Verifikacija softvera

— Simboličko izvršavanje —

Milena Vujošević Janičić

Matematički fakultet, Univerzitet u Beogradu

Sadržaj

1 Uvod u simboličko izvršavanje **2**
1.1 Simboličko izvršavanje kroz primer . . . 2
1.2 Istorija, alati, stablo izvršavanja . . . 3
1.3 Primene simboličkog izvršavanja . . . 6
1.4 Izazovi simboličkog izvršavanja . . . 8
1.5 Osnovni pojmovi . . . 11

2 Principi dizajna i konkoličko izvršavanje **11**
2.1 Principi dizajna . . . 11
2.2 Dinamičko simboličko izvršavanje . . . 13
2.3 Selektivno simboličko izvršavanje . . . 16

3 Strategije obilaska puteva **17**
3.1 Naivni pristupi, random strategija . . . 18
3.2 Izvršavanje vođeno pokrivenošću koda . . . 19
3.3 Razne strategije . . . 19
3.4 Izvršavanje unazad . . . 20

4 Modelovanje memorije **21**
4.1 Nizovi i pokazivači — problem . . . 21
4.2 Potpuno simbolička memorija . . . 22
4.3 Kompromisi . . . 27

5 Eksplozija broja stanja i putanja **29**
5.1 Odsecanje nedostižnih putanja . . . 29
5.2 Spajanje stanja . . . 29
5.3 Aproksimacije petlji . . . 30

6 Rešavači. Binarni kôd **31**
6.1 Rešavači . . . 31
6.2 Binarni kôd . . . 32

1 Uvod u simboličko izvršavanje

1.1 Simboličko izvršavanje kroz primer

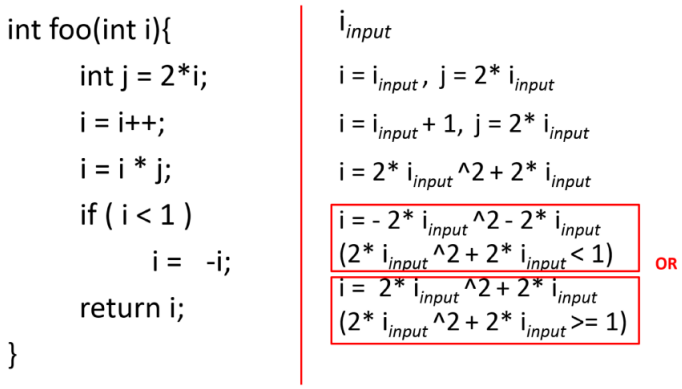
Konkretno izvršavanje

```
int foo(int i){                    | i = 1
        int j = 2*i;               | i = 1, j = 2
        i = i++;                   | i = 2, j = 2
        i = i * j;                 | i = 4, j = 2
        if ( i < 1 )               |
                i =  -i ;          |
        return i;                  | return 4
}                                  |
```

Simboličko izvršavanje

```
int foo(int i){                    | i_input
        int j = 2*i;               | i = i_input , j = 2* i_input
        i = i++;                   | i = i_input + 1, j = 2* i_input
        i = i * j;                 | i = 2* i_input ^2 + 2* i_input
        if ( i < 1 )               |
                i =  -i;           |
        return i;                  |
}                                  |
```

Simboličko izvršavanje



Simboličko izvršavanje

Neformalno...

- Izvršavamo program sa simbolima, tj pratimo simbolička stanja umesto konkretnih ulaza
- Izvršavamo puno putanja simultano, kada izvršavanje neke putanje može da se nastavi na više načina, pravimo nove putanje i dodajemo uslove nad simboličkim vrednostima
- Kada izvršavamo jednu putanju zapravo simuliramo veliki broj testova s obzirom da razmatramo sve ulaze koji prolaze kroz tu istu putanju

1.2 Istorija, alati, stablo izvršavanja

Poreklo ideje...

Tehnika nastala još 70tih godina prošlog veka
Naučni rad koji je najviše citiran: James C. King. 1976. **Symbolic execution and program testing.** Commun. ACM 19, 7 (July 1976), 385-394. DOI=http://dx.doi.org/10.1145/360248.360252 https://yurichev.com/mirrors/king76symbolicexecution.pdf

Poreklo ideje...

Ima i drugih sličnih radova iz tog perioda
Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. **SELECT — a formal system for testing and debugging programs by symbolic execution**. In Proceedings of the international conference on Reliable software. ACM, New York, NY, USA, 234-245. DOI=http://dx.doi.org/10.1145/800027.808445 Lori A. Clarke. 1976. **A program testing system**. In Proceedings of the 1976 annual conference (ACM '76). ACM, New York, NY, USA, 488-491. DOI=http://dx.doi.org/10.1145/800191.805647 Leon J. Osterweil and Lloyd D. Fosdick. 1976. **Program testing techniques using simulated execution**. In Proceedings of the 4th symposium on Simulation of computer systems (ANSS '76), Harold Joseph Highland (Ed.). IEEE Press, Piscataway, NJ, USA, 171-177.

Izazovi

Tek od 2005 — praktična upotreba simboličkog izvršavanja

U trenutku nastanka ideje, nije bilo jasno kako rešiti osnovne probleme koji su se odmah javili. Proboj su napravili alati

- DART — autori Godefroid and Sen, PLDI 2005 (uvođenje dinamičkog izvršavanja u simboličko izvršavanje)
- EXE — autori Cadar, Ganesh, Pawlowski, Dill, and Engler, CCS 2006 (STP: podrška za teoriju nizova)

Alati

Primeri alata zasnovanih na simboličkom izvršavanju

- KLEE (Stanford) — Open source, runs on top of LLVM, found lots of problems in open-source software
- SAGE — Microsoft internal tool, symbolic execution to find bugs in file parsers - e.g., JPEG, DOCX, PPT, etc.
- Pex symbolic execution for .NET
- Cloud9 parallel symbolic execution, also supports threads
- CUTE (UC Berkeley) and jCUTE (symbolic execution for Java)
- Java PathFinder (NASA) - symbolic execution
- S^2E (EPFL) — LLVM based platform
- SymDroid - symbolic execution on Dalvik Bytecode
- Kleenet - testing interaction protocols for sensor network

Alati

Symbolic engine	**References**	**Project URL** (last retrieved: August 2016)
CUTE	[Sen et al., 2005]	–
DART	[Godefroid et al., 2005]	–
jCUTE	[Sen and Agha, 2006]	https://github.com/osl/jcute
KLEE	[Cadar et al., 2006, Cadar et al., 2008]	https://klee.github.io/
SAGE	[Godefroid et al., 2008, Elkarablieh et al., 2009]	–
BitBlaze	[Song et al., 2008]	http://bitblaze.cs.berkeley.edu/
CREST	[Burnim and Sen, 2008]	https://github.com/jburnim/crest
PEX	[Tillmann and De Halleux, 2008]	http://research.microsoft.com/en-us/projects/pex/
Rubyx	[Chaudhuri and Foster, 2010]	–
Java PathFinder	[Păsăreanu and Rungta, 2010]	http://babelfish.arc.nasa.gov/trac/jpf
Otter	[Reisner et al., 2010]	https://bitbucket.org/khooyp/otter/
BAP	[Brumley et al., 2011]	https://github.com/BinaryAnalysisPlatform/bap
Cloud9	[Bucur et al., 2011]	http://cloud9.epfl.ch/
Mayhem	[Cha et al., 2012]	–
SymDroid	[Jeon et al., 2012]	–
S^2E	[Chipounov et al., 2012]	http://s2e.epfl.ch/
FuzzBALL	[Martignoni et al., 2012, Caselden et al., 2013]	http://bitblaze.cs.berkeley.edu/fuzzball.html
Jalangi	[Sen et al., 2013]	https://github.com/Samsung/jalangi2
Pathgrind	[Sharma, 2014]	https://github.com/codelion/pathgrind
Kite	[do Val, 2014]	http://www.cs.ubc.ca/labs/isd/Projects/Kite
SymJS	[Li et al., 2014]	–
CIVL	[Siegel et al., 2015]	http://vsl.cis.udel.edu/civl/
KeY	[Hentschel et al., 2014]	http://www.key-project.org/
Angr	[Shoshitaishvili et al., 2015, Shoshitaishvili et al., 2016]	http://angr.io/
Triton	[Saudel and Salwan, 2015]	http://triton.quarkslab.com/
PyExZ3	[Ball and Daniel, 2015]	https://github.com/thomasjball/PyExZ3
JDart	[Luckow et al., 2016]	https://github.com/psycopaths/jdart
CATG	–	https://github.com/ksen007/janala2
PySymEmu	–	https://github.com/feliam/pysymemu/
Miasm	–	https://github.com/cea-sec/miasm

