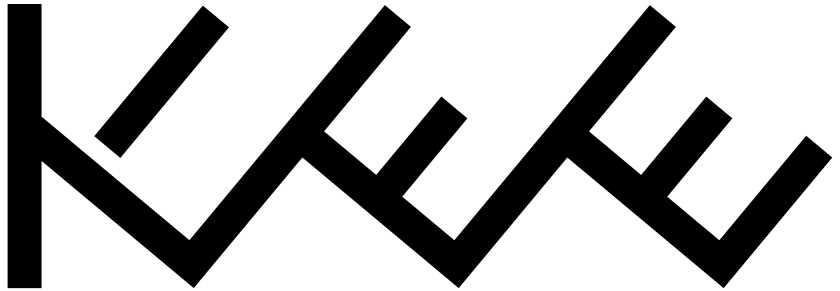

[1]

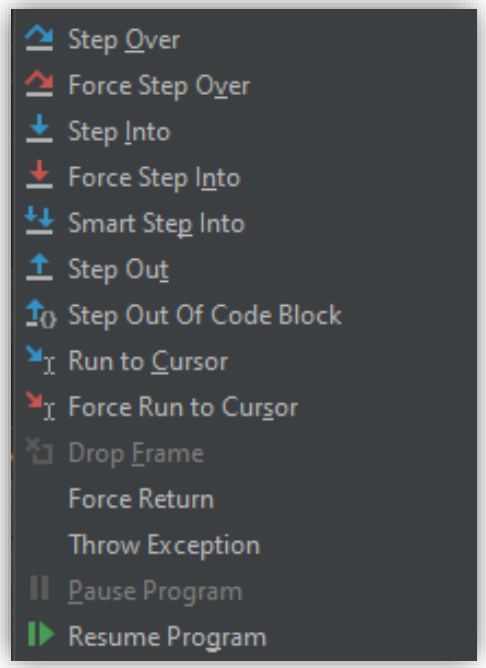
KLEE LLVM Execution Engine

Luca Ciegelski - 07.01.2020

Inhaltsverzeichnis

Probleme bei klassischem, manuellen Debugging

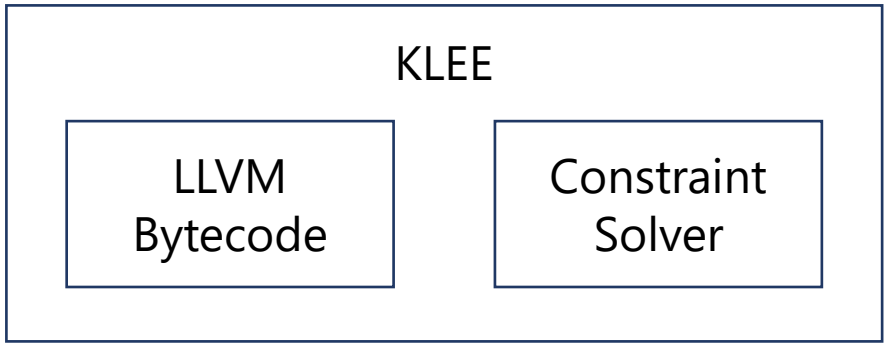
- Oft manuelle Eingriffe erforderlich
- Sehr zeitintensive Arbeit
- Ein Durchlauf ist immer nur ein bestimmter Kontext
- Menschliche Einflüsse täuschen Ergebnisse



[2]

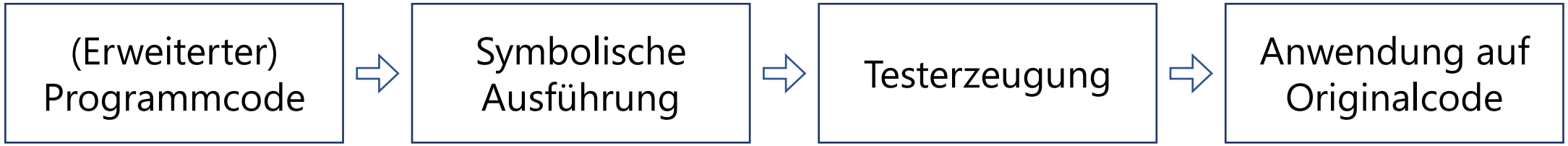
Was ist KLEE?

