

# TargetSearch algoritam kao vrsta pretrage alata KLEE

Seminarski rad u okviru kursa  
Verifikacija softvera  
Matematički fakultet

Luka Milošević, Filip Jovašević  
lukamilosevic11@gmail.com, filip.jovasevic96@gmail.com

29. mart 2020.

## Sažetak

Cilj ovog rada je da svim zainteresovanim osobama koji žele da znaju nešto više iz oblasti verifikacije i validacije softvera približi pojam i značenje simboličkog izvršavanja programa. Čitajući rad, čitalac će se upoznati sa alatom KLEE koji služi za simboličko izvršavanje programa. Pored pretraga koje ovaj alat podrazumeva detaljno će biti predstavljena nova pretraga koju smo mi, kao autori ovog rada, implementirali i nazvali TargetSearch.

## Sadržaj

<b>1</b>	<b>Uvod</b>	<b>2</b>
<b>2</b>	<b>Simboličko izvršavanje programa</b>	<b>2</b>
<b>3</b>	<b>KLEE</b>	<b>3</b>
<b>4</b>	<b>TargetSearch</b>	<b>4</b>
4.1	Implementacija . . . . .	5
4.2	Rezultati . . . . .	8
<b>5</b>	<b>Zaključak</b>	<b>10</b>
	<b>Literatura</b>	<b>11</b>

# 1 Uvod

Validacija i verifikacija softvera predstavljaju dve bitne oblasti čija je namena briga o kvalitetu softvera. Validacija proverava da li specifikacija zadovoljava korisničke potrebe, dok verifikacija proverava da li softver zadovoljava specifikaciju. Zbog teme ovog rada, više pažnje ćemo posvetiti verifikaciji.

Osnovne tehnike verifikacije softvera su:

- Dinamička verifikacija → proveravanje ispravnosti koda u fazi izvršavanja, najčešće testiranjem
- Statička verifikacija → proveravanje ispravnosti koda bez njegovog izvršavanja formalnim metodama ili ručnim proverama i pregledima koda

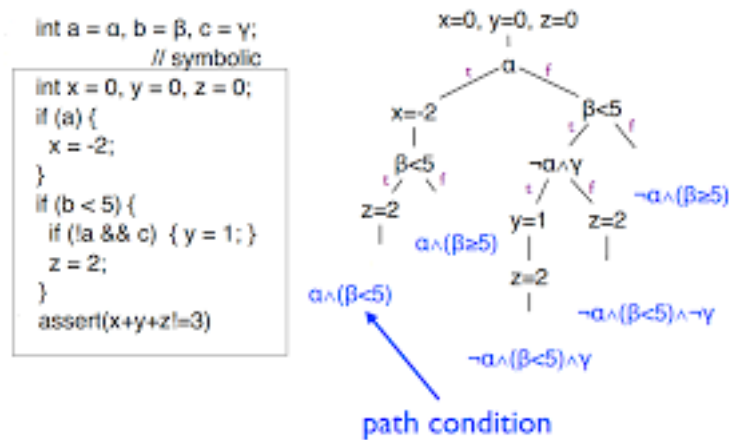
Što se tiče formalnih metoda kao vrste statičke verifikacije, ideja je da se one automatizuju. Ovo jeste moguće ali se ne može garantovati utvrđivanje ispravnosti proizvoljnog programa potpuno precizno [3]. U automatske formalne metode spadaju:

- Simboličko izvršavanje
- Proveravanje modela
- Apstraktna interpretacija

Kroz rad će detaljnije biti opisano simboličko izvršavanje kao i strategija obilaska puteva pod nazivom TargetSearch implementirana u softverskom alatu KLEE. Ova pretraga je implementirana od strane naših kolega sa fakulteta [1], a u ovom radu će biti opisano na koji način smo je mi unapredili.