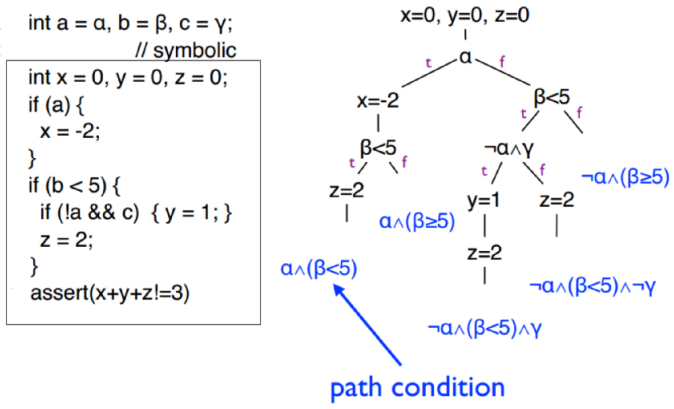
Table 1: Selection of symbolic execution engines, along with their reference article(s) and software project web site (if any).

Simboličko izvršavanje

Stablo izvršavanja

- Izvršavanje programa nad simboličkim vrednostima.
- Simbolička stanja preslikavaju promenljive u simboličke vrednosti.
- Uslov putanje (engl. *path condition*) je formula (bez kvantifikatora) nad simboličkim ulazima koja sadrži sve odluke koje su do te prilike donete
- Sve putanje programa formiraju stablo izvršavanja

Simboličko stablo izvršavanja



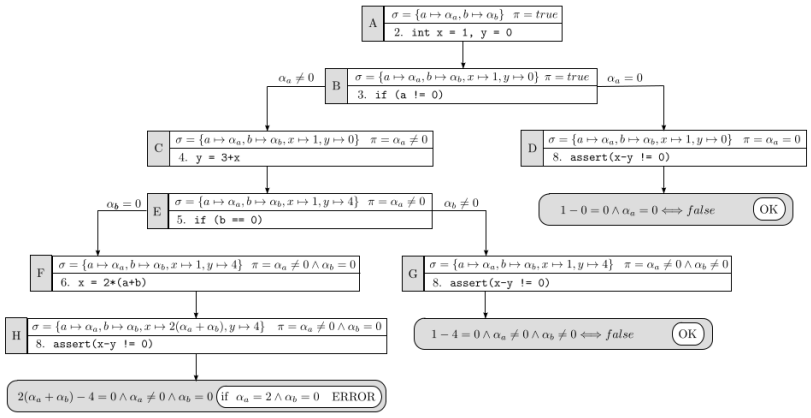
Simboličko stablo izvršavanja

- Program inputs are represented by symbols: $\boldsymbol{\alpha_a}, \boldsymbol{\alpha_b}$.
- Symbolic execution maintains a state $(\boldsymbol{stmt}, \boldsymbol{\sigma}, \boldsymbol{\pi})$:
  - $\boldsymbol{stmt}$: the next statement to evaluate
  - $\boldsymbol{\sigma}$: symbolic store
  - $\boldsymbol{\pi}$: path constraints
- Depending on $\boldsymbol{stmt}$, symbolic execution proceeds as follows:
  - $\boldsymbol{x = e}$: It updates the symbolic store $\boldsymbol{\sigma}$ by associating $\boldsymbol{x}$ with a new symbolic expression $\boldsymbol{e_s}$, where $\boldsymbol{e_s}$ is a symbolic expression obtained by evaluating $\boldsymbol{e}$ symbolically.
  - if $\boldsymbol{e}$ then $\boldsymbol{s_1}$ else $\boldsymbol{s_2}$: It is forked by creating two states with path constraints $\boldsymbol{\pi \wedge e_s}$ and $\boldsymbol{\pi \wedge \neg e_s}$.
  - `assert`$(\boldsymbol{e})$: The validity of $\boldsymbol{e}$ is checked.
    - ⋆ If $\boldsymbol{\neg e \wedge \pi}$ is unsatisfiable, the assertion is always true.
    - ⋆ If $\boldsymbol{\neg e \wedge \pi}$ is satisfiable, an assert-fail input is found.

Primer

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

Simboličko stablo izvršavanja



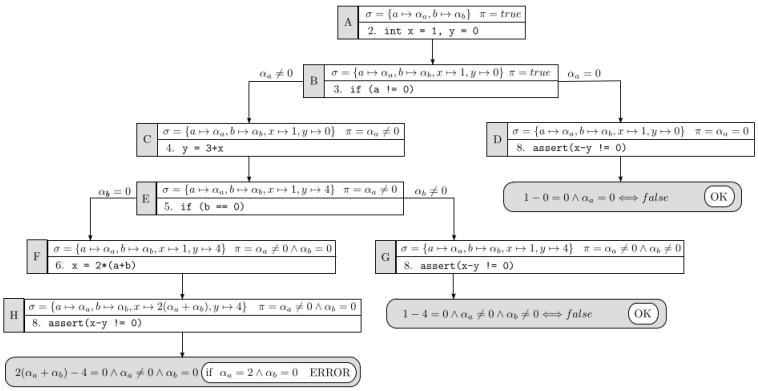
1.3 Primene simboličkog izvršavanja

Simboličko izvršavanje

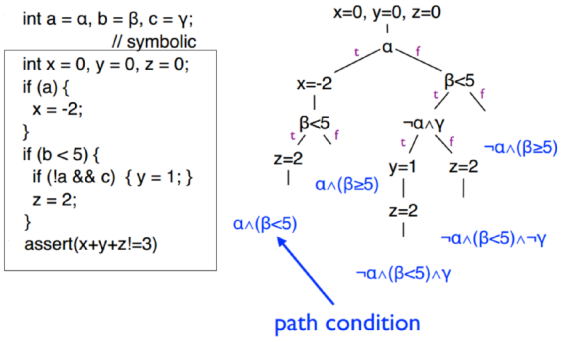
Primena simboličkog izvršavanja

- Pronalaženje grešaka
  - U Majkrosoftu, za vreme razvoja Windows 7, 30% grešaka je bilo otkriveno simboličkim izvršavanjem. Ove greške nisu bile otkrivene drugim analizama koda niti testiranjem
- Generisanje test primera
- Otkrivanje nedostižnih putanja
- Generisanje invarijnati programa, automatske ispravke programa....

Pronalaženje grešaka



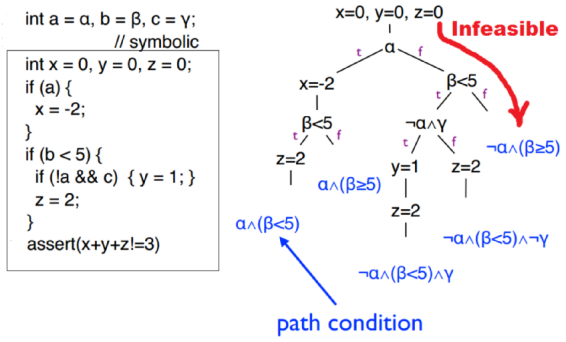
Generisanje test primera



Path 1: $\alpha = 1, \beta = 1$
Path 2: $\alpha = 1, \beta = 6$
Path 3 ...

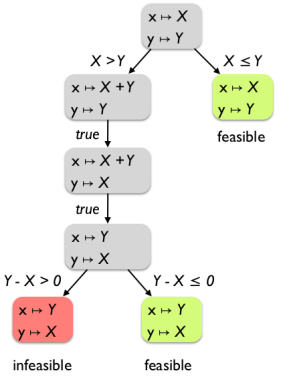
Nedostupne/nedostižne putanje

Suppose we require $\alpha = \beta$



Nedostupne/nedostižne putanje

```
def f (x, y):
  if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0):
      assert false
  return (x, y)
```



1.4 Izazovi simboličkog izvršavanja

Teorija i praksa

Teorija i praksa simboličkog izvršavanja

- Teorijski, simboličkim izvršavanjem možemo generisati sve moguće putanje kroz koje program može da prođe za vreme konkretnog izvršvanja programa na nekim konkretnim ulazima (pod pretpostavkom da se svako konkretno izvršavanje zaustavlja).
- Međutim, iako modelovanje svih mogućih izvršavanja daje mogućnost sprovođenja veoma interesantne analize koda, to je najčešće neizvodivo u praksi, posebno na softveru za realnu upotrebu.
- Postoje razni problemi koji se javljaju u ovom konktestu, a koji ne mogu da se reše praćenjem čisto simboličkog pristupa.

