UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Strahinja Stanojević

PROŠIRIVANJE ALATA KLEE NAPREDNIM ALGORITMOM PRETRAGE STABLA IZVRŠAVANJA PROGRAMA

master rad

Beograd, 2020.

Mentor:

doc. dr Milena Vujošević Janičić, docent
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

doc. dr Vesna Marinković, docent
Univerzitet u Beogradu, Matematički fakultet

prof. dr Saša Malkov, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: 15. januar 2016.

Sadržaj

Glava 1

Uvod

1.1 Softver

U današnje vreme softver je svuda oko nas. Postoji težnja ka tome da se što je moguće više poslova automatizuje. Takođe, teži se tome da se softverski olakšaju mnogi aspekti života (pametni satovi, telefoni, kuće). Softver je sveprisutan i u mnogim granam industrije (proizvodnja, fabrike, automobili, avioni itd.) ali pomaže i u nekim veoma važnim, pa čak i kritičnim zanimanjima (otkrivanje raka i drugih smrtonosnih bolesti u medicini). Zbog ovolike rasporstranjenosti veoma je važno da softver bude kvalitetan i pouzdan. Ukoliko bi postojale neke greške u softveru dešavale bi se mnoge nesreće (automobilske, železničke, avionske itd). Sem fatalnih ishoda i gubitka ljudskih života, greške u softveru dovode i do gubitka ogromne količine novca vrlo često. U delu 1.2 će biti opisane neke poznate softverske greške koje su probudile svest o važnosti ispravnog softvera. Dva važna koncepta u razvoju softvera su njegova validacija (eng. *validation*) i verifikacija (eng. *verification*). Validacija podrazumeva da projektna specifikacija softvera ispunjava korisničke potrebe. S druge strane, ukoliko je projektna specifikacija dobra, odnosno zadovoljava korisničke potrebe, potrebno je da i ona bude ispunjena, odnosno da softver radi baš ono što u istoj piše. Ovo predstavlja verifikaciju softvera. Verifikacija odgovara na pitanja da li softver radi ono što treba, da li ima grešaka, da li je u potpunosti ispravan.

Neke od osobina dobrog softvera su:

Ispravan kod - kod bez sintaksnih grešaka.

Ispravna i čitljiva projektna dokumentacija - dokumentacija koja zadovoljava

korisničke potrebe i iz koje je moguće razumeti zahteve koje softver treba da ispunjava.

Pouzdanost - softver bi trebalo da je pouzdan, da nema problema pri radu, da radi na odgovarajući način bez padova sistema.

Kompletnost - softver treba da ispunjava sve zahteve specifikacije i sve korisničke potrebe. Takođe, ne bi trebalo da ima grešaka.

Lakoća korišćenja - softver treba da bude jednostavan za korišćenje korisnicima. U današnje vreme je korisnicima veoma važna ova stavka. Čak 50% mobilnih aplikacija bude obrisano nakon otkrivanja samo jedne greške od strane korisnika.

Performanse - uz sve prethodno navedene stavke je veoma važno da softver radi efikasno i da ne zauzima previše memorije na uređaju na kome se pokreće. Ove osobine se nazivaju performansama..

1.2 Greške u softveru

Greške u softveru se neretko javljaju. Kako ljudi pišu softver, a ljudi greše, greške su neminovnost. I dok su neke prilično bezazlene i bezopasne, postoji veći broj poznatih softverskih problema i grešaka koji su dovodili do gubitaka ogromne količine novca, kao i do gubitaka ljudskih života.

Ljubavni virus - do današnjeg dana je ostao jedan od najrasprostranjenijih virusa i pokazao probleme koji ni danas nisu rešeni. Virus je prenošen preko mejla. Stizao je mejl sa naslovom „Volim te" u kome je pisalo „Molim Vas pogledajte ljubavno pismo od mene Vama koje se nalazi u prilogu". Otvaranjem priloga (eng. *attachment* bi se otvorila tekstualna datoteka koja je zapravo bila izvršni fajl (eng. *executable*) i isti mejl bi bio prosleđen svim kontaktima osobe koja je otvorila prilog. Na ovaj način se virus širio ekstremnom brzinom. Dodatna šteta koju je pravio je preimnovanje i brisanje ogromne količine datoteka u računaru. Na hiljade fajlova u mnogim računarima je zauvek nestalo. Procenjuje se da je ukupna šteta koju je naneo ovaj virus oko 10 milijardi dolara.

Mt. Gox - Japanski bitkoin (eng. *bitcoin*) koji je nastao 2010 godine je bio najveći bitkoin na svetu. Nakon što je hakovan u junu 2011 gode, Mt. Gox je izgubio

preko 850,000 bitkoina koji su u tom trenutku vredeli oko pola milijarde dolara. Ljudi iz kompanije su priznali da su ogromne gubitke pretrpeli zato što su imali grešku u bezbednosti sistema.

Ariane 5 - četvrtog juna 1996. godine raketa Arijana 5 (eng. *Ariane 5*) je eksplodirala pri prvom lansiranju nakon samo 40 sekundi. Pravljenje softvera za raketu i same rakete je trajalo preko jedne decenije i koštalo je oko sedam milijardi dolara. Cena same rakete je bila oko 500 miliona dolara. Uzrok ove nezgode je bio greška u softveru. Naime, 64-bitni broj u pokretnom zarezu je zapisan u promenljivu koja je bila označeni ceo broj, koja može da čuva broj 32,767 kao najveći. Broj koji je trebalo da bude zapisan je bio veći od ove vrednosti i došli je do greške pri konverziji.

Therac 25 - aparat za tretiranje raka radijacijom. Tokom prvih nekoliko godina od početka rada ovog uređaja 1983. godine nije bilo nikakvih problema. Međutim 1985. i 1986. godine se dogodilo ukupno 5 smrtnih slučajeva i jedan slučaj invaliditeta. Jedna od pacijenatkinja kojoj je bilo prepisano zračenje jačine 200 rada je bila ozračena jačinom između 10,000 i 20,000 rada.

1.3 Tehnike verifikacije softvera

Postoje dva osnovna pristupa verifikaciji softvera. Statička i dinamička analiza, odnosno verifikacija. Osnovna razlika između ove dve tehnike je u tome što se dinamička verifikacija vrši tokom izvršavanja programa, odnosno neophodno je pokrenuti isti, dok dinamička verifikacija analizira kôd i proverava ispravnost softvera bez njegovog pokretanja.

Dinamička verifikacija softvera

Dinamička verifikacija softvera se odnosi na proveru ispravnosti softvera u fazi izvršavanja. Dakle, neophodno je kompajlirati kôd, pokrenuti ga, i onda vršiti analizu. Dinamička verifikacija softvera se uglavnom sprovodi testiranjem. Testove uglavnom pišu i sprovode testeri, međutim u nekim situacijama to rade i programeri. Tester mora da ima znanje programiranja kako bi pisao kvalitetne testove, da poznaje specifičnosti jezika, da razume softver koji se razvija, njegove odlike, korisničke potrebe itd. Postoji više različitih vrsta i tehnika testiranja.

Statička verifikacija softvera

Statička verifikacija softvera se odnosi na proveru ispravnosti koda bez njegovog pokretanja. Kôd nije potrebno kompajlirati niti pokretati kako bi bile proverene eventualne greške u njemu. Postoje 2 načina statičke analize softvera.

Pregledi koda - tehnika statičke verifikacije softvera gde programeri pregledaju napisani kôd pre nego što on može da bude dodat u glavni repozitorijum, odnosno pre nego što bude priključen softveru koji će biti isporučen korisniku. Pregledima koda ne mogu biti otkrivene sve greške u kodu, ali postoje studije koje su pokazale da se broj grešaka značajno smanjuje ukoliko se koristi ova tehnika. Pregledi koda mogu da budu formalni, u vidu grupnih sastanaka gde se uglavnom čita i diskutuje napisani kôd. Takođe, to mogu biti i neke neformalne metode kao što su programiranje u paru, pregledi preko mejla, prosto objašnjavanje šta je i zašto urađeno u kodu.

Automatska statička analiza koda - tehnika statičke verifikacije softvera u kojoj postoje specijalizovani alati koji pomažu pri proveri ispravnosti koda. Postoji više tehnika koje se koriste u statičkoj analizi koda.

Apstraktna interpretacija - osnovna ideja ove tehnike je da se apstrahuje konkretna semantika datog programa, jer je semnatika sama po sebi previše kompleksna da bi se o njoj moglo rezonovati.

Simboličko izvršavanje - tehnika u kojoj je ideja da promenljive u kodu mogu da budu konkretne (one kojima je poznata unapred vrednost) i simboličke (one kojima se vrednost dodeljuje na osnovu nekog izračunavanja, učitava se iz fajla, dobija kao rezultat nekog upita itd). Simboličko izvršavanje će detaljnije biti opisano u delu 2.

Proveravanje modela - osnovna ideja proveravanja modela je da se sistemski ispitaju sve moguće putanje u izvršavanju nekog sistema (softvera). Na ovaj način se ispituje da li sistem ispunjava neku invarijantu ili neko svojstvo koje treba da bude ispoljeno. Potrebno je precizno i formalno opisati model sistema, kao i svojstva koja se proveravaju kako bi bilo moguće primeniti proveravanje modela na odgovarajući način. Uglavnom se oslanja na matematičku logiku, teoriju grafova i teoriju formalnih jezika.

Glava 2

Simboličko izvršavanje

U ovom poglavlju će biti opisani opšti principi na kojima se temelji simboličko izvršavanje u okviru mnogo šire oblasti računarstva, verfikacije softvera. U okviru verifikacije softvera postoje dve velike oblasti, *statička verifikacija softvera* i *dinamička verifikacija softvera*. Statička verifikacija softvera se odnosi na tehnike verifikacije softvera kod kojih je moguće otkrivanje grešaka i rizičnog ponašanja napisanog programskog koda bez njegove prethodne kompilacije i pokretanja. Kod dinamičke verifikacije je nepohodno kompajlirati kod, pokrenuti ga, i tek je onda moguće otkriti greške i kritične delove. Simboličko izvršavanje je jedna od tehnika statičke verifikacije softvera, odnosno bavi se statičkom analizom koda. Jedna je od najkorišćenijih tehnika statičke verifikacije. Na primer, još od 2008. nad većinom Majkrosoftovih (eng. *Microsoft*) aplikacija se neprekidno vrši verifikacija pomoću simboličkog izvršavanja, pre nego što aplikacije budu objavljene [7]. Dodatno, skoro 30% grešaka u operativnom sistemu Windows 7 je pronađeno baš korišćenjem ove tehnike. Ova činjenica se odnosi na greške koje nisu bile otkrivene korišćenjem drugih tehnika za statičku i dinamičku analizu softvera.

