Beam Stack Search algoritam pretrage u okviru alata za simboličko izvršavanje KLEE

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Aleksandar Stefanović, 1021/2023
Petar Đorđević, 1088/2022

18. avgust 2024.

Sažetak

Jedan od osnovnih problema simboličkog izvršavanja kao tehnike verifikacije softvera je eksplozija broja stanja, tj. putanja kojima izvršavanje programa može da teče. Zbog toga, praktično je nemoguće izvršiti potpunu pretragu svih putanja izvršavanja programa, pa alati za simboličko izvršavanje, poput alata `KLEE`, koriste različite heurističke pristupe za izbor podskupa putanja za istraživanje.

Ovaj rad je posvećen implementaciji *Beam Stack Search* algoritma pretrage u okviru alata za simboličko izvršavanje `KLEE`, modifikacije *Beam Search* algoritma koji svoje primene najčešće nalazi u oblasti mašinskog učenja, konkretno u obradi prirodnog jezika i mašinskog prevođenja. Eksperimentalnom evaluacijom pokazujemo da se *Beam Stack Search* algoritam u nekim realnim primerima ponaša bolje od podrazumevanog algoritma pretrage alata `KLEE`.

Sadržaj

1 Uvod

Simboličko izvršavanje u svom osnovnom obliku predstavlja tehniku statičke verifikacije programa, tj. verifikacije programa bez njegovog pokretanja, u kojoj se umesto konkretnog stanja programa tokom izvršavanja prati njegovo simboličko stanje [6], u cilju otkrivanja grešaka, automatskog generisanja testova sa velikom pokrivenošću koda i slično. Ovo podrazumeva praćenje simboličkih vrednosti promenljivih koje se javljaju u programu, kao i tzv. uslova putanja - logičkih formula nad simboličkim vrednostima tih promenljivih koje moraju da budu zadovoljene da bi izvršavanje stiglo do neke tačke u programu.

Svaka naredba kontrole toka, poput naredbi grananja ili petlji, može prouzrokovati više putanja izvršavanja programa. Ove putanje, zajedno sa pridruženim uslovima putanja i simboličkim vrednostima promenljivih, indukuju simboličko stablo izvršavanja programa. Osim za najtrivijalnije programe, simboličko stablo izvršavanja je nemoguće u potpunosti istražiti. Simboličko stablo izvršavanja program koji sadrži samo 30 naredbi grananja, zanemarujući moguće nedostižne putanje, sadržalo bi 2^{30} različitih putanja izvršavanja. Ukoliko program sadrži i petlje, njegovo stablo izvršavanja bi potencijalno bilo i beskonačno veliko (ukoliko i sam uslov prekida petlje ima simboličku vrednost).

Upravo eksplozija broja stanja u simboličkom stablu izvršavanja je jedan od glavnih problema sa kojim se susreću alati za simboličko izvršavanje. Neke od tehnika koje alati koriste da bi umanjili efekat ovog problema su odsecanje nedostižnih putanja, spajanje stanja, zadavanje preduslova i aproksimacija petlji [2]. Međutim, čak i uz primenu ovih tehnika, najčešće nije moguće eliminisati dovoljan broj stanja da bi se efikasno istražilo celokupno simboličko stablo izvršavanja, pa je od velike važnosti izbor stanja, tj. putanja, za ispitivanje.

Izbor narednog stanja za ispitivanje zavisi od primenjene strategije za obilazak puteva. Dve osnovne strategije su *DFS*, tj. pretraga u dubinu, i *BFS*, tj. pretraga u širinu. Iako imaju svoje primene, obe strategije imaju svoje mane - pretraga u dubinu ima tendenciju zaglavljivanja u „dubokim" putanjama indukovanih petljama ili rekurzijom, dok pretraga u širinu povlači izuzetno veliko memorijsko zauzeće usled potrebe za istovremenim čuvanjem čitavog nivoa, tj. sloja čvorova simboličkog stabla izvršavanja. Postoje i hibridne tehnike između ova dva pristupa, poput algoritma *BFS-DFS* [10]. Često se primenjuju i tehnike koje koriste randimozovane varijante algoritama pretrage, gde se stanjima koja imaju neku dobru osobinu može dodeliti prednost prilikom izbora, poput algoritma koji podrazumevano koristi alat za simboličko izvršavanje `KLEE` [3].