Izazovi simboličkog izvršavanja

Modelovanje hipa i rezonovanje o pokazivačima

- Modelovanje memorije: na koji način simboličko izvršavanje rukuje sa pokazivačima, nizovima i drugim kompleksnim objektima?
- Kôd koji radi sa pokazivačima i strukturama podataka može da stvori potrebu ne samo za simboličkim podacima veći i za adresama koje se opisuju simboličkim izrazima

Izazovi simboličkog izvršavanja

Modelovanje okoline (bibliotečke funkcije, sistemski pozivi)

- Na koji način simboličko izvršavanje interaguje sa pozivima bibliotečkih funkcija i sa sistemskim pozivima koje ne može simbolički da isprati.
- Na koji način simboličko izvršavanje modeluje sadržaj datoteka i sadržaj okoline koja utiče na izvršavanje programa?

Izazovi simboličkog izvršavanja

Eksplozija broja stanja i putanja

- Eksplozija broja putanja i broja stanja: na koji način se simboličko izvršavanje nosi sa eksplozijom broja putanja i eksplozijom novonastalih stanja?
- Jezički konstrukti, kao što su to petlje, mogu eksponencijalno da povećaju broj simboličkih stanja. Zbog toga je malo verovatno da simboličko izvršavanje može da pretraži sva moguća stanja u nekom razumnom vremenu.

Path Explosion

- Exponential in branching structure

```
1. int a = α, b = β, c = γ;    // symbolic
2. if (a) ... else ...;
3. if (b) ... else ...;
4. if (c) ... else ...;
```

  - Ex: 3 variables, 8 program paths

- Loops on symbolic variables even worse

```
1. int a = α;    // symbolic
2. while (a) do ...;
3.
```

  - Potentially 2^31 paths through loop!

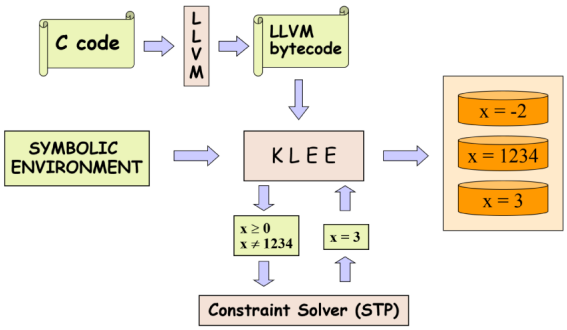
Izazovi simboličkog izvršavanja

Efikasni rešavači — SMT

- Na koji način se rešavaju uslovi koji se generišu simboličkim izvršavanjem?
- SMT rešavači mogu da dobro skaliraju na ograničenjima koja sadrže hiljade promenljivih. Međutim, rešavanje ograničenja koja sadrže nelinearnu aritmetiku ne može da bude prilično neefikasno...

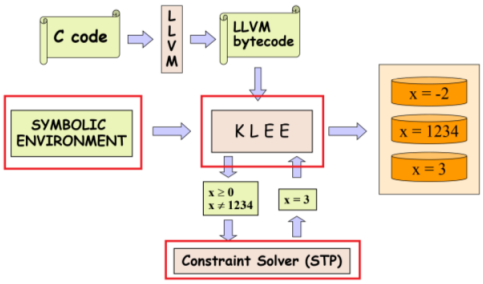
Unutrašnjost alata sa simboličko izvršavanje

Internal of Symbolic Executors: KLEE



Unutrašnjost alata sa simboličko izvršavanje

- Path explosion
- Modeling program statements and environment
- Constraint solving



Izazovi simboličkog izvršavanja

Binarni kôd

- Koji problemi se javljaju prilikom simboličkog izvršavanja binarnog koda?
- U nekim situacijama, nemamo izvorni kôd i binarni kôd je jedina postojeća reprezentacija programa

Izazovi simboličkog izvršavanja

Odluke se prave u zavisnosti od konteksta upotrebe

- U zavisnosti od konteksta u kojem se simboličko izvršavanje koristi, postoje različiti izbori i pretpostavke koje se prave da bi se adresirala prethodna pitanja.
- Iako ovi izbori utiču na saglasnost i kompletnost, u mnogim scenarijima je i delimično istraživanje prostora mogućih izvršavanja dovoljno za postizanje željenog cilja (u okviru zadatog budžeta)

1.5 Osnovni pojmovi

Lažno pozitivni i lažno negativni rezultati

Lažno pozitivan rezultat (lažno upozorenje)
Alat daje **lažno pozitivan rezultat** ili **lažno upozorenje** (engl. *false alarm* ili *false positive*) ukoliko za neku naredbu u kodu prijavi da može da izazove grešku iako ne postoji konkretno izvršavanje koje bi tu grešku izazvalo.

Lažno negativan rezultat
Alat prijavljuje da je sa kodom sve u redu, tj. daje **lažno negativan rezultat** (engl. *false negative*) ukoliko za neku naredbu u kodu ne prijavi da može da izazove grešku iako postoji konkretno izvršavanje koje bi tu grešku izazvalo.

Saglasnost i kompletnost

Saglasnost (engl. *soundness*)
Analiza je saglasna ako nema lažno negativnih rezultata, odnosno, za svaki kôd za koji tvrdi da je ispravan on stvarno i jeste ispravan.

Kompletnost (engl. *completeness*)
Analiza je kompletna ako nema lažno pozitivnih rezultata, odnosno, za svaki kôd za koji tvrdi da nije ispravan on stvarno nije ispravan.

2 Principi dizajna i konkoličko izvršavanje

2.1 Principi dizajna

Principi dizajna

Napredak, ponavljanje posla i ponovno korišćenje rezultata

- **Napredak**: izvršavanje bi trebalo da može da napreduje proizvoljno dugo vreme bez prelaženja zadatih granica resursa. Upotreba memorije je obično kritična, zbog mogućeg velikog broja različitih putanja i stanja
- **Ponavljanje posla**: prilikom izvršavanja posao ne bi trebao da se ponavlja, odnosno treba izbegavati započinjanje programa više puta od samog početka sa ciljem analize različitih putanja koje mogu imati isti prefiks

- **Ponovno korišćenje rezultata**: rezltati analize iz prethodnih pokretanja simboličkog izvršavanja treba da se ponovo iskoriste svuda gde je to moguće. Konkretno, skupi pozivi SMT rešavačima za ograničenja putanja koje su već ranije rešavana treba da se izbegavaju

Principi dizajna

Kako izabrati prioritete u dizajnu?

- *Online* izvršavanje
- *Offline* izvršavanje
- Hibridno izvršavanje (kombinacija prethodna dva)

Izbori dizajna

Online izvršavanje

- Simboličko izvršavanje pokušava da izvršava više putanja simultano. Da bi se ovo ostvarilo kloniraju se stanja na svakom mestu gde može da dođe do grananja (alati KLEE, AEG, S^2E).
- Na ovaj način nikada se ne izvršavaju instrukcije koje su jednom izvršene, odnosno izbegava se ponavljanje posla
- Međutim, veliki broj aktivnih stanja je potrebno održavati u memoriji i veoma lako se memorija prepuni i potrebno je na neki način to regulisati kako ne bi bio ugrožen napredak

Izbori dizajna

Offline izvršavanje

- S druge strane, posao može da se u velikoj meri ponavlja ako svako izvršavanje započinje od početka programa
- Kod *offline* izvršavanja, obično se izvršava konkretno izvršavanje tj program se napre izvršava konretno, prate se instrukcije koje su konkretnim izvršavanjem izvršene i onda se one izvrše i simbolički
- Upotreba memorije je značajno manja, a smanjuje se broj poziva solverima
- Mešanje konkretnog i simboličkog izvršavanja naziva se konkoličko izvršavanje

Konkretno i simboličko izvršavanje

Praktični problemi

- Iscrpno pretraživanje putanja u okviru poziva funkcija eksternih biblioteka vodi do eksponencijalnog rasta broja stanja i može da spreči da analiza dopre do interesantnih delova koda
- Simboličko praćenje poziva eksternih *third-party* komponenti često nije dostupno
- Simboličko izvršavanje često poziva SMT rešavač za vreme analize. Vreme koje se provede u rešavanju ograničenja je jedno od najveći barijera efikasnosti simboličkog izvršavanja, a programi nekada mogu voditi i ka ograničenjima koje čak ni najbolji rešavači ne mogu da reše efikasno

Konkretno i simboličko izvršavanje

Konkoličko izvršavanje (konkretno+simboličko)
Osnovna ideja za rešavanje prethodno pomenutih problema i za omogućavanje praktične upotrebe simboličkog izvršavanja je mešanje konkretnog i simboličkog izvršavanja, odnosno **konkoličko izvršavanje**.

