

Uvid u KLEE: Alat za automatsko generisanje testova visokog pokrića za programe složenih sistema

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Andrea Pilipović
andrea.pilipovic@yahoo.com

3. maj 2018.

Sažetak

U ovom tekstu upoznajemo se sa alatom za simboličko izvršavanje pod nazivom KLEE. To je alat koji automatski generiše test primere visokog pokrića na različitim skupovima kompleksnih programa. KLEE je korišćen za temeljnu proveru 89 samostalnih programa u okviru GNU COREUTILS, koji predstavljaju jezgro korisničkog okruženja instaliran na milione UNIX sistema. Testovi koje je generisao KLEE su postigli pokrivenost od preko 90% i značajno prevazišli pokrivenost ručno napisanih testova od strane programera. KLEE se koristi i kao alat za nalaženje bagova i primenjen je na najmanje 452 aplikacije (sa preko 430 hiljada linija koda) gde je pronašao 56 ozbiljnih bagova, uključujući i 3 бага u COREUTILS-u koje niko nije primetio više od 15 godina. [1]

Sadržaj

1	Uvod	2
2	Problemi sa kojim se suočava KLEE i njegova upotreba	2
2.1	Korišćenje alata	3
3	Arhitektura KLEE alata	3
3.1	Unapređenja koja koristi KLEE	4
3.2	Prolazak kroz stablo stanja	4
4	Modeli okruženja	5
5	Zaključak	5
	Literatura	5

1 Uvod

Simboličko izvršavanje (eng. Symbolic execution) 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, eng. path condition) 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. [2]

Pogledajmo naredni deo koda (slika 1), kako bi nam bilo jasnije šta znači simboličko izvršavanje.

```
1 int f() {
2   ...
3   y = read();
4   z = y * 2;
5   if (z == 12) {
6     fail();
7   } else {
8     printf("OK");
9   }
10 }
```

Slika 1: Primer jednostavne funkcije u okviru koda

Pri simboličkom izvršavanju, program čita simboličku vrednost (npr. simbol λ) i pridružuje je promenljivoj y . Program zatim množi tu vrednost i promenljivoj z se dodeljuje vrednost $\lambda * 2$. Nakon ovoga dolazimo do naredbe grananja i proverava se uslov $\lambda * 2 == 12$ (što je istovremeno jedan od moguća dva uslova putanje našeg primera). U ovom trenutku, λ može biti bilo koja vrednost pa se izvršavanje programa nastavlja u obe grane, prateći dve putanje. Svako putanji pridružuje se kopija stanja programa, tako da se obe putanje mogu samostalno simbolički izvršiti. U trenutku prekida programa (u našem slučaju naredba `fail()` u liniji broj 6), simboličko izvršavanje dodeljuje konkretnu vrednost simbolu λ na osnovu uslova putanje. Tako na kraju dobijamo skup ovih konkretnih vrednosti, koji bi u našem slučaju bio skup $\{6\}$. Na ovaj način programer može testirati isti program sa ulaznom vrednošću $\{6\}$ i otkriti grešku u kodu.

2 Problemi sa kojim se suočava KLEE i njegova upotreba

Iako su se ovakvi alati pokazali veoma uspešni u određenim situacijama, ostavlja se pitanje da li ovaj pristup može konstantno postizati isti dobar rezultat na pravim aplikacijama. Dva glavna problema su: (1)

Kompleksnost - Sa porastom broja naredbi grananja eksponencijalno raste broj različitih putanja koda. (2) Zavisnost od okruženja - Veći deo koda zavisi od ulaznih parametara koji dolaze iz okruženja u kojima se program izvršava. S obzirom da ovi parametri mogu biti netačni, kod mora obraditi sve greške koje ovakav ulaz može izazvati.

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. KLEE to postiže simboličkim izvršavanjem programa. 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.

2.1 Korišćenje alata

Korisnik može lako da proveri svoj program uz pomoć KLEE alata. Pokazaćemo način upotrebe ovog alata za GNU C. Potrebno je, pre svega, da se program kompajlira pomoću LLVM kompajlera. Kompajliramo neki `tr.c` program pomoću sledeće naredbe:

```
llvm-gcc -emit-llvm -c tr.c -o tr.bc
```

i dobijamo odgovarajući bajtkod (eng. *bytecode*). Korisnik zatim pokreće KLEE nad generisanim bajtkodom, uz mogućnost zadavanja broja, veličine i tipa simboličkih ulaza sa kojima se kod testira. Za naš `tr` program komanda bi izgledala npr. ovako:

```
klee -max-time 2 -sym-args 1 10 10 -sym-files 2 2000  
-max-fail 1 tr.bc
```

Opcija `-max-time` govori KLEE alatu da testira program maksimalno 2 minuta. Naredne opcije govore o simboličkim ulazima, pa tako opcija `-sym-args 1 10 10` govori alatu da koristi najviše tri argumenta komandne linije pri čemu je prvi argument dužine jednog karaktera, a druga dva dužine deset. Opcija `-sym-files 2 2000` govori alatu da koristi standardni ulaz i jedan fajl, pri čemu oba sadrže po 2000 bajtova simboličkih podataka. Poslednja opcija `-max-fail 1` govori da program prekine izvršavanje pri najviše jednom sistemskom pozivu pri prolasku kroz svaku putanju.

