prepare* Methode der *klee::KModule* Klasse. In dieser Funktion werden von KLEE einmalige Vorbereitungen wie das Transformieren mit Pässen sowie der Aufbau der Schattenstrukturen durchgeführt.

6.4.2. Z2: Pfad Explosion einschränken

Um Pfade zu Funktionen mit Schleifen bestimmen zu können, müssen in einem ersten Schritt die Schleifen gefunden werden. Hierzu wurden zwei weitere Schattenstrukturen zu KLEE hinzugefügt.

6.4.2.1. Schattenstrukturen

Die Klasse *KBasicBlock* (siehe Abbildung 6.3) dient als Wrapper um einen *llvm::BasicBlock* und speichert zu diesem, ob er sich in einer Schleife befindet und wie hoch die statischen

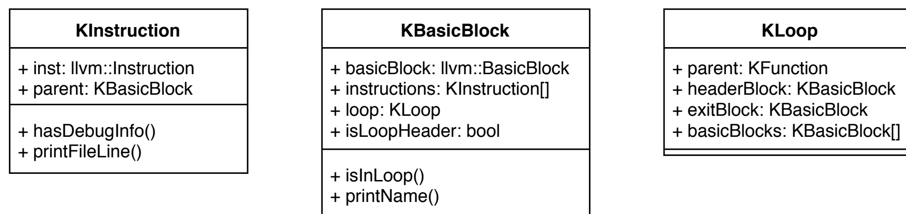


Abbildung 6.3.: UML Klassendiagramm der Schattenstrukturen.

Ausführungskosten der Instruktionen innerhalb des Blocks sind. Unter den statischen Ausführungskosten werden nur die Kosten der eigentlichen Instruktionen verstanden. Die Kosten einer Funktion bei einem Funktionsaufruf werden zu diesen nicht hinzugefügt. Außerdem enthält jeder *KBasicBlock* eine Liste der bereits vorhandenen *KInstruction* Schattenstrukturen.

Die Klasse *KLoop* (siehe Abbildung 6.3) enthält Informationen über eine Schleife im Quellcode. Dazu sind die *KFunction*, in der sich die Schleife befindet, der *KBasicBlock* Schleifenkopf und Exit-Knoten sowie die inneren BasicBlocks gespeichert.

Um die nötigen Informationen für die Schattenstrukturen zu erhalten wurde ein weiterer Pass implementiert. Die Klasse *ExtractExtendedInformationPass* ist ein *llvm::FunctionPass* und wird somit für jede Funktion des Moduls aufgerufen. Der Pass hängt von den Pässen *llvm::LoopInfo* (siehe Kapitel 6.2.2) und *llvm::TargetTransformInfo* (siehe Kapitel 6.2.3) ab. Für jede Funktion werden mit Hilfe des *llvm::LoopInfo* Objektes alle Schleifen ausgelesen. Da eine Schleife innere Schleifen besitzen kann, werden die Schleifeninformationen rekursiv extrahiert. Anschließend wird für alle *llvm::BasicBlock*s der Funktion bestimmt, ob sie sich innerhalb einer Schleife befinden. Zu jedem Block wird hierbei gespeichert in welcher Schleife er vorkommt. Für jede Instruktion innerhalb des Blocks werden mithilfe der *llvm::TargetTransformInfo* Klasse über die Funktion *getInstructionCost* die Ausführungskosten bestimmt.

Auch der *ExtractExtendedInformationPass* wird innerhalb der *prepare* Methode der Klasse *klee::KModule* durch einen *llvm::PassManager* ausgeführt. Mit den durch die Ausführung des Passes gewonnenen Informationen werden anschließend die *KLoop* und *KBasicBlock* Klassen erstellt. Außerdem werden die Ausführungskosten der Instruktionen in der *KInstruction* Klasse gespeichert.

6.4.2.2. Vorgabe der Ausführungspfade

Da durch den in Kapitel 6.4.2.1 beschriebenen Prozess alle Funktionen, die Schleifen enthalten, bekannt sind, können nun gültige Pfade für die symbolische Ausführung zu diesen vorgegeben werden. Um nicht alle möglichen Pfade bestimmen zu müssen, wird nur eine Menge von BasicBlocks vorgegeben, die sich auf dem Pfad zu einer Funktion mit Schleifen befinden. Um die Liste aufzubauen, wird die Funktion *buildCallableFromList* der Klasse *KCallGraph* verwendet. Diese startet mit dem *llvm::BasicBlock* eines Schleifenkopfes und fügt rekursiv alle Blöcke der Liste hinzu, die zum Schleifenkopf führen. Falls ein Block bereits in der Liste enthalten ist, wird die Rekursion unterbrochen, da der Block zuvor in einem anderen Durchgang verarbeitet wurde. Die Liste der gültigen Blöcke wird als Variable in der *klee::KModule* Klasse gespeichert.

Während der symbolischen Ausführung durch die *klee::Executor* Klasse wird für jede auszuführende Instruktion die Funktion *executeInstruction* aufgerufen. In dieser Funktion wird eine Fallunterscheidung je nach Art der Instruktion durchgeführt. Bei einem Verzweigungspunkt, also einer *llvm::BranchInst*, wird der Pfad aufgeteilt, indem der aktuelle

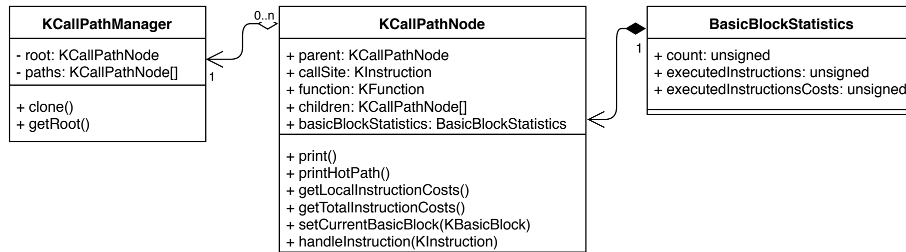


Abbildung 6.4.: UML Klassendiagramm der *KCallPathManager* Klasse.

klee::ExecutionState verdoppelt wird. Anschließend wird überprüft, welcher der Pfade seinen Weg durch einen erlaubten Block fortsetzt. Ist der nächste auszuführende Block nicht erlaubt, wird der Pfad mit der *terminateStateEarly* Funktion abgebrochen. Befindet sich die aktuelle Ausführung innerhalb einer Schleife, wird die Pfadbeschränkung deaktiviert, um alle Pfade durch die Schleife bestimmen zu können.

6.4.3. Z3: Schleifenerkennung

Um bestimmen zu können, ob eine Schleife betreten wird, wird bei der Ausführung einer *llvm::BranchInst* in der *executeInstruction* Funktion der *klee::Executor* Klasse überprüft, ob sich die Instruktion innerhalb des Schleifenkopfes befindet. Falls das der Fall ist, werden aus der aktuellen *klee::Expr*-Path Condition des *klee::ExecutionState*s mit der *findSymbolicObjects* Funktion alle symbolischen Variablen extrahiert. Die hierdurch gewonnene Liste an Variablen ist für das Erreichen der Schleife notwendig. Die gefundene Schleife wird mit den symbolischen Variablen im aktuellen *klee::ExecutionState* gespeichert.

