Novi algoritam pretrage u KLEE-u

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Miloš Lončarević, Filip Miljaković, Jelena Ivković, Marija Mijailović
milosloncarevic494@gmail.com, filipmiljakovic1994@gmail.com,
jelena.ivkovic5@gmail.com, mijailovicmarija@hotmail.com

septmbar 2018.

Sažetak

U ovom radu je predstavljen nov algoritam pretrage u okviru KLEE alata. Pre samog opisa algoritma, predstavljeni su osnovni teorijski koncepti na kojima leži implementacija KLEE-a. Zatim je na malom skupu test primera, prikazano poređenje postojećih algoritama pretrage sa novim.

Sadržaj

1 Šta je KLEE? 2

2 Arhitektura 2

3 Prolazak kroz stablo 3

4 Upotreba 3

5 Opis rešenja problema 3
5.1 Primeri 4
5.1.1 Primer deljenje nulom 4
5.1.2 Primer paran 4
5.1.3 Primer maze 5
5.1.4 Primer armstrong 6
5.1.5 Ostali primeri 7

6 Zaključak 7

1 Šta je KLEE?

KLEE je javno dostupan alat koji služi za simboličko izvršavanje programa i za automatsko generisanje test primera. KLEE vrši analizu nad LLVM kodom i koristi SMT rešavač *STP* za proveravanje uslova ispravnosti koje generiše. KLEE koristi nekoliko optimizacija i heuristika za poboljšavanje pokrivenosti koda prilikom simboličkog izvršavanja. Koristi se i kao sastavni deo raznih platformi za razvijanje novih alata za analizu programa. Dva cilja su da pokrije svaku liniju izvornog koda u programu i da detektuje svaku opasnu operaciju ako postoji ijedna ulazna vrednost koja može da prouzrokuje grešku. Istraživanja su pokazala da su testovi koje je generisao, postigli pokrivenost od preko 90% i značjno prevazišli pokrivenost ručno napisanih testova od strane programera.

Simboličko izvršavanje automatski generiše ulaze za testiranje. Ideja ovakvog izvršavanja je da se izbegne ručno testiranje i nasumično biranje ulaznih vrednosti pri testiranju programa. Ulazi pri ovakvom načinu izvršavanja su proizvoljno izabrani simboli (npr. simbol λ) koji normalno učestvuju u svim izračunavanjima u okviru koda (prateći normalan tok izvršavanja programa). Kada dođe do uslovnog grananja u okviru izvršavanja programa, u čijem uslovu učestvuju pomenuti simboli, sistem prati obe grane i generiše skup ograničenja (tkz. uslov putanje) koja moraju da važe u toj grani. U trenutku kada se u jednoj od grana otkrije bag, izabrani simbol dobija konkretan skup vrednosti na osnovu uslova te putanje u kojoj se bag dogodio.

Pri izvršavanju, KLEE ima za cilj da prođe kroz svaku liniju koda koji je moguće izvršiti i da detektuje ulazne vrednosti koje potencijalno mogu dovesti do greške pri izvršavanju određenih računskih operacija. Za razliku od normalnog izvršavanja gde se na osnovu ulaza dobija rezultat operacija, simboličkim izvršavanjem se generišu ograničenja koja jasno opisuju skup mogućih vrednosti tih istih operacija.

2 Arhitektura

Svaki simbolički proces ima njemu pridružen registarski fajl, stek, hip, programski brojač i uslov putanje. Ovakvu reprezentaciju simboličkih procesa nazivamo stanje. KLEE prolazi kroz veliki broj ovih stanja prilikom generisanja testova za jedan program. On se izvršava u jednoj petlji koja određuje redosled odabira stanja čije se vrednosti uzimaju pri simboličkom izvršavanju koda. Petlja se izvršava sve dok se ne obiđe kod sa svim stanjima ili dok ne istekne maksimalno zadato vreme za izvršavanje. Ova stanja, za razliku od stanja normalnih procesa, predstavljena su u vidu stabla izraza gde su listovi simboličke promenljive ili konstante, a čvorovi su operacije u LLVM jeziku asemblera (npr. operacije poređenja ili aritmetičke operacije).