2.1 Osnovni principi simboličkog izvršavanja

Svaki program može biti izvršavan konkretno ili simbolički. Konkretno izvršavanje podrazumeva da su vrednosti svih promenljivih u kodu poznate i onda se prati kojim putem se prolazi izvršavanjem (zavisno od uslova grananja koji su ispunjeni). S druge strane, kôd je moguće simbolički izvršavati čak i ako nisu poznate početne vrednosti za sve promenljive. Na primer, ukoliko u kodu postoji promenljiva a tipa *int* kojoj na početku nije poznata vrednost (vrednost joj se dodeljuje čitanjem sa

standardnog ulaza ili iz datoteke npr) ona se naziva simboličkom promenljivom. Ova promenljiva može da uzme bilo koju vrednost iz opsega vrednosti koje mogu biti čuvane u promenjivoj konkretnog tipa (u ovom slučaju je to celobrojni tip, odnosno *int*).

Osnovna ideja simboličkog izvršavnja je da se vrši simboličko, a ne konkretno izvršavanje. Za svaku od otkrivenih putanja tokom izvršavanja koda se čuvaju sledeći podaci: formula logike prvog reda koja predstavlja uslove koji su bili zadovoljeni u grananjima kroz koja se prošlo i koji su doveli do konkretne putanje i simbolički memorijski prostor u kome se čuvaju vrednosti simboličkih promenljivih. Grananja dovode do ažuriranja formule, dok dodeljivanje vrednosti simboličkim promenljivima dovodi do ažuriranja simboličkog memorijskog prostora. U primeru 2.1, promenljive a i b se nazivaju *simboličke* i one mogu uzeti bilo koju vrednost iz opsega vrednosti koje mogu biti upisane u promenljivu tipa *int*.

Nakon određivanja neke od putanja vrši se provera zadovoljivosti logičke formule koja odgovara putanji. Za ove potrebe se obično koristi SMT rešavač [5]. Svakoj putanji kroz koju se može proći izvršavanjem postoji odgovarajuća logička formula. Za svaku od tih formula SMT rešavač pokušava da nađe valuaciju u kojoj je formula nezadovoljiva. Ukoliko postoji takva valuacija, to znači da je putanja kritična, odnosno da postoje vrednosti ulaznih promenljivih koje mogu dovesti do neke greške. Alat za simboličko izvršavanje onda može da pruži informaciju o tome koje vrednosti ulaznih podataka mogu dovesti do grešaka u kodu.

Kod simboličkog postoje *stanja* koja nastaju simboličkim izvršavanjem programa. Sva stanja su oblika $(stmt, \sigma, \pi)$, gde važi sledeće:

- *stmt* je naredba koja se izvršava u datom stanju, odnosno sledeća naredba na koju se naišlo u kodu.

- σ predstavlja simbolički memorijski prostor u koji se smeštaju vrednosti koje su dodeljene simboličkim promenljivima. Primećujemo da se ove vrednosti dok nisu poznate označavaju sa α_i. Iste oznake se koriste i za druge podatke koji nisu unapred poznati (nije ih moguće odrediti statičkom analizom koda) kao što su rezultzati izvršavanja sistemskih poziva, čitanje iz tokova (eng. *streams*) i slično.

- π je formula logike prvog reda koja predstavlja skup uslova koji važe u trenutnom stanju. Ovi uslovi se nadograđuju prolaskom kroz različite naredbe

grananja u kodu. Na početku, dok još ne postoji nijedno ograničenje koje treba da važi, odnosno dok se ne naiđe na neko grananje, važi $\pi = true$.

Zavisno od naredbe *stmt* alat za simboličko izvršavanje radi sledeće:

- ako je u pitanju naredba dodele `x = p` simbolički memorijski prostor σ se menja tako što se promenljivoj x u tom prostoru dodeljuje vrednost p_s gde p_s predstavlja vrednost izraza sa desne strane operatora dodele. To može biti konkretna vrednost, ili vrednost nekog kompleksnijeg izraza koja se evaluira u konkretnom stanju zavisno od uslova koji važe. Dodelu označavamo sa $x \rightarrow p_s$.
- ako je *stmt* naredba grananja `if` e `then` s_{true} `else` s_{false}, ažurira se formula π. Stanje u kome je došlo do grananja se deli na dva druga stanja, jedno u kome se dodaje uslov s_{true} i drugo u kome se dodaje uslov s_{false}. Formule koje odgovaraju ovim stanjima su $\pi \wedge e_s$ i $\pi \wedge \neg e_s$ redom, gde važi da je e_s simbolički izraz koji se dobija evaluacijom izraza e.

2.2 Izazovi simboličkog izvršavanja

Ideja simboličkog izvršavanja je da se detaljnom analizom svih putanja kroz koje se može proći u kodu pronađu ulazne vrednosti koje bi mogle dovesti do grešaka. Teorijski gledano ovo je odličan koncept koji garantuje *saglasnost* (eng. *soundness*) i *kompletnost*. Saglasnost garantuje da neće biti lažno pozitivnih rezlutata. Lažno pozitivni rezultati predstavljaju pojavu gde se neki ulazni podaci prograšavaju kritičnim tj. kaže se da mogu dovesti do greške, a to nije tačno, odnosno sa tim ulaznim podacima ne može doći do prijavljene greške. Kompletnost garantuje da neće biti lažno negativnih rezultata. To znači da se neće desiti da za neke ulazne podatke koji mogu dovesti do greške alat kaže da su validni. U praksi, vrlo često ovo nije moguće postići jer se obično radi sa kompleksnim softverom, tako da se obično zarad performansi žrtvuje saglasnost. Glavni problem predstavljaju petlje i rekurzija. Ukoliko broj iteracija petlje zavisi od neke simboličke promenljive ili nekog spoljnog faktora nije moguće unapred znati koliko puta će se petlja izvršavati. Zbog ovoga se gubi na preciznosti. Neki od glavnih izazova sa kojima se suočava simboličko izvršavanje su:

Eksplozija stanja: na koji način se rešava problem eksplozije stanja? Petlje, rekurzija ili ugnježdene naredbe grananja unutar petlji dovode do ogromnog

broja stanja. Nije realno očekivati da alat za simboličko izvršavanje može sve ove putanje da istraži u razumnom vremenu.

Memorija: na koji način alati za simboličko izvršavanje rade sa pokazivačima, nizovima, višedimenzionim nizovima, strukturama, klasama i drugim kompleksnim objektima.

Rešavanje ograničenja: SMT rešavaču može biti potrebno mnogo vremena da reši neka kompleksna ograničenja (formule logike prvog reda). Pitanje je na koji način smanjiti broj poziva rešavaču kako bi i samo izvršavanje bilo što je moguće efikasnije.

Način rešavanja navedenih problema u alatu KLEE će biti razmotren u poglavlju 3.

2.3 Osnovni primer simboličkog izvršavanja

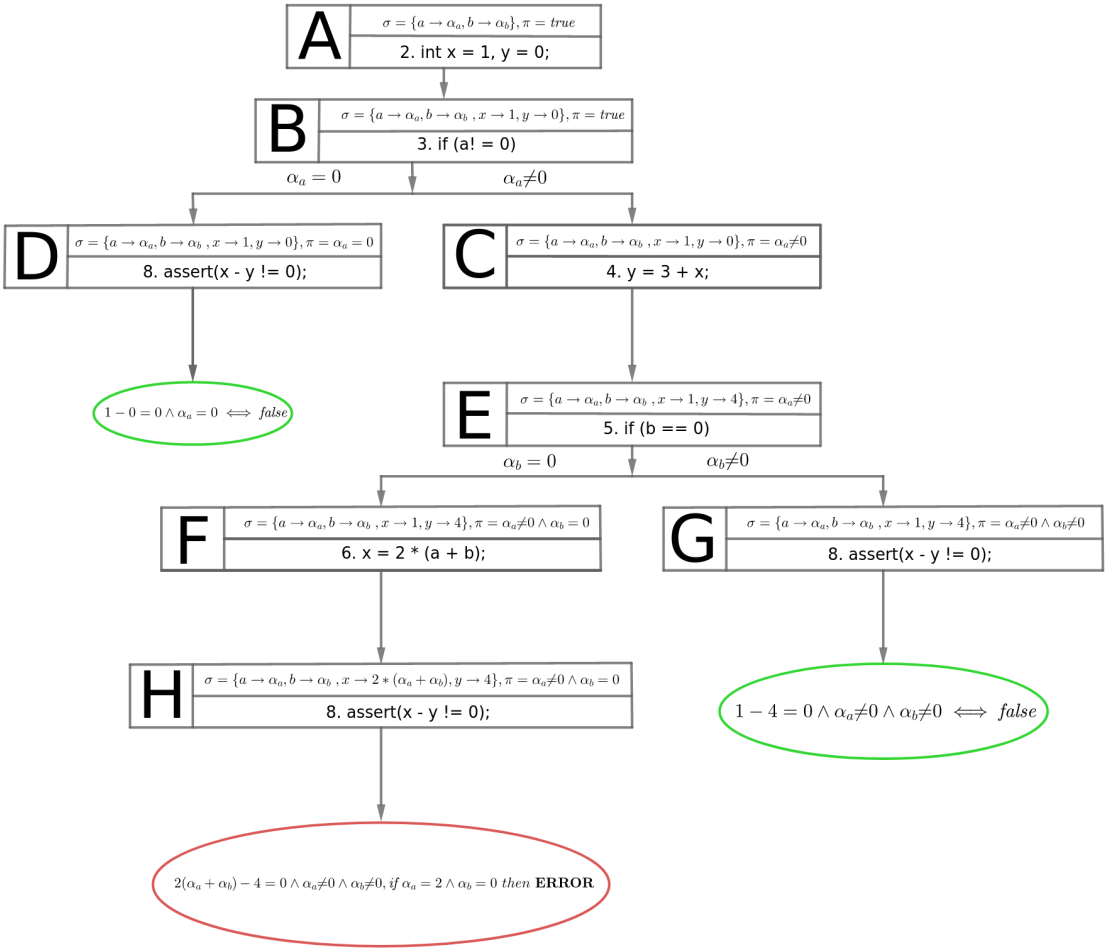
Na primeru jedne jednostavne funkcije bice objasnjeno na koji način simboličko izvršavanje funkcioniše.

```
1. void foo(int a, int b) {
2.   int x = 1, y = 0;
3.   if (a != 0) {
4.     y = x + 3;
5.     if (b == 0)
6.       x = 2 * (a + b);
7.   }
8.   assert(x - y != 0);
9. }
```