6.4.4. Z4: Hot-Path Analyse

Um den Hot-Path unter allen Pfaden bestimmen zu können, müssen in einem ersten Schritt die Ausführungskosten der Pfade bestimmt werden. Hierfür wurde die Klasse *KCallPathManager* implementiert, die die Ausführungskosten während dem Ausführen von Funktionen protokolliert (siehe Abbildung 6.4). Die Klasse enthält eine Instanz der *KCallPathNode* Klasse, die die Ausgangsfunktion[23] repräsentiert. Jede *KCallPathNode* Instanz kennt ihren Vorgänger und verwaltet eine Liste mit Nachfolgern. Bei jedem in der Anwendung ausgeführten Funktionsaufruf wird ein neuer *KCallPathNode* erstellt und dem aktuell aktiven Knoten als Nachfolger hinzugefügt. Um mehrfache Funktionsaufrufe derselben Funktion verarbeiten zu können, ist sowohl die aufzurufende Funktion als auch die aufrufende Instruktion ein Schlüssel für den Knoten. Auf diese Weise können mehrere Funktionsaufrufe unabhängig voneinander protokolliert werden. Innerhalb einer *KCallPathNode* Instanz wird der aktuell ausgeführte *KBasicBlock* verwaltet. Zu jedem Block existiert eine Instanz der *BasicBlockStatistics* Klasse. Diese enthält Zähler für die aktuellen Ausführungskosten und die Häufigkeit, wie oft der Block betreten wurde. Wird während der symbolischen Ausführung der aktuelle BasicBlock durch die Funktion *transferToBasicBlock* geändert, wird dies dem aktuellen *KCallPathNode* mitgeteilt, sodass die Kosten für diesen Block gezählt werden können. Kehrt eine Funktion nach der Ausführung zu ihrem Aufrufer zurück, wird der aktuelle *KCallPathNode* des *klee::ExecutionState* wieder auf den Vorgängerknoten festgelegt.

Durch den auf diese Weise aufgebauten Graph ist es nach dem Beenden der Ausführung eines Pfads möglich dessen gesamte Ausführungskosten zu berechnen. Auf diese Weise können Pfade miteinander verglich und somit der Pfad mit den höchsten Kosten bestimmt werden.

[23] Im Normalfall wird dies die *main* Funktion der Anwendung sein.

6.4.5. Z5: Ausgabe von Pfadinformationen

Die Ausgabe der Pfadinformationen erfolgt nach dem erfolgreichen Durchlaufen eines Pfades durch die Anwendung. Sobald ein Pfad beendet wurde, erzeugt KLEE bereits unterschiedliche Ausgaben (siehe Kapitel 2.7.3). An dieser Stelle wurde die Ausgabe um die Datei *Test<N>.pathinfo* erweitert. In diese Datei werden die vom aktuellen Pfad verwendeten symbolischen Variablen und die Ausgabe der für Z4 implementierten *KCallPathManager* Klasse des Pfades generiert. Zu den symbolischen Variablen werden die konkreten berechneten Werte angegeben. Zur verbesserten Darstellung wird versucht die Werte im Datentyp *unsigned* auszugeben. Da allerdings nicht bekannt ist, welchen Typ eine Variable hat, wird zusätzlich noch eine hexadezimale Darstellung des rohen Byte-Arrays ausgegeben, das KLEE intern verwendet hat. Die Ausgabe des *KCallPathManagers* beinhaltet den vollständigen ausgeführten Pfad, der durch Funktionsnamen und Quelltextreferenzen nachvollzogen werden kann. Gefundene Schleifen werden zusammen mit ihren Ausführungskosten den jeweiligen Funktionen zugeordnet. Ein Beispiel für die erzeugte Ausgabe ist in Kapitel 7.3 aufgeführt.

Eine weitere Datei *hotpath* enthält die gesonderte Ausgabe des Pfades mit den höchsten Ausführungskosten. Um diesen zu bestimmen, wird nach jedem Beenden eines Pfades überprüft, ob dessen Kosten höher als die bisher bekannten maximalen Kosten sind. Übersteigen die Kosten das bisherige Maximum, wird dieser Pfad zur späteren Ausgabe vorgemerkt. Nachdem die symbolische Ausführung aller Pfad beendet wurde, erfolgt die Ausgabe der Hotspots des Pfades mit den höchsten Ausführungskosten. Die Ausgabe enthält hierbei den teuersten Pfad innerhalb des Ausführungspfades. Darunter ist zu verstehen, dass falls von einer Funktion mehrere andere Funktionen aufgerufen wurden, nur die Informationen der Funktion ausgegeben werden, die die höchsten Kosten verursacht hat. Zu jeder Funktion sind die am häufigsten ausgeführten BasicBlocks unter Angabe ihrer Quelltextreferenzen aufgeführt.

6.4.6. Z6: Performanzsteigerung durch Entfernen von Funktionalität

Um keine Funktionalität von KLEE entfernen zu müssen, wurden lediglich die Standardeinstellungen der *check-div-zero* und *check-overshift* Parameter auf *false* festgelegt. Hierdurch werden beide Überprüfungen nicht mehr durchgeführt, können allerdings trotzdem noch per Kommandozeilenparameter nachträglich aktiviert werden.

7. Evaluation

In diesem Kapitel wird beschrieben, wie die in Kapitel 6 vorgestellten Implementierungen getestet wurden. Da zum Vergleichen eine passende Implementierung fehlt, wurden kurze Programme erstellt, mit denen die jeweiligen Ziele getestet werden können.

7.1. Z1: Interprocedural Control-Flow Graph aufbauen

Um die Erstellung des Interprocedural Control-Flow Graph zu testen, wurde das in Quelltextausschnitt 7.1 dargestellte Programm verwendet. Es ist kurz genug, sodass es bei manueller Durchsicht der LLVM Zwischensprache verständlich bleibt. Trotzdem besitzt es alle Eigenschaften, die für den Graphen wichtig sind.

Quelltextausschnitt 7.1: Beispielcode für den Aufbau des Interprocedural Control-Flow Graph.

```
1  int baz()
2  {
3     return 0;
4  }
5
6  int bar()
7  {
8     return 1;
9  }
10
11 int foo(int x)
12 {
13    if (x)
14    {
15       return baz();
16    }
17    else
18    {
19       return bar();
20    }
21 }
22
```

```
23 int main(int argc, char** argv)
24 {
25     return foo(argc);
26 }
```

Der Quelltext in der Datei *test.c* wurde mit Clang in den LLVM Zwischencode übersetzt:

```
1 # clang -emit-llvm -c -g test.c
```

Die hierdurch erstellte Datei *test.bc* wurde anschließend mit *llvm-dis* in eine lesbare Textrepräsentation umgewandelt:

```
1 # llvm-dis test.bc
```

In Listing 7.2 ist der gekürzte Inhalt der neu erzeugten Datei *test.ll* dargestellt. Bei der Funktion *foo* kann die durch die Verzweigungspunkte ausgelöste Aufteilung in einzelne Blöcke erkannt werden.