Vrste konkoličkog izvršavanja

- Dinamičko simboličko izvršavanje
- Selektivno simboličko izvršavanje

2.2 Dinamičko simboličko izvršavanje

Dinamičko simboličko izvršavanje

Simboličko izvršavanje vođeno konkretnim vrednostima
Kao dodatak simboličkom skladištu i uslovima putanje, čuva se i **konkretno skladište**. Nakon što se izabere proizvoljni ulaz sa kojim se počinje izvršavanje, program se izvršava i konkretno i simbolički tako što se simultano ažuriraju oba skladišta i uslovi putanje. Kada god konkretno izvršavanje krene nekom konkretnom granom, simboličko izvršavanje se usmerava prema istoj grani i uslovi putanje se ažuriraju u skladu sa uslovom koji je zadovoljen da bi se tom granom krenulo. Ukratko, **simboličko izvršavanje je vođeno specifičnim konkretnim izvršavanjem**. Kao posledica, simbolčko izvršavanje ne mora da poziva rešavač ograničenja da odluči da li je neka grana dostupna, odnosno da li je neka formula koja odgovara uslovima putanje zadovoljiva: to se direktno proverava konkretnim izvršavanjem.

Dinamičko simboličko izvršavanje

Simboličko izvršavanje vođeno konkretnim vrednostima
Da bi se istražile različite putanje, uslovi putanje koji su zadati sa jednim ili više grananja se mogu negirati, i tako dobijena formula se može dati SMT rešavaču da pronađe nove ulaze koji će onda ići tom novom putanjom. Ova strategija se

može ponavljati onoliko puta koliko je to potrebno da se dostigne odgovarajuća pokrivenost.

Pogledati na slajdovima animaciju koja kroz primer objašnjava dinamičko simboličko izvršavanje.

Izbor uslova za negiranje?

Tehnike pretrage
Iako dinamičko simboličko izvršavanje koristi konkretne ulaze da navodi simboličko izvršavanje nekom konkretnom putanjom, i dalje postoji problem izbora uslova koji će biti negiran, odnosno nove putanje koja će svaki naredni put biti istražena. Treba imati u vidu da svako konkretno izvršavanje može da doda nova grananja koja treba da budu posećena. Kako skup grana koje nisu istražene može biti veoma velik, veoma je važno usvojiti efikasnu heuristiku za pretragu.

- DART — DFS pretraga u dubinu
- SAGE — generacijska pretraga
- ...

Izbor početnih vrednosti

Uticaj na pretragu

- Kako se prostor svih mogućih stanja samo parcijalno istraži, početni ulaz igra suštinsku ulogu u učinku celokupnog pristupa.
- Značaj prvog ulaza je sličan onome što se dešava u okviru tehnika rasplinutog testiranja crne kutije (engl. *black-box fuzzing*) i iz tog razloga se alati koji koriste dinamičko simboličko izvršavanje (kao, na primer, alat SAGE), često nazivaju i alatima za rasplinuto testiranje bele kutije (engl. *white-box fuzzing*)

Kako pratiti funkcijske pozive za koje kôd nije dostupan?

```
void foo(int x, int y) {
   int a = bar(x);
   if (y < 0) ERROR;
}
```
(a)

```
void qux(int x) {
   int a = bar(x);
   if (a > 0) ERROR;
}
```
(b)

```
void baz(int x) {
   abs(&x);
   if (x < 0) ERROR;
}
```
(c)

Figure 4: Concolic execution: (a) testing of function `foo` even when `bar` cannot be symbolically tracked by an engine, (b) example of false negative, and (c) example of a path divergence, where `abs` drops the sign of the integer at `&x`.

Pretpostavimo da funkcija `bar` ne može da se isprati konkoličkim izvršavanjem. Na primer, to je funkcija koja je obezbeđena kroz spoljnu komponentu za koju nije dostupan kôd ili je napisana u drugom programskom jeziku

Kako pratiti funkcijske pozive za koje kôd nije dostupan?

Razmotrimo funkciju `foo` na slici 4a

- Ako pretpostvimo da su vrednosti `x = 1` i `y = 2` slučajno izabrane kao početni ulazni parameteri, konkoličko izvršavanje će ispratiti izvršavanje funkcije `bar` koja će vratiti neku vrednost i preskočiće granu koja bi dovela do greške.
- U isto vreme, simboličko izvršavanje prati uslove putanje $\alpha_y \geq 0$ u okviru funkcije `foo`.
- Uslovi grananja u funkciji `bar` nisu poznati konkoličkom izvršavanju.

Kako pratiti funkcijske pozive za koje kôd nije dostupan?

Razmotrimo funkciju `foo` na slici 4a

- Da bi se istražila alternativna putanja, negira se ograničenje putanje u grananju u okviru funkcije `foo`.
- To gneriše, na primer, ulaze $x = 1$ i $y = -4$ koji navode konkretno izvršavanje alternativnom putanjom.
- Sa ovakvim pristupom, obe putanje u `foo` mogu da budu istražene iako funkcija `bar` nije simbolički ispraćena.

Kako pratiti funkcijske pozive za koje kôd nije dostupan?

Razmotrimo funkciju `qux` na slici 4b

- Funkcja `qux`, za razliku od funkcije `foo`, uzima jedan ulazni parametar ali proverava rezultat funkcije `bar` u uslovu grananja
- Iako simbolicko izvršavanje prati uslove putanja u granama u okviru funkcije `qux`, ne postoji garancija da će ulaz moći da navede izvršavanje alternativnim putem: relacija između `a` i `x` je nepoznata konkoličkom izvršavanju jer se izvršavanje funkcije `bar` ne prati simbolički
- U ovom slučaju, izvršavanje koda može da bude pokrenuto puno puta sa različitim random generisanim ulazima, ali na kraju nemamo garanciju da ćemo uspeti tj. veoma je moguće da nećemo uspeti da istražimo interesantne putanje kroz funkciju `foo`.

Kako pratiti funkcijske pozive za koje kôd nije dostupan?

Razmotrimo funkciju `baz` na slici 4c

- Funkcija `baz` poziva eksternu funkciju `abs` koja računa apsolutno vrednost broja
- Biranjem $x = 1$ za ulazni parametar konkretnog izvršavanja, konkretno izvršavanje ne uzrokuje grešku, ali simboličko izvršavanje beleži uslov putanje $\alpha_x \geq 0$ na osnvou grananja u funkciji `baz` i pokušava da generiše novi izlaz njegovim negiranjem, npr $x = -1$

- Međutim, novi ulaz ne prolazi kroz alternativnu putanju zbog uticaja funkcije `abs`
- U ovom slučaju, nakon generisanja novog ulaza izvršavanje detektuje **divergenciju putanje**, tj. konkretno izvršavanje koje ne prati predviđeni put
- U ovom primeru, ne postoji ulaz koji može da prođe kroz putanju koja uzrokuje grešku, ali konkoličko izvršavanje nije u stanju da detektuje ovu osobinu

Posledice

Simboličko izvršavanje može da ima lažne negativne rezultate

- Kao što je pokazano primerima, lažno negativni rezultati (tj. promašene putanje) ili divergencija putanja su značajne loše osobine dinamičkog simboličkog izvršavanja.
- Dinamičko simboličko izvršavanje žrtvuje saglasnost pristupa zarad performansi i lakše implementacije: propuštene greške su moguće jer neka izvršavanja programa, a samim tim i moguća pogrešna ponašanja, mogu da budu propuštena
- Analiza ostaje kompletna