Listing 2.1: Osnovni primer simboličkog izvršavanja

Razmotrimo primer 2.1. Postavlja se pitanje u kojim situacijama može doći do narušavanja ograničenja koje je zadato naredbom *assert*(). Konkretno izvršavanje programa radi isključivo sa konkretnim (unapred određenim) vrednostima svih promenljivih. U datom primeru znamo vrednosti za promenljive x i y i one se nazivaju konkretnim promenljivima. Međutim, ukoliko se prethodni kôd izvršava konkretno, za promenljive a i b je neophodno odrediti vrednosti pre izvršavanja. Kako promenljive ovog tipa mogu uzeti samo neku od unapred definisanih vrednosti (promenljive su tipa *int*, i mogu da čuvaju samo opseg vrednosti koje mogu biti smeštene u promenljive ovog tipa), male su šanse da će se konkretnim izvršavanjem, za promenljive

kojima nisu dodeljene vrednosti u kodu, generisati baš one vrednosti koje bi dovele do greške. Za razliku od konkretnog izvršavanja, simboličko izvršavanje može odjednom da istražuje veći broj putanja u kodu kroz koje se može proći zavisno od ulaznih podataka (u prethodnom primeru, ulazne vrednosti su vrednosti promenljivih a i b). Na slici 2.1 su prikazana stanja koja odgovaraju simboličkom izvršavanju koda prikazanog u primeru 2.1.



Slika 2.1: Stablo stanja simboličkog izvršavanja koje odgovara primeru. Crvenom elipsom je predstavljeno završno stanje kod koga može doći do greške, dok su zelenim elipsama označena završna stanja kod kojih nema grešaka.

Graf simboličkog izvršavanja može biti predstavljen stablom, što se može primetiti na slici 2.1. Incijalno u korenu stabla (stanje A) imamo formulu π koja ima vrednost **tačno** (eng. *true*), i promenljivima a i b su dodeljene simboličke vrednosti α_a i α_b. Stanje B odgovara prvoj naredbi unutar funkcije foo u kojoj se promenljivi-

ma x i y dodeljuju vrednosti 1 i 0 redom, čime one postaju konkretne promenljive. U stanju B se memorijski prostor σ ažurira tako što se menjaju vrednosti promenljivih x i y. Sledeća naredba je naredba grananja pa se stanje B deli na dva stanja, C i D. Stanja C i D odgovaraju uslovima $\alpha_a \neq 0$ i $\alpha_a = 0$ redom, što se može i videti na osnovu ažuriranja formule π u odgovarajućim stanjima. Daljom analizom se može zaključiti da su stanja D, G i H završna stanja, tj. stanja u kojima se izvršava poslednja naredba funkcije foo. Jedina formula koja može dovesti do greške je ona u stanju H. Bilo koje ulazne vrednosti za simboličke promenljive a i b za koje važi

$$2(\alpha_a + \alpha_b) - 4 = 0 \wedge \alpha_a \neq 0 \wedge \alpha_b = 0$$

će dovesti do narušavanja uslova naredbe *assert*. Vrednosti koje bi dovele do greške se mogu odrediti pozivanjem SMT rešavača, u datom primeru to su vrednosti `a = 2` i `b = 0`.

2.4 Osnovni principi rada alata za simboličko izvršavanje

Teorijska ideja simboličkog izvršavanja je da se istraže sve moguće putanje u okviru nekog koda (softvera) i da se odrede sve kombinacije ulaznih podataka koje bi dovele do grešaka. U praksi to nije moguće zbog ranije navedenih problema, pa samim tim alati za simboličko izvršavanje ne pretražuju sve moguće putanje. Jedan od dodatnih problema pored ranije navedenih zbog kog je nemoguće pronaći sve moguće putanje je taj što u realnom softveru postoji kôd iz eksternih biblioteka kojima sam alat nema pristup, samim tim ne može da generiše sve putanje kroz taj deo koda. Sporedni efekti prilikom izvršavanja koda (kako programski koji se tiču simboličkih promenljivih tako i neki drugi koji se tiču arhitekture računara) mogu predstavljati problem pri rekonstrukciji kompletnog steka softvera koji se analizira. Osnovna ideja za rešavanje ovih problema je kombinovanje konkretnog i simboličkog izvršavanja, takozvano *konkoličko izvršavanje* (eng. *concolic*), što je kovanica reči *concrete* i *symbolic*.

Dinamičko simboličko izvršavanje

Jedna od popularnih metoda konkoličnog simboličkog izvršavanje je dinamičko simboličko izvršavanje (eng. *Dynamic Symbolic Execution* ili DSE). Ideja iza ovog metoda je da simboličko izvršavanje bude vođeno konkretnim izvršavanjem. Pored simboličkog memorijskog prostora i formula koje predstavljaju ograničenja na različitim putanjama, alat za simboličko izvršavanje čuva i konkretan memorijski prostor σ_c. Na početku izvršavanja je potrebno odrediti konkretne vrednosti za ulazne podatke, a zatim se sam kôd izvršava i konkretno i simbolički, paralelno ažurirajući oba memorijska prostora i odgovarajuća ograničenja koja odgovaraju putanjama. Kada se naiđe na naredbu grananja, na osnovu konkretnih vrednosti promenljivih se određuje kojom granom se nastavlja izvršavanje. Granu koja se bira konkretnim izvršavanjem bira i simboličko izvršavanje i na odgovarajući način se ažurira formula π koja odgovara putanji. Dakle, simboličko izvršavanje prati konkretno izvršavanje. Ovakav način izvršavanja omogućava da SMT rešavač ne mora da se poziva nakon svakog grananja kako bi se proverilo da li je formula koja se dobija dodavanjem izraza koji odgovara izabranoj putanji nezadovoljiva, jer se to rešava konkretnim izvršavanjem, s obzirom da su konkretne vrednosti promenljivih poznate. Kako bi se istraživale različite putanje, moguće je negirati uslov nekog od grananja i pozvati SMT rešavač kako bi odredio koje su vrednosti ulaznih podataka koje bi dovele do kretanja baš tom putanjom, odnosno koje početne vrednosti ulaznih podataka odgovaraju negiranom uslovu u naredbi grananja.

Razmotrimo primer 2.1. Neka su ulazne vrednosti `a = 1` i `b = 1`. Izvršavanje bi krenulo od stanja A, zatim se prelazi u stanje B gde postoji grananje. Kako je uslov `a != 0` ispunjen, izvršavanje bi nastavilo stanjem C, nakon čega se nastavlja stanjem E u kome opet imamo grananje. S obzirom uslov $b == 0$ nije ispunjen (jer imamo konkretu vrednost promenljive b koja je 1, izvršavanje ulazi u završno stanje G. Konkretan memorijski prostor se kroz ovo izvršavanje menja na sledeći način:

- $\sigma_c = a \to 1, a \to 1$ u stanju A,
- $\sigma_c = a \to 1, a \to 1, x \to 1, y \to 4$ u stanjima B i C i
- $\sigma_c = a \to 1, a \to 1, x \to 1, y \to 0$ u stanjima E i G.

U stanju G bi se ustanovilo da je sve u redu, tj. nema narušavanja ograničenja koje je zadato naredbom *assert*, pa se nova putanja može generisati negacijom nekog od uslova koji su bili ispunjeni u naredbama grananja. Recimo da se negira uslov

$\alpha_b \neq 0$. SMT rešavač će generisati nove ulazne podatke koji bi zadovoljavali ovaj uslov, recimo `a = 1`, `b = 0` i na taj način bi bila otkrivena nova putanja. U ovom slučaju bi izvršavanje išlo kroz stanja $A \rightarrow B \rightarrow C \rightarrow E \rightarrow F$.

Iako dinamičko simboličko izvršavanje počinje dodelom konkretnih vrednosti promenljivima, nove putanje se generišu negiranjem određenih uslova. Potrebno je odabrati uslov koje naredbe grananja će biti negiran. Broj naredbi grananja (samim tim i broj različitih putanja) u nekom kodu može biti jako veliki, s toga je neophodno na neki način vršiti odabir uslova koji će biti negirani jer je nemoguće u razumnom vremenu proći kroz sve putanje. Načini na koje različiti alati koji vrše dinamičko simboličko izvršavanje biraju koji će uslov biti negiran će biti diskutovan dalje u tekstu.