Quelltextausschnitt 7.2: Ausschnitt aus der Datei *test.ll*.

```
1  ; ModuleID = 'test.bc'
2  target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-
       i32:32:32-i64:64:64-f32:32:32-f64:64:64-v64:64:64-v128
       :128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-S128"
3  target triple = "x86_64-pc-linux-gnu"
4
5  ; Function Attrs: nounwind uwtable
6  define i32 @baz() #0 {
7    ret i32 0, !dbg !22
8  }
9
10 ; Function Attrs: nounwind uwtable
11 define i32 @bar() #0 {
12   ret i32 1, !dbg !23
13 }
14
15 ; Function Attrs: nounwind uwtable
16 define i32 @foo(i32 %x) #0 {
17   %1 = alloca i32, align 4
18   %2 = alloca i32, align 4
19   store i32 %x, i32* %2, align 4
20   %3 = load i32* %2, align 4, !dbg !26
21   %4 = icmp ne i32 %3, 0, !dbg !26
22   br i1 %4, label %5, label %7, !dbg !26
23
24 ; <label>:5
25   %6 = call i32 @baz(), !dbg !28
26   store i32 %6, i32* %1, !dbg !28
27   br label %9, !dbg !28
28
29 ; <label>:7
30   %8 = call i32 @bar(), !dbg !30
31   store i32 %8, i32* %1, !dbg !30
32   br label %9, !dbg !30
```

```
33
34 ; <label>:9
35   %10 = load i32* %1, !dbg !32
36   ret i32 %10, !dbg !32
37 }
38
39 ; Function Attrs: nounwind uwtable
40 define i32 @main(i32 %argc, i8** %argv) #0 {
41   %1 = alloca i32, align 4
42   %2 = alloca i32, align 4
43   %3 = alloca i8**, align 8
44   store i32 0, i32* %1
45   store i32 %argc, i32* %2, align 4
46   store i8** %argv, i8*** %3, align 8
47   %4 = load i32* %2, align 4, !dbg !36
48   %5 = call i32 @foo(i32 %4), !dbg !36
49   ret i32 %5, !dbg !36
50 }
```

Um die Daten der *KCallGraph* Klasse ausgeben zu können, wurde sie um eine *dump* Funktion erweitert. Diese gibt eine Liste von BasicBlocks aus. Zu jedem Block wird der Name der übergeordneten Funktion, sowie seine direkten Vorgänger und Nachfolger ausgegeben. Die *dump* Funktion wurde zum Testen nach der Ausführung des *KCallGraphBuilderPass*es aufgerufen. In Listing 7.3 ist die hierbei erhaltene Ausgabe dargestellt:

Quelltextausschnitt 7.3: Ausgabe der Funktion *KCallGraph.dump*.

```
1  BasicBlock 1 (main)
2  Nachfolger: BB2
3  Vorgänger:
4
5  BasicBlock 2 (foo)
6  Nachfolger: BB4 BB3
7  Vorgänger: BB1
8
9  BasicBlock 3 (foo)
10 Nachfolger: BB5 BB7
11 Vorgänger: BB2
12
13 BasicBlock 4 (foo)
14 Nachfolger: BB5 BB6
15 Vorgänger: BB2
16
17 BasicBlock 5 (foo)
18 Nachfolger:
19 Vorgänger: BB4 BB3
20
21 BasicBlock 6 (bar)
22 Nachfolger:
23 Vorgänger: BB4
24
25 BasicBlock 7 (baz)
26 Nachfolger:
```

```
27 Vorgänger: BB3
```

Man kann erkennen, dass die Ausgabe den Erwartungen an den Quelltext entspricht. Jeder BasicBlock hat die passende Zuordnung seiner Vorgänger und Nachfolger erhalten. Durch die Auflösung von Funktionsaufrufen wurden außerdem funktionsübergreifende Kanten hinzugefügt.

Ein Problem der aktuellen Implementierung des Interprocedural Control-Flow Graphen ist die fehlende Erkennung von dynamischen Funktionsaufrufen. Es können nur Funktionsaufrufe erkannt werden, die während der statischen Analyse bekannt sind. Wird zur Laufzeit das Ziel eines Aufrufs dynamisch bestimmt, fehlt diese Kante im Graphen. Ein Beispiel hierfür ist beispielsweise Vererbung. Wenn Vererbung eingesetzt wird, können Funktionen der Basisklasse überschrieben werden. Zur Laufzeit wird anhand der aufrufenden Klasseninstanz entschieden, welche Funktion aufzurufen ist. Da diese Informationen während der Analyse nicht verfügbar sind, werden bei Anwendungen, die diese Technik einsetzen, unvollständige Graphen erstellt. Dies führt dazu, dass nicht alle möglichen Pfade erkannt werden. Eine mögliche Lösung für das Problem wäre, dass alle möglichen Aufrufziele statisch bestimmt werden und diese anschließend als statische Funktionsaufrufe behandelt werden.

7.2. Z2: Pfad Explosion einschränken

Um die Einschränkung der Pfad Explosion zu testen, wurde eine Anwendung geschrieben, deren Struktur an ein Labyrinth erinnert. Nur durch einen der möglichen Pfade kann eine Schleife erreicht werden. Der Quelltext sowie die Textrepräsentation des LLVM Zwischencodes sind in den Quelltextausschnitten 8.1 und 8.2 dargestellt. Wird die Schleife durch ein weiteres *return* Statement ersetzt, so zeigt KLEE zu Beginn der Analyse die Ausgabe:

```
1 # klee maze.bc
2 KLEE: output directory is "/klee/examples/maze/klee-out-0"
3 KLEE: Using STP solver backend
4 KLEE: Used target: x86-64
5 KLEE: Directed blocks: 0/78
```

Sie zeigt an, dass keiner der 78 Blöcke für eine gerichtete symbolische Ausführung verwendet wird. Es gibt somit keine Beschränkung und alle möglichen Pfade werden ausgeführt werden. Enthält der Quelltext allerdings die Schleife, wird diese bei der Analyse erkannt und gültige Pfade zu dieser berechnet. Die Ausgabe von KLEE zeigt anschließend:

```
1 # klee maze.bc
2 KLEE: output directory is "/klee/examples/maze/klee-out-1"
3 KLEE: Using STP solver backend
4 KLEE: Used target: x86-64
5 KLEE: Directed blocks: 7/82
```

Es wurden die in Quelltextausschnitt 8.1 mit *<label>:21*, *<label>:17*, *<label>:16*, *<label>:13*, *<label>:5* und *<label>:0* benannten Blöcke der Funktion *maze* sowie der Einstiegsblock der *main* Funktion als gültige Blöcke erkannt. Führt eine Verzweigung zu einem nicht erlaubten Block, so wird dieser Pfad frühzeitig abgebrochen und KLEE erzeugt eine *test<N>.early* Datei mit dem Inhalt „Path not allowed“. Durch das Abbrechen der Pfade wird die Pfad Explosion effektiv eingeschränkt, da nur noch die Pfade weiter beachtet werden, die zum gesuchten Ziel führen.

7.3. Z3, Z4, Z5: Zusammenfasung

Da sich die drei Ziele Z3, Z4 und Z5 (siehe Kapitel 4.4.3, 4.4.4 und 4.4.5) nicht gut eigenständig testen lassen, wurden sie zu einem Kapitel zusammengefasst. Zum Testen wurde eine Anwendung verwendet, die eine naive Implementierung einer Primzahlbestimmung enthält. Die Quelltextausschnitte 7.4 und 8.3 zeigen sowohl den C-Code als auch den erzeugten LLVM Zwischencode.

Quelltextausschnitt 7.4: Inhalt der Datei *main.c* zur Primzahlprüfung.

```
1 #include <klee/klee.h>
2
3 int isPrimImpl(int value)
4 {
5     for (int i = 2; i < value; ++i)
6     {
7         if (value % i == 0)
8         {
9             return 0;
10        }
11    }
12
13    return 1;
14 }
15
16 int isPrim(int value)
17 {
18    if (value <= 0)
19    {
20        return 0;
21    }
22    if (value == 1 || value == 2)
23    {
24        return 1;
25    }
26    return isPrimImpl(value);
27 }
28
29 int main() {
30    int value;
31    klee_make_symbolic(&value, sizeof(value), "value");
32    klee_assume((value > -1) & (value < 100));
33    return isPrim(value);
34 }
```

Die Funktion *isPrim* wird mit Werten aus dem Bereich $[0, 99]$ aufgerufen. Sie enthält zu Beginn Prüfungen auf Spezialfälle und ruft die Funktion *isPrimImpl* auf. Diese wiederum prüft in einer Schleife, ob die Zahl durch eine kleinere Zahl teilbar ist. Bei der Ausführung mit KLEE wird die Schleife gefunden und Pfade zu dieser berechnet. Nachdem die symbolische Ausführung beendet wurde, zeigen die erstellten Dateien, dass zwei Pfade frühzeitig abgebrochen wurden.