U nastavku rada će biti predstavljena upotreba *Beam Stack Search* algoritma pretrage, kao i njegova implementacija u okviru alata za simboličko izvršavanje `KLEE`. Na kraju, biće izvršena uporedna analiza performansi ovog algoritma sa podrazumevanim algoritmom pretrage koji koristi alat `KLEE`.

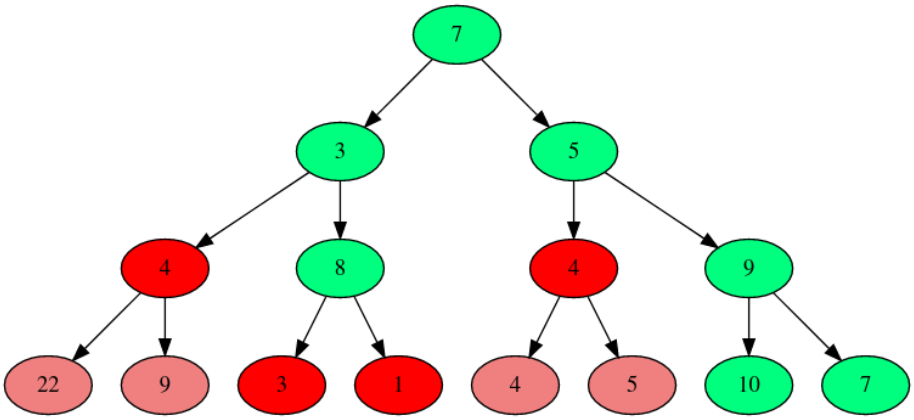
2 Opis algoritma

Algoritam *Beam Stack Search* [11] predstavlja proširenje heurističkog algoritma pretrage *Beam Search*, pa će u nastavku prvo biti opisan ovaj algoritam. Iako u opštem slučaju ovi algoritmi imaju mogućnost obilaska proizvoljnog grafa, u nastavku rada će se, zbog potreba u simboličkom izvršavanju, razmatrati samo obilazak stabla.

2.1 *Beam Search* algoritam pretrage

Algoritam pretrage *Beam Search*, nadalje *BS* algoritam, svoje najveće primene nalazi u oblasti mašinskog učenja, konkretno u obradi prirodnog jezika i mašinskog prevođenja [8, 4, 9]. Međutim, ideje koje on uvodi se mogu primeniti i u simboličkom izvršavanju. Naime, *BS* algoritam, po cenu gubitka kompletnosti pretrage, heurističkim pristupom brzo i uz malu memorijsku potrošnju pronalazi dobre, po nekoj izabranoj meri kvaliteta, putanje u težinskom grafu. U kontekstu simboličkog izvršavanja, ova ideja se može primeniti na pronalaženje putanja koje, na primer, maksimizuju pokrivenost koda.

Algoritam kao parametar prihvata ceo broj w, tj. širinu pretrage, i zasnovan je na *BFS* pretrazi, sa modifikacijom da na svakom nivou stabla pretrage zadržava samo w najboljih čvorova (najveće ili najmanje težine, zavisno od izabrane mere kvaliteta čvora), a ostale čvorove odbacuje. Primer izvršavanja za $w = 2$ nad stablom gde se prednost daje čvorovima veće težine prikazan je na slici 1. Zeleni čvorovi označavaju čvorove izabrane za pretragu, crveni čvorove koji su odbačeni, a svetlo crveni čvorove koji, kao potomci odbačenih čvorova, nisu ni razmatrani.



Slika 1: Izvršavanje *BS* algoritma nad stablom za širinu pretrage $w = 2$

Pošto se na svakom nivou stabla zadržava samo w čvorova, a prethodno posećeni čvorovi se ne pamte, memorijska složenost ovog algoritma pretrage je $O(w)$, tj. ne zavisi od veličine stabla. Međutim, problem u ovom pristupu je to što će samo w putanja biti u potpunosti istraženo, pod pretpostavkom da su one konačne dužine. U kontekstu simboličkog izvršavanja, ovo bi dovelo do gubitka saglasnosti analize. Da bi se ovaj problem rešio, potrebno je izvršiti modifikaciju algoritma, koja je data u vidu *Beam Stack Search* algoritma pretrage.