U nekim situacijama alat za simboličko izvršavanje koji radi na principu dinamičkog simboličkog izvršavanja ne može da isprati ceo kôd simbolički. Razlog za to je što se u kodu može desiti da se pozivaju funkcije koje su deo neke eksterne biblioteke čiji izvorni kôd nije dostupan. Razmotrimo primer 2.2:

```
void foo(int a, int b)
{
  int x = bar(a);

  if (b > 5)
    ERROR;
}
```

Listing 2.2: Primer gde rezultat izvršavanja eksterne funkcije nije važan

Alat bi u ovom slučaju bi promenljivima a i b dodelio neke slučajne vrednosti na početku izvršavanja funkcije *foo*. Pretpostavimo da važi `a = 1` i `b = 0`. Sva grananja i sve naredbe koje se izvršavaju u okviru funkcije *bar* su potpuno nepoznate alatu. Kako bi se otkrila neka nova, potencijalno zanimljiva putanja, potrebno je negirati neki od uslova u grananjima kroz koja se prolazi tokom izvršavanja koda. Jedino grananje koje je poznato je `if (b > 5)` (izvorni kôd funckije *bar* nije dostupan), tako da je moguće negirati uslov baš tog grananja. Na taj način bi se pomoću SMT rešavača dobili novi ulazni podaci za koje mora da važi `b > 5`, recimo `a = 1` i `b = 7`. Na ovaj način bi bila otkrivena nova putanja, i otkrilo bi se da može doći do greške u izvršavanju navedene funkcije.

```
void foo(int a)
{
  int x = bar(a);

  if (x > 0)
    ERROR;
}
```

Listing 2.3: Primer gde je rezultat izvršavanja eksterne funkcije važan

Posmatrajmo sada primer 2.3. Pretpostavimo da izvorni kôd funkcije *bar* nije poznat ni u ovom slučaju. Ono što se može primetiti je da uslov grananja koje je poznato direktno zavisi od rezultata njenog izvršavanja. Recimo da je alat za simboličko izvršavanje generisao ulazne podatke (vrednost za simboličku promenljivu a) tako da uslov poznate naredbe grananja nije poznat. SMT rešavač bi pokušao da negira uslov i generiše nove ulazne podatke koji bi trebalo da omoguće otkrivanje nove putanje. Međutim, kako izvorni kôd funkcije *bar* nije poznat a uslov grananja direktno zavisi od rezultata njenog izvršavanja nije moguće garantovati da će generisanjem novih ulaznih vrednosti biti otkrivena putanja koja može dovesti do greške. U nekim situacijama je alatu nemoguće da otkrije da ne postoje ulazni podaci koji mogu otkriti novu putanju, tj. da postoji nedostižan kod. Taj slučaj je ilustrovan primerom
2.4:

```
void foo(int a)
{
  double x = pow(a, 2);

  if (x < 0)
    ERROR;
}
```

Listing 2.4: Primer u kome može doći do divergencije putanje

U ovom primeru imamo slučaj gde funkcija foo poziva funkciju *pow* koja vraća kvadrat svog prvog argumenta argumenta. Neka je alat generisao $a = 5$ kao ulazni podatak. Kako uslov u naredbi grananja nije bio ispunjen, rešavač će otkriti da je potrebno negirati uslov $a \geq 0$. Novi ulazni podatak bi mogao da bude $a = -5$. Međutim, ni na ovaj način se neće desiti da je uslov u naredbi grananja ispunjen.

Alat nema mogućnost da otkrije da je nemoguće da dođe do greške, tj. da je naredba u kojoj dolazi do greške nedostižna.

Na osnovu prethodnih primera možemo videti da postoje problemi u dinamičkom simboličkom izvršavanju. Neotkrivanje nekih interesantnih putanja, nemogućnost otkrivanja nedostižnog koda. Pozivi eksternih funkcija, kastovanje i simboličkli pokazivači su neki od najvažnijih aspekata o kojima se mora voditi računa prilikom dinamičkog simboličkog izvršavanja kako ne bi došlo do divergencije putanja.[1]

Selektivno simboličko izvršavanje

Postoji i drugačiji pristup konkoličkom simboličkom izvršavanju koji vrši alat za selektivno simboličko izvršavanje (eng. *Selective Symbolic Execution* ili S^2E). Kod ovog alata simboličko i konkretno izvršavanje se kombinuju na drugačiji način u odnosu na dinamičko simboličko izvršavanje. Neka postoje dve funkcije A i B pri čemu funkcija A poziva funkciju B. Postoje dva osnovna pristupa:

1. **od konkretnog ka simboličkom i nazad** - funkcija A se izvršava konkretno. Kada se dođe do narebe u kojoj se poziva funkcija B, argumenti funkcije se prave simboličkim kako bi se cela funkcija simbolički izvršila i otkrile eventualne greške u istoj. Funkcija B se takođe izvršava i konkretno, a zatim se rezultat konkretnog izvršavanja vraća funkciji A kako bi ona nastavila da se izvršava konkretno.

2. **od simboličkog ka konkretnom i nazad** - funkcija A se izvršava simbolički. Kada se dođe do narebe u kojoj se poziva funkcija B, argumenti funkcije se konkretizuju i funkcija se u potpunosti izvršava konkretno. Nakon izvršavanja funkcije B, izvršavanje se vraća na funkciju A koja se nastavlja simbolički.

Ovakav pristup utiče i na saglasnost i na kompletnost rezultata.

kompletnost - potrebno je na neki način izbeći lažno pozitivne rezultate. Kada se funkcija B izvršava simbolički potrebno je voditi računa kroz koje je grane moguće porći zavisno od konkretizacije argumenata, jer izvršavanje funkcije B direktno zavisi od vrednosti promenljivih u funkciji A. Kod selektivnog simboličkog izvršavanja se pri kreiranju formule određene putanje u simboličkom

[1] Divergencija putanja je pojava u dinamičkom simboličkom izvršavanju gde izvršavanje vodi nekom neočekivanom putanjom. Recimo u primeru 2.4 bi bilo očekivano da negacijom uslova grananja bude otkrivena nova putanja, međutim to se nikada neće desiti jer je kvadrat bilo kog broja uvek pozitivan broj.

izvršavanju (π) vodi računa o načinu na koji su argumenti konkretizovani, koje vrednosti mogu da uzmu, koji su to sporedni efekti koji mogu da se dese u funkciji B i koja će biti povratna vrednost funkcije za konkretne vrednosti argumenata.

saglasnost - takođe može da dođe do lažno negativnih rezultata što je potrebno izbeći. S obzirom da se argumenti funkcije B konkretizuju nakon povratka u funkciju A je moguće da se kroz neka grananja ne prolazi. Kako bi se ovaj problem rešio, funkcija B se konkretno izvršava veći broj puta, pri čemu se za svako od izvršavanja vodi računa kojom putanjom se prošlo kroz funkciju B, i nove konkretne ulazne vrednosti parametara se biraju tako da se kroz B u svakom sledećem izvršavanju ide kroz neke nove putanje.

Simboličko izvršavanje unazad

Još jedan vid simboličkog izvršavanja je simboličko izvršavanje unazad (eng Symbolic Backward Execution ili SBE). Osnovna ideja ovog pristupa je da se krene od neke naredbe u kodu i da se izvršavanje kreće ka ulaznu, tj. početku rada programa. Ova tehnika se obično koristi kada je potrebno odrediti pod kojim uslovima se može doći do neke konkretne naredbe u kodu, i kojom se putanjom do nje stiže. Osnovna mana ovog pristupa je što je neophodno da postoji graf (stablo) izršavanja samog programa, a konstrukcija tog grafa je često izuzetno skupa i komplikovana operacija.

2.5 Algoritmi simboličkog izvršavanja

Postoji veliki broj različitih algoritama i heuristika koje koriste različiti alati za simboličko izvršavanje. U ovom delu će biti dat njihov kratak prikaz.

DFS

Jedan od osnovnih i najpoznatijih algoritama koji se koriste u simboličkom izvršavanju je pretraga grafa u dubinu (eng. *Depth-First Search* ili DFS). Kako simboličko izvršavanje koda možemo predstaviti kao stablo stanja kroz koja se prolazi, a s obzirom da je stablo zapravo aciklički graf, možemo koristiti algoritme za pretragu grafova kako bismo birali sledeće stanje koje će biti izvršavano. Algoritam DFS bira

putanju kojom krece inicijalno, prilikom prvog grananja, zatim tu putanju istražuje do kraja. Ide maksimalno u dubinu sve dok se ne dođe do kraja putanje ili dok ne bude ispunjen neki drugi uslov zasutavljanja (dubina rekurzije, broj naredbi koje su izvršene na putanji, vreme koje je provedeno istražujući odrešenu putanju itd). Nakon toga algoritam vrši odsecanje te putanje, vraća se unazad do nekog prethodnog grananja i nastavlja drugim putem. Zatim se ta nova putanja istražuje maksimalno (do kraja, ili nekog drugog uslova izlaska). Algoritam pretrage grafa u dubinu se obično koristi kada je memorija kojom se raspolaže ogranična (ne dovoljno velika za čuvanje velikog broja stanja). Najveća mana ovog pristupa je što jako dugo radi (vrlo često ne može da završi u razumnom vremenu) kada u kodu postoji rekurzivna funkcija koja se izvršava ili ukoliko postoje petlje čije ograničenje nisu konkretne vrednosti već simboličke promenljive.

BFS

Još jedan često korišćeni grafovski algoritam za pretragu stabla stanja u simboličkom izvršavanju je pretraga grafa u širinu (eng. *Breadth-First Search* ili BFS). Pretraga u širinu ima prednosti u odnosu na pretragu u dubinu jer paralelno izučava veliki broj putanja. Na ovaj način se mogu otkriti neka zanimljiva opažanja ranije u odnosu na pretragu u dubinu i češće se koristi ukoliko je vreme ograničeno (previše kratko da bi se čekalo da DFS pretraga istraži jednu po jednu putanju do kraja). Mana ovog algoritma je to što čuva ogroman broj stanja i puni memoriju koja je dodeljena alatu za simboličko izvršavanje. Pošto se veliki broj putanja paralelno izučava potrebno je za sve njih čuvati stanja, formule logike prvog reda čija se zadovoljivost proverava SMT rešavačem (ograničenja koja važe za svaku od putanja), održavati simbolički memorijski prostor i slično. Iz navedenih razloga je potrošnja memorije znatno veća u odnosu na algoritam pretrage u dubinu. Još jedna mana ovog pristupa je što nije realno očekivati da sve putanje mogu biti izučene do kraja. Vreme je često ograničavajući faktor kada govorimo o analizi realnog softvera gde postoji jako veliki broj linija koda pa u tim situacijama algoritam pretrage u širinu ne uspeva da izuči sve putanje do kraja. Međutim, činjenica je da paralelnim izučavanjem većeg broja putanja lakše i brže dolazi do interesantnih zapažanja u odnosu na pretragu u dubinu.

Slučajan odabir narednog stanja

Strategija kod koje se naredno stanje koje će biti obrađeno bira na potpuno slučajan način. Od svih trenutno dostupnih stanja u stablu pretarge na slučajan način se bira jedno od njih. Ova ideja je sama po sebi jako loša jer nema konkretne strategije po kojoj bira sledeće stanje, međutim često se obogaćuje dodatno različitim heuristikama. Neke od heuristika su:

Izvršavanje vođeno pokrivenošću koda - osnovna ideja ove heuristike je da se za svaku putanju određuje njena važnost na osnovu toga koliko instrukcija je u određenoj putanji izvršeno od otkrivanja poslednje nove instrukcije, koliko je udaljena naredna neposećena instrukcija i slično.