Zur Veranschaulichung der erzeugten Ausgaben sind nachfolgend die Dateiinhalte der *test<N>.pathinfo* Dateien stellvertretend für je einen abgebrochenen als auch einen gültigen Ausführungspfad aufgelistet (siehe Ausschnitte 7.5 und 7.6).

Quelltextausschnitt 7.5: Inhalt der Datei *test000001.pathinfo* mit Informationen zu einem frühzeitig abgebrochenen Pfad.

```
1  Symbolics:
2  value: 0 0x0000
3
4  Function: <root>
5      Function: main
6      Callsite: none
7      Call Count: 1
8      Costs: 10
9          Function: isPrim
10         Callsite: /klee/examples/isPrim/main.c:33
11         Call Count: 1
12         Costs: 1
```

Zu Beginn sind die symbolischen Variablen mit den für diesen Pfad konkreten Werten aufgelistet. Mit dem Wert 0 führte der Pfad in der Funktion *isPrim* zu einem der nicht erlaubten Blöcke, da von dort keine Schleife erreicht werden kann.

Quelltextausschnitt 7.6: Inhalt der Datei *test000030.pathinfo* mit Informationen zu einem abgeschlossenen Pfad.

```
1  Symbolics:
2  value: 97 0x00061
3
4  Function: <root>
5      Function: main
6      Callsite: none
7      Call Count: 1
8      Costs: 393
9          Function: isPrim
10         Callsite: /klee/examples/isPrim/main.c:33
11         Call Count: 1
12         Costs: 384
13             Function: isPrimImpl
14             Callsite: /klee/examples/isPrim/main.c:26
15             Call Count: 1
16             Costs: 381
17             Detected loops:
18               Loop 1:
19               /klee/examples/isPrim/main.c:5
20               Costs: 381
21               Symbolics (1): value
```

Für diesen Pfad zeigt die Ausgabe, dass er die Schleife in der *isPrimImpl* Funktion erreicht und ausgeführt hat. Für die Schleife sind die aufsummierten Ausführungskosten, sowie die von der Schleife verwendeten symbolischen Variablen aufgelistet.

Zusätzlich wurde als Zusammenfassung über alle Pfad die Datei *hotpath* erzeugt, deren Inhalt in Ausschnitt 7.7 dargestellt ist.

Quelltextausschnitt 7.7: Inhalt der Datei *hotpath* mit Informationen zum teuersten Pfad.

```
1  Path: 30
```

```
2  Costs: 393
3
4  Symbolics:
5  value: 97 0x00061
6
7  Function: <root>
8      Function: main (Call Count: 1, Total Costs: 393)
9      Hotspots:
10       <block 0> (Enter Count: 1, Executed instructions:
            15, Executed instructions costs: 9)
11       /klee/examples/isPrim/main.c:31
12         Function: isPrim (Call Count: 1, Total Costs: 384)
13         Hotspots:
14           <block 5> (Enter Count: 1, Executed instructions
                : 2, Executed instructions costs: 0)
15           /klee/examples/isPrim/main.c:26
16           <block 3> (Enter Count: 1, Executed instructions
                : 2, Executed instructions costs: 0)
17           /klee/examples/isPrim/main.c:26
18           <block 2> (Enter Count: 1, Executed instructions
                : 3, Executed instructions costs: 2)
19           /klee/examples/isPrim/main.c:22
20           <block 0> (Enter Count: 1, Executed instructions
                : 2, Executed instructions costs: 1)
21           /klee/examples/isPrim/main.c:18
22             Function: isPrimImpl (Call Count: 1, Total
                  Costs: 381)
23             Hotspots:
24               <block 1> (Enter Count: 96, Executed
                    instructions: 288, Executed instructions
                    costs: 96)
25               /klee/examples/isPrim/main.c:5
26               <block 3> (Enter Count: 95, Executed
                    instructions: 285, Executed instructions
                    costs: 190)
27               /klee/examples/isPrim/main.c:7
28               <block 6> (Enter Count: 95, Executed
                    instructions: 190, Executed instructions
                    costs: 95)
29               /klee/examples/isPrim/main.c:5
30               <block 8> (Enter Count: 1, Executed
                    instructions: 2, Executed instructions
                    costs: 0)
31               /klee/examples/isPrim/main.c:5
32               <block 0> (Enter Count: 1, Executed
                    instructions: 1, Executed instructions
                    costs: 0)
33               /klee/examples/isPrim/main.c:5
```

Die ersten Zeilen verweisen auf den Ausführungspfad und dessen gesammelte Kosten. Anschließend sind die symbolischen Variablen mit konkreten Werten aufgelistet, die zu diesem Pfad geführt haben. Den größten Teil der Ausgabe nehmen die während der Ausführung

gesammelten Pfadinformationen ein. Es sind die teuersten der ausgeführten Funktionen baumförmig aufgelistet. Im Anschluss an jede Funktion werden maximal fünf BasicBlocks ausgegeben. Zu jeden Block ist angegeben, wie oft dieser aufgerufen wurde und welche Ausführungskosten er verursacht hat. Die Blöcke sind hierbei nach ihrer Aufrufzahl absteigend sortiert.

7.4. Z6: Performanzsteigerung durch Entfernen von Funktionalität

Die Deaktivierung der beiden Fehlerüberprüfungen wirkt sich nur positiv bei Anwendungen aus, die eine große Anzahl an Divisionen ausführen. Das Problem der Pfad Explosion stellt einen viel größeren Aufwand bei der symbolischen Ausführung dar.

8. Zusammenfassung und Ausblick

Die Steigerung der Leistungsfähigkeit aktueller Prozessorgenerationen wird durch das Hinzufügen von zusätzlichen Rechenkernen erreicht. Sie ermöglichen hiermit die nebenläufige Ausführung von Teilstücken einer Anwendung. Da die Teilstücke nicht mehr sequentiell nacheinander abgearbeitet werden müssen, kann die Ausführungszeit signifikant verringert werden. Damit eine Anwendung von den Möglichkeiten zur Parallelisierung profitieren kann, müssen hierfür passende Stellen im Quellcode vom Entwickler erkannt und parallelisiert werden. Ein lohnenswertes Ziel für Parallelisierung sind Schleifen. Da in einer Schleife der Schleifenkörper wiederholt ausgeführt wird, kann durch eine Verteilung der durchgeführten Iterationen auf unterschiedliche Prozessorkerne deren Verarbeitung stark beschleunigt werden.

Um bestimmen zu können, bei welchen Schleifen sich eine Parallelisierung lohnt, müssen deren Ausführungskosten bestimmt werden. Problematisch ist hierbei, dass diese im Normalfall von den Eingaben abhängt, die die Anwendung entgegennimmt. Auch kann es schwierig sein, passende Eingaben zu bestimmen, die einen großen Anteil von Programmcode abdecken, um mögliche Flaschenhälse zu entdecken.

