Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Ausarbeitung

KLEE LLVM Execution Engine

vorgelegt von

Luca Ciegelski

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Seminar Effiziente Programmierung
Betreuer: Jannek Squar
Hamburg, 27. Februar 2020

Inhaltsverzeichnis

1 Einleitung

Wer Software entwickelt, kommt um das Testen dieser in der Regel nicht herum. Durch komplexere Projekte, steigende Datenmengen und größere Teams entstehen immer mehr Fehlerquellen, die durch intensives und vollständiges Testen abgedeckt werden müssen. Häufig erfordert dies aber viel manuelle Arbeit: Es muss entschieden werden, welche Testfälle genutzt werden, in welchem Kontext das Programm getestet wird, ob die ausgegebenen Ergebnisse tatsächlich stimmen und weitere Einschätzungen, welche insbesondere durch den menschlichen Faktor nicht immer verlässlich sind[5]. Auf diese Art der Überprüfung kann sich speziell bei sicherheitskritischen Systemen nicht verlassen werden, aber auch der vergleichsweise hohe Zeitaufwand der Handarbeit ist generell zu vermeiden. Um dieses Problem zu lösen wurde eine Vielzahl von automatisierten Testwerkzeugen entwickelt, darunter auch KLEE [2].

2 Symbolische Ausführung

Ein grundlegender Bestandteil von KLEE ist die symbolische Ausführung, welche es von anderen, automatischen Testerzeugungs-Tools absetzt.[2, 3, 5, 7]

Statt wie bei einem regulären Programm einzelnen Variablen konkrete Werte zuzuweisen, definiert sich ein Programmzustand während der symbolischen Ausführung durch Bedingungen, die an möglichen Verzweigungen im Programmfluss erzeugt und zusammengefasst werden. Dadurch werden effektiv alle Zustandsmöglichkeiten für den speziellen Programmpfad abgedeckt, ohne exzessiv über alle Variablenbelegungen, Speicherzustände, etc. zu iterieren. Der Anwender definiert dafür einige Variablen mit Hilfe der KLEE-Funktion `klee_make_symbolic` als symbolisch (siehe Listing 1), diese werden dann durch symbolische Variablen ersetzt. Der Code wird nun schrittweise durchgegangen, wobei nicht zwingend immer ein Pfad zu Ende geführt wird, sondern mehrere aktive *States* des Durchlaufs koexistieren können. Jeder dieser States enthält folglich eigene Informationen über die Variablenbelegungen und Pfadbedingungen, welche bis zum aktiven Zeitpunkt angesammelt wurden.

Jedes Mal, wenn eine solche hinzugefügt wird, gelangen alle als Anfrage an einen sogenannten *Constraint Solver*. Dieser ist dafür verantwortlich zu überprüfen, ob alle aktuell gültigen Bedingungen gleichzeitig erfüllt werden können oder kein solcher Zustand existieren kann und das Programm somit fehlerhaften Code enthält. Gleichzeitig werden auch Bedingungen, welche sich gegenseitig einschließen, vereinfacht, um bei nachfolgenden Überprüfungen schneller arbeiten zu können.

Zusätzlich zur Nutzung mit dem Constraint Solver werden die Pfadbedingungen auch genutzt um allgemeinere Fehler wie Overflows oder Assert-Fehlschläge zu erkennen. Sobald ein solches Problem entdeckt wurde, wird der aktuell betrachtete Programmpfad terminiert und entsprechend der angesammelten Bedingungen ein Testfall mit konkreten Variablenbelegungen erzeugt. Dieser wird dann im unmodifizierten Originalprogramm angewendet und erzeugt im Regelfall genau den Fehler, welcher auch bei der symbolischen Ausführung festgestellt wurde.

3 KLEE

Das seit 2009 unter Open Source frei verfügbare Projekt KLEE ist ein Tool zur automatischen Testgenerierung und Fehlerfindung, welches auf der LLVM-Compilerinfrastruktur basiert. Es ist auf die Programmiersprache C und Systemprogramme ausgelegt, wurde allerdings bereits erfolgreich in weiteren Bereichen angewendet.

Das Ziel ist es, möglichst umfassende Testfälle für das zu testende Programm zu erzeugen, mit denen einerseits viele Codezeilen abgedeckt, andererseits auch möglichst alle Zustandsmöglichkeiten erfasst werden. Dazu bedient es sich einiger spezieller Ansätze. Das wohl wichtigste Merkmal ist die symbolische Ausführung, außerdem simuliert KLEE nicht nur die Internen Zustandsmöglichkeiten des Programms, sondern auch Interaktionen und Reaktionen von externen Komponenten, mit welchen das Programm interagiert, wie beispielsweise dem Dateisystem.