Izvršavanje vođeno podputanjama - ideja je birati ono stanje koje vodi podputanjom koja je bila izvršena manji broj puta do tog trenutka. Podputanja se definiše kao n uzasptonih naredbi u okviru jedne putanje. Kod ove heuristike ključnu ulogu igra odabir vrednosti n. Do sada nije pronađena optimalna vrednost parametra n koja je univerzalna.

Najkraće simboličko izvršavanje - ova heurstika se obično koristi kada je potrebno doći do određene instrukcije u samom kodu. Uvek se bira ono stanje koje je najbliže traženoj naredbi. Malo podseća na pretragu vođenu pokrivenošću koda, međutim razlika je u tome da u ovoj strategiji težimo jednoj konkretnoj naredbi, a ne pokrivenošću celokupnog koda.

Iscrpljivanje petji - osnovna ideja ove heuristike je da se biraju putanje koje sadrže petlje. Opravdanje je se krije u opažanju da u praksi veliki broj grešaka u radu sa petljama vodi ka prekoračenju bafera[2] i drugim memorijskim problemima.

Prvo putanje sa greškama - kod ove heuristike se prvo izučavaju one putanje kod kojih su u nekim ranijim delovima pronađene sitne greške (eng. *bugs*). Intuicija iza ovog pristupa je da ako je putanja u ranijem delu imala neku grešku nije dovoljno dobro istestirana.

[2]Divergencija putanja je pojava u dinamičkom simboličkom izvršavanju gde izvršavanje vodi nekom neočekivanom putanjom. Recimo u primeru 2.4 bi bilo očekivano da negacijom uslova grananja bude otkrivena nova putanja, međutim to se nikada neće desiti jer je kvadrat bilo kog broja uvek pozitivan broj.

Generacijska pretraga

Ova tehnika uglavnom predstavlja kombinaciju pretrage u dubinu i pretrage vođene pokrivenošću koda. Osnovna ideja je da se pretraga vrši po generacijama. Kreće se od nulte generacije u kojoj se slučajno odabrana putanja izvršava do kraja pomoću algoritma pretrage grafa stanja u dubinu. U prvoj generaciji se iz putanje nulte generacije istražuju nove putanje i to tako što se odredi jedan od uslova grananja u putanji nulte generacije i njegovim negiranjem se dobija sledeća putanja. U N-toj generaciji se iz svih putanja prethodne generacije bira uslov koji se negira i pomoću koga se kreće u istraživanje nove putanje. Biranje putanje kojom će se krenuti se vrši heuristikom vođenom pokrivenošću koda.

Hibridne strategije

Uglavnom se svode na kombinaciju većeg broja algoritama koji se na određene načine smenjuju prilikom pretrage. Zavisno od specifičnih potreba (otkrivanje putanje do određene naredbe, otkrivanje što većeg broja putanja, pokrivenost koda itd.) koriste se kombinacije različitih algoritama koji se pri izvršavanju smenjuju i kombinuju na različite načine.

2.6 Pregled alata za simboličko izvršavanje

Postoji veliki broj različitih alata za simboličko izvršavanje. Većina ovih alata počiva na istim principima:

Napredak - izvršavanje bi trebalo da može da traje neko određeno vreme bez potrošnje svih resursa koji su mu dodeljeni. Tu se pre svega misli na memoriju koja često može biti resurs koji biva brzo iskorišćen u potpunosti zbog velikog broja raličitih putanja i stanja.

Ponovljen posao - ne bi trebalo da se neki posao ponavlja. Odnosno, nije dobro da se izvršavanje kreće pokreće od samog početka da bi se otkrile neke nove putanje ako već postoji zajednički prefiks te putanje sa nekim postojećim.

Analiza i ponovno korišćenje već odrađenog posla - rezultati prethodnih izračunavanja i saznanja iz prethodnih putanja bi trebalo da se koriste u što

većoj meri kako bi se uštedelo vreme, i smanjilo ponavljanje posla što se poklapa sa prethodnom stavkom. Tu se pre svega misli na skupe pozive SMT rešavaču za formule za koje je zadovoljivost proverena ranije. Takve pozive treba izbegavati.

Na osnovu toga da li pokušavaju da obrade veći broj putanja odjednom ili jednu po jednu alati za simboličko izvršavanje se mogu podeliti u dve grupe:

***online* alati** - alati koji pokušavaju da izvršavaju i izučavaju veći broj putanja odjednom. Kada god se naiđe na grananje vrši se kloniranje trenutnog stanja. Dobra strana ovih alata je što nema ponavljanja posla jer se jednom izvršena instrukcija nikada ne izvršava ponovo. Loša strana je to što veoma brzo pune memoriju koja im je dodeljena, jer moraju da održavaju veliki broj putanja i stanja. Primeri ovakvih alata su KLEE [2], AEG [1], S^2E [4].

***offline* alati** - za razliku od *online* alata, *offline* alati izvršavaju jednu po jednu putanju. Primer ovakvog alata bi bio SAGE [8]. Ideja je izvršiti celu jednu putanju do samog kraja, a zatim preći na neku drugu. Dobra strana ovog pristupa je jako malo korišćenje memorije jer se čuvaju stanja vezana za samo jednu putanju, ali je problem što obično ima mnogo ponavljanja posla. Pretraga nove putanje uglavnom kreće od samog početka. Opšti princip rada *offline* alata je da kreću sa konkretnim izvršavanjem, što znači da je potrebno generisati ulazne podatke. Pamti se put kojim se prošlo pri konkretnom izvršavanju, a zatim se isti put prolazi simbolički.

***hibridni* alati** - hibridni alati kao što je Mayhem [3] pokušavaju da pronađu balans između prevelikog utroška memorije i vremena koje je potrebno za izvršavanje. Na početku počinju kao *online* rešavači, a kada se memorija popuni do određene količine prelaze u *offline* izvršavanje.

DART [6] - (eng. *Directed Automated Random Testing*) je alat koji vrši konkoličko simboličko izvršavanje i to dinamičko simboličko izvršavanje. Za odabir uslova grananja koji će biti negiran kako bi se otkrila nova putanja se koristi algoritam pretrage u dubinu (DFS). DART je alat koji počiva na tri osnovna principa:

1. *Automatski* izdvaja interfejs programa statičkom analizom izvornog koda.

2. *Slučajno* (eng. *Random*) generiše testove za izdvojeni interfejs programa kako bi se simuliralo što opštije okruženje rada programa.

3. Dinamička analiza ponašanja programa prilikom slučajnog testiranja (prethodna stavka) i automatsko generisanje novih test primera koji *usemravaju* (eng. *Direct*) izvršavanje ka drugim mogućim putanjama.

DART se uglavnom koristi za testiranje programa napisanih u programskom jeziku *C*.

SAGE [8] - još jedan alat koji vrši dinamičko simboličko izvršavanje. Za razliku od DART−a, SAGE koristi generacijsku pretragu. Osnovna ideja ovog alata je da sistematski, ali ipak delimično istraži prostor mogućih stanja softvera pri čemu se izbegava ponavljanje posla, a ujedno maksimizuje broj generisanih različitih testova. Pošto se izučava samo deo prostora stanja, ključan je izbor ulaznih vrednosti simboličkih promenljivih. Važnost inicijalnih vrednosti podseća na važnost istih kod tradicionalnog nejasnog testiranja metodom crne kutije (eng. *black-box fuzzy testing*), pa se stoga alati poput SAGE−a često nazivaju nejasnim testerima metodom bele kutije (eng. *white-box fuzzy testers*). SAGE je primer *offline* alata.

S^{2}E [4] - još jedan poznat alat za simboličko izvršavanje je ranije pomenuti alat S^{2}E koji vrši selektivno simboličko izvršavanje. Jedan je od glavnih predstavnika *online* alata. Pored ranije navedenih osobina treba napomenuti da se za izbor narednog stanja koje se izvršava, samim tim i izbor putanje koja se izučava, vrši heuristikom vođenom pokrivenošću koda.

Mayhem [3] - primer *hibridnog* alata za simboličko izvršavanje. Ovaj alat kombinuje tehnike *online* i *offline* alata i to tako što koristi prednosti svake od tehnika. Ova jalat radi direktno sa izvršnim kodom. Kako bi ovo bilo moguće postoje dve važne stavke o kojima Mayhem mora da vodi računa:

- Održavanje putanja koje se izučavaju bez potpune potrošnje dostupne memorije.

- Oprez pri radu sa simboličkom memorijom, odnosno sa simboličkim adresama kako ne bi dolazilo do prekoračenja bafera i sličnih problema.

Pored koncepta hibridnog alata, Mayhem još uvodi i memoriju zasnovanu na indeksima, tehniku koja ovom alatu omogućava da efikasno radi sa simboličkom memorijom na binarnom nivou (rad sa izvršnim fajlovima).

AEG [1] - **Automatic Exploit Generation** je alat koji je napravljen kao nadgradnja drugog alata za simboličko izvršavanje, *KLEE-a*. Osnovna ideja ovog alata je da se ne istražuju sve moguće putanje u programu, već da fokus bude na onim putanjama za koje se veruje da imaju veću verovatnoću izvršavanja zavisno od vrednosti ulaznih podataka. Osnovna tehnika kojom se služi ovaj alat je preduslovno simboličko izvršavanje (eng. *preconditioned symbolic execution*). Jedan zasad poznati problem ovog alata je skalabilnost na velikim programima, ali se na tome aktivno radi.

Glava 3

Alat za simboličko izvršavanje KLEE

KLEE je alat otvorenog koda koji je nastao na Univerzitetu Ilinois. U pitanju je potpuno simbolički alat što znači da su sve promenljive simboličke i nema konkoličkog simboličkog izvršavanja. Ideja alata je da pokuša kompletne programe da izvršava simbolički uz dodatak konkretnih vrednosti koje se u kodu javljaju kao konstante. Dakle, ne generiše se slučajan ulaz i na taj način se prati kojim se putem prolazi zavisno od vrednosti već pokušava da ide različitim putevima odjednom menjajući uslove koji važe. Na primer ako imamo uslov grananja da je $x > 3$ onda KLEE putanju deli na dve disjunktne putanje. U jednoj će važiti $\pi \wedge x > 3$ a u drugoj negacija tog uslova, odnosno $\pi \wedge x \leq 3$. Na ovaj način KLEE prolazi kroz obe putanje. Odlika ovog alata je i ta da sve putanje obilazi paralelno, tj. ne izučava jednu putanju do kraja. Stanja koja su vezana za svaku od aktivnih putanja se čuvaju sve vreme bez obzira na algoritam koji se koristi. Razlog za ovako nešto leži u činjenici da je KLEE jedan od *online* alata za simboličko izvršavanje.