- Ausführungsumgebung für LLVM-Programmcode
- Seit 2009 Open Source
- Automatische Testgenerierung und Fehlerfindung
- Symbolische Ausführung für breite Wertebereichabdeckung
- Ausgelegt auf Systemprogramme



KLEE Funktionsweise

- Ziele
  - Automatische Tests erzeugen
  - Hohe Abdeckung des Codes erreichen
  - Zuständsmöglichkeiten und damit Fehlermöglichkeiten vollständig abdecken
  - Keine false positives erzeugen (nicht vorhandene Fehler)
  - Einfache Reproduktion der gefundenen Fehler



Symbolische Ausführung

- Keine feste Wertebindung an Variablen während der Überprüfung
- Stattdessen werden Bedingungen erzeugt, überprüft und angepasst
- Vergleiche erfolgen vorerst auf den symbolischen Ersatzvariablen
- Testfälle entspringen dem modifizierten Code, Anwendung erfolgt auf den Ursprungscode

KLEE Funktionsweise - Operationen

- *klee_make_symbolic()*
  - Macht eine Variable symbolisch, sodass alle Belegungen erkundet werden
- *klee_assume()*
  - Definiert eine Annahme im Code, Nichterfüllbarkeit zählt als Fehler
  - Praktisch eine weitere Pfadbedingung
- *klee_prefer_cex()*
  - Bevorzugt bestimme Variablenbelegungen bei der symbolischen Ausführung

KLEE Funktionsweise – CMDLine Optionen

- *-entry-point=<Funktion>*
  - Beginnt bei der angegebenen Funktion
- *-exit-on-error(-type=TYPE)*
  - Bricht nach dem ersten Fehler (vom Typ TYPE) ab
- *-sym-arg*
  - Macht ein Kommandozeilenargument symbolisch
- *-sym-files <Menge> <Größe>*
  - Erstellt *Menge* symbolische Dateien als Umgebung

Symbolische Ausführung

Programmcode

```
 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(&y, 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 }
```

Symbolische Ausführung

Programmcode

```
 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(&y, 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 }
```

Beispieldurchlauf

```
$ clang -I ../klee/include -emit-llvm -c -g -O0
example.c
$ ls
example.bc example.c
$ klee example.bc
KLEE: output directory is ".../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
$ ls
klee-last klee-out-0 example.bc example.c
$ ls klee-out-0
assembly.ll         messages.txt
run.stats           test000002.div.err
test000002.ktest    test000004.ktest
info                run.istats
test000001.ktest    test000002.kquery
test000003.ktest    warnings.txt
```

Dateiübersicht

– testXXXXXX.ktest – testXXXXXX.kquery – testXXXXXX.<errtype>.err	Informationen über einzelne Testfälle
– assembly.ll	Assembly-Version der .bc-Datei
– info – run.stats – run.istats – messages.txt – warnings.txt – …	Durchlaufinformationen

Dateiübersicht

test000003.ktest

```
ktest file : 'test000003.ktest'
ars        : ['example.bc']
num objects: 2
object 0: name: 'x'
object 0: size: 4
object 0: data: n'\x02\x00\x00\x00'
object 0: hex : 0x01000000
object 0: int : 2
object 0: uint: 2
object 0: text: ....
object 1: name: 'y'
object 1: size: 4
object 1: data: n'\xff\xff\xff\x7f'
object 1: hex : 0xffffff7f
object 1: int : 2147483647
object 1: uint: 2147483647
object 1: text: ....
```

test000003.kquery

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

Symbolische Ausführung

Programmcode

```
 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(&y, 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 }
```