2.3 Selektivno simboličko izvršavanje

Selektivno simboličko izvršavanje S^2E

Drugačiji pristup mešanju simboličkog i konkretnog izvršavanja

- Konkretno i simboličko izvršavanje se može kombinovati na različite načine
- Jedan pristup koji je drugačiji od dinamičkog simboličkog izvršavanja je selektivno simboličko izvršavanje
- Osnovna ideja je da neko može da želi da istraži samo neke delove koda u potpunosti, tj. simbolički, a da pritom nije zainteresovan za neke druge delove koda
- Selektivno simboličko izvršavanje prepliće konkretno i simboličko izvršavanje održavajući pritom analizu smislenom
- Na primer, pretpostavimo da funkcija A poziva funkciju B i da se način izvršavanja prilikom poziva menja

Od konkretnog prema simboličkom i nazad

Konkretno izvršavanje A, simboličko B
Argumenti funkcije B postaju simbolički i B se istražuje u potpunosti simbolički. Takođe, B se izvršava i konkretno i konkretan rezultat se vraća u A tako da kada se završi istraživanje funkcije B, funkcija A nastavlja normalno dalje sa radom.

Od simboličkog prema konkretnom i nazad

Simboličko izvršavanje A, konkretno B
Argumenti funkcije B se konkretizuju i B se izvršava konkretno, a zatim izvršavanje u okviru A se nastavlja simbolički. Ovakav pristup može da utiče i na saglasnost

i na kompletnost analize.

Saglasnost

Lažno negativni rezultati

- Konkretizacija može da uzrokuje da se u simboličkom izvršavanju preskoče grane koje su dostižne nakon što se vratimo u A, što može da dovede do lažno negativnih rezultata.
- Da bi to izbegli, skupljena ograničenja se markiraju kao soft ograničenja: kada god neka grana, nakon vraćanja u A bude obeležena kao nedostižna zbog soft ulsova, izvršavanje radi bektreking i bira na drugi način argumente za B. Za vođenje nove konkretizacije argumenata za B, u tom slučaju se koriste uslovi grana koji su skupljeni za B i biraju se one konkretne vrednosti koje omogućavaju drugačije konkretno izvršavanje kroz B.

Kompletnost

Lažna upozorenja
Da bi bili sigurni da simboličko izvršavanje preskače sve putanje koje nisu dostižne u skladu sa izvršenom konkretizacijom, tj da ne bi imali lažno pozitivne rezultate, potrebno je skupiti sva ograničenja putanje koja je izvršena u skladu sa odgovarajućom konkretizacijom, sve bočne efekte koje je B napravila kao i povratnu vrednost koju B proizvodi.

3 Strategije obilaska puteva

Strategije obilaska puteva

Heuristike vođene ciljevima

- Kako obilzak svih mogućih putanja programa može da bude preskup ili vremenski/memorijski nedostižan, u okviru raznih aktivnosti vezanih za testiranje i debagovanje pretraga se navodi tako što se ispituju najpre putanje za koje deluje da najviše obećavaju.
- Postoje razne strategije za izbor naredne putanje koja će biti istražena i sada ćemo dati pregled nekih koje se koriste.
- Heuristike za izbor putanja se obično kroje da pomognu u ostvarivanju specifičnih ciljeva.

Pronalaženje univerzalno optimalne strategije je otvoren istraživački problem.

3.1 Naivni pristupi, random strategija

Najčešće strategije: DFS i BFS

Naivne tehnike: DFS i BFS
Tehnike zasnovane na strukturi koda

***Depth-first search* (DFS)**

- Pretraga u dubinu: prati se putanja dok je god to moguće pre nego što se uradi *backtracking* na najdublju neistraženu granu
- DFS se primenjuje kada je prioritet smanjena upotreba memorije, ali često biva uhvaćena putanjama koje sadrže petlje ili rekurzivne pozive

Najčešće strategije: DFS i BFS

***Breadth-first search* (BFS)**

- Pretraga u širinu, prati sve putanje paralelno
- Uprkos većem zauzeću memorije i velikom vremenu koje je potrebno da se neka putanja završi do kraja, neki alati koriste BFS jer on omogućava da se brzo istraže skroz različite putanje i da se rano detektuju interesantna ponašanja sistema
- S druge strane, ako krajnji cilj zahteva da se kompletno istraži jedna ili više putanja, BFS može da zahteva jako puno vremena

Random strategija

Random izbor putanje

- Kako se sprovodi random pretraga?
  - Ideja 1: Izaberi sledeću putanju za istraživanje random metodom
  - Ideja 2: Random metodom restartuj pretragu ukoliko se ništa novo ne dešava već neko vreme
  - Ideja 3: Kada imamo da istražimo dve jednako prioritetne putanje, izaberi sledeću random
  - ...
- Problem sa reprodukovanjem, pesudo-random se koristi i čuva se *seed*

Random strategija

Random izbor putanje sa težinama

- KLEE dodeljuje težine putanjama koje se zasnivaju na njihovoj dužini i na arnosti njihovog grananja: favorizuje putanje koje su manji broj puta bile istraživane čime sprečava zaglavljivanje u petljama i u drugim uzrocima eksplozije broja putanja

3.2 Izvršavanje vođeno pokrivenošću koda

Izvršavanje vođeno pokrivenošću koda

Maksimizovati pokrivenost
Izaberi putanju koja će najverovatnije da izvrši neku novu instrukciju.

- Pokušaj da posetiš instrukcije koje ranije nisu bile izvršavane.
- Ukoliko takve putanje nema, izaberi onu putanju u kojoj su instrukcije izvršavane najmanji broj puta.

Dobra osobina: greške su često u delovima programa koji se retko izvršavaju, a ova strategija pokušava da dopre svuda.

Izvršavanje vođeno pokrivenošću koda — stanja

Davanje težina stanjima
Izvršavanje vođeno pokrivenošću koda postoji u različitim alatima, na primer i u okviru alata KLEE. Za svako stanje, izračunava se težina koja se koristi prilikom random izbora stanja sa kojim se izvršavanje nastavlja. Težina se izračunava razmatranjem koliko daleko se nalazi instrukcija koja nije pokrivena, da li je novi kôd otkriven iz tog stanja relativno skoro, kao i iz kojih stanja se došlo do datog stanja

Izvršavanje vođeno pokrivenošću koda — putanje

Pretraga vođena pod-putanjama
Pretraga vođena pod-putanjama pokušava da istražuje putanje kojima je manji broj puta prolaženo tako što bira pod-putanje grafa kontrole toka koje su obilažene manji broj puta. Ovo se ostvaruje čuvanjem informacija o istraženim pod-putanjama, gde se pod-putanja definiše kao uzastopna sekvenca stanja dužine n. Veličina n igra ključnu ulogu za ostvarivanje dobre pokrivenosti koda ovom heuristikom, ali ne postoji jedna specifična vrednost n koja je univerzalno optimalna.

3.3 Razne strategije

Izvršavanje vođeno najkraćim rastojanjem

Izvršavanje vođeno najkraćim rastojanjem (engl. *shortest-distance symbolic execution*)
Ova vrsta izvršavanja nema za cilj povećanje pokrivenosti koda već pronalaženje ulaznih parametara koji će uzrokovati izvršavanje neke izabrane tačke u programu. Ova heuristka se zasniva, slično kao i izvršavanja vođena pokrivenošću koda, na metrikama za evaluaciju najkraćeg rastojanja do ciljne tačke. Ovo rastojanje se računa kao dužina najkraće putanje u grafu intereproceduralne kontrole toka i putanjama koje imaju najkraću distancu se daje prioritet.