Das Problem zur Bestimmung von aussagekräftigen Eingaben kann durch das Prinzip der symbolischen Ausführung gelöst werden. Bei der symbolischen Ausführung wird der Code der Anwendung schrittweise analysiert und Werte für die verwendeten Variablen bestimmt, sodass jeder Pfad im Programm erreichbar ist. Dazu startet die Analyse mit einem einzigen Ausführungspfad, der an Verzweigungspunkten in immer neue Ausführungspfade aufgeteilt wird. Jede mit einem Verzweigungspunkt verknüpfte Bedingung wird hierbei mit den bereits durch den Pfad gesammelten Bedingungen verknüpft. Mithilfe eines Solvers können die verknüpften Bedingungen ausgewertet werden, um konkrete Werte für verwendete Variablen zu bestimmen, die diese Bedingungen erfüllen.

Bei der Aufteilung der Ausführungspfade an Verzweigungspunkten kommt es zum sog. Pfad Explosion-Problem. Die fortschreitende Teilung der Pfade führt zu einem exponentiellen Wachstum der Anzahl an Pfaden. Ab einem bestimmten Punkt kann deren Anzahl zu groß werden, als dass sie die Testanwendung sinnvoll verwalten könnte.

Das Ziel dieser Arbeit war es, einen Entwickler dabei zu unterstützen, in einer bereits vorhandenen Anwendung Schleifen im Quellcode zu finden die Parallelisierungspotential besitzen. Hierzu sollte das auf der Compiler-Infrastruktur LLVM aufbauende Test-Programm KLEE erweitert werden. Die primäre Aufgabe von KLEE ist es, durch symbolische Ausführung Ausführungspfade durch Anwendungen zu bestimmen und deren Wege auf vorge-

gebene Fehlerfälle wie beispielsweise eine mögliche Division durch Null zu prüfen. Durch die symbolische Ausführung sollten Variablenbelegungen bestimmt werden, die zu hohen Ausführungskosten bei einem Pfad führen. Das Ziel dieser Arbeit war es, einen Entwickler dabei zu unterstützen, in einer bereits vorhandenen Anwendung Schleifen im Quellcode zu finden die Parallelisierungspotential besitzen. Hierzu sollte das auf der Compiler-Infrastruktur LLVM aufbauende Test-Programm KLEE erweitert werden. Die primäre Aufgabe von KLEE ist es, durch symbolische Ausführung Ausführungspfade durch Anwendungen zu bestimmen und deren Wege auf vorgegebene Fehlerfälle wie beispielsweise eine mögliche Division durch Null zu prüfen. Durch die symbolische Ausführung sollten Variablenbelegungen bestimmt werden, die zu hohen Ausführungskosten bei einem Pfad führen.

Um die Ausführungskosten zu bestimmen, wurde das Kostenmodell von LLVM verwendet. Es bietet die Möglichkeit die Kosten für die Ausführung einer Instruktion auf einer Zielarchitektur zu bestimmen. Dies ist notwendig, da nicht für alle LLVM Instruktionen passende native Übersetzungen vorhanden sind. Ein Beispiel hierfür ist die *switch* Instruktion, die plattformspezifisch durch eine Verkettung von Verzweigungspunkten oder durch den Einsatz einer Lookup-Tabelle umgesetzt wird.

Durch die Analyse der Pfade auf Hot-Spots, also Bereiche, die besonders hohe Ausführungskosten verursachen, wird einem Entwickler ermöglicht, gezielt diese auf ihr Parallelisierungspotential zu untersuchen. Dazu wurde KLEE dahingehend erweitert, dass das in einem vorgelagerten Schritt erstellte Kostenmodell verwendet wurde, um während der symbolischen Ausführung die Kosten aller Pfade zu bestimmen. Diese werden abschließend in einer Ergebnisdatei zur späteren Auswertung protokolliert. In der Auswertung sind für jeden Pfad die durchlaufenen Schleifen sowie die hierbei verwendeten symbolischen Variablen mit ihren für diesen Pfad gültigen konkreten Werten aufgelistet.

Um dem Problem der Pfad Explosion entgegen zu wirken, wurde in KLEE eine Einschränkung bei der Wahl von Ausführungszweigen implementiert. Hierzu wird einmalig zu Beginn der Analyse ein Interprocedural Control-Flow Graph aufgebaut. Dieser ermöglicht es Pfade zu Funktionen zu bestimmen, die Schleifen enthalten. Auf diese Weise können potentielle Pfade vorgegeben werden, die an Verzweigungspunkten gewählt werden sollen. Führt ein Pfad nicht in Richtung einer Schleife wird er vorzeitig abgebrochen und muss somit nicht weiter bearbeitet werden. Auf diese Weise wird das Problem der Pfad Explosion effektiv eingeschränkt.

Außerdem wurde die Funktionalität von KLEE eingeschränkt. Da der Einsatzzweck primär die Fehlersuche in Anwendungen ist, können bestimmte Funktionalitäten entfernt werden. Die Fehlersuche wird im Kontext dieser Arbeit nicht verwendet und ihre Deaktivierung führt somit zu einer Steigerung der Performanz, da überflüssige Prüfungen und Auswertungen durch den verwendeten Solver unterbunden werden.

Eine mögliche Erweiterung zur bestehenden Implementierung ist eine Verbesserung des eingesetzten Interprocedural Control-Flow Graphs. Er ist aktuell so umgesetzt, dass nur statische Aufrufziele verarbeiten werden. Dies sind Funktionsaufrufe, bei denen ohne dynamische Ausführung feststeht, welche Funktion aufgerufen wird. Viele Techniken in der Programmierung beruhen allerdings darauf, dass zur Laufzeit dynamisch entschieden wird, welche Funktion aufgerufen wird. Dies ist zum Beispiel bei einer Klasse mit Vererbung der Fall, die eine Funktion der Basis-Klasse überschreibt. Zur Laufzeit wird bei einem Funktionsaufruf anhand der aktuellen Instanz entschieden, ob die ursprüngliche oder die überschriebene Variante der Funktion aufzurufen ist. Dies führt dazu, dass diese Funktionsaufrufe aktuell nicht beachtet werden und somit mögliche Pfade zu Funktionen mit Schleifen nicht erkannt werden. Da verhindert wird, dass Pfade weiter ausgeführt werden, die nicht zu einer Schleife führen, fehlen diese in der abschließenden Auswertung. Lösen

könnte man das Problem, indem alle möglichen Aufrufziele statisch bestimmt werden und diese als statische Funktionsaufrufe behandelt werden.

Außerdem wäre es möglich bei den gefundenen Schleifen sog. false-positves auszufiltern. Im Kontext dieser Arbeit wären das Schleifen, die beispielsweise aufgrund von Datenabhängigkeiten zwischen den einzelnen Iterationen nicht einfach parallelisiert werden können. Hierzu könnte geprüft werden, ob Datenabhängigkeiten vorhanden sind und ob sich diese durch eine Umstrukturierung des auszuführenden Codes auflösen lassen.

Eine zusätzliche Erweiterung stellt das Bestimmen der tatsächlichen Ausführungszeit für einen Pfad dar. Die Ausführungskosten lassen sich nicht direkt in zeitliche Kosten übersetzen. Der Grund hierfür ist, dass identische Instruktionen auf unterschiedlichen Plattformen und Prozessoren unterschiedlich schnell verarbeitet werden. Zusätzliche Probleme bereiten nicht vorhersagbare Ereignisse wie das Umsortieren von Instruktionen durch den Prozessor, der aktuelle Zustand der Ausführungspipeline oder ob benötigte Daten bereits im schnellen Cache vorliegen oder von einem langsameren Speicher geladen werden müssen.