Genutzte Eingabewerte

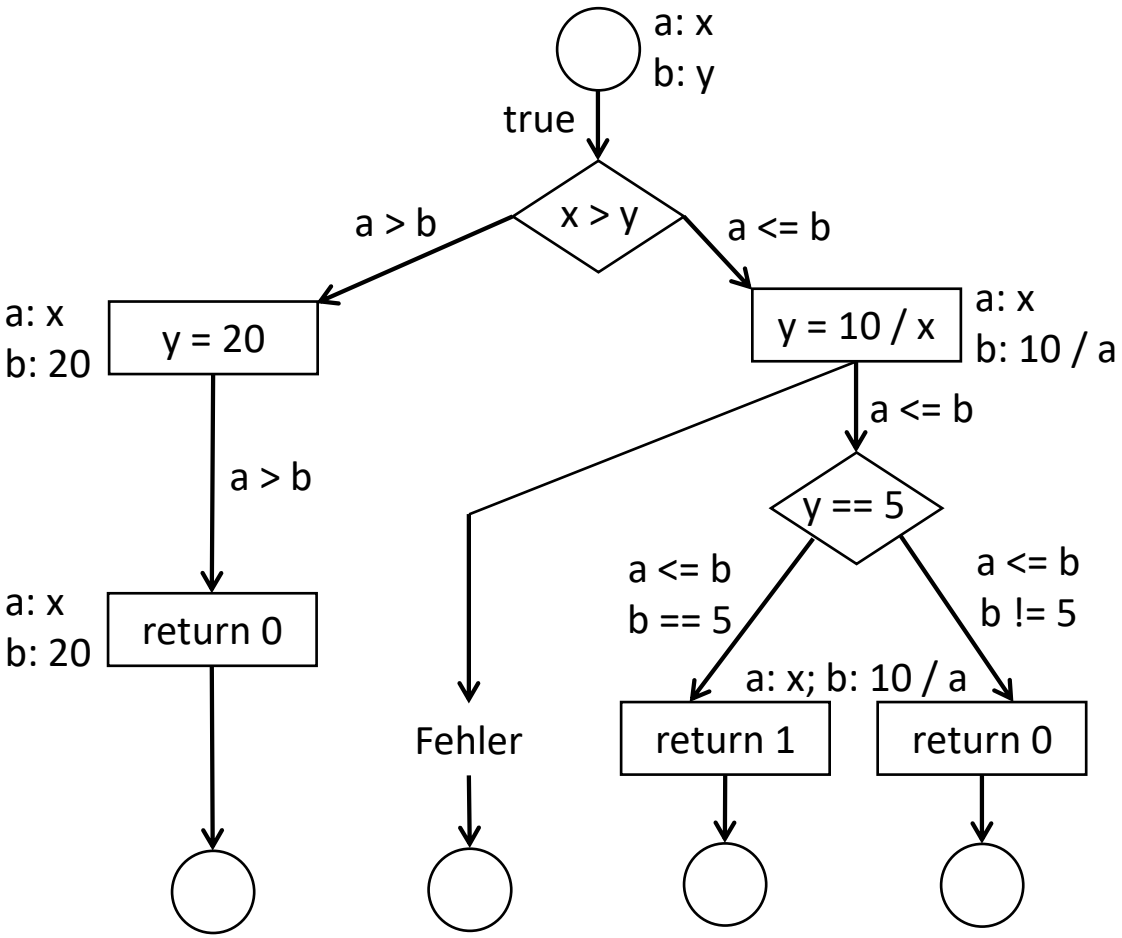
x	y	Ergebnis
1	0	*Fehlerfrei (0)*
0	0	*Divide by 0*
2	2147483647	*Fehlerfrei (1)*
16043009	2147483647	*Fehlerfrei (0)*

Symbolische Ausführung

Programmcode

```
 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(&y, 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 }
```