## 2 Simboličko izvršavanje programa

Simboličko izvršavanje programa predstavlja izvršavanje programa sa simbolima, tačnije praćenje simboličkih stanja umesto konkretnih ulaza. Ovom tehnikom se izvršava mnogo putanja istovremeno i svaka putanja simulira veliki broj testova jer se razmatraju svi ulazi koji prolaze kroz tu putanju. Tehnika je nastala sedamdesetih godina prošlog veka, ali je tek od 2005. godine omogućena praktična primena simboličkog izvršavanja. Alati koji su prvo bili korišćeni su DART I EXE. Neki od alata koji se koriste i danas su: SAGE, CUTE, Cloud9, Kleenet i KLEE o kojem će biti nešto više reči u narednoj sekciji. Simboličko izvršavanje u toku svog rada gradi stablo izvršavanja 1. Ono se sastoji od svih putanji programa. Uslov putanje (path condition) koji je označen na slici predstavlja formulu (bez kvantifikatora) koja sadrži sve odluke koje su do tog trenutka donete [4]. Za obilazak puteva koriste se različite strategije kao što su DFS, BFS, Random Path, izvršavanje vođeno pokrivenošću koda itd. O njima neće biti reči u ovom radu. U sekciji 4 možete naći sve informacije o strategiji obilaska kojoj smo mi posvetili pažnju.



Slika 1: Stablo izvršavanja

### 3 KLEE

KLEE je simbolička virtualna mašina koja je izgrađena koristeći LLVM infrastrukturu kompajlera kao osnovu. Omogućava automatsko generisanje testova koji postižu visoku pokrivenost nad različitim skupom kompleksnih programa [5]. Kao što je pomenuto u prethodnoj sekciji, EXE je bio jedan od prvih alata za simboličko izvršavanje. Baš taj alat koji su napravili Cristian Cadar i Dawson Engler je kasnije preimenovan u KLEE. Zadatak ovog alata je da u kodu pronađe bagove i generiše test primere za njih. Do sada, KLEE je pronašao veliki broj problema u softveru otvorenog koda [2]. Da biste mogli da koristite KLEE, neophodno je prvo da vaš program prevedete u bajtkod. To možete postići sledećom komandom:

Prevođenje programa u bajtkod

```
clang-6.0 -emit-llvm -c -g ime_programa.c
```

Nakon što je program preveden u bajtkod, možete koristiti klee. Jedan jednostavan primer bi bio:

Primer korišćenja KLEE alata

```
klee ime_programa.bc
```

KLEE nudi mogućnost odabira strategije obilaska puteva. U sekciji 4 možete videti primer poziva alata KLEE gde je za strategiju obilaska puteva odabrana strategija target-searcher 4.

## 4 TargetSearch

TargetSearch je algoritam koji predstavlja unapređenje već pomenutog algoritma. Osnovna ideja je da se omogući pretraga na nivou funkcije i da različite funkcije mogu da se obilaze različitim načinom pretrage. Tačnije, neophodno je da korisnik definiše vrstu pretrage, a nakon nje i funkcije koje će biti izvršene tom vrstom. Korisnik je u mogućnosti da definiše više od jedne vrste pretrage, a maksimalno pet i za svaku od njih navede imena funkcija koje želi da izvrši tom vrstom. Mogućnost da se definiše i do pet pretraga predstavlja prvo bitno unapređenje u odnosu na postojeći rad. Vrste pretraga koje se mogu odabrati su:

- DFS search → `-target-dfs` komanda
- BFS search → `-target-bfs` komanda
- Random search → `-target-rs` komanda
- Random-path search → `-target-rps` komanda
- Weighted-random search → `-target-wrs` komanda

Ispod možete videti kako izgleda poziv našeg algoritma na veštačkom primeru.

### Primer korišćenja TargetSearch pretrage

```
klee -search=target-searcher -target-bfs=f1,f2, f3 -target-dfs=f4  
-target-rps=f5, f6 program.bc
```

Nakon komande *search* navodimo vrstu pretrage koju želimo da koristimo. U ovom slučaju smo odabrali našu vrstu pretrage. Da bismo definisali koje funkcije želimo da izvršimo kojom vrstom pretrage, neophodno je da te funkcije navedemo u komandnoj liniji. U ovom konkretnom slučaju, želimo funkcije *f1*, *f2*, *f3* da izvršimo *bfs* pretragom, funkciju *f4* *dfs* pretragom, a funkcije *f5* i *f6* *random – path* pretragom.