3.1 Memorijski model

KLEE čuva vrednosti različitih promenljivih u skladištima. Postoje dva skladišta u okviru alata.

konkretno - skladište u okviru kog se čuvaju vrednosti konkretnih promenljivih. To su konstante ili promenljive kojima je dodeljena vrednost u kodu. Recimo `int x = 5;` predstavlja konkretnu promenljivu. Nije potrebno simbolički pratiti koju potencijalnu vrednost ova promenljiva ima. Ona je unapred poznata. Ukoliko se desi da se konkretna promenljiva koristi u nekoj operaciji sa

simboličkom ona takođe postaje simbolička, jer zavisno od uslova koji važe za simboličku promenljivu i konkretna menja svoju vrednost u određenoj putanji.

simboličko - skladište gde se čuvaju simboličke promenljive. Kako one nemaju konkretne vrednosti sve do trenutka kada se dođe u završno stanje, veći deo vremena se čuva samo njihova adresa. Kada se dođe u završno stanje, potrebno je generisati konkretne vrednosti simboličkih promenljivih na osnovu uslova koji važe u tom stanju. U tom trenutku one dobijaju konkretne vrednosti, i generiše se test primer koji govori o tome koje su vrednosti promenljivih kada se program izvrši kretanjem kroz određenu putanju. Te vrednosti ne moraju biti jednistvene, npr ako u stanju važi uslov $x > 5$ promenljiva x može uzeti bilo koju vrednost koja je veća od 5. Ukoliko u nekom stanju dolazi do greške pod uslovom $x = 7$ onda će vrednost promenljive x koja se generiše u tom stanju biti baš 7.

Količina radne memorije koja se dodeljuje samom alatu je ograničena i zadaje se kao jedan od parametara prilikom pokretanja samog KLEE-a. Zbog osobine alata da čuva sva stanja koja su vezana za sve putanje koje se izučavaju vrlo često dolazi do situacije gde se memorija popuni. Kada se ovo desi na neki način mora da dođe do „pražnjenja". To se radi tako što se odbacuju neka stanja. Broj stanja koja se izbacuju se određuje slučajnom metodom. Generiše se pseudo-slučajan broj i on predstavlja broj stanja koja se izbacuju. Postavlja se pitanje na koji način se biraju stanja koja bivaju izbačena. Ovo se takođe radi slučajno uz još jedan dodatan uslov, a to je da se ne izbacuju stanja koja su otkrila novu instrukciju. Generiše se pseudo-slučajan broj koji govori o indeksu stanja u nizu koje treba da bude izbačeno i ako u njemu nije otkrivena nova instrukcija ono može biti izbačeno. U suprotnom se generiše nov indeks i bira se drugo stanje za izbacivanje iz skupa aktivnih stanja.

3.2 Kontrola toka programa

Rad sa funkcijama u alatu KLEE se ne razlikuje previše od standardnog rada sa funkcijama u nekom programskom jeziku. Neka je potrebno simbolički izvršiti funkciju f. I neka to bude funkcija koja se poziva iz funkcije g sa parametrom a bez povratne vrednosti. Dakle, `void f(int a)`. Postoji „stek" na kome se čuvaju lokalne promenljive i parametri funkcije f. Takođe se pamti koja je funkcija pozivalac kao i koju naredbu funkcije g treba izvršiti nakon što f bude izvršena. Sve promenljive se

i dalje čuvaju u ranije opisanim skladištima. Nakon što se završi izvršavanje funkcije memorija na kojoj su se nalazile lokalne promenljive postaje „slobodna", odnosno tu je moguće smeštait nove promenljive neke nove funkcije. Sve rečeno se odnosi samo na funkcije za koje je izvorni kôd poznat i napisan u programskim jezicima C i C++. Ono što se može desiti je da izvorni kôd neke funkcije nije poznat uopšte, ili da je napisan u drugom programskom jeziku. Ukoliko se nešto ovako desi KLEE nije u mogućnosti da prati izvršavanje funkcije simbolički. U tim situacijama se argumenti konkretizuju na osnovu uslova koji važe na putanji gde se nalazi poziv funkcije i funkcije se jednostavno poziva i čuva se povratna vrednost ukoliko postoji. Ukoliko je funkcija bez povratne vrednosti pozvana samo se izvršava bez simboličkog praćenja i izvršavanje se nastavlja simbolički u okviru funkcije pozivaoca. Kada je reč o petljama KLEE ima prilično specifičan način rada. Naime, kako je alat potpuno simbolički, sve petlje se izvršavaju do kraja kroz sve potencijalne iteracije. Ako važi da je uslov izlaska iz petlje `i < n` gde je n simbolička promenljiva, KLEE će pokušavati da prođe sve moguće iteracije. Ukoliko bi se znalo da je vrednost promenljive n na primer 1000 onda bi se petlja vrtela simbolički do vrednosti 1000. Međutim, kada je granica broja iteracija petlje simbolička promenljiva KLEE nema mogućnost da izvrti određen broj iteracija i vidi ponašanje programa, već se ide do maksimalne vrednosti koju može da čuva promenljiva tipa koji odgovara tipu promenljive n. Iz ovog razloga KLEE ima veliki problem sa petljama bez obzira na to koji se algoritam koristi. Često u okviru petlji postoje grananja što dovodi do toga da se kreira veliki broj novih stanja u okviru istih. Na ovaj način se i memorija brže puni što dovodi do potrebe za ranije opisanim odbacivanjem nekih stanja. Kada se govori o rekurzivnim funkcijama situacija je veoma slična. Rekurzija se izvršava do uslova izlaska. Ukoliko uislov izlaska iz rekurzije zavisi od simboličke promenljive javlja se isti problem kao i pri radu sa petljama u okviru alata KLEE.

3.3 Rad sa pokazivačima i nizovima

Rad sa pokazivačima se takođe ne razlikuje previše od onoga na šta smo navikli u programskom jeziku C. Postoji poseban deo memorije koji se zove *Known Symbolics* gde se smeštaju promenljive poslate preko pokazivača. Kada je promenljiva prosleđena preko pokazivača funkciji nju je moguće menjati u okviru iste. Ako je u pitanju simbolička promenljiva potrebno je znati koja simbolička promenljiva se „menja", tj. koji dodatni uslovi za istu treba da važe. Zato postoji pomenuti deo

simboličkog skladišta koji čuva sve simboličke promenljive koje su uvedene do tad. Objekat okazivačke promenljive koja se kreira prilikom poziva funkcije kao jedno od polja čuva i adresu promenljive na koju „pokazuje". Na taj način je moguće u okviru dela *Known Symbolics* moguće menjati promenljivu, odnosno uslove koji važe za nju. Ukoliko je promenljiva koja se šalje preko pokazivača konkretna, onda se „prati" adresa u okviru konkretnog skladišta i vrednost konkretne promenljive se menja. Rad sa nizovima dosta podseća na rad sa pokazivačima. Suština obe tehnike je ista. Postoji deo memorije gde se čuvaju simbolički nizovi. Kada se javi potreba za pristupom nekom elementu niza na osnovu indeksa, kao i kod programskog jezika C se to radi tako što se u odnosu na početak niza vrši pomeranje za određeni broj, tj. baš traženi indeks. Zavisno da li se radi o konkretnom ili simboličkom nizu, sve promenljive koje pripadaju istom se čuvaju u odgovarajućem skladištu (konkretnom odnosno simboličkom). Nakon što se javi potreba za pristupom i-tom elementu niza, određuje se njegova adresa, tj. kreira se pokazivač na taj element kako bi mu se pristupilo i kako bi eventualno mogao da bude izmenjen. Kao i kod pokazivačkih promenljivih (nizovi u suštini i jesu pokazivači) i kod nizova se elementi konkretnih nizova menjaju u konkretnom skladištu, a elementima simboličkih nizova se dodatna ograničenja dodaju u okviru *Known Symbolics* dela simboličkog skladišta.

3.4 Algoritmi pretrage stabla stanja

KLEE kao i većina drugih alata u okviru svoje implementacije sadrži vreći broj algoritama za pretragu grafa (stabla) stanja simboličkog izvršavanja. Alat se svrstava u grupu *online* alata što znači da bez obzira na to koji se konkretan algoritam koristi za analizu nekog koda, sve putanje i sva stanja koja su vezana za njih će biti čuvani tokom izvršavanja (dok se putanja ne završi čime se brišu sva stanja vezana za nju pa i ona sama).

DFS

Jedan od osnovnih algoritama pretrage grafa stanja u alatu KLEE je DFS algoritam. U pitanju je tradicionalna pretraga grafa u dubinu. Ideja je istražiti jednu putanju do njenog samog kraja (završnog stanja ili dok se ne ispuni neki drugi uslov prekida istraživanja putanje), a zatim se vrši povratak na neko prethodno grananje gde se ide suprotnim putem od izabranog prilikom tog grananja (slično

kao kod pretrage bilo kog grafa u dubinu). Ukoliko imamo stanje A u kome se vrši naredba grananja, u narednom koraku je potrebno da postoje dva stanja (B i C), jer postoji grananje koje može da odvede izvršavanje na dva različita puta. Interna implementacija algoritma pretrage u dubinu u okviru alata KLEE koristi *std::vector<>* pokazivača na stanja izvršavanja (ExecutionState). Svaki put kada se naidje na grananje u stanju A vrši se dodavanje stanja C na kraj vektora, ali se stanje B ne kreira eksplicitno. Umesto toga KLEE vrši „evoluciju" stanja A. Ovom stanju se samo dodaju dodatna ograničenja koja treba važe u stanju B, menja se *trenutna* naredba koja se izvršava u stanju da odgovara naredbi koja je trenutna u stanju B i menja se *sledeća* naredba koja takođe odgovara novonastalom stanju. Dakle, jedno stanje evoluira i jedno stanje se dodaje dodaje na kraj vektora stanja svaki put kada imamo naredbu grananja. Kako bi se nad ovakvim grafom (na početku nemamo sve čvorove grafa, već se on formira dinamički) simulirala pretraga u dubinu uvek se za naredno stanje bira poslednje stanje iz vektora. Kasnije u tekstu će biti prikazan primer izvršavanja algoritma DFS gde će se videti kako se vrši simulacija navedenog algoritma. Ukoliko u nekom stanju nema naredbe grananja, vrši se samo „evolucija" stanja, tj. dodaju se nova ograničenja, menja se *trenutna* i *sledeća* naredba tog stanja i izvršavanje se nastavlja. Ovo stanje ne menja svoju poziciju u vektoru, već ostaje poslednje. Kada se dođe do završnog stanja ono se samo uklanja iz vektora stanja i uzima se poslednje stanje koje ostaje u vektoru kao naredno čime se nastavlja pretraga stabla u dubinu. Treba obratiti pažnju da iako je uobičajena implementacija DFS algoritma rekurzivna, u implementaciji u alatu KLEE nema rekurzije niti steka kojim bi se simulirala ista, već se algoritam simulira pomoću vektora na već opisani način.