Generacijska pretraga (SAGE)

Generacijska pretraga
Hibrid DFSa i izvršavanja vođenog pokrivenošću koda

- Generacija 0: Izvrši random putanju do kraja
- Generacija 1: Uzmi sve putanje iz generacije 0, negiraj jedan uslov tako da vodi do novog prefiksa putanje, nadji rešenje te putanje i onda je izvrši
- ...
- Generacija N: slično, samo što se grananje vrši u odnosu na putanju generacije N-1 (za biranje putanje se koristi heuristika pokrivenosti koda)

Kombinovana strategija

Pretraga istovremeno sa različitim algoritmima

- Izvršavaj više različitih algoritama istovremeno i alterniraj između njih
- Zavisi od uslova koji su potrebni da se pronađe greška u kodu, ponaša se kao najbolji od njih, sa konstantnim faktorom izgubljenog vremena i memorije sa svim ostalim algoritmima
- Mogu se koristiti različiti algoritmi da bi se došlo do različitih delova programa

3.4 Izvršavanje unazad

Simboličko izvršavanje unazad

Simboličko izvršavanje unazad (engl. *Symbolic Backward Execution* (SBE))

- Simboličko izvršavanje unazad je varijanta simboličkog izvršavanja u kojem izvršavanje počinje od ciljne tačke prema tački ulaza u program, tj analiza se izvršava u obrnutom smeru.
- Osnovni cilj ovog pristupa je da se napravi test primer koji uzrokuje izvršavanje specifične linije koda (obično nekog *assert*-a ili neke *throw* naredbe)
- Ovo je takođe veoma korisno za debagovanje ili regresiono testiranje

Izvršavanje unazad (SBE)

Sličnost sa običnim simboličkim izvršavanjem

- Kako izvršavanje počinje od ciljne linije koda, ograničenja putanje se skupljaju po grananjima unazad.
- Više putanja se istražuje u jednom trenutku i kao kod običnog simboličkog izvršavanja, putanje se povremeno proveravaju da li su dostižne.
- Ako putanja nije dostižna, ona se odbacuje i radi se *backtracking*.

Izvršavanje unazad (CCSBE)

Call-chain backward symbolic execution (CCBSE)

- Tehnika započinje utvrđivanjem validne putanje u okviru funkcije gde je ciljna linija locirana.
- Kada se putanja pronađe, pomeramo se na funkciju pozivaoca ove funkcije i pokušavamo da rekonstruišemo validnu putanju od njenog ulaza do poziva funkcije u kojoj je ciljna linija koda.
- Proces se rekurzivno nastavlja dok ne dođemo do *main* funkcije
- Osnovna razlika između SBE i CCSBE je što se u okviru svake funkcije za CCSBE izvršava obično simboličko izvršavanje dok se za SBE izvršava unazad.

Izvršavanje unazad

Ograničenja

- Da bi izvršavanje unazad moglo da se primeni, potrebno je da postoji na raspolaganju inter-proceduralni CFG (control-flow graph) koji obezbeđuje tok kontrole za ceo program i omogućava da se odrede mesta poziva svih funkcija koje učestvuju u istraživanju.
- Nažalost, konstruisanje takvog grafa često je vrlo složen posao u praksi.
- Dodatno, svaka funkcija može biti pozvana sa više mesta u kodu što dodatno otežava (usporava) pretragu.

4 Modelovanje memorije

4.1 Nizovi i pokazivači — problem

Modelovanje memorije

Nizovi i pokazivači

- Primeri koje smo razmatrali su koristili jednostavnu memoriju koja sve podatke čuva kao skalarne vrednosti, tj. bez redirekcije.
- Važan aspekt simboličkog izvršavanja je modelovanje memorije tako da podrži programe sa pokazivačima i nizovima.
- To zahteva proširivanje memorije tako da preslikava ne samo promenljive u njihove simboličke vrednosti, već i memorijske adrese u simbličke izraze ili konkretne vrednosti.

Modelovanje memorije

Simboličko skladište σ

- Simboličko skladište σ može se posmatrati kao preslikavanje koje adresama u memoriji (umesto samim promenljivama) dodeljuje konkretne ili simboličke vrednosti.
- Na ovaj način i dalje možemo da podržavamo obične promenljive koristeći njihove adrese umesto imena u tom preslikavanju. Na primer, umesto da se x preslikava u izraz e (u zapisu $x \to e$) možemo da to razmatramo zapravo kao $\&x \to e$ gde je $\&x$ konkretna adresa promenljive x.
- Takođe, ako je v niz i c konstanta, onda za $v[c] \to e$ zapravo razmatramo $\&v + c' \to e$ (c' je odgovarajući pomeraj za konstantu c)

Modelovanje memorije

Problem simboličkih adresa

- Za konkretne vrednosti to ne predstavlja problem (tj. kada je indeks konstanta) ali izazov je obraditi situacija kada je indeks simbolički izraz. To se naziva **problem simboličkih adresa**
- Izbor modelovanja memorije je važan izbor za dizajn simboličkog izvršavanja jer utiče direktno i na ostvarenu pokrivenost koda istraživanjem putanja i na skalabilnost rešavača.
- Postoje različiti pristupi rešavanju ovog problema.

4.2 Potpuno simbolička memorija

Potpuno simbolička memorija (engl. fully symbolic memory)

Simbolička memorija

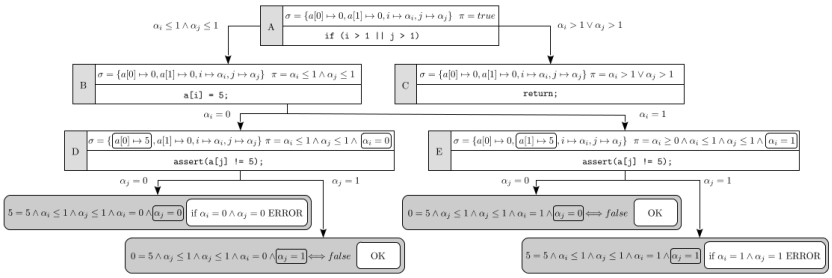
- Kao najopštiji pristup, memorija može da se tretira potpuno simbolički
- Dva osnovna pristupa, koja je još King opisao u svom radu iz '76. godine su
  1. Pravljenje novih stanja
  2. Pravljenje *if-then-else* formula
- Teorija nizova — SMT rešavači

Pravljenje novih stanja

Pravljenje novih stanja (engl. state forking)
Ako operacija čita ili piše na neku simboličku adresu, prave se nova stanja razmatranjem svih mogućih stanja koje mogu da rezultuju kao posledica te operacije. Uslovi putanje se dopunjavaju za svako novokreirano stanje. Primer:

```
1.  void foobar ( unsigned i, unsigned j ) {
2.    int a[2] = { 0, 0 };
3.    if (i>1 || j>1) return;
4.    a[i] = 5;
5.    assert (a[j] != 5);
6.  }
```

Pravljenje novih stanja — primer



***if-then-else* formule**

***ite* formule**

- Alternativni pristup: umesto kreiranja novih stanja, prebaciti posao pretrage na rešavač
- Pristup se sastoji u enkodiranju neodređenosti mogućih vrednosti simboličkog pokazivača u izraze koji se čuvaju u okviru simboličkog skladišta i ograničenja putanja, bez pravljenja novih stanja.
- Osnovna ideja je da se iskoristi mogućnost rešavača da rezonuju o formulama koje sadrže *if-then-else* izraze, tj. u formi *ite(cond, trueExp, falseExp)*

***if-then-else* formule**

ite(cond, trueExp, falseExp)

- Pristup radi drugačije za operacije čitanja i pisanja.
- Neka je α simbolička adresa koja može da ima konkretne vrednosti a_1, a_2 ...
  - čitanje sa lokacije α podrazumeva izraz

$$ite(\alpha = a_1, \sigma(a_1), ite(\alpha = a_2, \sigma(a_2), ...))$$

– pisanje izraza e na lokaciju α menja simboličko skladište σ tako što svaku vrednost promenljivih na adresama a_1, a_2 ... postavlja na sledeći način:

$$\sigma(a_i) = ite(\alpha = a_i, e, \sigma(a_i))$$

if-then-else formule

ite(cond, trueExp, falseExp)

- Na primer, neka je α simbolička adresa koja može imati konkretne vrednosti a_1, a_2, a_3 i a_4
  - čitanje sa lokacije α podrazumeva izraz