3.1 Anwendungsbeispiel

Betrachten wir nun einen beispielhaften Durchlauf von KLEE anhand des folgenden Programms *example.c*:

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

Listing 1: KLEE Codebeispiel, eigener Entwurf

In diesem Beispiel wurden beide Variablen x und y symbolisch definiert, entsprechend wurde dafür auch keine Initialisierung durchgeführt. Zunächst wird der Code in den LLVM-Bytecode übersetzt (je nach Umgebung können hier andere Argumente genutzt werden):

```
$ clang -I ../klee/include -emit-llvm -c -g -O0
```

Dies erzeugt die Datei `example.bc`; die Ausführung von KLEE kann darauf nun direkt gestartet werden. Für eine einfache, automatische Analyse des gesamten Programms reicht dafür ein parameterloser Aufruf aus:

```
$ klee example.bc
```

Aus diesem Aufruf entsteht für den gegebenen Beispielcode die folgende Ausgabe:

```
KLEE: output directory is "\ldots/klee-out-0"
KLEE: Using STP solver backend
KLEE: ERROR: example.c:12: divide by zero
KLEE: NOTE: now ignoring this error at this location
KLEE: done: total instructions = 44
KLEE: done: completed paths = 4
KLEE: done: generated tests = 4
```

KLEE findet in diesem Fall vier unterschiedliche Pfade durch das Programm, zu unterschiedlichem Verhalten führen. Drei davon entstehe durch die Fallunterscheidungen in Zeile 9 und 14, den letzten erzeugt die Division in Zeile 12 mit einem potenziellen Fehler im Falle einer Division durch 0. Auch dieser Fehler wird samt Fehlertyp explizit in der Ausgabe angegeben.

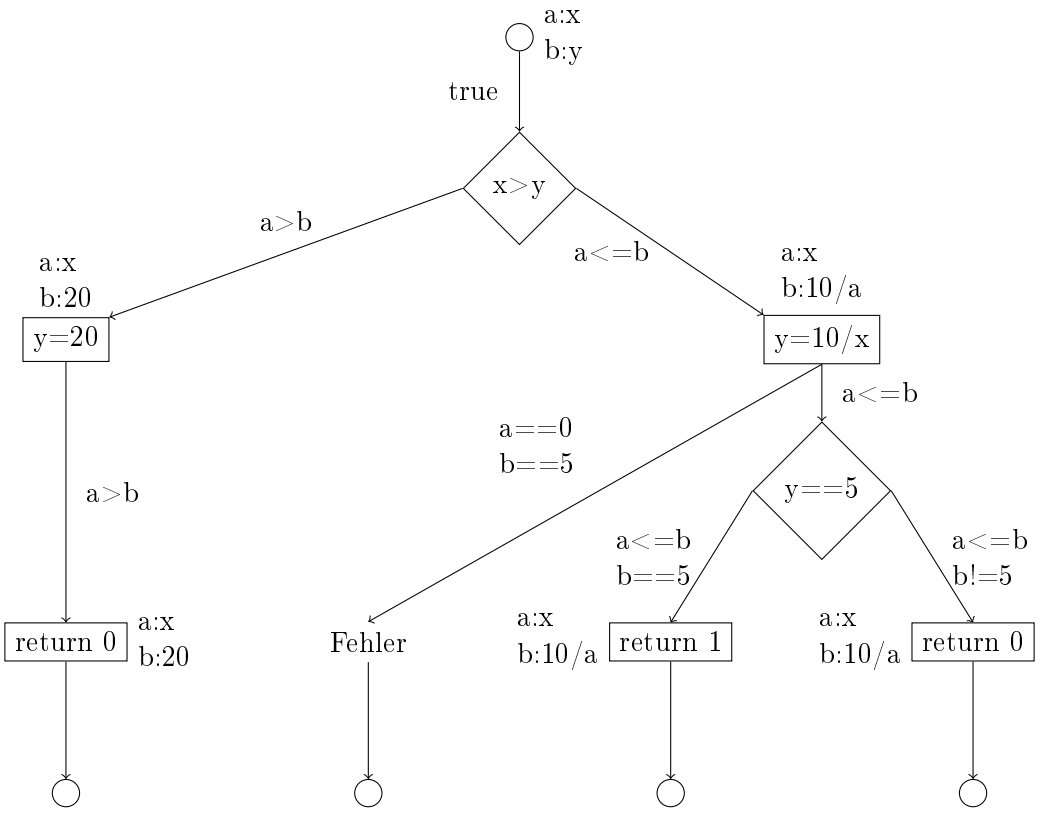

Abbildung 1: Pfadverarbeitungsbaum von example.c (Eigene Darstellung nach [9])