Sve ostale funkcije koje se nalaze u kodu koji testiramo će u ovom konkretnom slučaju biti pretražene *bfs* pretragom. Razlog za ovakvo ponašanje je taj što smatramo da ukoliko je najveći broj funkcija koje smo naveli izvršen *bfs* pretragom (*bfs* → tri funkcije, *dfs* → jedna funkcija, *rps* → dve funkcije), velika je verovatnoća da će i preostalim funkcijama najviše pogodovati ta vrsta pretrage.

Važno je i napomenuti da je u slučaju *Weighted – random* pretrage moguće dodati i željenu heuristiku, ali nije neophodno.

Poslednji argument komandne linije je program koji testiramo koji je prethodno preveden u bajtkod.

Kroz sekcije 4.1 i 4.2 biće prikazano na koji način su sve ove mogućnosti postignute kao i upoređivanje rezultata našeg algoritma u odnosu na rezultate korišćenja pretraga koje nudi alat KLEE.

## 4.1 Implementacija

U ovom odeljku će detaljnije biti objašnjeno kako je algoritam konstruisan, to jest koji su fajlovi izmenjeni i koje su funkcionalnosti dodate.

Počecemo od fajla “ExecutionState.h“. Ovde je bilo neophodno definisati kojim sve načinima pretrage može biti izvršeno neko stanje. Zato smo iskoristili tip *enum* prilikom definisanja pet vrsti pretraga koje smo spominjali u sekciji 4.

```
1000 enum searcherType {sDFS,sBFS,sWRS,sRPS,sRandomSearcher,sNotDet};
```

Na primeru se može videti i oznaka *sNotDet* koju do sada nismo spominjali jer nije važna za razumevanje algoritma već služi isključivo implementaciji.

Inicijalizacija svih stanja na podrazumevanu vrstu pretrage je izvršena u fajlu “ExecutionState.cpp“. Podrazumevanom pretragom se smatra ona pretraga koja je od strane korisnika dobila najveći broj funkcija, kao što smo to prethodno naveli.

“Interpreter.h“ fajlu je dodata virtualna funkcija koja za cilj ima da vrati sve funkcije koje je korisnik eksplicitno naveo želeći da ih izvrši navedenim vrstama pretrage.

```
1000 virtual const std::unordered_map<std::string, searcherType>&
      getAllTargetFunctions() = 0;
```

Osim toga, kao argument funkcije *runFunctionAsMain* dodat je tip pretrage da bismo mogli da izvršimo inicijalizaciju koju smo malopre pomenuli.

```
1000 virtual void runFunctionAsMain(llvm::Function *f,
1002                               int argc,
                               char **argv,
                               char **envp, searcherType type, std
::vector<bool>& ifSearchers) = 0;
```

Mogućnost da kao argumente komandne linije zadajemo vrste pretrage je implementirana u fajlu “main.cpp“

```
1000 cl::opt<std::string>
      TargetFunctionDFS("target-dfs",cl::desc("Target Functions for
DFS"),cl::init(""));
```

Definisanje imena pretrage za komandnu liniju

Ovde je implementirana još jedna bitna funkcionalnost, a to je manipulacija funkcijama koje je korisnik naveo. Ona je postignuta funkcijom *processTargetFunction* koja popunjava matricu funkcijama koje je korisnik naveo korišćenjem funkcije *fillTargetFunctions*

```
1000 while(!targetFunction.empty()){
1002     int pos=targetFunction.find_first_of(',');
    std::string s = targetFunction.substr(0,pos);
    m_allTargetFunctions[s] = searcherType;
1004     if(pos == -1)
        return;
1006     targetFunction=targetFunction.substr(pos+1);
}
```

Popunjavanje matrice

a zatim i određuje najučestaliju pretragu.

```

1000     for(int i = sDFS; i < sNotDet; i++){
1001         if(nums[i] > mmax){
1002             mmax = nums[i];
1003             m_maxType = (searcherType)i;
1004         }
1005     }

```

### Određivanje najučestalije pretrage

Modifikovanjem fajla “UserSearcher.cpp“ izvršeno je instanciranje TargetSearch klase. Razlika u odnosu na projekat koji unapređujemo je ta što se konstruktoru klase TargetSearch dodaje argument typeWRS za slučaj da korisnik želi neke funkcije da izvrši WeightedRandom pretragom.