Symbolische KLEE-Verarbeitung

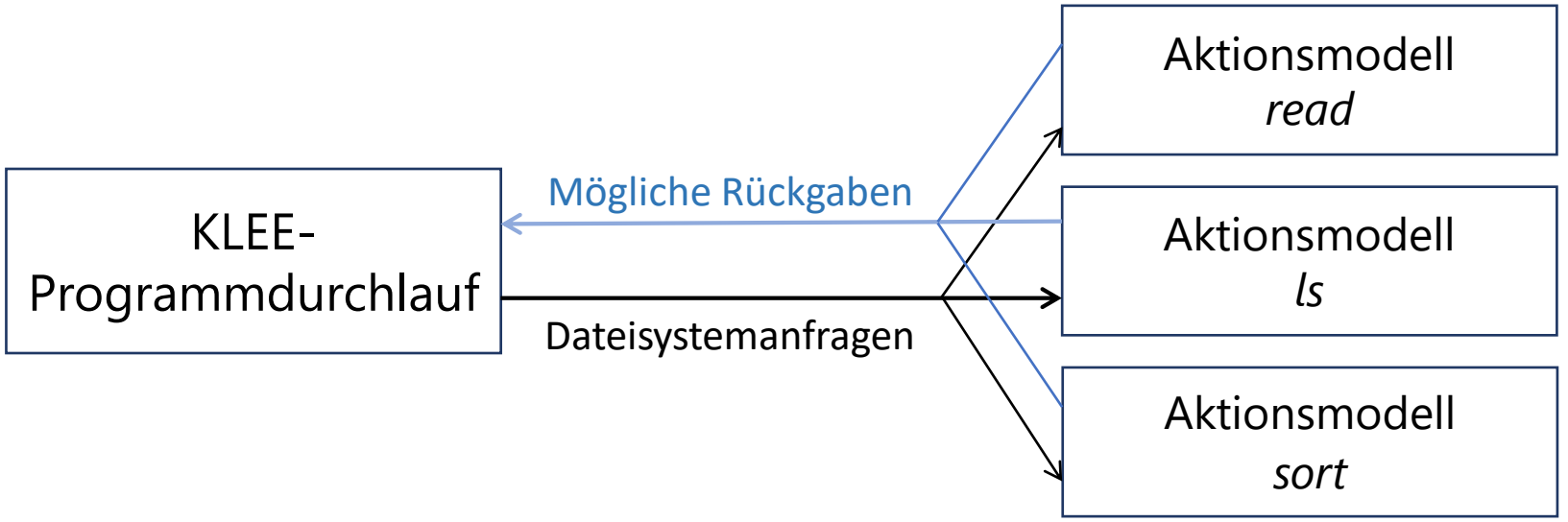


Eigene Darstellung nach [5]

Umgebungssimulation

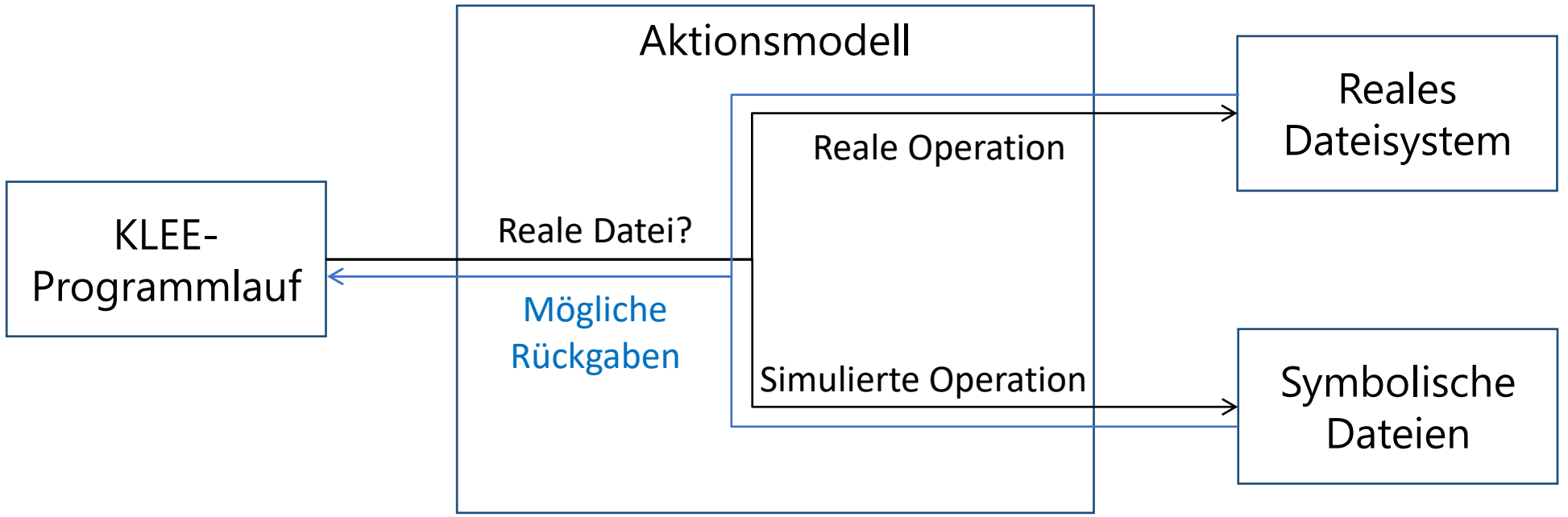
- Zur Einbeziehung von Benutzereingaben, Dateien, …
- Auch hier sollen alle möglichen Werte abgedeckt werden

- Vorgehensweise:



Umgebungssimulation

- Beispiel Dateisystem
  - Neben den realen Dateien existieren symbolische Dateien, welche beim Programmaufruf definiert werden
  - Modelle für nutzen beide Systeme möglicherweise gleichzeitig



Bisherige Anwendungen

- Test der GNU-Coreutils und Busybox

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

[4]

	Coreutils		Busybox	
Coverage (w/o lib)	**KLEE tests**	**Devel. tests**	**KLEE tests**	**Devel. tests**
100%	16	1	31	4
90-100%	40	6	24	3
80-90%	21	20	10	15
70-80%	7	23	5	6
60-70%	5	15	2	7
50-60%	-	10	-	4
40-50%	-	6	-	-
30-40%	-	3	-	2
20-30%	-	1	-	1
10-20%	-	3	-	-
0-10%	-	1	-	30
Overall cov.	84.5%	67.7%	90.5%	44.8%
Med cov/App	94.7%	72.5%	97.5%	58.9%
Ave cov/App	90.9%	68.4%	93.5%	43.7%

"The GNU Core Utilities are the basic file, shell and text manipulation utilities of the GNU operating system.