BFS

Još jedan osnovni algoritam kako pretrage grafova (samim tim i stabala), tako i pretrage stabla stanja u alatima za simboličko izvršavanje je BFS. Za razliku od pretrage grafa u dubinu, ideja pretrage u širinu je, baš kao i kod svakog grafa, da se što veći broj putanja izučava odjednom, odnosno da se izvršavaju sva stanja koja su na jednom nivou stabla, a nakon što se sva stanja sa nekog nivoa izvrše prelazi se na naredni nivo. Ovaj pristup ima prednost u odnosu na prethodni jer ima potencijala da otkrije veći broj zanimljivih stanja i pojava u kodu u ranim fazama

putanja. Zamislimo recimo da imamo potpuno binarno stablo.[3]. Neka se izvršavanje nalazi na nivou dva gde imamo osam čvorova (stanja). Dodatno, neka je memorija ograničavajući faktor u toj meri da neka stanja moraju biti odbačena jer je nema dovoljno da se izvrši sve do kraja. Zamislimo da na narednom nivou nakon grananja prvog stanja (krajnji levi čvor na nivou) može doći do greške u „evoluiranom" stanju. DFS neće biti u stanju da pronađe ovu grešku jer će birati uvek poslednje stanje i kretaće se njime do samog kraja. S druge strane, BFS algoritam će izvršti svih osam stanja drugog nivoa i nakon toga će biti u stanju da već kod narednog stanja („evoluiranog") pronađe grešku. U alatu KLEE interna implementacija algoritma pretrage u širniu koristi *std::deque<>* pokazivača na stanja koja se izvršavaju (ExecutionState), kako bi operacije skidanja sa početka i dodavanja na kraj bile efikasne. Slično kao kod algoritma DFS, i BFS samo vrši „evoluciju" stanja kada dođe do grananja. Ako postoji stanje A u kome se vrši grananje, u sledećem koraku treba da postoje stanja B i C. Stanje B će biti zapravo samo „evoluirano" stanje A kome su dodata nova ograničenja, i promenjene *trenutna* i *naredna* instrukcija koje se izvršavaju, dok će stanje C biti novo stanje koje se kreira i dodaje na **kraj** reda stanja kao što se to i obično radi kod algoritma pretrage grafa u širinu. Ukoliko u stanju nema grananja, ono će samo „evoluirati" u novo stanje dodavanjem ograničenja i menjanjem odgovarajućih instrukcija kao i u algoritmu pretragu u dubinu. Razlika u odnosu na algoritam DFS je ta što „evoluirano" stanje neće ostati na istoj poziciji u redu stanja već će se prebaciti na kraj reda kako bi sva ostala stanja na istom nivou u stablu bila izvršena pre njega. Ovim postupkom ovo stanje „prelazi" u naredni nivo što i jeste očekivano ponašanje.

Slučajan odabir narednog stanja

Verovatno i najjednostavniji ali retko korišćen algoritam za pretragu stabla stanja je algoritam slučajnog odabira narednog stanja. Naredno stanje koje će biti izvršeno se bira slučajno (eng. *random*) iz skupa dostupnih stanja. Interna implementacija ovog algoritma čuva pokazivače na stanja u vektoru (*std::vector<>*). Kao i kod ostalih stanja ako postoji grananje, trenutno stanje „evoluira" i dodaje se jedno novo. Ukoliko ne postoji grananje samo se dodaje jedno novo stanje na kraj vektora stanja. Generisanjem pseudo-slučajnog broja u intervalu $[0 - n\text{-}1]$ gde je n trenutni broj stanja u vektoru stanja se bira naredno stanje koje se izvršava, bez ikakvog

[3]Svi čvorovi osim listova imaju tačno dva potomka

pravila.

Težinsko slučajno biranje narednog stanja

Nešto napredniji algoritam pretrage grafa stanja u odnosu na prethodno navedene je algoritam težinskog slučajnog biranja narednog stanja. Osnovna ideja ovog algoritma je da se svakom stanju koje treba da bude izvršeno dodeljuje težina prema odabranoj heuristici. Intera implmentacija ovog algoritma koristi implementaciju crveno crnog stabla koja je implementirana u okviru alata. U ovoj strukturi podatakase čuvaju pokazivači na stanja koja se izvršavaju. Kao i kod ostalih algoritama ukoliko postoji grananje u nekom stanju ono „evoluira" i dodaje se jedno novo stanje, a ukoliko nema grananja trenutnom stanju se menjaju *trenutna* i *sledeća* naredba, kao i formula logike prvog reda π koja čuva ograničenja koja važe u stanju tj. putanji kojoj stanje pripada. Svako od stanja dobija određenu težinu zavisno od heuristike koja se koristi i stanja se u stablo ubacuju na osnovu težine, i to tako što se pri ubacivanju stanja ono smešta u list a zatim zavisno od pozicije *crvenih* i *crnih* čvorova može doći do rotacija čvorova unutar stabla. Svaki put kada se neko stanje doda u stablo, njegova težina se propagira uz put kojim se došlo do pozicije na kojoj se čvor smestio u stablu. Samim tim koren stabla čuva sumu svih težina čvorova u stablu. Izbor narednog stanja koje će se stanje izvršavati se vrši slučajnim izborom. Najpre se genriše slučajan broj p iz intervala $[0, 1)$. Zatim se težina korena pomnoži ovom vrednošću čime se dobija težina w. Dok god postoji levi sin i njegova težina je veća od w ide se levo niz stablo. Ukoliko je w veće od težine u levom sinu w se smanjuje za tu vrednost. Ukoliko se dođe do čvora u kojima važi $w < w_n$ pri čemu w_n predstavlja težinu trenutnog čvora kažemo da baš taj čvor čuva stanje koje se naredno izvršava. Inače, ide se desno niz stablo pri čemu se vrednost w pri svakom spuštanju umanjuje za težinu desnog potomka u koji se pretraga spušta. Detaljinije o operacijama dodavanja i biranja narednog stanja se može naći u [2]. Heursitke koje postoje u okviru alata KLEE su:

Dubina stanja - težina stanja se određuje kao dubina na kojoj se nalazi stanje, tj. nivo u stablu na kom se nalazi čvor koji čuva odgovarajuće stanje.

RP - težina stanja se određuje kao $(\frac{1}{2})^d$ gde je d nivo stabla na kom se nalazi stanje.

Broj instrukcija - heuristika koja stanju dodeljuje težinu na osnovu broja instrukcija koje su izvršene da bi se do njega došlo.

Cena upita - težina stanju se dodeljuje na osnovu vremena koje je potrebno da se izvrši upit (provera zadovoljivosti formule logike prvog reda pomoću SMT rešavača).

Udaljenost najbliže neotkrivene instrukcije - najkorišćenija heuristika u alatu KLEE i heuristika koja je preporučena od strane autora alata. Ona daje najveću pokrivenost koda koja može da se postigne. Težina stanja se određuje na osnovu udaljenosti trenutnog stanja od najbliže neotkrivene (do tada neizvršene) naredbe. Što je stanje bliže neotkrivenoj instrukciji to je težina koja mu se dodeljuje veća.

Batch pretraga

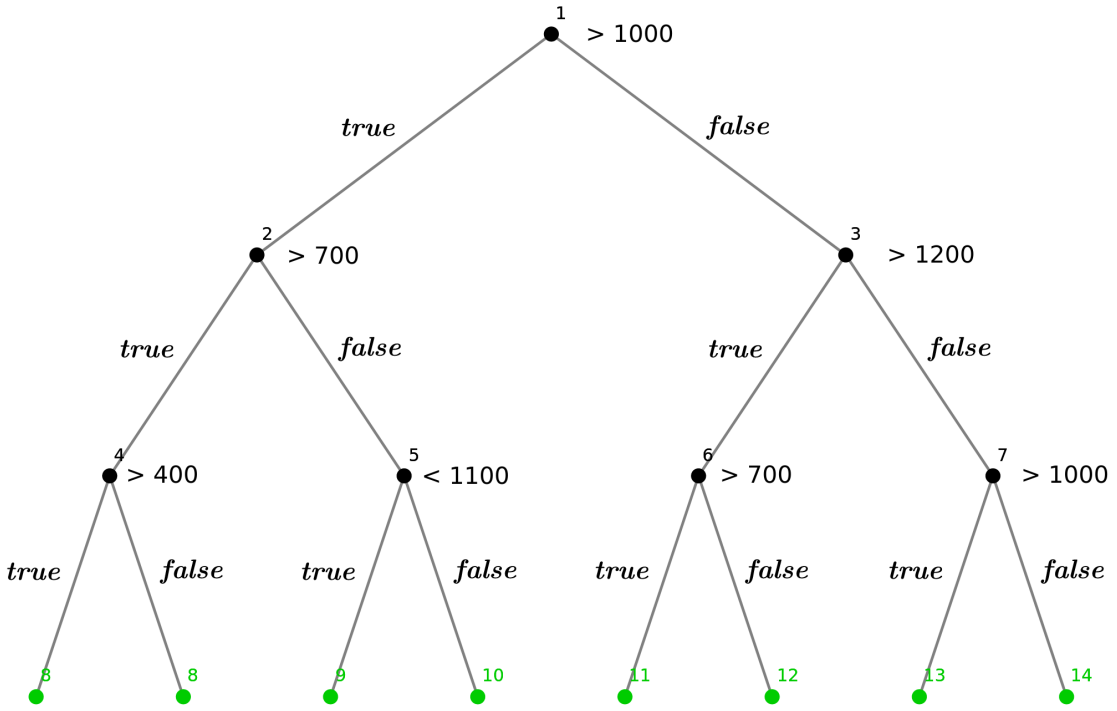
Batch pretraga predstavlja dodatak bilo kom od prethodno navedenih algoritama. Postoje dva dodatka:

- Ako određeni broj instrukcija bude izvršen bez otkrivanja nove instrukcije, pređi na neku drugu putanju.
- Ako određeno vreme nema novootkrivene instrukcije, pređi na neku drugu putanju.

