Modelovanje okoline (Simboličko izvršavanje)

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Nikola Stojevic 1111/2018

10. decembar 2018

Sažetak

Seminarski rad bavi se načinima interakcije simboličkog izvršavanja sa okruženjem.Veoma bitne su simboličke promenljive, koje nemaju konkretnu vrednost, vec mogu uzimati proizvoljne vrednosti. Time mogu pronaći sve putanje izvršavanja programa i vratiti vrednosti koje su dovele do izbora konkretne putanje. Ovo predstavlja način verifikacije softvera, jer upravo ove vrednosti koje su vraćene predstavljaju testove slučaja. Problem je kada se radi sa špoljnim"fukcijama, čije ponašanje ne može da odredi simboličkim promenljivima. Recimo korišćenje Swing-a, Android-a, .NET-a, kao i biblioteka. Način za prevazilaženje prethodnih problema je kreiranje apstraktnog modela koji sadrži oponašanje spoljašnje interakcije.Kod ovog načina postoje određeni neželjeni sporedni efekti ovog pristupa. Bolje je da analizirajući programi intereaguju sa stvarnim okruženjem dok pretražuju različite putanje, nego interakcija sa modelima. Ovo se postiže pravljenjem virtuelne mašine za svako stanje/putanju.

Sadržaj

1 Uvod

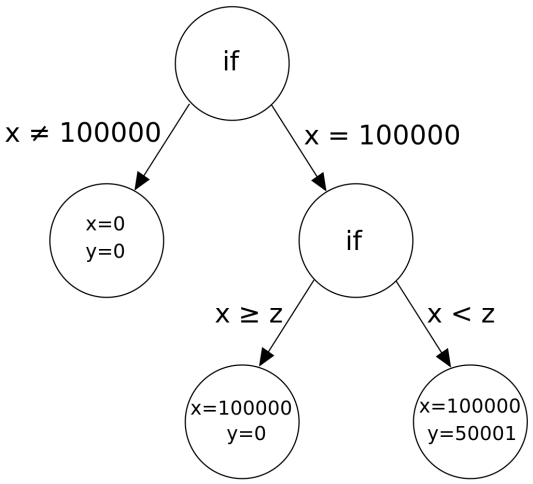
1.1 Simboličko izvršavanje

Simboličko izvršavanje (symbolic execution ili symbolic evaluation) podrazumeva analizu programa u cilju određivanja koji ulazni parametri uzrokuju izvršavanje određenog dela koda. Nisu u pitanju "normalni"ulazni parametri, već apstraktne, simboličke vrednosti. Ta vrednost može biti proizvolja, sa ciljem da omogući prolaz kroz sve putanje izvršavanja programa.

Kada se neka putanja završi može se odrediti konkretna vrednost simboličke vrednost na osnovu ograničenja konrektne putanje. Primer, vrednost zbog koje putanja nije uspela sa izvršavanje. Samim tim ove vrednosti su neke vrste test slučajeva, koje pomažu programerima da reprodukuju ponovno isto ponašanje programa.[3]

Pogledajmo prosti primer ispod:[4]

```
void f(int x, int y) {
    int z = 2*y;
    if (x == 100000) {
        if (x < z) {
            assert(0); /* error */
        }
    }
}
```



Slika 1: Drvo izvršavanja putanja za primer[4]

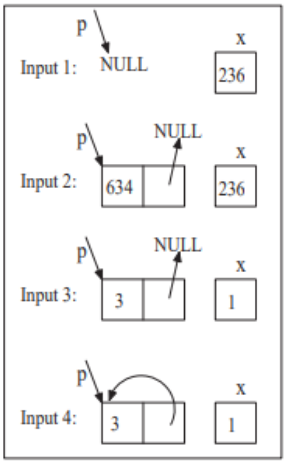
U ovom slučaju simboličke promenljive su x,y. Ograničenja putanje se nalaze u naredbi if, gde kao i kod ostalih grananja izvršavanje se račva na više putanja. Prva putanja je da x ne ispunjavanje uslov x = 100 000. Drugo i treće rešenje ispunjavaju ovaj uslov, s tim da drugo ne ispunjava sledeći uslov x < z , dok treće rešenje ili putanja ispunjava ovaj uslov.

Simboličko izvršavanje svih putanja programa nije izvodljivo kod velikih programa. Neka rešenja su primenjivanje heurstika za traženje putanja, sa ciljem što veće pokrivenosti koda, paralelno izvršavanje nezavisnih putanja ili spajane sličnih putanja.

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;

int
f(int v) {
  return 2*v + 1;
}

int
testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          ERROR;
  return 0;
}
```



Slika 2: Primer simboličkog izvršavanja[1]

2 Interakcija sa okruženjem

Programi često komuniciraju sa svojim okruženjem. Najčešće sa operativnim sistemom preko sistemskih poziva, koriste fajl sistem, promenljive okruženja, mrežu. Problem je kada izvršavanje putanje dođe do komponente koja nije pod kontrolom simboličkog izvršavanja. Na primer, kernel ili razne biblioteke. Te fukcije kontrolisane od sistemskog okruženja se često nazivaju spoljašnje. Takođe moderne aplikacije koriste komponente Android, Swing. Izostanak simboličkog izvršavanja kroz ove komponente, utiče na kompletnost analize programa.[3]

Pogledajmo primer ispod:[3]