These are the core utilities which are expected to exist on every operating system."

[9]

Bisherige Anwendungen

- Test der GNU-Coreutils und Busybox
  - Vergleich der Endergebnisse zwischen beiden Varianten einzelner Programme

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		

[4]

Bisherige Anwendungen

- Statistiken
  - Coreutils: 10 bisher unbekannte Bugs [4]
  - Busybox: 21 Bugs im Schnelltest [4]
  - Minix: 21 Bugs im Schnelltest [4]
  - HiStar OS Kernel: Ein benannter Sicherheitsbug [4]
  - Coppelia: 31 reproduzierte Bugs [12]

Probleme und Limitierungen

- Annahme eines deterministischen Programms
- Durch vollständige Überprüfung extreme Laufzeiten möglich
- Vorbereitungen / Einschränkungen für KLEE oft nicht trivial
- Es werden nur Zwischenergebnisse überprüft, nicht aber die entgültige Ausgabe des Prorgamms

KLEE als Grundlage

- KleeNet [7]
  - Unterstützung bei verteilten Systemen
- Shadow of a Doubt [8]
  - Findet Bugs durch Verarbeitungsunterschiede in Patches
- KLOVER [10]
  - Symbolischs Tool für C++ Programme
- Coppelia [12]
  - Exploitfindung bei Prozessordesign
- Docovery [13]
  - Symbolische Dokumentenwiederherstellung

Zusammenfassung

- Einfache, bei Bedarf aber auch umfangreiche Testumgebung
- Während des Durchlaufs keine Eingriffe erforderlich
- Symbolische Ausführung als Grundlage für hohe Abdeckung
- Ergebnisse sind direkt nachvollziehbar und theoretisch reproduzierbar
- Aus Vollständigkeit resultiert hoher Aufwand
- Anpassbar auf die gewünschten Reaktionen und Randbedingungen

Literatur - 1

(1) https://klee.github.io/ *(28.12.2019)*.

(2) IntelliJ IDEA Ultimate by JetBrains, Version 2019.3.1.

(3) Renshaw, D.; Kong, S.: *Symbolic Execution on Difficult Environments*. 2011. http://www.cs.cmu.edu/~dwrensha/15-745/final.pdf. *(22.12.2019)*.

(4) Cadar, C.; Dunbar D., Engler, D.: *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Compley Systems Program*. https://llvm.org/pubs/2008-12-OSDI-KLEE.pdf. *(25.12.2019)*.

(5) http://simplysomethings.de/computer+science+%26+information+technology/symbolische+ausf%C3%BChrung.html *(23.12.2019)*.

(6) Kneuper, R.: *Validation und Verifikation von Software durch symbolische Ausführung*. http://www.kneuper.de/English/Publications/validation-verification.pdf *(20.12.2019)*.

Literatur - 2

(7) Sasnauskas, R.; Dustman, O.; Wehrle, K. et al.: *KleeNet – Symbolic Execution of Distributed Systems*. https://www.comsys.rwth-aachen.de/research/past-projects/kleenet/ *(29.12.2019)*.

(8) Palikerava, H.; Kuchta, T.; Cadar.: Shadow of a Doubt: Testing for Divergences Between Software Versions. 2016. https://srg.doc.ic.ac.uk/files/papers/shadow-icse-16.pdf (28.12.2019).

(9) https://www.gnu.org/ *(28.12.2019)*.

(10) Li, G.; Ghosh, I. Rajan, S.: *KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs*. http://www.cs.utah.edu/~ligd/publications/KLOVER-CAV11.pdf *(29.12.2019)*.

(11) https://en.wikipedia.org/wiki/Symbolic_execution *(29.12.2019)*.

(12) Zhang, R.; Deutschbein, C.; Huang, P.; Sturton, C.: *End-to-End Automated Exploit Generation for Validating the Security of Processor Design*. https://www.cs.unc.edu/~rzhang/files/MICRO2018.pdf *(03.01.2019)*.

(13) Kuchta, T.; Cadar, C.; Castro, M.; Costa, M.: *Docovery: Toward Generic Automatic Document Recovery. 2014*. https://srg.doc.ic.ac.uk/files/papers/docovery-ase-14.pdf. *(03.01.2019)*