Durch solch einen Ausführungsbaum 3.1 lassen sich die einzelnen Zustände während der Ausführung darstellen. Neben der genauen Pfade durch das Programm werden

dort auch die Pfadbedingungen deutlich, welche zu einem bestimmten Zeitpunkt bereits angesammelt wurden; diese sind an den Übergangspfeilen notiert. Die aktuelle Belegung der Variablen findet sich jeweils an den Anweisungsknoten.

Während der Ausführung werden diverse Dateien erzeugt. Dazu gehören neben Statistiken über den Durchlauf und der von KLEE ausgeführten Assembly-Version des Codes `assembly.ll` auch detaillierte Informationen über die einzelnen Pfade welche abgesucht wurden, sowie die generierten Testfälle:

testXXXXXX.ktest

In den *.ktest-Dateien wird angegeben, welche konkreten Werte für die Überprüfung der Testfälle genutzt wurden. Diese wurden am Ende des jeweiligen Pfades basierend auf den Pfadbedingungen erzeugt und daraufhin auf den Ursprungscode angewendet.

```
1  ktest file : 'test000001.ktest'
2  args       : ['example.bc']
3  num objects: 2
4  object 0: name: 'x'
5  object 0: size: 4
6  object 0: data: n'\x02\x00\x00\x00'
7  object 0: hex : 0x01000000
8  object 0: int : 2
9  object 0: uint: 2
10 object 0: text: ....
11 object 1: name: 'y'
12 ...
```

Listing 2: Ausschnitt einer .ktest-Datei

testXXXXXX.kquery

Die Anfragestrukturen mit den Pfadbedingungen an den Constraint Solver lassen sich in den *.kquery-Dateien einsehen. Anfangs erfolgt eine Anpassung der genutzten Werte, darunter sind die Bedingungen in einer geschachtelten Struktur dargestellt.

```
1  array x[4] : w32 -> w8 = symbolic
2  array y[4] : w32 -> w8 = symbolic
3  (query [(Eq false
4               (Slt (ReadLSB w32 0 y)
5                    N0:(ReadLSB w32
                             0 x)))
6           (Eq false (Eq 0 N0))
7           (Eq 5 (SDiv w32 10 N0))]
8         false)
```

Listing 3: Ausschnitt einer .kquery-Datei

testXXXXXX.XXX.err

Für alle gefundenen Fehler werden außerdem Fehlerdateien angelegt. Diese enthalten Informationen zum Fehlertypen, den Auftrittsort des Fehlers sowie den dortigen Aufrufstack.

```
1  Error: divide by zero
2  File: example.c
3  Line: 12
4  assembly.ll line: 36
5  Stack:
6          #000000071 in
              klee_div_zero_check ()
              at ...
7          #100000036 in main () at
              example.c:12
```

Listing 4: Ausschnitt einer .err-Datei

3.2 Variationen in der Ausführung

Durch Vorkenntnisse des Programms sowie potenziellen Fehlern kann die Effizienz von KLEE gesteigert werden, wenn über Kommandozeilenoptionen entsprechende Anpassung bei der Ausführungen gemacht werden.

3.2.1 Pfadheuristiken

Die Abarbeitungsreihenfolge des Codes ist nicht zwingend rein zufällig, sondern kann beim Starten angepasst werden. Dafür existieren vier grundsätzliche Ansätze, die teilweise weiter variierbar sind:

Tiefensuche
Bei der Tiefensuche wird der aktive State wird so lange beibehalten, bis der Pfad beendet wird, wie es bei einem Programmfehler der Fall ist. Anschließend wird der so entstandene Pfad zurückgegangen, bis eine Abzweigung erreicht wurde, von der ein Unterpfad noch nicht abgearbeitet wurde. Dieser wird dann ebenfalls vollständig abgearbeitet und das Vorgehen wiederholt.