Klasu čiju smo instancu malopre pomenuli smo morali negde i da definišemo. To smo uradili u fajlu “Searcher.h“. Pomenuti argument za WeightedRandom pretragu ne mora da se navede, u tom slučaju podrazumevana vrednost je minDistUncovered.

```

1000 class TargetSearcher : public Searcher{
1001     WeightedRandomSearcher::WeightType m_typeWRS;
1002     Executor &m_executor;
1003
1004     DFSSearcher searcherDFS;
1005     BFSSearcher searcherBFS;
1006     RandomSearcher searcherRandomSearcher;
1007     WeightedRandomSearcher searcherWRS{m_typeWRS};
1008     RandomPathSearcher searcherRPS{m_executor};
1009
1010     std::vector<Searcher*> choose;
1011 public:
1012     TargetSearcher(Executor& executor, std::vector<bool>&
1013         ifSearchers, WeightedRandomSearcher::WeightType typeWRS =
1014         WeightedRandomSearcher::MinDistToUncovered);
1015     ~TargetSearcher();
1016     ExecutionState &selectState();
1017     void update(ExecutionState *current,
1018         const std::vector<ExecutionState*> &addedStates,
1019         const std::vector<ExecutionState*> &removedStates);
1020     bool empty() { return ((searcherDFS?searcherDFS->empty():true)
1021         && (searcherBFS?searcherBFS->empty():
1022             true)
1023         && (searcherRandomSearcher?
1024             searcherRandomSearcher->empty():true)
1025         && (searcherWRS?searcherWRS->empty():
1026             true)
1027         && (searcherRPS?searcherRPS->empty():
1028             true)); }
1029     void printName(llvm::raw_ostream &os) {
1030         os << "TargetSearcher\n";
1031     }
1032 };

```

### TargetSearcher klasa

Drugo bitno unapređenje rada na koji referišemo je implementirano u fajlu “Searcher.cpp“, jer se umesto imitacije pretraga koristi postojeća implementacija pretraga KLEE-a.

```

1000     ExecutionState &TargetSearcher::selectState(){
1002         if(searcherDFS && !searcherDFS->empty()){
1003             return searcherDFS->selectState();
1004         }else if(searcherBFS && !searcherBFS->empty()){
1005             return searcherBFS->selectState();
1006         }else if(searcherWRS && !searcherWRS->empty()){
1007             return searcherWRS->selectState();
1008         }else if(searcherRandomSearcher && !searcherRandomSearcher
->empty()){
1009             return searcherRandomSearcher->selectState();
1010         }else if(searcherRPS && !searcherRPS->empty()){
1011             return searcherRPS->selectState();
1012         }
    }

```

### Odabir sledećeg stanja

Kao i u slučaju odabira sledećeg stanja, tako je i funkcija update pozivana za svaku pretragu posebno i tako je iskorišćena originalna implementacija.

```

1000         if(searcherDFS) searcherDFS->update(current, addedStatesDFS
,removedStatesDFS);
1001         if(searcherBFS) searcherBFS->update(current, addedStatesBFS
,removedStatesBFS);
1002         if(searcherRandomSearcher) searcherRandomSearcher->update(
current, addedStatesRandomSearcher, removedStatesRandomSearcher
);
1003         if(searcherWRS) searcherWRS->update(current, addedStatesWRS
,removedStatesWRS);
1004         if(searcherRPS) searcherRPS->update(current, addedStatesRPS
,removedStatesRPS);

```

### Update funkcija

Ovo unapređenje je implementirano tako što smo umesto čuvanja stanja u jednom vektoru (što je do sada bio slučaj jer je korišćena jedna pretraga) stanja čuvali u mapi sa vrstom pretrage kojom treba da budu obidena. Neophodno je bilo da kao polja klase TargetSearcher dodamo pet željenih pretraga, što se može videti u kodu 4.1.