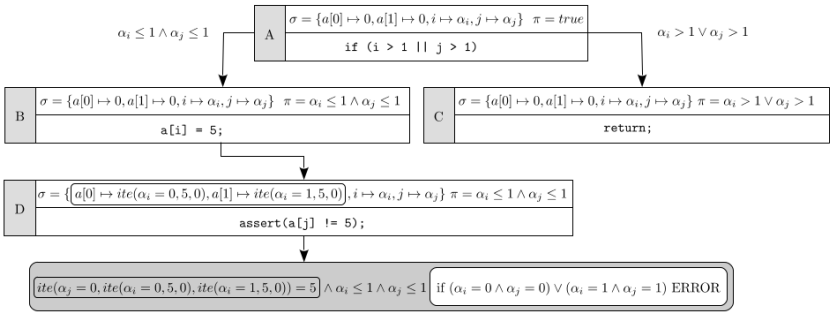
$$ite(\alpha = a_1, \sigma(a_1), ite(\alpha = a_2, \sigma(a_2), ite(\alpha = a_3, \sigma(a_3), \sigma(a_4))))$$

  - pisanje izraza e na lokaciju α menja simboličko skladište σ na sledeći način:

$$\sigma(a_1) = ite(\alpha = a_1, e, \sigma(a_1)) \qquad \sigma(a_2) = ite(\alpha = a_2, e, \sigma(a_2))$$

$$\sigma(a_3) = ite(\alpha = a_3, e, \sigma(a_3)) \qquad \sigma(a_4) = ite(\alpha = a_4, e, \sigma(a_4))$$

if-then-else formule — primer



Teorija nizova

Teorija nizova (engl. theory of arrays)

- Kada se koristi teorija nizova u okviru SMT rešavača, to nam omogućava da operacije nad nizovima jednostavno izrazimo tj i oni su *građani prvog reda* u okviru formula
- *Teorija nizova* (eng. theory of arrays, skraćeno ARR) uvodi ternarni funkcijski simbol *store* (nekada se naziva i *write*) i binarni funkcijski simbol *select* (nekada se naziva i *read*).

SMT theory of arrays

Funkcija *store...*
... vrši izmenu vrednosti na odgovarajućem mestu u nizu i ima tri argumenta:

(i) niz,

(ii) poziciju (indeks) u nizu na koju se smešta vrednost,

(iii) vrednost koja se smešta u niz.

Za zadati niz a, celobrojnu vrednost i i vrednost v tipa nad kojim je niz definisan, term $store(a, i, v)$ označava niz koji je identičan sa nizom a, osim što je vrednost na poziciji i jednaka v.

SMT teorija nizova

Funkcija *select...*
... vrši čitanje vrednosti sa odgovarajućeg mesta u nizu i ima dva argumenta:

(i) niz,

(ii) poziciju (indeks) u nizu sa koje se čita vrednost.

Za zadati niz a i celobrojnu vrednost i, term $select(a, i)$ označava vrednost na poziciji i niza a.

Teorija nizova

Aksiome
Teorija nizova je teorija koja ima sledeće dve aksiome:

$$\forall a\ \forall i\ \forall v\ (select(store(a, i, v), i) = v) \qquad \text{(A1)}$$

$$\forall a\ \forall i\ \forall j\ \forall v\ (i \neq j \Rightarrow select(store(a, i, v), j) = select(a, j)) \qquad \text{(A2)}$$

NP-kompletan problem

Odlučiv i NP-kompletan
Univerzalno kvatnifikovani fragment ove teorije je odlučiv i problem zadovoljivosti u ovom fragmentu teorije je NP-kompletan.

Aksioma *proširivanja*
Pored prethodne dve aksiome, ponekad se dodaje i aksioma *proširivanja* (engl. *extensionality*)

$$\forall a\ \forall b\ ((\forall i\ (select(a, i) = select(b, i)) \Rightarrow a = b) \qquad \text{(A3)}$$

Primer zadovoljive formule

Primer 4.1. *Naredna formula je primer zadovoljive formule teorije nizova:*

$$a' = store(a, 5, select(a, 3)) \ \wedge\ select(a, 3) = select(a', 5).$$

Intuitivno: niz a' nastaje kada se na petom mestu niza a upiše vrednost koja se nalazi na trećem mestu niza a (npr: `a[5] = a[3];`), i vrednost na trećem mestu niza a i vrednost na petom mestu niza a' su jednake.

Primer nezadovoljive formule

Primer 4.2. *Naredna formula je primer nezadovoljive formule teorije nizova:*

$$a = b \ \wedge \ i = j \wedge \ store(a, i, x) \neq b$$
$$select(b, i) = y \wedge \ select(store(b, i, x), j) = y$$

Da bi se dokazala nezadovoljivost ove formule, potrebno je koristiti i aksiomu (A3).

Intuitivno, ova formula govori da su a i b nizovi sa svim jednakim vrednostima ($a = b$), i da su i i j jednake pozicije u tim nizovima ($i = j$). Ako na poziciji i niza a upišemo vrednost x, ta vrednost x menja niz a ($store(a, i, x) \neq b$). Takođe, na poziciji i niza b se nalazi vrednost y. Iz ovoga možemo da zaključimo da je $x \neq y$. Zbog toga ne može da važi da kada se izabere vrednost sa pozicije na koju je upisana vrednost x da to bude vrednost y ($select(store(b, i, x), j) = y$).

Primer modelovanja

```
a[i] = 5;
b[i] = a[i]+1;
c[i] = 100 / (a[i] - b[i]);
```

$$a \rightarrow store(a, i, 5)$$
$$b \rightarrow store(b, i, select(store(a, i, 5), i) + 1)$$

ili, sa uvođenjem novih promenljivih a' i b'

$$a \rightarrow a' \ \wedge \ a' = store(a, i, 5)$$
$$b \rightarrow b' \ \wedge \ b' = store(b, i, select(a', i) + 1)$$

Primer modelovanja

```
a[i] = 5;
b[i] = a[i]+1;
c[i] = 100 / (a[i] - b[i]);
```

Provera ispravnosti: `a[i] - b[i] != 0` odnosno `a[i] != b[i]`?

$$select(a', i) \neq select(b', i)$$

odnosno

$$select(store(a, i, 5), i) \neq select(store(b, i, select(store(a, i, 5), i) + 1), i)$$
$$5 \neq select(store(b, i, 5 + 1), i)$$

$5 \neq 6$

Simbolička memorija

Ograničenja

- Zahvaljujući svojoj opštosti, puna simbolička memorija podržava najpreciznije opis stanja memorije i ponašanja programa i uzima u obzir sve moguće manipulacije sa memorijom
- U mnogim praktičnim scenarijima upotrebe, skup mogućih adresa sa kojima memorija operiše je mali, što dozvoljava preciznu analizu korišćenjem razumne količine resursa
- Međutim, u opštem slučaju, simbolička adresa može da referencira bilo koju ćeliju memorije, što vodi do velike ekplozije broja stanja koju nije moguće u razumnom vremenu ispratiti. Iz tih razloga, postoji veliki broj tehnika koje su dizajnirane sa ciljem da se poboljša skalabilnost upotrebe teorije nizova i simboličke memorije

4.3 Kompromisi

Umesto simboličke memorije

Kompromisi (engl. *trade-off*)

- Kako puna simbolička memorija ne skalira dobro, potrebno je napraviti neki kompromis
- **Otvoren problem**

Konkretna memorija

Konkretizacija simboličkih adresa

- Ukoliko je kombinatorna kompleksnost analize prevelika jer vrednosti pokazivača ne mogu da se ograniče na dovoljno male opsege, često se koristi strategija **konkretizacije simboličke pokazivačke vrednosti** u neku konkretnu vrednost.
- U ovom slučaju imamo gubitak saglasnosti zarad dobijanja na performansama
- Ovo može da smanji broj stanja i kompleksnost formula koje se daju rešavaču i na taj način da poboljša vreme izvršavanja, ali može da uzrokuje i da budu preskočene neke putanje koje zavise od nekih specifičnih vrednosti pokazivača.

Konkretna memorija

Konkretizacija simboličkih adresa

- Konkretizacija se prirodno javlja u *offline* simboličkom izvršavanju

- Konkretizacija pokazivačke vrednosti tipa T* u NULL ili u adresu novoalociranog objekta veličine sizeof(T). Ovaj izbor može da ide random (DART) ili da imamo različita stanja za različite izbore (CUTE - prvo proba sa NULL, pa zatim sa konkretnom adresom). Ako je T struktura, ista konkretizacija se rekurzivno sprovodi na sva polja na koja pokazuje objekat.