Trotz der erkannten Limitierungen der Implementierung konnte erfolgreich gezeigt werden, dass sich KLEE dazu nutzen lässt, gezielt Pfade zu Schleifen innerhalb einer Anwendung zu finden. Durch die symbolische Ausführung können Variablenbelegungen erzeugt werden, die es einem Entwickler ermöglichen Rückschlüsse über die Ausführungskosten der gefundenen Pfade zu ziehen und hierdurch eine Parallelisierung von Hot-Spots zu ermöglichen.

Literaturverzeichnis

[Al 07] Al Aho: *Compilers and Translators: Software Verification Tools Lecture 8: Loops in Flow Graphs*. `http://dragonbook.stanford.edu/lecture-notes/Columbia-COMS-W4117/07-09-27.html`, 2007. – [abgerufen am 19. September 2018] (zitiert auf Seite 32).

[BCD+18] Baldoni, Roberto ; Coppa, Emilio ; D'Elia, Daniele C. ; Demetrescu, Camil ; Finocchi, Irene: A Survey of Symbolic Execution Techniques. In: *ACM Comput. Surv.* 51 (2018), Nr. 3 (zitiert auf Seite 9).

[BCR11] Bethea, Darrell ; Cochran, Robert A. ; Reiter, Michael K.: Server-side verification of client behavior in online games. In: *ACM Transactions on Information and System Security (TISSEC)* 14 (2011), Nr. 4, S. 32 (zitiert auf Seite 20).

[BJS09] Burnim, Jacob ; Juvekar, Sudeep ; Sen, Koushik: WISE: Automated test generation for worst-case complexity. In: *Proceedings of the 31st International Conference on Software Engineering* IEEE Computer Society, 2009, S. 463–473 (zitiert auf Seite 25).

[BL96] Ball, Thomas ; Larus, James R.: Efficient path profiling. In: *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture* IEEE Computer Society, 1996, S. 46–57 (zitiert auf Seite 24).

[CDE+08] Cadar, Cristian ; Dunbar, Daniel ; Engler, Dawson R. u. a.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: *OSDI* Bd. 8, 2008, S. 209–224 (zitiert auf den Seiten 2, 14 und 20).

[CDF12] Coppa, Emilio ; Demetrescu, Camil ; Finocchi, Irene: Input-sensitive profiling. In: *ACM SIGPLAN Notices* 47 (2012), Nr. 6, S. 89–98 (zitiert auf Seite 24).

[CDFM13] Coppa, Emilio ; Demetrescu, Camil ; Finocchi, Irene ; Marotta, Romolo: Multithreaded input-sensitive profiling. In: *arXiv preprint arXiv:1304.3804* (2013) (zitiert auf Seite 24).

[CGP+08] Cadar, Cristian ; Ganesh, Vijay ; Pawlowski, Peter M. ; Dill, David L. ; Engler, Dawson R.: EXE: automatically generating inputs of death. In: *ACM Transactions on Information and System Security (TISSEC)* 12 (2008), Nr. 2, S. 10 (zitiert auf Seite 16).

[CLL16] Chen, Bihuan ; Liu, Yang ; Le, Wei: Generating performance distributions via probabilistic symbolic execution. In: *Proceedings of the 38th International Conference on Software Engineering* ACM, 2016, S. 49–60 (zitiert auf Seite 24).

[DMB08] De Moura, Leonardo ; Bjørner, Nikolaj: Z3: An efficient SMT solver. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems* Springer, 2008, S. 337–340 (zitiert auf Seite 16).

[DMB11] De Moura, Leonardo ; Bjørner, Nikolaj: Satisfiability Modulo Theories: Introduction and Applications. In: *Commun. ACM* 54 (2011), September, Nr. 9, 69–77. http://dx.doi.org/10.1145/1995376.1995394. – DOI 10.1145/1995376.1995394. – ISSN 0001–0782 (zitiert auf Seite 10).

[ES03] Eén, Niklas ; Sörensson, Niklas: An extensible SAT-solver. In: *International conference on theory and applications of satisfiability testing* Springer, 2003, S. 502–518 (zitiert auf Seite 9).

[HFF+11] Haedicke, Finn ; Frehse, Stefan ; Fey, Görschwin ; Grosse, Daniel ; Drechsler, Rolf: metaSMT: Focus on Your Application not on Solver Integration. In: *DIFTS FMCAD*, 2011 (zitiert auf Seite 16).

[Kin76] King, James C.: Symbolic execution and program testing. In: *Communications of the ACM* 19 (1976), Nr. 7, S. 385–394 (zitiert auf Seite 2).

[KKBC12] Kuznetsov, Volodymyr ; Kinder, Johannes ; Bucur, Stefan ; Candea, George: Efficient State Merging in Symbolic Execution. In: *SIGPLAN Not.* 47 (2012), Juni, Nr. 6, 193–204. http://dx.doi.org/10.1145/2345156.2254088. – DOI 10.1145/2345156.2254088. – ISSN 0362–1340 (zitiert auf Seite 9).

[KLE08] KLEE: *The reference manual for the KQuery language.* http://klee.github.io/docs/kquery/, 2008. – [abgerufen am 27. August 2018] (zitiert auf Seite 14).

[LA04] Lattner, Chris ; Adve, Vikram: LLVM: A compilation framework for lifelong program analysis & transformation. In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* IEEE Computer Society, 2004, S. 75 (zitiert auf den Seiten 2 und 10).

[LKP17] Luckow, Kasper ; Kersten, Rody ; Păsăreanu, Corina: Symbolic Complexity Analysis using Context-preserving Histories. In: *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on* IEEE, 2017, S. 58–68 (zitiert auf Seite 25).

[LLS+12] Li, Guodong ; Li, Peng ; Sawaya, Geof ; Gopalakrishnan, Ganesh ; Ghosh, Indradeep ; Rajan, Sreeranga P.: GKLEE: concolic verification and test generation for GPUs. In: *ACM SIGPLAN Notices* Bd. 47 ACM, 2012, S. 215–224 (zitiert auf Seite 21).

[LLV08] LLVM: *LLVM's Analysis and Transform Passes.* https://llvm.org/docs/Passes.html, 2008. – [abgerufen am 18. August 2018] (zitiert auf Seite 13).

[LLV12] LLVM: *Cost Model Analysis.* http://llvm.org/doxygen/CostModel_8cpp_source.html, 2012. – [abgerufen am 19. September 2018] (zitiert auf Seite 40).

[LR91] Landi, William ; Ryder, Barbara G.: Pointer-induced aliasing: A problem classification. In: *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* ACM, 1991, S. 93–103 (zitiert auf Seite 6).

[LT79] Lengauer, Thomas ; Tarjan, Robert E.: A fast algorithm for finding dominators in a flowgraph. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1 (1979), Nr. 1, S. 121–141 (zitiert auf den Seiten 32 und 40).

[MMZ+01] Moskewicz, Matthew W. ; Madigan, Conor F. ; Zhao, Ying ; Zhang, Lintao ; Malik, Sharad: Chaff: Engineering an efficient SAT solver. In: *Proceedings of the 38th annual Design Automation Conference* ACM, 2001, S. 530–535 (zitiert auf Seite 9).