2.2 *Beam Stack Search* algoritam pretrage

Algoritam *Beam Stack Search*, nadalje *BSS*, prvi put je opisan u radu [11] i predstavlja nadogradnju osnovnog *Beam Search* algoritma koja rešava prethodni problem odbacivanja potencijalno važnih putanja izvršavanja, ali po cenu povećane memorijske složenosti. Naime, osnovna ideja *BSS* algoritma je u tome da se, umesto odbacivanja, neposećena stanja čuvaju za kasniju obradu u poseb-

no uređenoj stek strukturi. Stek je uređen po nivoima - čvorovi na proizvoljnom nivou stabla koji su razmatrani, ali ne i izabrani za obradu, se postavljaju kao jedan nivo čvorova na steku. U trenutku kada bi standardna *Beam Search* pretraga stigla do kraja stabla, sa vrha steka se uzima jedan nivo čvorova i nastavlja se sa pretragom. Algoritam je prikazan pseudokodom 1.

Algoritam 1 Tree Beam Stack Search

```
1:  procedure BSS(Tree t, Int w)
2:      Vec<Node> currentLevel = {t.root()}
3:      Vec<Node> nextLevel = {}
4:      Stack<Vec<Node>> beamStack = {}
5:      while not currentLevel.empty() do
6:          for Node n in currentLevel do
7:              visit(n)
8:              nextLevel.append(t.adjacent(n))
9:          end for
10:         nextLevel.sort()
11:         Int beamWidth = min(nextLevel.size(), w)
12:         currentLevel = nextLevel[0 : beamWidth]
13:         if beamWidth != nextLevel.size() then
14:             beamStack.push(nextLevel[beamWidth : nextLevel.size()])
15:         end if
16:         nextLevel.clear()
17:         if currentLevel.empty() and not beamStack.empty() then
18:             currentLevel = beamStack.top()
19:             beamStack.pop()
20:         end if
21:     end while
22: end procedure
```

Upotreba steka sa neposećenim čvorovima garantuje da će svi čvorovi stabla u nekom trenutku biti posećeni, ali njegovo održavanje zahteva dodatnu memoriju u odnosu na klasičan *Beam Search* algoritam. Ukoliko pretpostavimo da je stablo nad kojem se algoritam izvršava binarno, što i jeste validna pretpostavka za osnovni slučaj simboličkog stabla izvršavanja, može se pokazati da je memorijska složenost algoritma ograničena sa dubinom stabla i zadatim parametrom širine pretrage w.

Lema 2.1. *Ukoliko se algoritam Beam Stack Search primenjuje nad binarnim stablom, memorijska složenost algoritma je $O(dw)$, gde d predstavlja dubinu stabla, a w je zadati parametar širine pretrage.*

Dokaz. Za dokaz je dovoljno pokazati da je (1) veličina svakog nivoa čvorova na steku najviše w i (2) na steku je u svakom trenutku najviše d nivoa čvorova.

1. Posmatrajmo skup izabranih čvorova u i-toj iteraciji algoritma 1, u oznaci $currentLevel$. Pre ulaska u petlju, ovaj skup sadrži samo koren stabla, te je njegova kardinalnost jednaka $card(currentLevel) = 1$. U $i-toj$ iteraciji petlje, skup čvorova $currentLevel$ se ažurira tako što se od njima susednih čvorova, u oznaci $nextLevel$, bira $min(w, card(nextLevel))$ njih.

Prema tome, važi da je $card(currentLevel) \leq w$ u svakoj iteraciji petlje. Usled uslova da je stablo binarno, dodatno važi i $card(nextLevel) \leq 2 \cdot card(currentLevel) \leq 2w$. Pošto se na stek dodaju skupovi čvorova veličine $card(nextLevel) - w$, iz prethodnog važi da će se na steku u svakom trenutku nalaziti samo skupovi čvorova veličine manje ili jednake od w.