Zufällige Statewahl
Nachdem eine Operation an einem State ausgeführt wurde, wird zufällig ein neuer State ausgewählt, an dem die Ausführung fortgesetzt wird.

Zufällige Pfadwahl
Vom Startpunkt aus werden nacheinander die Operationen bis zur Pfadterminierung ausgeführt. Dabei werden bei Verzweigungen zufällig die nächste Operation ausgewählt, nach Beendigung eines Pfades wird erneut von vorne begonnen. Nach [5] führt dies zu einer breiteren Codeabdeckung, da durch das erneute Beginnen die Pfadbedingungen niedrig gehalten werden und die Suche daher mehr Freiraum hat.

Nicht-uniforme randomisierte Suche (NURS)
Auch hier wird nach jeder Aktion ein zufälliger nächster State ausgewählt, jedoch beeinflusst diesmal eine Wahrscheinlichkeitsverteilung die Auswahl. Optionen dafür sind nach [2] unter anderen die Aussicht auf Verarbeitung von neuem Code, die Kosten für die Anfrage an den Constraint Solver sowie eine pfadtiefenbasierte Verteilung.

Die Verfahren können zudem in Kombination miteinander genutzt werden, falls ein einzelnes nicht den gewünschten Effekt oder die gewünschte Performance liefert. Standardmäßig verwendet KLEE eine Kombination aus der zufälligen Pfadwahl und Auswahl nach Entdeckung von bisher unbearbeitetem Code. Um dieses Verhalten anzupassen, wird die Option –search verwendet

```
$ klee -search=dfs [...] example.bc
```

wobei bei [...] weitere Heuristiken angegeben werden können, um eine Mischung der Verfahren zu erzeugen, welche dann alternierend genutzt werden.

3.2.2 Abbruchsteuerung

Auch wenn es aus dem gegebenen Beispiel nicht direkt ersichtlich ist, kann die Ausführung eines KLEE-Durchlaufs große Ausmaße annehmen. Die Menge der möglichen Programmpfade wächst schnell mit der Größe des Programms und führt dabei nicht nur zu einem hohen Speicheraufwand, sondern auch zu hohem Aufwand für den Constraint Solver und damit zu Laufzeiten, welche mehrere Tage lang sein können. Dieses Verhalten ist zu erwarten und führt zu einer verbesserten Überprüfungsquote, unter Realbedingungen ist es jedoch Hilfreich, die Überprüfung schon vorzeitig beenden zu können. Dies hilft bei der Suche nach einem bestimmten bekannten oder vermuteten Fehler, ohne dass Zeit für die weitere Überprüfung des Codes aufgewendet wird. Mit der Option `-exit-on-error[-type]` ermöglicht KLEE dafür den Abbruch der Analyse nach dem auftreten eines beliebigen oder bestimmten Fehlers wie einem Overflow, einem Speicherproblem oder dem Fehlschlag einer eingebauten Assertion.

Umgekehrt bricht KLEE automatisch bei einem Problem mit dem Constraint Solver ab, dies kann mit der Option `-ignore-solver-failures` unterdrückt werden.

3.2.3 Zusätzliche Optionen

Neben diesen Einstellungen lassen sich die Durchläufe weiter anpassen. Um einen gezielten Teil des Programms zu analysieren bietet die Option `-entry-point` die Möglichkeit der Angabe einer einzelnen Funktion. Die Ausführung beginnt nun an dieser Stelle und nicht am Programmanfang.

Außerdem können nicht nur Programmvariablen, sondern auch die Startparameter symbolisch genutzt werden, um die Belegung beliebig zu gestalten. Mit `-sym-arg(s)` werden ein oder mehrere Kommandozeilenargumente symbolisch dargestellt und entsprechend im Programmfluss von KLEE behandelt

Zudem bietet KLEE die Erzeugung von symbolischen Dateien an, diese werden im nächsten Abschnitt 3.3 näher erläutert.

3.3 Umgebungssimulation

Durch die symbolische Ausführung werden bereits viele Fehlerquellen abgedeckt, welche mit anderen Methoden schwer auffindbar sein können. Jedoch sind die Zustände innerhalb des Programms nicht die einzigen Komponenten, welche Fehler und unvorhersehbares Verhalten verursachen können. Auch Interaktionen mit dem umliegenden Dateisystem, Benutzereingaben und weiteren Unbekannten müssen vollständig simuliert und kontrolliert werden, um die größtmögliche Fehlerzahl zu finden.