3 Arhitektura KLEE alata

Svaki simbolički proces ima njemu pridružen registarski fajl (eng. *register file*), stek, hip, programski brojač i uslov putanje. Ovakvu reprezentaciju simboličkih procesa nazivamo stanje (eng. *state*). 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 (eng. *trees*) gde su listovi simboličke promenljive ili konstante a čvorovi (eng. *nodes*) su operacije u LLVM jeziku asemblera (npr. operacije poređenja ili aritmetičke operacije). Pogledajmo primer:

```
%dst = add i32 %src0, %src1
```

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 granjanja su bulovski izrazi (eng. boolean expressions) 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 mogućih 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.

3.1 Unapređenja koja koristi KLEE

U praksi, broj stanja koja pravi KLEE prilikom izvršavanja brzo raste. Prilikom pokretanja Coreutils programa sa 1GB memorije, maksimalan broj stanja iznosio je 95,982 stanja, a prosek ovih stanja bio je 51,385. [1] Kako bi smanjili ove brojeve, KLEE primenjuje tkz. copy-on-write tehniku deljenja memorije stanja, pri čemu se značajno smanjuje memorija koja je potrebna za čuvanje jednog stanja. Ovim se smanjio i sam broj stanja koja se prave prilikom izvršavanja, jer se podaci o više istih stanja ne kopiraju.

Najskuplja operacija KLEE alata je rešavanje skupa ograničenja. Pri tome, bilo je potrebno pronaći način optimizovanja ovih upita ograničenja. Navešćemo najbitnija pronađena rešenja:

- Pojednostavljanje izraza - najjednostavnija optimizacija koja se primenjuje. Primer ovakve optimizacije bi bio linearno pojednostavljanje ($2*x - x = x$).
- Pojednostavljanje skupa ograničenja - u slučaju kada imamo skup ograničenja $\{x < 10\}$ a zatim je potrebno dodati novo ograničenje $x=5$, KLEE pojednostavi skup tako što iz njega izbriše prethodno ograničenje i ostavi novo ($x=5$).
- Konkretizacija podrazumevane vrednosti - u trenutku kada se ograničenje $x+1=10$ dodaje uslovu putanje, promenljivoj x se dodeljuje konkretna vrednost (u ovom slučaju $x=9$).
- Međusobna zavisnost ograničenja - Ovaj način optimizacije grupiše ograničenja iz skupa na osnovu simboličkih promenljivih koje ograničenja sadrže. Na ovaj način KLEE eliminiše nepotrebna ograničenja za određeni upit koji se šalje STP-u (rešavač ograničenja).

Primenjujući ova i mnoga druga ograničenja, znatno se smanjuje potrebno vreme za rešavanje skupa ograničenja (samim tim i generisanja testova).

3.2 Prolazak kroz stablo stanja

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 Modeli okruženja

Mnogi programi prilikom izvršavanja komuniciraju sa okruženjem (koriste argumente komandne linije, fajlove i promenljive iz okruženja, itd.). S toga, potrebno je generisati i testove koji pokrivaju sve ove slučajeve komunikacije programa i okruženja. Ovo se postiže koristeći modele (eng. models) koji generišu testove za pozive operacija (npr. open, read, write, stat, lseek...) koje se pozivaju prilikom ovakvih komunikacija. Ovi modeli napisani su u C programskom jeziku i mogu se lako modifikovati po potrebi.

Model koji obrađuje sistemske operacije nad fajlovima proverava da li se operacija (npr. lseek, open...) poziva nad konkretnim fajlom i u tom slučaju se operacija i izvršava nad tim fajlom. Ukoliko se radi o simboličkim fajlovima, operacija se simbolički izvršava pri čemu korisnik zadaje broj N simboličkih fajlova nad kojima se operacija izvršava kao i veličine tih N fajlova. Ovakvim pozivom operacije kreira se N mogućih putanja izvršavanja koda, kao i jedna koja rezultuje grešku (ukupno N+1 putanja).

Pored operacija nad fajlovima, sistemski pozivi mogu da prouzrokuju grešku i iz drugih neočekivanih razloga (npr. write() može da izbaci grešku u slučaju kada je disk pun). KLEE zna da opciono obradi i ovakve greške simulirajući greške okruženja. Međutim, ovakva obrada nije važna svakoj aplikaciji, pa se ta opcija neretko isključuje.

5 Zaključak

Zajedno sa porastom linija koda programa, raste i mogućnost grešaka koje se mogu javiti u njemu. Alati koji nam pomažu da otkrijemo što veći broj ovakvih grešaka ne garantuju pronalazak svih. KLEE je jedan od alata koji se u praksi pokazao kao jako dobar što se tiče pronalaska grešaka i pokrivenosti koda koji proverava. Glavni cilj za budućnost ovog moćnog alata je da se rutinski pokrije 90% koda koji se proverava, čime bi se pokrili svi zanimljivi testni ulazi. Za dalji razvoj ovog i sličnih alata, potrebno je da se što više programera bolje upozna sa njima i da ih u praksi koriste. Više reči o alatu o kome smo govorili može se pronaći na njihovom sajtu, zajedno sa načinima na koje možemo pomoći njegovom razvijanju.

Literatura

- [1] Dawson Engler Cristian Cadar, Daniel Dunbar. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, 2008. on-line at: <https://lvm.org/pubs/2008-12-OSDI-KLEE.pdf>.
- [2] Koushik Sen Cristian Cadar. Symbolic Execution for Software Testing: Three Decades Later, 2013. on-line at: <https://people.eecs.berkeley.edu/~ksen/papers/cacm13.pdf>.