2. Dovoljno je primetiti da nivoi na steku, počev od njegovor vrha, odgovaraju strogo opadajućem redosledu nivoa u stablu, gde najmanji nivo ima koren stabla. Naime, svakim ponovnim pokretanjem pretrage od nekog nivoa na steku se taj nivo skida sa steka, a dalja pretraga do ponovnog pokretanja od nekog nivoa sa steka teče strogo niz stablo, pa sledi da je nivo koji odgovara svakom skupu čvorova koji se dodaje na stek strogo manji od onog koji je prethodno izbačen sa steka, kao i od onog koji je prethodno dodat na stek. Pošto stablo ima najviše d nivoa, sledi da se na steku u svakom trenutku može naći najviše d nivoa čvorova.

□

Ovako ograničena memorijska složenost sprečava prebrzo zauzeće memorije, kao što je to slučaj kod klasičnog BFS algoritma, ali pritom zadržava mogućnost šireg prostora pretrage i istovremenog praćenja više potencijalno zanimljivih putanja pomoću podesivog parametra širine pretrage w. Zapravo, za vrednost parametra $w = 1$, algoritam bi se ponašao kao DFS algoritam pretrage, pri čemu se u svakom koraku bira bolji čvor za nastavak pretrage, dok bi se za dovoljno veliko w, vrednosti veće ili jednake širini celokupnog stabla, ponašao kao BFS algoritam. Dodatno, izbor čvorova za posetu prema nekoj izabranoj meri kvaliteta omogućava brži pronalazak kvalitetnih putanja, što bi u kontekstu simboličkog izvršavanja moglo da podrazumeva putanje čije izvršavanje osigurava veliki stepen pokrivenosti koda.

3 Implementacija u okviru alata za simboličko izvršavanje KLEE

Implementacija *BSS* algoritma pružena je kroz klasu `BeamSearcher`, specijalizaciju apstraktne klase `Searcher` - bazne klase svih algoritama pretrage u okviru alata `KLEE`. Potpuna implementacija dostupna je na repozitorijumu [1]. Klasa `BeamSearcher` implementira naredne čisto virtualne metode iz klase `Searcher`:

- **selectState:** Ova metoda bira jedno stanje za dalje istraživanje. U kontekstu *BSS* algoritma, bira se prvo neistraženo stanje u trenutnom nivou pretrage.
- **update:** Ova metoda obaveštava pretraživač o novim ili uklonjenim stanjima. Koristi se za ažuriranje slojeva pretrage nakon što su stanja razgranata ili uklonjena.
- **empty:** Ova metoda proverava da li su sva stanja istražena, tj. da li pretraživač više nema stanja za istraživanje.

3.1 Modifikacija u implementaciji algoritma

U implementaciji algoritma uvedena je modifikacija u odnosu na prethodno opisanu *BSS* pretragu - dodatni parametar, `beamStackSwapLimit`, koji je uveden kako bi se izbeglo zaglavljivanje u lokalnim maksimumima tokom pretrage. Ovaj parametar omogućava pretraživaču da, nakon određenog broja posećenih stanja u kojima nije pronađeno novo pokrivanje (`statesSinceLastCoveredNew`), pretragu nastavi od nivoa stanja sa dna steka, čime se postiže vraćanje pretrage na više nivoe stabla. Postavljanjem vrednosti parametra `beamStackSwapLimit` na 0, algoritam se ponaša kao standardna *BSS* pretraga. Iako se ovom modifikacijom gubi garancija memorijske složenosti od $O(dw)$, eksperimentalno se pokazalo da nema značajan negativan uticaj na performanse algoritma, dok istovremeno povećava njegovu robusnost u analizi složenijih programa.

3.2 Mera kvaliteta pri izboru stanja

Mera kvaliteta za izbor stanja za naredni nivo pretrage zasnovana je na istoj strategiji kao i podrazumevani *nurs:covnew* (*Non-Uniform Random Search*) algoritam. Ova mera kombinuje dve komponente:

1. Minimalna udaljenost do najbliže nepokrivene instrukcije u grafu kontrole toka.
2. Broj instrukcija izvršenih od poslednje pokrivene nove instrukcije.