Dabei wird die folgenden Hilfestellung verwendet: Neben den symbolischen Variablen werden nun auch symbolische Dateien erzeugt, die Menge dieser wird beim Programmstart mit `-sym-files` angegeben. Wie symbolische Variablen können auch diese einen beliebigen Inhalt haben, je nachdem, wie weit sie bereits durch den bisherigen Programmpfad beschränkt wurden. Auch für sie werden an den Pfadenden

Inhalte für die Testfälle generiert, sodass bei der Anwendung auf das Originalprogramm dieselben Ergebnisse entstehen.

Um das Problem nun zu lösen werden für alle Operationen welche auf externe Komponenten zugreifen, wie read, ls und sort Modelle genutzt, welche die Reaktionen wahrheitsgetreu simulieren. Grundsätzlich gilt dabei, dass wenn auf eine real existierende Datei zugegriffen wird, auch die originale Funktion für den entsprechenden Aufruf genutzt und das Ergebnis durchgereicht wird. Handelt es sich dagegen um eine symbolische Datei, so wird anhand des Modells eine Reaktion vorhergesagt, mit mit welcher die symbolische Ausführung dann weiterarbeitet oder gegebenenfalls durch einen Fehler abbricht.

3.4 Probleme und Limitierungen

Auch wenn viele der genannten Aspekte eine sehr komfortable und effiziente Testumgebung erzeugen, bleibt auch KLEE nicht ohne Nachteile. Zunächst einmal benötigt KLEE für die Reproduzierbarkeit der während der symbolischen Ausführung generierten Testfälle einen deterministischen Programmfluss. Nach Aussage der Entwickler heißt es jedoch: "However, non-determinism in checked code and bugs in KLEE or its models have produced false positives in practice."[5] Entsprechend ist KLEE in der Originalversion nicht geeignet für parallele Programme, es existieren jedoch aufbauende Projekte, welche diese Funktionalität ermöglichen. [12]

Auch wird keine Aussage darüber getroffen, ob das Programm tatsächlich das richtige Ergebnis liefert, da im Kern nur einzelne Operationen an sich und im Kontext überprüft werden. Es sei zudem in Teilen schwer festzulegen, welche Anpassungen im Voraus gemacht werden müssen, damit KLEE auf die relevanten Programmteile beschränkt wird. [10]

4 Anwendung

Im folgenden Abschnitt wird näher erläutert, inwiefern KLEE bereits erfolgreich verwendet und weiterentwickelt wurde.

4.1 Nutzung von KLEE

Die Entwickler beschreiben in ihrer Veröffentlichung [5] ihre Anwendung auf unter anderem die GNU Coreutils[1] und die Busybox[4] Tools. Dabei wurden mit dem folgenden Befehl die Codeabdeckung verglichen, welche einerseits von KLEE und andererseits von den Entwicklertests der GNU Coreutils erzeugt wurde:

```
./run <tool-name> -max-time 60 -sym-args 10 2 2 -sym-files 2 8
[-max-fail 1]
```

Im Durchschnitt erreichte dabei KLEE eine wesentlich höhere Codeabdeckung, wobei die Auswertung den mehrfach genutzten Code von externen Bibliotheken nicht einschließt, um eine Verfälschung der Ergebnisse zu vermeiden.

Input	BUSYBOX	COREUTILS
`comm t1.txt t2.txt`	[does not show difference]	[shows difference]
`tee -`	[does not copy twice to stdout]	[does]
`tee "" <t1.txt`	[infinite loop]	[terminates]
`cksum /`	`"4294967295 0 /"`	`"/: Is a directory"`
`split /`	`"/: Is a directory"`	
`tr`	[duplicates input on stdout]	`"missing operand"`
`[ 0 ''<'' 1 ]`		`"binary operator expected"`
`sum -s <t1.txt`	`"97 1 -"`	`"97 1"`
`tail -2l`	[rejects]	[accepts]
`unexpand -f`	[accepts]	[rejects]
`split -`	[rejects]	[accepts]
`ls --color-blah`	[accepts]	[rejects]
t1.txt: a *t2.txt:* b		

Abbildung 2: Inkonsistenzen bei Ausgaben zwischen Busybox und den GNU-Coreutils [5]

Als weiterer Schritt sollten die Reaktionen von Busybox und den Coreutils auf bestimmte Eingaben verglichen werden. Dafür wurden die beiden Ergebnisse mit einem einfachen Assert-Befehl auf Gleichheit überprüft, sodass mit dieser Herangehensweise in Teilen nun doch die Gesamtausgabe des Programms überprüft werden kann. Die Tabelle 2 zeigt einige der gefundenen Unterschiede, sowie die Eingaben und Dateien, welche diese herbeigeführt haben.