Pogledajmo primer:

$$\%dst = add\ i32\ \%src0,\ \%src1 \tag{1}$$

U kojem operacija sabiranja ima dva argumenta $\%src0$ i $\%src1$, izraz $Add(\%src0, \%src1)$ dodeljuje se registru $\%dst$. Prilikom građenja ovog izraza proverava se da li su argumenti konkretne vrednosti (konstante) i ukoliko jesu primenjuje se operacija sabiranja i rezultat se dodeljuje registru.

Uslovi grananja su bulovski izrazi koji mogu imati vrednosti *true* ili *false*, od kojih zavisi dalji tok programa. Takođe, KLEE može proveriti da li su ovi uslovi uvek zadovoljeni ili uvek nezadovoljeni i na osnovu toga ispratiti samo jedan (od moguća dva) tok programa. U suprotnom, obe grane se moraju ispratiti pri čemu se kopira stanje simboličkih procesa. Ono što je još zanimljivo za KLEE su potencijalno opasne operacije koje mogu uticati na generisanje skupa testova, npr. deljenje je opasna operacija zbog mogućnosti deljenja nulom.

3 Prolazak kroz stablo

Stablo stanja može biti veoma složeno, pa je potrebno pronaći optimalan način za prolazak kroz sve delove stabla. KLEE koristi sledeće dve heuristike pretrage:

- Nasumično biranje putanje - Sledeće stanje za izvršenje se bira tako što se putuje kroz binarno stablo od korena i na svakom grananju se nasumično bira putanja, tako da svaki skup stanja ima istu šansu da bude izabran bez obzira na veličinu podstabla.
- Pretraga bazirana na pokrivenosti - Izračunava koja stanja imaju najveću šansu da prođu kroz novi kod u bližoj budućnosti i na osnovu toga dodeljuje određenu težinu svim stanjima.

KLEE ove dve strategije koristi naizmenično i time umanjuje mane pojedinačnih strategija i podiže sveukupnu efektivnost.

4 Upotreba

Statističke informacije u vezi sa izvršavanjem Klee alata nad nekim bitkodom, kao što su broj izvršenih instrukcija, pokrivenost instukcija bitkoda, pokrivenost grana bitkoda i slično, možemo dobiti pomoću Python skripta *mesaure_time.sh*. Informacije se prikazuju u tabelarnoj formi. Legenda je prikazana u tabeli 1, a možemo je dobiti pomoću opcije --*help*.

Instrs	number of executed instructions
Time	total wall time (s)
Icov	instruction coverage in the LLVM bitcode (%)
Bcov	branch coverage in the LLVM bitcode (%)
Icount	total static instructions in the LLVM bitcode
Tsolver	time spent in the constraint solver

Tabela 1: Legenda tabele

5 Opis rešenja problema

U okviru KLEE-a postoji veći broj različitih vrsta pretrage (što možemo da vidite npr. u okviru *UserSearcher.cpp* fajla). Naš zadatak je bio da u okviru KLEE-a osmislimo i implementiramo nov algoritam pretrage kao i da napravimo eksperimente kojima ćemo uporediti postojeće algoritme pretrage sa našim algoritmom.

Naša ideja ovde je bila da KLEE-u prosleđujemo stanje tako što ćemo praviti AVL stablo, čiji čvorovi bi bili struktura podataka koja bi kao podatke imala ExecutionState, pokazivače na levo i desno podstablo, visinu stabla (koja služi da bi se vršilo balansiranje stabla), a njegov koren prosledjivali dalje. AVL smo izabrali iz razloga što nam daje brže osnovne operacije umetanja, brisanja, i pretrage čvorova stabla. Kao kriterijum za pravljenje i balansiranje smo uveli novi parametar *nasWeight*, koji se postavlja random za sve čvorove stabla. U nastavku ćemo videti primere korišćenja ovog načina pretrage u KLEE-u i poređenje sa već postojećim algoritmima koji postoje.