Algoritam maksimizuje pokrivenosti koda tako što bira stanja koja verovatnije vode do, do tog trenutka, nepokrivenih delova programa. Pošto manja vrednost za obe komponente daje veći prioritet stanju, one se invertuju kako bi se postigao željeni efekat.

3.3 Pokretanje algoritma u okviru alata KLEE

Alat `KLEE` omogućava korisnicima da prilagode način pretrage kroz veliki broj različitih opcija. Za podešavanje parametara `BeamSearcher` pretrage, pružene su dve nove opcije:

1. **--beam-width**: Ova opcija postavlja širinu pretrage, tj. broj stanja koja se zadržavaju za dalje istraživanje u svakom sloju pretrage. Širina pretrage određuje koliko različitih grana se istražuje u jednom trenutku. Podrazumevana vrednost za ovu opciju je 256.
2. **--beam-stack-swap-limit**: Ova opcija određuje maksimalni broj stanja koja će biti posećena u nizu bez pokrivanja nove instrukcije pre nego što se pretraga vrati na najranije odložena stanja, uzimanjem sloja sa dna steka za dalju pretragu. Postavljanjem vrednosti na 0, ova opcija se isključuje. Podrazumevana vrednost je 1 000 000.

Za pokretanje alata `KLEE` koristeći `BeamSearcher` pretragu sa specifičnim vrednostima za širinu pretrage 128 i maksimalno 500 000 stanja bez pokrivanja nove instrukcije pre prelaska na ranija stanja, može se koristiti naredna komanda:

```
klee --searcher=beam \
     --beam-width=128 \
     --beam-stack-swap-limit=500000 \
     /putanja/do/programa.bc
```

4 Eksperimentalna evaluacija algoritma

Radi utvrđivanja performansi *Beam Stack Search* algoritma, sprovedeni su eksperimenti nad slučajno odabranim podskupom `GNU Coreutils` alata [5], konkretno nad alatima `printf`, `mkdir`, `sha256sum`, `touch` i `sort`. Nad svakim od njih je izvršeno po jedno pokretanje *Beam Stack Search* pretrage i podrazumevane pretrage alata `KLEE` - kombinovane *nurs:covnew* i *Random Path* pretrage.

Kako bi se omogućila adekvatna uporedivost rezultata između *Beam Stack Search* i podrazumevanog algoritma pretrage, sva testiranja su sprovedena u konzistentnim uslovima sa parametrima što sličnijim onim koji su korišćeni u originalnom radu o alatu `KLEE` [3, 7]. *Beam Stack Search* algoritam je pokretan sa podrazumevanim vrednostima parametara `--beam-width` od 256 i `--beam-stack-swap-limit` od 1 000 000. Kao i u originalnom eksperimentu, sve iteracije su trajale po jedan sat, a memorija je bila ograničena na 1GB. Svi eksperimenti su pokrenuti na računaru sa `AMD Ryzen 7730U` procesorom, `32GB` radne memorije, pod `Debian 12 - Bookworm` operativnim sistemom.

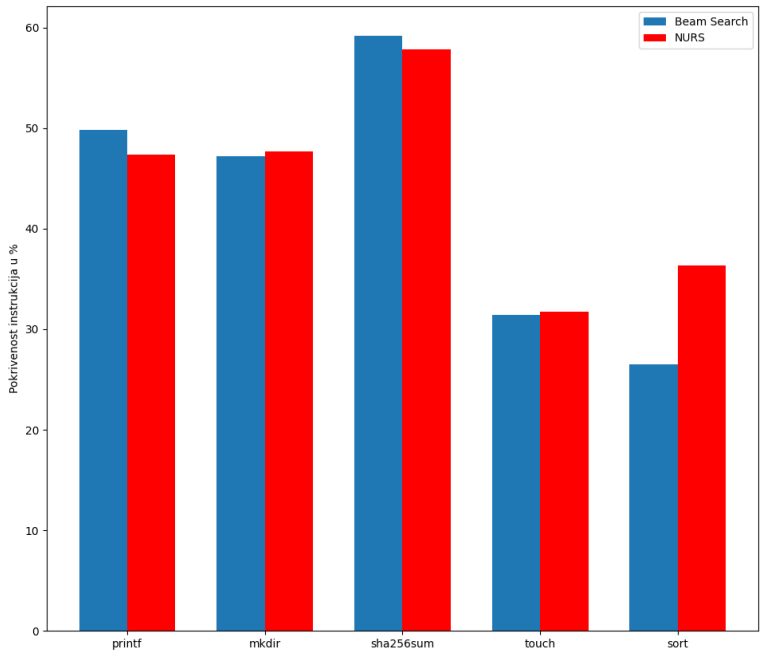
Za deo testiranih programa, *Beam Stack Search* algoritam pokazuje blago poboljšanje u pokrivenosti instrukcija u odnosu na podrazumevani algoritam pretrage. Na primer, kod programa `printf`, *BSS* algoritam postiže 49.81% pokrivenosti instrukcija, što je za 2.46% više u odnosu na podrazumevani algoritam. Slično tome, `sha256sum` beleži poboljšanje od 1.31% sa *BSS* algoritmom.