[MPFH11] Ma, Kin-Keung ; Phang, Khoo Y. ; Foster, Jeffrey S. ; Hicks, Michael: Directed Symbolic Execution. In: *Proceedings of the 18th International Conference on Static Analysis*. Berlin, Heidelberg : Springer-Verlag, 2011 (SAS'11). – ISBN 978–3–642–23701–0, 95–111 (zitiert auf den Seiten 9, 26 und 30).

[PFK+15] Pedrosa, Luis ; Fogel, Ari ; Kothari, Nupur ; Govindan, Ramesh ; Mahajan, Ratul ; Millstein, Todd D.: Analyzing Protocol Implementations for Interoperability. In: *NSDI*, 2015, S. 485–498 (zitiert auf Seite 20).

[Pre10] Preuss, Adam: Implementation of Path Profiling in the Low-Level Virtual-Machine (LLVM) Compiler Infrastructure. (2010) (zitiert auf Seite 23).

[RSW+10] Ray, Abhijit ; Srikanthan, Thambipillai ; Wu, Jigang u. a.: Rapid techniques for performance estimation of processors. In: *Journal of Research and Practice in Information Technology* 42 (2010), Nr. 2, S. 147 (zitiert auf Seite 26).

[RSW+14] Riener, Heinz ; Soeken, Mathias ; Werther, Clemens ; Fey, Görschwin ; Drechsler, Rolf: MetaSMT: a unified interface to SMT-LIB2. In: *Specification and Design Languages (FDL), 2014 Forum on* Bd. 978 IEEE, 2014, S. 1–6 (zitiert auf Seite 16).

[SLA+10] Sasnauskas, Raimondas ; Landsiedel, Olaf ; Alizai, Muhammad H. ; Weise, Carsten ; Kowalewski, Stefan ; Wehrle, Klaus: KleeNet: discovering insidious interaction bugs in wireless sensor networks before deployment. In: *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks* ACM, 2010, S. 186–196 (zitiert auf Seite 20).

[URS14] Upadhyaya, Ganesha ; Rajan, Hridesh ; Sondag, Tyler: ParaSCAN: A Static Profiler to Help Parallelization. (2014) (zitiert auf Seite 23).

[WL94] Wu, Youfeng ; Larus, James R.: Static branch frequency and program profile analysis. In: *Proceedings of the 27th annual international symposium on Microarchitecture* ACM, 1994, S. 1–11 (zitiert auf Seite 24).

[YLK+17] Yoshida, Hiroaki ; Li, Guodong ; Kamiya, Takuki ; Ghosh, Indradeep ; Rajan, Sreeranga ; Tokumoto, Susumu ; Munakata, Kazuki ; Uehara, Tadahiro: KLOVER: Automatic Test Generation for C and C Programs, Using Symbolic Execution. In: *IEEE Software* 34 (2017), Nr. 5, S. 30–37 (zitiert auf Seite 21).

Anhang

A. KLEE Quelltextänderungen

Nachfolgend sind alle Quelltextdateien von KLEE aufgelistet, die im Rahmen dieser Arbeit verändert wurden. Die Ausgabe wurde durch git diff –stat erzeugt.

```
Implementierung/klee/.gitignore                     |   5 +
Implementierung/klee/examples/ICFG/test.c           |  26 ++
Implementierung/klee/examples/ICFG/test.ll          | 101 +++++++
Implementierung/klee/examples/isPrim/main.c         |  34 +++
Implementierung/klee/examples/isPrim/main.ll        | 174 ++++++++++++
Implementierung/klee/examples/loop/loop.bc          | Bin 0 -> 2416 bytes
Implementierung/klee/examples/loop/loop.c           |  20 ++
Implementierung/klee/examples/loop/loop.ll          | 109 ++++++++
Implementierung/klee/include/klee/ExecutionState.h |  15 ++
.../klee/Internal/Module/InstructionInfoTable.h     |   5 +-
.../include/klee/Internal/Module/KInstruction.h     |  11 +-
.../klee/include/klee/Internal/Module/KModule.h     |  47 +++-
Implementierung/klee/include/klee/Statistics.h      |   8 +
Implementierung/klee/lib/Core/CallPathManager.cpp   | 244 +++++++++++++++-
Implementierung/klee/lib/Core/CallPathManager.h     |  86 +++++-
Implementierung/klee/lib/Core/ExecutionState.cpp    |  29 +-
Implementierung/klee/lib/Core/Executor.cpp          |  94 ++++++-
Implementierung/klee/lib/Core/Executor.h            |   2 +
Implementierung/klee/lib/Module/CMakeLists.txt      |   1 +
.../klee/lib/Module/CallGraphBuilderPass.cpp        | 159 +++++++++++
.../klee/lib/Module/CallGraphBuilderPass.h          | 170 ++++++++++++
.../klee/lib/Module/ExtendedInfoStore.h             | 130 +++++++++
.../klee/lib/Module/ExtractFunctionInfoPass.h       | 300 ++++++++++++++++++++
.../klee/lib/Module/InstructionInfoTable.cpp        |  14 +-
Implementierung/klee/lib/Module/KInstruction.cpp    |   6 +-
Implementierung/klee/lib/Module/KModule.cpp         | 181 ++++++++++++-
Implementierung/klee/tools/klee/main.cpp            |  87 +++++-
27 files changed, 2019 insertions(+), 39 deletions(-)
```

B. Quelltextausschnitte

Quelltextausschnitt 8.1: Inhalt der Datei *maze.c* zum Testen der Pfadsuche.

```
1  #include <klee/klee.h>
2
3  int maze(int value)
4  {
5      if (value > 0)
6      {
7          if (value > 50)
8          {
9              if (value > 75)
10             {
```

```
11                return 1;
12            }
13            else
14            {
15                return 2;
16            }
17        }
18        else
19        {
20            if (value > 25)
21            {
22                int x = 3;
23                for (int i = 1; i < value; ++i)
24                {
25                    x += x * i;
26                }
27                return x;
28            }
29            else
30            {
31                return 4;
32            }
33        }
34    }
35    else
36    {
37        if (value > -50)
38        {
39            if (value > -25)
40            {
41                return 5;
42            }
43            else
44            {
45                return 6;
46            }
47        }
48        else
49        {
50            if (value > -75)
51            {
52                return 7;
53            }
54            else
55            {
56                return 8;
57            }
58        }
59    }
60 }
61
62 int main() {
```

```
63   int value;
64   klee_make_symbolic(&value, sizeof(value), "value");
65   return maze(value);
66 }
```