Parcijalno modelovanje memorije

Između simboličke i konkretne memorije

- Da bi se prevazišli problemi skaliranja pune simboličke memorije, a da bi se preskočio gubitak informacija koji nastaje konkretizacijom, jedan pravac je parcijalno modelovanje memorije.
- Ključna ideja je da adrese na koje se piše uvek budu konkretizovane, a da adrese sa kojih se čita se modeluju simbolički ako je neprekidan interval svih mogućih vrednosti koje se mogu pretpostaviti dovoljno mali

Parcijalno modelovanje memorije

Između simboličke i konkretne memorije

- Kompromis: korišćenje izražajnih formula u odnosu na konkretizaciju, jer se enkodiraju višestruke vrednosti pokazivača po stanju, ali se ne pokušava enkodiranje svih simboličkih vrednosti kao što je to u punom simboličkom modelovanju memorije
- Osnovni pristup vezivanja skupa mogućih vrednosti koje mogu da se pretpostave za adresu je pokušavanje upotrebe različitih konkretnih vrednosti i proveravanje da li one zadovoljavaju tekuće uslove putanje. Na taj način se mogu isključiti veliki delovi adresnog prostora pri svakom pokušaju, sve dok se ne nađe odgovarajući dovoljno uzan prostor
- Ovaj algoritam zvuči logično i jednostavno, ali zapravo ima veliki broj dodatnih pretpostavki i posledica koje treba da se uzmu u razmatranje

Lenja inicijalizacija

Kompleksne strukture podataka u C++-u i Javi

- Simboličko izvršavanje u prisustvu struktura podataka kao što su liste i drveta je značajno teže.
- Koristi se okvir verifikacije softvera koji kombinuje simboličko izvršavanje i proveravanje modela
- Uopštavanje simboličkog izvršavanja uvođenjem lenje inicijalizacije za efikasno rukovanje sa dinamički alociranim objektima
- Kombinovanje lenje inicijalizacije sa korisnički obezbeđenim preduslovima, tj. uslovima koji se pretpostavlja da su tačni pre izvršavanja metoda

5 Eksplozija broja stanja i putanja

Eksplozija broja stanja i putanja

Osnovni izazov simboličkog izvršavanja

- Jedna od osnovnih izazova simboličkog izvršavanje je eksplozija broja putanja i broja stanja: simboličkim izvršavanjem mogu se praviti nova stanja na svakom grananju programa i ukupan broj stanja vrlo lako postaje eksponencijalan po broju grananja.
- Praćenje velikog broja stanja i grana koje treba da budu istražene utiču i na vreme izvršavanja i na ukupnu potrošnju memorije

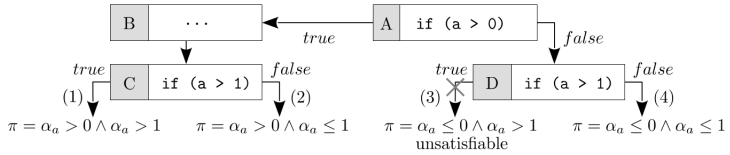
5.1 Odsecanje nedostižnih putanja

Odsecanje nedostižnih putanja

Odsecanje nedostižnih putanja

Broj stanja se može redukovati pozivom SMT rešavača da detektuje nedostižna stanja (putanje). Na primer, za kôd

```
if (a > 0) {...}
if (a > 1) {...}
```



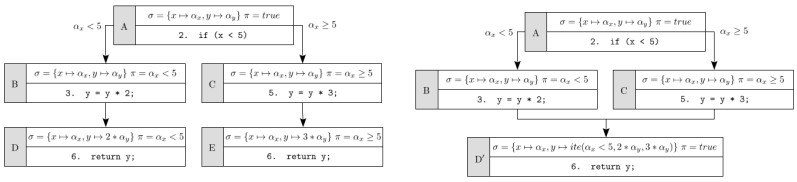
Poziv rešavača kod svakog grananja (kako se ne bi formirale putanje koje su nedostižne) vs poziv rešavača povremeno (štednja poziva rešavaču jer su pozivi rešavača skupi)

5.2 Spajanje stanja

Spajanje stanja

Tehnika spajanja putanja u jedno stanje - *ite* formule

```
1. int foo(int x, int y) {
2.   if(x < 5)
3.     y = y * 2;
4.   else
5.     y = y * 3;
6.   return y;
7. }
```



Spajanje stanja

Spajanje stanja

- Ako imamo dva stanja $(stmt, \sigma_1, \pi_1)$ i $(stmt, \sigma_2, \pi_2)$ spojeno stanje je

$$(stmt, \sigma', \pi_1 \vee \pi_2)$$

  pri čemu σ' spaja stanja σ_1 i σ_2 sa *ite* izrazima

- Kompromis: spajanje smanjuje broj putanja koje treba istražiti ali prebacuje i otežava posao rešavaču
- Postoje razne heuristike za određivanje kada spajati stanja

5.3 Aproksimacije petlji

Aproksimacije petlji

Primer
Razmotrimo naredni kod:

```
void f (unsigned int n) {
  i = 0;
  while (i < n) {
    i = i + 1;
  }
}
```

Simboličko izvršavanje može da pravi nova stanja za sve moguće vrednosti promenljive `n`

Aproksimacija petlji

Jedno rešenje: razmotavanje petlje fiksiran broj puta

```
void f (unsigned int n) {
  i = 0;
  if (i < n) {
    i = i + 1;
  }
  if (i < n) {
    i = i + 1;
  }
}
```

Na ovaj način gubi se saglasnost.

Aproksimacija petlji

Invarijante petlji
Obezbediti invarijantu petlje koja će dozvoliti simboličkom izvršavanju da preskoči analizu petlje. Na primer:

```
void f (unsigned int n) {
  i = 0;
  while (i < n) { // inv: i <= n
    i = i + 1;
  }
}
```

Ovakva analiza je polu-automatska (ako korisnik obezbeđuje invarijante) ili je u pitanju automatsko generisanje invarijante što ponovo dovodi do aproksimacije koda u smeru uopštavanja rezultata izvršavanja petlje (engl. *over-approximation*)

6 Rešavači. Binarni kôd

6.1 Rešavači

Rešavači

Ključna komponenta efikasnosti simboličkog izvršavanja su performanse rešavača

- Pozivi rešavača su skupi
  - Simboličko izvršavanje može da održava preslikavanje iz formula u odgovarajuće vrednosti koje ih zadovoljavaju. Na primer $x + y < 10 \wedge x > 5 \rightarrow \{x = 6, y = 3\}$
  - Ako nam je potrebno rešenje za slabiju formulu, npr za $x + y < 10$, možemo da iskoristimo prethodno sračunate vrednosti bez poziva rešavača
  - Ako je formula jača, npr. $x + y < 10 \wedge x > 5 \wedge y \geq 0$ možemo najpre da proverimo da li je sačuvano rešenje ok, ako jeste, iskoristimo ga, ako nije, teko onda ponovo pozivamo rešavač
- Ograničenja koja se javljaju u realnom softveru su teška za rešavanje
  - Nelinearna ograničenja, ograničenja teorije nizova...

Tehnike smanjivanja opterećenja rešavača

Smanjivanje opterećenja rešavača

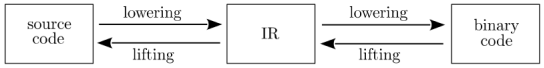
- Simplifikovati izraz u hodu
- Inkrementalno rešavanje
- Čuvanje dobijenih rezultata i njihovo ponovno krišćenje
- Zamena konkretnih vrednosti simboličkim i obratno u okviru kompleksnih uslova

6.2 Binarni kôd

Simboličko izvršavanje binarnog koda

Na postojeće probleme i tehnike, dodaje se:

- Podizanje do međureprezentacije



- Rekonstruisanje grafa kontrole toka
- Problem obfuskacije koda

Literatura

Pregled simboličkog izvršavanja
A Survey of Symbolic Execution Techniques
autori: Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi
ACM Computing Surveys (CSUR), 51(3), pp.1-39. 2018.
https://dl.acm.org/doi/10.1145/3182657
Rad je slobodno dostupan na:
https://arxiv.org/abs/1610.00502
Dodatni materijali:
http://www.verifikacijasoftvera.matf.bg.ac.rs/