Implementacija funkcije koja reaguje na svako pojavljivanje funkcije u kodu koja je navedena od strane korisnika napisana je u fajlu "Executor.h". U pitanju je funkcija *ifTargetFunction* i ona ima za cilj da odredi kojoj pretrazi ta funkcija pripada.

```

1000     for(int i = sDFS; i < sNotDet; i++){
1001         for(std::vector<std::string>::iterator it=
allTargetFunctions[i].begin(), ie =
1002         allTargetFunctions[i].end(); it != ie; it++){
1003             switch(i){
1004                 case sDFS:
1005                     if(functionName == *it){
1006                         return sDFS;
1007                     }
1008                     break;
1009                 case sBFS:
1010                     if(functionName == *it){
1011                         return sBFS;
1012                     }
1013                     break;
1014                 case sWRS:
1015                     if(functionName == *it){
1016                         return sWRS;
1017                     }
1018                     break;
1019                 case sRPS:
1020                     if(functionName == *it){
1021                         return sRPS;
1022                     }

```

```

1022         }
1023         break;
1024     case sRandomSearcher:
1025         if(functionName == *it){
1026             return sRandomSearcher;
1027         }
1028         break;
1029     }
1030 }
1031 }
1032 return sNotDet;

```

Provera kojoj pretrazi funkcija treba da pripada

“Executor.cpp“ je fajl u kojem je rešen problem prelaska jednog načina pretrage u drugi nad istim stanjem. Kao što smo već rekli, koristimo pet vektora i neophodno je postarati se da se trenutno stanje nalazi u pravom vektoru. Zahvaljujući prethodno opisanoj funkciji, znamo u koji vektor treba da smestimo trenutno stanje.

```

1000     searcherType method = ifTargetFunction(f);
1001
1002     if(method != sNotDet && method != state.s){
1003         searcher->removeState(&state);
1004         state.s = method;
1005         searcher->addState(&state);
1006     }

```

Prebacivanje stanja u odgovarajući vektor

## 4.2 Rezultati

Nakon što smo videli kako algoritam funkcioniše i na koji način smo to postigli, sada je red došao da ga uporedimo sa postojećim algoritmima KLEE-a. Fajl nad kojim smo vršili testiranje se sastoji od sedam funkcija. Svaka od tih funkcija je specijalno prilagođena određenim strategijama pretrage sa ciljem da se demonstrira moć našeg algoritma. U svakoj funkciji se nalazi greška koju KLEE treba da prepozna.

Prvo ćemo prikazati rezultate primene našeg algoritma pretrage 2. Komanda koju smo iskoristili da bismo pokrenuli našu pretragu za fajl nad kojim vršimo testiranje je sledeća:

Primer poziva TargetSearch algoritma nad test fajlom

```

klee -search=target-searcher -target-dfs=f4,f5,f6 -target-bfs=f7 -
target-rs=f1 -target-rps=f8 -target-wrs=f8 -type-wrs=Depth tar-
getSearcherTest.bc

```

Path	Instrs	Time(s)	ICov(%)	BCov(%)	ICount	TSolver(%)
klee-last	1135371	47.39	91.99	83.33	949	77.14

Slika 2: TargetSearch

TargetSearch strategiju smo primenili i u slučaju kada nismo naveli sve funkcije. Kao što je već pomenuto nenavedene funkcije će biti pretražene



onom strategijom koja u pozivu ima najveći broj funkcija da pretraži. Primer poziva možete videti ispod.

Primer poziva TargetSearch algoritma nad test fajlom

```
klee -search=target-searcher -target-dfs=f4,f5,f6 -target-bfs=f7
targetSearcherTest.bc
```

Rezultate koje smo dobili nakon poziva ove komande možete videti [ovde](#) [3](#)

Vidimo da su dobijeni rezultati u ovom slučaju bolji nego u prethodnom slučaju. Razlog za tako nešto je to što se prilikom poziva random-path i random-state strategija pravi veliki broj stanja koja usporavaju izvršavanje algoritma.