Quelltextausschnitt 8.2: Ausschnitt aus der Datei *maze.ll*.

```
1  ; ModuleID = 'maze.bc'
2  target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-
      i32:32:32-i64:64:64-f32:32:32-f64:64:64-v64:64:64-v128
      :128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-S128"
3  target triple = "x86_64-pc-linux-gnu"
4
5  @.str = private unnamed_addr constant [6 x i8] c"value\00"
      , align 1
6
7  ; Function Attrs: nounwind uwtable
8  define i32 @maze(i32 %value) #0 {
9    %1 = alloca i32, align 4
10   %2 = alloca i32, align 4
11   %x = alloca i32, align 4
12   %i = alloca i32, align 4
13   store i32 %value, i32* %2, align 4
14   %3 = load i32* %2, align 4, !dbg !17
15   %4 = icmp sgt i32 %3, 0, !dbg !17
16   br i1 %4, label %5, label %33, !dbg !17
17
18 ; <label>:5
19   %6 = load i32* %2, align 4, !dbg !19
20   %7 = icmp sgt i32 %6, 50, !dbg !19
21   br i1 %7, label %8, label %13, !dbg !19
22
23 ; <label>:8
24   %9 = load i32* %2, align 4, !dbg !22
25   %10 = icmp sgt i32 %9, 75, !dbg !22
26   br i1 %10, label %11, label %12, !dbg !22
27
28 ; <label>:11
29   store i32 1, i32* %1, !dbg !25
30   br label %46, !dbg !25
31
32 ; <label>:12
33   store i32 2, i32* %1, !dbg !27
34   br label %46, !dbg !27
35
36 ; <label>:13
37   %14 = load i32* %2, align 4, !dbg !29
38   %15 = icmp sgt i32 %14, 25, !dbg !29
39   br i1 %15, label %16, label %32, !dbg !29
40
41 ; <label>:16
42   store i32 3, i32* %x, align 4, !dbg !34
43   store i32 1, i32* %i, align 4, !dbg !37
```

```
44   br label %17, !dbg !37
45
46 ; <label>:17
47   %18 = load i32* %i, align 4, !dbg !37
48   %19 = load i32* %2, align 4, !dbg !37
49   %20 = icmp slt i32 %18, %19, !dbg !37
50   br i1 %20, label %21, label %30, !dbg !37
51
52 ; <label>:21
53   %22 = load i32* %x, align 4, !dbg !38
54   %23 = load i32* %i, align 4, !dbg !38
55   %24 = mul nsw i32 %22, %23, !dbg !38
56   %25 = load i32* %x, align 4, !dbg !38
57   %26 = add nsw i32 %25, %24, !dbg !38
58   store i32 %26, i32* %x, align 4, !dbg !38
59   br label %27, !dbg !40
60
61 ; <label>:27
62   %28 = load i32* %i, align 4, !dbg !37
63   %29 = add nsw i32 %28, 1, !dbg !37
64   store i32 %29, i32* %i, align 4, !dbg !37
65   br label %17, !dbg !37
66
67 ; <label>:30
68   %31 = load i32* %x, align 4, !dbg !41
69   store i32 %31, i32* %1, !dbg !41
70   br label %46, !dbg !41
71
72 ; <label>:32
73   store i32 4, i32* %1, !dbg !42
74   br label %46, !dbg !42
75
76 ; <label>:33
77   %34 = load i32* %2, align 4, !dbg !44
78   %35 = icmp sgt i32 %34, -50, !dbg !44
79   br i1 %35, label %36, label %41, !dbg !44
80
81 ; <label>:36
82   %37 = load i32* %2, align 4, !dbg !47
83   %38 = icmp sgt i32 %37, -25, !dbg !47
84   br i1 %38, label %39, label %40, !dbg !47
85
86 ; <label>:39
87   store i32 5, i32* %1, !dbg !50
88   br label %46, !dbg !50
89
90 ; <label>:40
91   store i32 6, i32* %1, !dbg !52
92   br label %46, !dbg !52
93
94 ; <label>:41
95   %42 = load i32* %2, align 4, !dbg !54
```

```
96    %43 = icmp sgt i32 %42, -75, !dbg !54
97    br i1 %43, label %44, label %45, !dbg !54
98
99  ; <label>:44
100   store i32 7, i32* %1, !dbg !57
101   br label %46, !dbg !57
102
103 ; <label>:45
104   store i32 8, i32* %1, !dbg !59
105   br label %46, !dbg !59
106
107 ; <label>:46
108   %47 = load i32* %1, !dbg !61
109   ret i32 %47, !dbg !61
110 }
111
112 ; Function Attrs: nounwind readnone
113 declare void @llvm.dbg.declare(metadata, metadata) #1
114
115 ; Function Attrs: nounwind uwtable
116 define i32 @main() #0 {
117   %1 = alloca i32, align 4
118   %value = alloca i32, align 4
119   store i32 0, i32* %1
120   %2 = bitcast i32* %value to i8*, !dbg !64
121   call void @klee_make_symbolic(i8* %2, i64 4, i8*
         getelementptr inbounds ([6 x i8]* @.str, i32 0, i32
         0)), !dbg !64
122   %3 = load i32* %value, align 4, !dbg !65
123   %4 = call i32 @maze(i32 %3), !dbg !65
124   ret i32 %4, !dbg !65
125 }
126
127 declare void @klee_make_symbolic(i8*, i64, i8*) #2
```

Quelltextausschnitt 8.3: Ausschnitt aus der Datei *main.ll* für die Primzahlbestimmung.

```
1  ; ModuleID = 'main.bc'
2  target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-
      i32:32:32-i64:64:64-f32:32:32-f64:64:64-v64:64:64-v128
      :128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-S128"
3  target triple = "x86_64-pc-linux-gnu"
4
5  ; Function Attrs: nounwind uwtable
6  define i32 @isPrimImpl(i32 %value) #0 {
7    br label %1, !dbg !135
8
9  ; <label>:1
10   %i.0 = phi i32 [ 2, %0 ], [ %7, %6 ]
11   %2 = icmp slt i32 %i.0, %value, !dbg !135
12   br i1 %2, label %3, label %8, !dbg !135
13
14 ; <label>:3
```

```
15    %4 = srem i32 %value, %i.0, !dbg !137
16    %5 = icmp eq i32 %4, 0, !dbg !137
17    br i1 %5, label %8, label %6, !dbg !137
18
19  ; <label>:6
20    %7 = add nsw i32 %i.0, 1, !dbg !135
21    br label %1, !dbg !135
22
23  ; <label>:8
24    %.0 = phi i32 [ 0, %3 ], [ 1, %1 ]
25    ret i32 %.0, !dbg !140
26  }
27
28  ; Function Attrs: nounwind uwtable
29  define i32 @isPrim(i32 %value) #0 {
30    %1 = icmp sle i32 %value, 0, !dbg !141
31    br i1 %1, label %5, label %2, !dbg !141
32
33  ; <label>:2
34    %value.off = add i32 %value, -1, !dbg !143
35    %switch = icmp ult i32 %value.off, 2, !dbg !143
36    br i1 %switch, label %5, label %3, !dbg !143
37
38  ; <label>:3
39    %4 = call i32 @isPrimImpl(i32 %value), !dbg !145
40    br label %5, !dbg !145
41
42  ; <label>:5
43    %.0 = phi i32 [ %4, %3 ], [ 0, %0 ], [ 1, %2 ]
44    ret i32 %.0, !dbg !146
45  }
46
47  ; Function Attrs: nounwind uwtable
48  define i32 @main() #0 {
49    %value = alloca i32, align 4
50    %1 = bitcast i32* %value to i8*, !dbg !147
51    call void @klee_make_symbolic(i8* %1, i64 4, i8*
         getelementptr inbounds ([6 x i8]* @.str, i32 0, i32
         0)), !dbg !147
52    %2 = load i32* %value, align 4, !dbg !148
53    %3 = icmp sgt i32 %2, -1, !dbg !148
54    %4 = zext i1 %3 to i32, !dbg !148
55    %5 = load i32* %value, align 4, !dbg !148
56    %6 = icmp slt i32 %5, 100, !dbg !148
57    %7 = zext i1 %6 to i32, !dbg !148
58    %8 = and i32 %4, %7, !dbg !148
59    %9 = sext i32 %8 to i64, !dbg !148
60    call void @klee_assume(i64 %9), !dbg !148
61    %10 = load i32* %value, align 4, !dbg !149
62    %11 = call i32 @isPrim(i32 %10), !dbg !149
63    ret i32 %11, !dbg !149
64  }
```

```
65
66  declare void @klee_make_symbolic(i8*, i64, i8*) #2
67
68  declare void @klee_assume(i64) #2
```