Međutim, postoji izuzetak kod programa `sort`, gde *BSS* algoritam značajno zaostaje sa pokrivenošću od 26.54%, što je za 9.83% manje u poređenju sa podrazumevanim algoritmom. Za preostala dva programa rezultati su približno isti, uz malu prednost podrazumevanog algoritma pretrage. Rezultati se mogu videti na slici 2 i u tabeli 1.

Program	**nurs ICov (%)**	**beam ICov (%)**	**Razlika ICov (%)**
`printf`	47.35	49.81	2.46
`mkdir`	47.71	47.18	-0.53
`sha256sum`	57.85	59.16	1.31
`touch`	31.71	31.44	-0.27
`sort`	36.37	26.54	-9.83

Tabela 1: Poređenje pokrivenosti instrukcija za *Beam Stack Search* i podrazumevani algoritam pretrage

Kada je reč o pokrivenosti grana, slično kao i kod pokrivenosti instrukcija, *Beam Stack Search* algoritam pokazuje poboljšanje u analizi `printf` i `sha256sum` programa i približno iste rezultate analizi u `mkdir` i `touch` programa. Na primer, kod programa `printf`, *BSS* algoritam povećava pokrivenost grana za 1.32% u poređenju sa podrazumevanim algoritmom. Takođe, kod `sha256sum`, *BSS* algoritam beleži blago poboljšanje od 0.91%. Opet, kod programa `sort`, primećujemo značajno smanjenje pokrivenosti grana, gde *BSS* algoritam postiže samo 17.80%



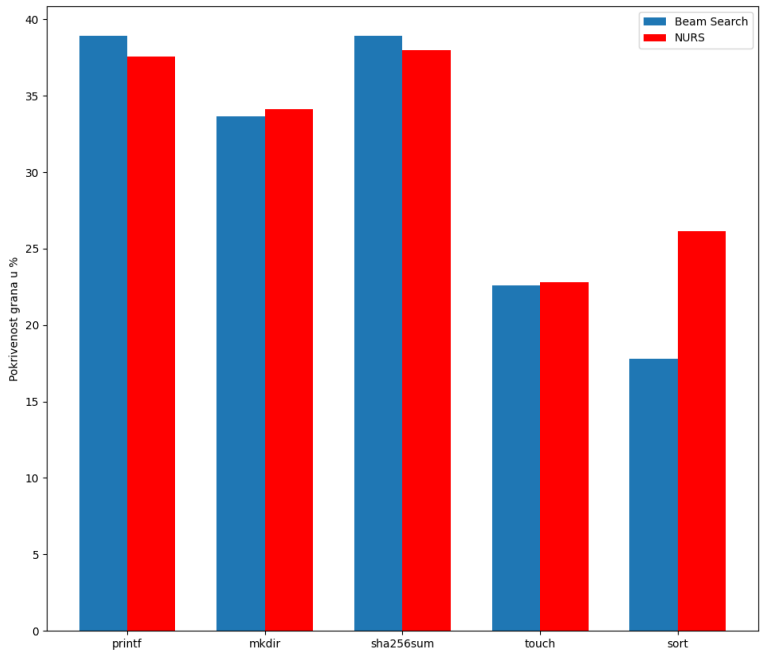
Slika 2: Poređenje pokrivenosti instrukcija za *Beam Stack Search* i podrazumevani algoritam pretrage

pokrivenih grana, što je za 8.35% manje od podrazumevanog algoritma. Rezultati se mogu videti na slici 3 i u tabeli 2.