Path	Instrs	Time(s)	ICov(%)	BCov(%)	ICount	TSolver(%)
klee-last	1135371	41.13	91.99	83.33	949	75.24

Slika 3: TargetSearch2

Naredne slike će demonstrirati dobijene rezultate koristeći algoritme koje KLEE omogućava. Za upoređivanje smo koristili dfs [4](#), bfs [5](#), random-state [6](#), random-path [7](#), weighted-random [8](#).

Path	Instrs	Time(s)	ICov(%)	BCov(%)	ICount	TSolver(%)
klee-last	1135371	37.04	91.99	83.33	949	73.39

Slika 4: DFS

Path	Instrs	Time(s)	ICov(%)	BCov(%)	ICount	TSolver(%)
klee-last	1135371	42.46	91.99	83.33	949	74.59

Slika 5: BFS

Path	Instrs	Time(s)	ICov(%)	BCov(%)	ICount	TSolver(%)
klee-last	1135371	54.13	91.99	83.33	949	70.98

Slika 6: Random State

Path	Instrs	Time(s)	ICov(%)	BCov(%)	ICount	TSolver(%)
kleee-last	1135371	50.51	91.99	83.33	949	73.40

Slika 7: Random Path

Path	Instrs	Time(s)	ICov(%)	BCov(%)	ICount	TSolver(%)
kleee-last	1135371	48.81	91.99	83.33	949	70.50

Slika 8: Weighted Random

Kao što možemo videti na ovim slikama, naš algoritam ne daje najbolje rezultate poredeći ga sa ostalim algoritmima pretrage, zbog navedenog problema sa kreiranjem velikog broja stanja. Ipak, temeljnim testiranjem, utvrdili smo da je naš algoritam izuzetno efikasan u traženju grešaka. U takvim situacijama je bolji od svih testiranih algoritama. To su situacije u kojima je prilikom pozivanja TargetSearch algoritma navedena komanda `-exit-on-error`. Takođe, bitno je naglasiti da smo testiranje vršili nad funkcijama sa maksimalno dvanaest simboličkih promenljivih, jer je za veći broj izvršavanje trajalo previše dugo, pa samim tim, ne možemo ništa konkretnije reći o ponašanju našeg algoritma u takvim situacijama.

## 5 Zaključak

Algoritam TargetSearch je naša verzija strategije obilaska putanja. Ideja je bila da se omogući odabir strategije obilaska putanja za svaku funkciju posebno u zavisnosti od toga koja strategija im najviše pogoduje, a da kao posledica toga dobijemo na efikasnosti. Efikasnost je nešto na čemu dalje može da se radi da bismo u svakoj situaciji dobili rezultate koji su bolji u poređenju sa postojećim strategijama, što sada nije slučaj.

## Literatura

- [1] Klee symbolic execution engine. <https://github.com/salesh/klee>.
- [2] Dawson Engler Cristian Cadar, Daniel Dunbar. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. <https://llvm.org/pubs/2008-12-OSDI-KLEE.pdf>, 2008.
- [3] Milena Vujošević Jančić. Briga o kvalitetu softvera. [http://www.verifikacijasoftvera.matf.bg.ac.rs/vs/predavanja/01\\_uvod/02\\_motivacija.pdf](http://www.verifikacijasoftvera.matf.bg.ac.rs/vs/predavanja/01_uvod/02_motivacija.pdf).
- [4] Milena Vujošević Jančić. Simboličko izvršavanje. [http://www.verifikacijasoftvera.matf.bg.ac.rs/vs/predavanja/05\\_simbolicko\\_izvrsavanje/05\\_simbolicko\\_izvrsavanje-prvi\\_deo.pdf](http://www.verifikacijasoftvera.matf.bg.ac.rs/vs/predavanja/05_simbolicko_izvrsavanje/05_simbolicko_izvrsavanje-prvi_deo.pdf).
- [5] The KLEE Team. Klee llvm execution engine. <https://klee.github.io/>.