Ovaj dodatak se može koristiti uz bilo koji od ranije navedenih algoritama, ali se najčešće koristi u kombinaciji sa algoritmom *težinskog slučajnog biranja narednog stanja* i heuristikom *udaljenost najbliže neotkrivene instrukcije*. U tom slučaju su dobijeni rezultati najbolji.

3.5 Primeri rada osnovnih algoritama u okviru alata KLEE

U ovom delu će biti prikazan rad algoritama DFS i BFS na jednom primeru. Razmotrimo primer 3.1:



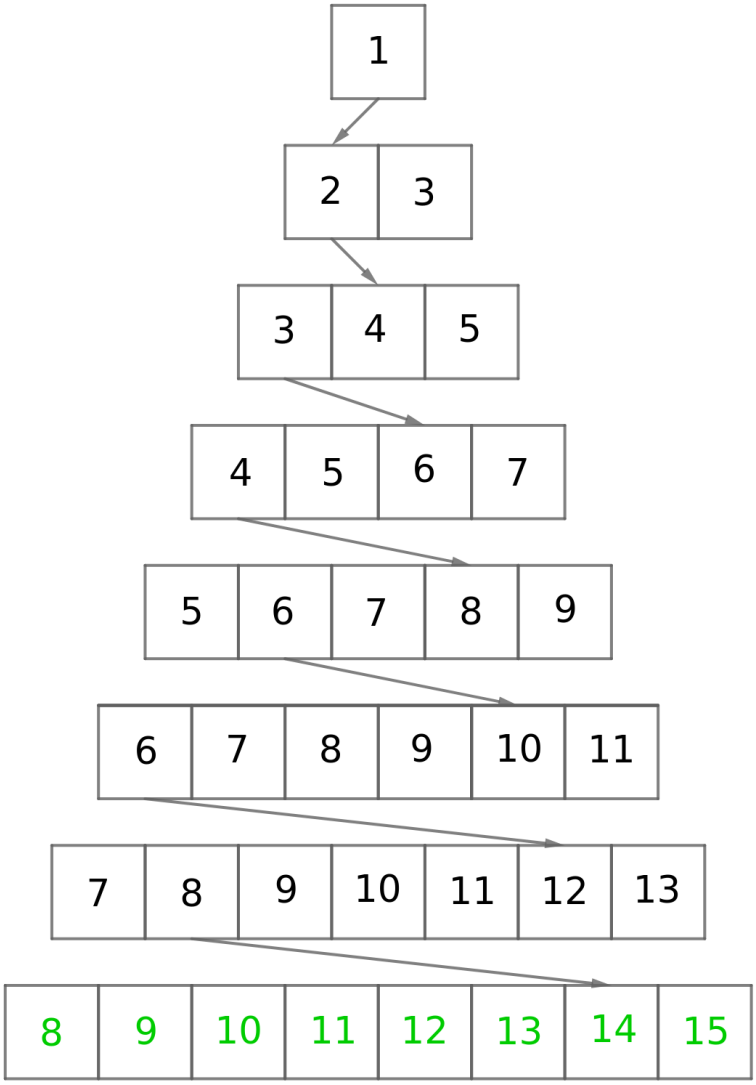
Slika 3.1: Stablo stanja simboličkog izvršavanja koje odgovara primeru 2. Zelenom bojom su označena završna stanja.

```
void foo(int a) {
  if (a > 1000) {
    a /= 2;
    if (a > 700) {
      a /= 2;
      if (a > 400) {
        a /= 2;
      }
      else {
        return ;
      }
    }
    else {
      a *= 2;
      if (a < 1100) {
        return ;
      }
      else {
        a *= 2;
      }
    }
  }
  else {
    a *= 2;
    if (a > 1200) {
      a /= 2;

      if (a > 700) {
        a /= 2;
      }
      else {
        return ;
      }
    }
    else {
      a *= 2;

      if (a > 1000) {
        a *= 2;
      }
      else {
        a *= 2;
      }
    }
  }
}
```

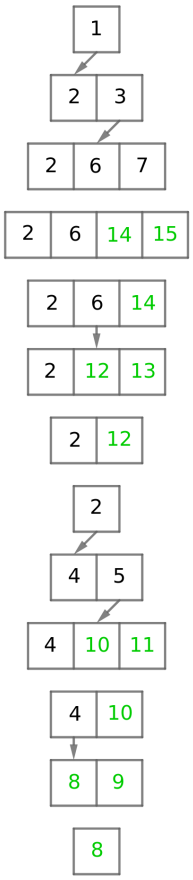
Listing 3.1: Primer simboličkog izvršavanja u alatu KLEE



Slika 3.2: Pregled izgleda vektora stanja tokom izvršavanja DFS algoritma. Zelenom bojom su prikazana završna stanja, a strelice predstavljaju „evoluciju" stanja.

BFS - Ako pogledamo kako se menja red koji čuva stanja koja se izvršavaju vidimo da se kreće iz stanja 1. Zatim stanje 1 „evoluira" u stanje 2 i na kraj reda se dodaje stanje 3. Sledeće stanje koje se uzima je upravo 2, koje dalje „evoluira" u stanje 4. Takođe se dodaje novo stanje 5. Stanje 3 mora da ostane na početku reda kako bi bilo naredno izvršeno jer je u pitanju algoritam BFS i potrebno je prvo izvršiti sva stanja na jedno nivou stabla, pa tek onda preći na naredni. Od stanja 3 se dobijaju stanja 6 i 7 koji idu na kraj reda. Sledeće stanje koje se izvršava je 4, od koga nastaju 8 i 9. Postupak se nastavlja na prikazani način sve dok u redu ne ostanu

samo završna stanja (8 - 15). U tom trenutku će se sva završna stanja izvršiti redom i biti izbacivana iz reda u redosledu izvršavanja. Pošto iz završnog stanja nema gde da se ode dalje, ono se samo izbacuje, time se putanja završava i prelazi se na naredno stanje koje treba izvršiti.



Slika 3.3: Pregled izgleda reda stanja tokom izvršavanja BFS algoritma. Zelenom bojom su prikazana završna stanja, a strelice predstavljaju „evoluciju" stanja.

DFS - kod izvršavanja algortima pretrage grafa u dubinu niz stanja se menja na drugačiji način u odnosu na pretragu grafa u širinu. U pitanju je drugačija priroda algoritma. Kod DFS-a se jedna putanja izučava do kraja pa se onda pretraga vraća na grananje gde se bira drugi put. Kod BFS-a se paralelno izučavaju sve putanje. Vidimo da se na početku kreće iz stanja 1 koje evoluira u stanje 2 i dodaje se novo stanje 3 na kraj vektora. Međutim onda se bira baš stanje 3, kao poslednje u vektoru kako bi se simulirala pretraga grafa u dubinu. Nakon toga stanje 3 evoluira u stanje 6 i dodaje se novo stanje 7, dok stanje 2 ostaje i dalje u vektoru ali na početku.

Pretraga se nastavlja iz stanja 7 koje „evoluira" u stanje 14 i dodaje se stanje 15. Kako je sledeće stanje koje se izvršava broj 15, a ono je ujedno i završno stanje, samo se izbacuje iz vektora i prelazi se na stanje 14. Ono je takođe završno pa se pretraga vraća na stanje 6, čime se zapravo izvršio povratak na prethodno grananje (kod stanja 3 na slici). Iz stanja 6 se dobijaju 2 završna stanja, koja nakon izvršavanja bivaju izbačena iz vektora gde ostaje samo stanje 2. Primećujemo da je cela desna polovina stabla stanja izvršena, tj. istražene su sve putanje, pa se pretraga dalje na isti način fokusira na levu plovinu stabla. Iako je u pitanju DFS algoritam koji jednu putanju izvršava do kraja pre nego što pređe na narednu, primećujemo da se sve vreme čuvaju i stanja koja su vezana za putanje koje se ne izučavaju u tom trenutku. Razlog za to je taj što je KLEE *online* alat, kao što je već rečeno u poglavlju 2.6.

Glava 4

Zaključak

Literatura

[1] Thanassis Avgerinos i dr. „AEG: Automatic Exploit Generation". U: *NDSS*. 2011.

[2] Cristian Cadar, D. Dunbar i D. Engler. „KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". U: *OSDI*. 2008.

[3] S. K. Cha i dr. „Unleashing Mayhem on Binary Code". U: *2012 IEEE Symposium on Security and Privacy*. 2012, str. 380–394.

[4] Vitaly Chipounov, Volodymyr Kuznetsov i George Candea. „The S2E Platform: Design, Implementation, and Applications". U: *ACM Trans. Comput. Syst.* 30.1 (Feb. 2012). ISSN: 0734-2071. DOI: 10.1145/2110356.2110358. URL: https://doi.org/10.1145/2110356.2110358.

[5] Sanjit A. Seshia Clark Barrett Roberto Sebastiani i Cesare Tinelli. „Satisfiability Modulo Theories". U: (). URL: https://people.eecs.berkeley.edu/~sseshia/pubdir/SMT-BookChapter.pdf.

[6] Patrice Godefroid, Nils Klarlund i Koushik Sen. „DART: Directed Automated Random Testing". U: *SIGPLAN Not.* 40.6 (Jun 2005), 213–223. ISSN: 0362-1340. DOI: 10.1145/1064978.1065036. URL: https://doi.org/10.1145/1064978.1065036.

[7] Patrice Godefroid, Michael Y. Levin i David Molnar. „SAGE: Whitebox Fuzzing for Security Testing". U: *Queue* 10.1 (Jan. 2012), 20–27. ISSN: 1542-7730. DOI: 10.1145/2090147.2094081. URL: https://doi.org/10.1145/2090147.2094081.

[8] Patrice Godefroid, Michael Y. Levin i David A. Molnar. „Automated Whitebox Fuzz Testing". U: *NDSS*. 2008.

Biografija autora