5.1 Primeri

Rad našeg algoritma pretrage smo uporedili sa *bfs, dfs, random path i random state* algoritmima pretrage, a izbor putanja različitih heuristika testirali smo na velikom broju primera. Sledi kratak opis svakog primera i prikaz dobijenih statističkih informacija.

Sve primere i dobijene rezultate možeete naći u folderu primeri

5.1.1 Primer deljenje nulom

Ovo su primeri ex002-{1|2|v}.c -ex002-{1|2|v}.c

Ovde imamo N simboličkih promenljivih i za svaku od njih imamo **if-else** grane. Nakon tih provera imamo i računanje količnika. Zbog toga što je nedozvoljeno deliti sa 0, KLEE implicitno za tu naredbu dodaje proveru, da li je delilac različit od nule. Takvo ponašanje se javlja kod svih kritišnih mesta u kodu. Imamo 3 verzije programa. Verzije sa sufiksom -1.c i -2.c imaju deljenje 0 samo na različitim putanjama, dok verzija sa sufiksom -v.c nema spornu putanju.

Ukoliko otvorimo generisane datoteke sa izveštajima dfs, bfs, random-path, random-state, naš primećujemo da se naš algoritam pretrage ponaša gotovo isto kao i ostali, i da uglavnom odmah pronalazi spornu putanju. Takođe za 12 promenljivih pre došao do sporne putanje u odnosu na random-state.

5.1.2 Primer paran

Program: ex020-even.c

Ovde se vrši provera da li je argument paran ili neparan po vrednosti. U nastavku je dat prikaz dobijenih statističkih informacija i možemo da vidimo da je vreme koje je potrošeno na solver za naš algoritam najgore.

DFS izveštaj:

```
ex020-even.bc

------------------------------------------------------------------------
|  Path   |  Instrs|  Time(s)|  ICov(%)|  BCov(%)|  ICount|  TSolver(%)|
------------------------------------------------------------------------
|klee-last|      42|     0.01|    12.16|     5.71|     296|       85.82|
------------------------------------------------------------------------
```

BFS izveštaj:

```
ex020-even.bc

------------------------------------------------------------------------
```

```
|  Path   |  Instrs|  Time(s)|  ICov(%)|  BCov(%)|  ICount|  TSolver(%)|
--------------------------------------------------------------------------
|klee-last|      42|     0.01|    12.16|     5.71|     296|       86.63|
--------------------------------------------------------------------------
```

Random-state izveštaj:

```
ex020-even.bc

--------------------------------------------------------------------------
|  Path   |  Instrs|  Time(s)|  ICov(%)|  BCov(%)|  ICount|  TSolver(%)|
--------------------------------------------------------------------------
|klee-last|      42|     0.02|    12.16|     5.71|     296|       89.88|
--------------------------------------------------------------------------
```

Random-path izveštaj:

```
ex020-even.bc

--------------------------------------------------------------------------
|  Path   |  Instrs|  Time(s)|  ICov(%)|  BCov(%)|  ICount|  TSolver(%)|
--------------------------------------------------------------------------
|klee-last|      42|     0.02|    12.16|     5.71|     296|       90.86|
--------------------------------------------------------------------------
```

Naš izveštaj:

```
ex020-even.bc

--------------------------------------------------------------------------
|  Path   |  Instrs|  Time(s)|  ICov(%)|  BCov(%)|  ICount|  TSolver(%)|
--------------------------------------------------------------------------
|klee-last|      42|     0.05|    12.16|     5.71|     296|       96.31|
--------------------------------------------------------------------------
```

5.1.3 Primer maze

Program: ex022-maze.c

Primer jednostavne igrice, zadatak je pronaći put kroz lavirint koji vodi do nagrade gde je startna pozicija igrača je gornji levi ugao lavirinta. U ovakim zadacima takođe možemo iskoristiti KLEE za pronalaženje rezltata. Iz docijenih izveštaja možemo da primetimo da je BFS u ovom slučaju najbrže pronašao put do cilja. Što se tiče našeg algoritma, u ovom primeru, je pokazao vreme koje je u rangu sa ostalim heuristikama, a ponajviše sa random-path.