```
int main()
{
  FILE *fp = fopen("doc.txt");
  ...
  if (condition) {
    fputs("some data", fp);
  } else {
    fputs("some other data", fp);
  }
  ...
  data = fgets(..., fp);
}
```

Program radi sa fajlovima. Šta piše zavisi od naredbe if, pa bi normalno simboličko izvršavanje videlo da postoje dve putanje za if i svaka putanja bi posebno bila analizirana. Problem je što operacije nad fajlovima su implementirane kao sistemski pozivi ka kernelu operativnog sistema, samim tim su izvan kontrole simboličkog izvršavanja. Postoje načini prevazilaženja tih problema.[3]

2.1 Konkretno i simboličko izvršavanje

Hibridna tehnika verifikacije softvera (Concolic tj. Concrete and symbolic). Jednostava za implementaciju. Koristi simboličke promenljive dok je moguće, a ako nije onda konretne vrednosti (testiranje sa pojedinačnim ulazima). Cilj je pronalaženje grešaka, sa što većom pokrivenost, pre nego dokazivanje ispravnosti programa.[4]

Opis koncepta je predstavljem u člancima:

[2]'DART: Directed Automated Random Testing',autori: Patrice Godefroid, Nils Klarlund,Koushik Sen

[1]'CUTE: A concolic unit testing engine for C',autori: Koushik Sen, Darko Marinov, Gul Agha

Jednostavna implementacija jer nije potrebno implementirati simbolički interpretator za programski jezik, već se oslanja na postojeći program implementiranjem instrumentalizacije, deo koda koji se ubacuje u program radi praćenja stanja i njegove analize. Podaci se čuvaju u fajlu, da bi se kasnije mogao tok izvršavanja ispratiti.

Strukturom stabla se predstavljaju moguće izvršavajuće putanje programa. Koristi se DFS (Depth first search) algoritam za pretragu stabla. U slučaju prevelikog broja putanja i sprečavanja da provede previše vremena na nekom malom delu programa, pretraga može biti ograničene dubine.[2]

Uključuje okruženje sistema u analizu programa, tako što izvršava spoljne pozive koristeći konkretne argumente za pozive. To ograničava ponašanja koje može da se analizira u poređenju sa kompletno simboličkom strategijom.

Pošto ne postoji mehanizam praćenja sporednih efekata spoljašnjih poziva, postoji potencijalni rizik od nekonzistentnosti.[1]

Primer, putanja izvršavanja može da čita iz fajla, dok istovremeno druga putanja pokušava da obriše isti fajl.

Tako u prethodnom primeru koda sa fajlovima ispis fajla bi vratio šome datasome other data"ili šome other datasomedata"u zavisnosti od redosleda izvršavanja.

Zanimljivo vršena su istraživanja gde se pokušava automatsko kreiranje modela, što je jedina održiva opcija za komponente čiji kod nije dostupan. Model se pravi da zadovoljava potrebne funkcionalnosti. [3]

2.2 Modelovanje okruženja

Način za prevazilaženje prethodnih problema je kreiranje apstraktnog modela koji sadrži oponašanje spoljašnje interakcije. Recimo, efekat sistemskih poziva se simulira, i skladište se svi sporedni efekti. Ovo daje dobre rezultate u interakciji simboličkog izvršavanja sa okruženjem.

Mana je što treba implementirati i održavati mnogo potencijalno kompleksnih fukcija standardnih biblioteka. To je skupo i podložno greškama. Modeli se implementiraju na nivou sistemskih poziva, pre nego na nivou biblioteka.

Alati poput KLEE, Cloud9 ,AEG i ostali, koriste ovaj pristup implementirajući modele za operacije nad fajl sistemima, soketima,...

KLEE (simbolička virtuelna mašina igrađena na LLVM kompajler infrastrukturi. EE prestavlja Execution Engine) na primer simboličke fajlove podržava kroz simbolički fajl sistem za svako izvršavajuće stanje. Direktorijum sadrži n fajlova, čije brojeve i veličinu određuje korisnik. Svaka operaciju nad fajlom, rezultuje u račvanju na n+1 granu stanja.

Po jedna za svaki mogući fajl, i dodatno jedna za hvatanje neočekivane greške u operaciji.

AEG (Automatic Exploit Generation) modelira okruženja sistema koji mogu biti korišćeni od strane napadača kao ulazne parametre, fajl, sokete i promenljive okruženja. Modelirano je preko 70 biblioteka. Soketi modelirani slično kao kod KLEE.

CLOUD9 podržava dodatne POSIX biblioteke.[3]

2.3 Račvanje čitavog stanja sistema

Alati simboličkog izvršavanja bazirani na virtuelnim mašinama, rešavaju problem okruženja kreiranjem VM za svako stanje. Ovakav način rešava nas brige pisanje i održavanje kompleksnih modela. Modeli iako su skupi, retko postižu potpunu preciznost i veoma su podložni promenama. Bolje je da analizirajući programi intereaguju sa stvarnim okruženjem dok pretražuju različite putanje, nego interakcija sa modelima. Dozvoljava virtuelno svakom programu da se izvršava simbolički. Svakako mana je mnogo veća potrošnja memorije, zbog kreiranja velikog broja VM.

S2E recimo za svako stanje pravi posebne VM, koje se izvršavaju odvojene. Time sprečava dešavanje sporednih efekata među zavisnim izvršnim putanjama, kada intereaguju sa realnim okruženjem.

QEMU se koristi da emulira virtualizaciju hardvera.[3]

Literatura

[1] Gul Agha Koushik Sen, Darko Marinov. CUTE: A Concolic Unit Testing Engine for C, 2005.

[2] Koushik Sen Patrice Godefroid, Nils Klarlund. DART: Directed Automated Random Testing, 2005.

[3] Daniele Cono D'Elia Camil Demetrescu Irene Finocchi Roberto Baldoni, Emilio Coppa. A Survey of Symbolic Execution Techniques, 2018.

[4] Wikipedia. Concolic testing, 2018.