4.2 Aufbauende Projekte

Um Programme testen zu können, welche außerhalb der üblichen Umgebungen für KLEE fallen, wurden von Dritten unterschiedliche Projekte entwickelt, um die bestehende Funktionalität für neue Probleme nutzbar zu machen.

Dazu gehören unter anderem Shadow of a Doubt zur Analyse von Programmunterschieden die entstehen, wenn bei Patchvorgängen Code verändert und nicht im neuen Kontext getestet wird[11]. KLOVER wurde entwickelt um den Funktionsumfang auch für C++ nutzbar zu machen[6]. Als letztes intressantest Konzept steht Docovery, welches mit symbolischer Verarbeitung beschädigte Dokumente reparieren können soll, ohne vorher die Dokumentenstruktur zu kennen[8].

5 Zusammenfassung

In dieser Ausarbeitung wurde die symbolische Ausführungsumgebung KLEE vorgestellt. Sie nutzt zur automatischen Testerzeugung symbolische Variablen und Dateien, um so viele mögliche Programmzustände wie möglich zu simulieren. Dabei werden für alle relevanten Programmzustände Testfälle generiert, die einfach auf das unveränderte Originalprogramm angewendet werden können.

Der Anwender kann dabei an vielen Stellen eingreifen und so gezielter bestimmte Funktionen testen und auch die Laufzeit teils erheblich reduzieren. Dazu gehören die Suchheuristik mit welcher der Code durchlaufen wird, sowie diverse Optionen zur Einschränkung der Analyse.

Zudem wurde die Umgebungssimulation erläutert, mit der KLEE externe Interaktionen simuliert und einen weiteren Fehlerbereich abdeckt.

Abschließend wurden Schwachpunkte aufgezeigt und Projekte, welche sich teilweise dieser annehmen und darüber hinaus die Grundfunktionalität nutzen, um in anderen Bereichen Tests erzeugen zu können.

Literatur

[1] 2019. URL: https://www.gnu.org/.

[2] KLEE LLVM Execution Engine. 2019. URL: https://klee.github.io.

[3] 2020. URL: https://en.wikipedia.org/wiki/Symbolic_execution.

[4] Erik Andersen. 2008. URL: https://busybox.net/.

[5] Dawson Engler Christian Cadar, Daniel Dunbar. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex System Programs. pages 1–15, 12 2008. URL: https://llvm.org/pubs/2008-12-OSDI-KLEE.pdf.

[6] Takuki Kamiya Indradeep Ghosh Sreeranga Rajan Susumu Tokumoto Kazuki Munakata Tadahiro Uehara Hiroaki Yoshida, Guodong Li. KLOVER: Automatic Test Generation for C and C++ Programs, Using Symbolic Execution. pages 30–37, 2017. URL: http://www.cs.utah.edu/~ligd/publications/KLOVER-CAV11.pdf.

[7] Ralf Kneuper. Validation und Verifikation von Software durch symbolische Ausführung. URL: http://www.kneuper.de/English/Publications/validation-verification.pdf.

[8] Tomasz Kuchta, Cristian Cadar, Miguel Castro, and Manuel Costa. Docovery: Toward generic automatic document recovery. In *International Conference on Automated Software Engineering (ASE 2014)*, pages 563–574, 9 2014.

[9] Andreas MOtzek. Symbolische Ausführung. URL: http://simplysomethings.de/computer+scienceㅤ+%26+information+technology/symbolische+ausf%C3%BChrung.html.

[10] Lee Rosenbaum Mark R.Tuttle Vincent Zimmer Oleksandr Bazhaniuk, John Loucaides. Symbolic execution for BIOS security. pages 3–4, 5 2015. URL: https://www.usenix.org/system/files/conference/woot15/woot15-paper-bazhaniuk.pdf.

[11] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. Shadow of a doubt: Testing for divergences between software versions. In *International Conference on Software Engineering (ICSE 2016)*, pages 1181–1192, 5 2016.

[12] Cristian Zamfir George Candea Stefan Bucur, Vlad Ureche. Parallel Symbolic Execution for Automated Real-World Software Testing. page 1, 4 2011. URL: https://dslab.epfl.ch/pubs/cloud9.pdf.