BFS izveštaj:

```
ex022-maze.bc

--------------------------------------------------------------------------
|  Path   |  Instrs|  Time(s)|  ICov(%)|  BCov(%)|  ICount|  TSolver(%)|
--------------------------------------------------------------------------
|klee-last|   40898|     3.10|    38.16|    17.07|     435|       12.24|
--------------------------------------------------------------------------
```

Random-state izveštaj:

```
ex022-maze.bc

------------------------------------------------------------------------
|  Path   |  Instrs|  Time(s)|  ICov(%)|  BCov(%)|  ICount|  TSolver(%)|
------------------------------------------------------------------------
|klee-last|   42020|     3.09|    38.16|    17.07|     435|       12.34|
------------------------------------------------------------------------
```

DFS izveštaj:

```
ex022-maze.bc

------------------------------------------------------------------------
|  Path   |  Instrs|  Time(s)|  ICov(%)|  BCov(%)|  ICount|  TSolver(%)|
------------------------------------------------------------------------
|klee-last|   38010|     3.11|    38.16|    17.07|     435|       12.85|
------------------------------------------------------------------------
```

Naš izveštaj:

```
ex022-maze.bc

------------------------------------------------------------------------
|  Path   |  Instrs|  Time(s)|  ICov(%)|  BCov(%)|  ICount|  TSolver(%)|
------------------------------------------------------------------------
|klee-last|   40244|     3.08|    38.16|    17.07|     435|       13.14|
------------------------------------------------------------------------
```

Random-path izveštaj:

```
ex022-maze.bc

------------------------------------------------------------------------
|  Path   |  Instrs|  Time(s)|  ICov(%)|  BCov(%)|  ICount|  TSolver(%)|
------------------------------------------------------------------------
|klee-last|   41019|     3.12|    38.16|    17.07|     435|       13.17|
------------------------------------------------------------------------
```

5.1.4 Primer armstrong

Program: ex025-armstrong.c

Vrši se provera da li n-tocifren prirodan broj jednak zbiru n-tih stepa svojih cifara. Prilikom izvršavanja ovog primera koristeći BFS vidimo da je 100% vremena izvršavanja prošlo na solveru, dok su ostale heuristike, pa i naša, odmah dale odgovor na postavljeno pitanje. Dakle i u ovom slučaju naš algoritam parira postojećim.

BFS izveštaj:

```
ex022-maze.bc

------------------------------------------------------------------------
|  Path   |  Instrs|  Time(s)|  ICov(%)|  BCov(%)|  ICount|  TSolver(%)|
------------------------------------------------------------------------
|klee-last|   40898|     3.10|    38.16|    17.07|     435|       12.24|
------------------------------------------------------------------------
```

5.1.5 Ostali primeri

Ostali primeri nad kojima smo vršili poredjenje različitih heuristika pretrage su ex021-inverse.c ex023-password.c ex024-pointer.c ex026-guess_game.c ex027-misionari.c

U ovim primerima dobijeni izveštaji pokazuju da nam za svaku heuristiku KLEE odmah nailazi na traženi zahtev.

6 Zaključak

Nad testiranim primerima naš algoritam je zadovoljavajuće parirao već postojećim. U slučaju programa sa većim brojem promeljivih, i samim tim velikim broj putanja, izvršavanje, svih heuristika pretrage, je trajalo dugo stoga da ne možemo da garantujemo da i u tim slučajevima naš algoritam parira ostalima. KLEE se u praksi pokazao kao jako dobar što se tiče pronalaska grešaka i pokrivenosti koda koji proverava. Za dalji razvoj ovog, a i slličnih alata potrebno je da se što više programera bolje upozna sa njima i da ih u praksi koristi.