Program	**nurs BCov (%)**	**beam BCov (%)**	**Razlika BCov (%)**
`printf`	37.59	38.91	1.32
`mkdir`	34.11	33.67	-0.44
`sha256sum`	38.00	38.91	0.91
`touch`	22.83	22.58	-0.25
`sort`	26.15	17.80	-8.35

Tabela 2: Poređenje pokrivenosti grana vrednosti za nurs i beam algoritme

Nad programima `printf`, `sha256sum` i `sort` dodatno je pokrenuta i analiza sa *BSS* algoritmom bez modifikacije sa uzimanjem sloja stanja sa dna steka, tj. sa vrednošću 0 za parametar `--beam-stack-swap-limit`. U sva tri slučaja, dobijena pokrivenost, kako instrukcija, tako i grana, je bila manja nego sa modifikacijom (47.24%, 56.20% i 26.52% za pokrivenost instrukcija i 37.48%, 36.30% i 17.78% za pokrivenost grana, redom za `printf`, `sha256sum` i `sort`).



Slika 3: Poređenje pokrivenosti grana za nurs i beam algoritme

5 Zaključak

Eksperimentalni rezultati pokazuju da *Beam Stack Search* algoritam u velikom broju slučajeva postiže veoma dobre rezultate u kontekstu nivoa pokrivenosti koda - najčešće približno ili čak bolje od podrazumevanog algoritma pretrage alata `KLEE`. Međutim, zaglavljivanje u pretrazi programa velike složenosti je i dalje moguće, što se može videti iz loših rezultata prilikom analize programa `sort`. Jedan mogući način rešavanja ovog problema je kombinovano pokretanje nekog od randomizovanih algoritama pretrage, poput *Random Path Selection* algoritma, zajedno sa *Beam Stack Search* pretragom.

Potencijalno poboljšanje algoritma *Beam Stack Search* u kojem bi se dalje istraživanje moglo kretati je uvođenje nedeterminizma u sam algoritam radi izbegavanja zaglavljivanja, na primer izborom nasumičnog nivoa stanja sa steka umesto isključivo sa dna prilikom korišćenja `--beam-stack-swap-limit` parametra. Dodatno, potrebno je sprovesti dalja istraživanja o ponašanju pretrage u zavisnosti od karakteristika ulaznog programa i vrednosti parametara `--beam-width` i `--beam-stack-swap-limit`.

Literatura

[1] Aleksandar Stefanović, Petar Đorđević. Beam stack search algoritam pretrage u okviru alata za simboličko izvršavanje klee. https://github.com/MATF-Software-Verification/2023_VLC-Media-Player_KLEE/tree/main/klee, 2024.

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[4] Markus Freitag and Yaser Al-Onaizan. Beam search strategies for neural machine translation. In *Proceedings of the First Workshop on Neural Machine Translation*. Association for Computational Linguistics, 2017.

[5] Coreutils - GNU core utilities. https://www.gnu.org/software/coreutils.

[6] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, jul 1976.

[7] OSDI'08 Coreutils Experiments. http://klee-se.org/docs/coreutils-experiments.

[8] B. T. Lowerre. *The Harpy speech recognition system*. PhD thesis, Carnegie Mellon University, Pennsylvania, April 1976.

[9] Clara Meister, Tim Vieira, and Ryan Cotterell. Best-first beam search, 2022.

[10] Strahinja Stanojević. Proširivanje alata klee naprednim algoritmom pretrage stabla izvršavanja programa. Master's thesis, University of Belgrade, Faculty of Mathematics, 2020.

[11] Rong Zhou and Eric A. Hansen. Beam-stack search: integrating backtracking with beam search. In *Proceedings of the Fifteenth International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'05, page 90–98. AAAI Press, 2005.