Selektivno simboličko izvršavanje

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Irena Blagojević, 1097/2017
9irenab@gmail.com

Gorana Vučić, 1095/2017
goranavucic94@gmail.com

11. decembar 2018

Sažetak

Analiza velikih, kompleksnih sistema koji rade u realnom vremenu zahteva korišćenje novih tehnika kojima se na drugačiji način pristupa ovakvom problemu. Stoga će biti predstavljena platforma S^2E koja koristi selektivno simboličko izvršavanje i relaksirani model konzistentnosti izvršavanja, na način kojim istovremeno izvršava čitave familije izvršnih putanja skalabilno. Spomenute tehnike daju mogućnost S^2E direktnog analiziranja izvršnih fajlova, kao i mogućnost prikazivanja realnog ponašanja izvršavanjem analiza unutar realnog softverskog steka.

Sadržaj

1 Uvod

Kako bi se analiziralo ponašanje onoga što se razvija postoje različite analize koje mogu da se sprovedu. Jedna od osnovnih jeste razumevanje posmatranog ponašanja, dok neke složenije imaju za cilj da predvide buduće ponašanje sistema u ranije neviđenim okolnostima. Iskusni developeri često za neke manje programe mogu samostalno na osnovu koda da sagledaju odgovore na postavljena pitanja u vezi sa ponašanjem programa. Međutim cilj je da se na takva pitanja odgovori za velike, kompleksne sisteme koji rade u relanom vremenu.

Biće predstvljena platforma koja omogućava jednostavnu izgradnju alata za analizu i nudi naredne tri osobine:

- Efikasno analiziranje čitavih familija izvršnih putanja;
- Prikazivanje realnog ponašanja izvršavanjem analiza unutar realnog softverskog steka;
- Mogućnost direktnog analiziranja izvršnih fajlova;

Prediktivne analize umesto uvida u samo jednu putanju, često moraju da imaju uvid u čitave familije putanja sistema koji se analizira. Familija putanja (*eng. family of paths*) predstavlja skup putanja koje imaju neko specifično svojstvo [6]. Na primer, sigurnosne analize moraju da istraže da ne postoje nekakvi granični slučajevi koji mogu da naruše željenu politiku sigurnosti nekog programa. Pošto treba ispitati ogroman broj putanja, bitno je da analiza bude što efikasnija.

Kako bi se odredila tačna procena ponašanja programa, to često zahteva da se u obzir uzme čitavo okruženje programa, za šta je potrebno da se analize izvršavaju in-vivo[1]. Okruženje (eng. *environment*) programa predstavljaju biblioteke, jezgro operativnog sistema, drajveri, itd. I veoma mali programi interaguju sa svojim okruženjem, stoga je za razumevanje njihovog ponašanja neophodno razumeti prirodu takvih interakcija. Pojedini alati izvršavaju realno okruženje, ali pri tom dozvoljavaju da se pozivi iz različitih izvršnih putanja neprekidno međusobno mešaju, dok većina drugih pristupa apstrahuje okruženje nezavisno od modela. Međutim pisanje apstraktnih modela je veoma zahtevno. Takvi modeli su retko stoprocentno tačni, i pri tom imaju tendenciju da gube tačnost sa razvojem modelovanog sistema. Iz navedenog razloga, poželjno je da ciljni program tokom analize direktno interaguje sa svojim realnim okruženjem na način koji analizu više putanja održava konzistentnom.

Realni sistemi sastavljeni su od mnogo raličitih komponenti, i pristup svim odgovarajućim izvornim kodovima je retko dostupno. Upravo iz tog razloga, analize je potrebno izvršavati direktno nad izvršnim datotekama.

Ključni izazov za izvođenje in-vivo analize više putanja (eng. *in-vivo multi-path analysis*) koje operišu nad izvršnim datotekama predstavlja skalabilnost. Pri prelasku sa analize jedne putanje na anlizu više putanja, linearni problem se pretvara u eksponencijalni, jer se broj putanja kroz program eksponencijalno povećava u broju grana koje treba obići. Ovo predstavlja problem *"eksplozije putanja"* (eng. *path explosion*). Iz tog razloga danas je nemoguće potpuno simbolički izvršavati ceo softverski stek kog sačinjavaju programi, biblioteke, jezgro operativnog sistema, drajveri,

[1] Latinski izraz koji u direktnom prevodu označava *"unutar živog"* i odnosi se na eksperimentisanje korišćenjem celog sistema u nekom trenutku vremena. In-vivo programska analiza obuhvata sve interakcije analiziranog koda sa sistemom koji ga okružuje, ne samo sa pojednostavljenom apstrakcijom tog sistema [5].

itd., što bi bilo neophodno ukoliko je potrebno izvršiti in-vivo analizu više putanja koja je konzistentna.

S^2E predstavlja opštu platformu za razvijanje alata koji izvršavaju in-vivo analizu više putanja. Ova platforma praktična je i za velike, kompleksne sisteme. Korišćenjem selektivnog simboličkog izvršavanja (eng. *selective symbolic execution*) i relaksiranog modela konzistentnosti izvršavanja (eng. *relaxed execution consistency models*), S^2E istovremeno izvršava čitave familije izvršnih putanja na skalabilan način. Za predstavljanje želejnih analiza in-vivo koristi se virtuelizacija. Takođe S^2E koristi i dinamičko binarno prevođenje (eng. *dynamic binary translation*) kako bi se direktno interpretirao x86 mašinski kod, što omogućava analizu različitih softvera.

S^2E platforma nudi mehanizam automatskog istraživanja putanja i modularne analizatore putanja. Istraživač (eng. *explorer*) paralelno prolazi kroz sve izvršne putanje od interesa, dok analizatori (eng. *analyzers*) proveravaju njihova svojstva ili sakupljaju određene informacije. S^2E poseduje gotove selektore i analizatore. Nekakav alat koji se pravi na S^2E platformi zajedno spaja selektore putanja sa analizatorima putanja. Selektori (eng. *selectors*) specifikacijom putanja od interesa navode istraživač putanja. Analizatori (eng. *analyzers*) putanja mogu se spojiti sa onim analizatorima koje poseduje S^2E ili se mogu od početka pisati korišćenjem S^2E API-ija. Korisnik S^2E u konfiguracionom fajlu treba da definiše selektore i analizatore koje želi da koristi, kao i da postavi odgovarajuće parametre. Potom treba da startuje željeni softverski stek unutar S^2E virtuelne mašine, i da pokrene S^2E launcher u gostujućem operativnom sistemu, čime se startuju željena aplikacija i komunikacija sa S^2E virtuelnom mašinom.

U daljem tekstu biće objašnjeno selektivno simboličko izvršavanje 2, modeli konzistentnosti izvršavanja 3 i opšta platforma (eng. *general platform*) 5 [5].

2 Selektivno simboličko izvršavanje

Ideja simboličkog izvršavanja jeste da se program tretira kao superpozicija mogućih izvršnih putanja. Na primer, program koji je većim delom linearan osim što poseduje jednu uslovnu naredbu *if* $(x > 0)$ *then ... else ...*, može se posmatrati kao superpozicija dve moguće putanje: jedne za uslov $x > 0$ i druge za $x <= 0$. Za izvršavanje svih putanja nije neophodno da se ispitaju sve vrednosti za x, već je dovoljno ispitati jednu vrednost veću od nule i jednu manju od nule. Superpozcija putanja se razvija u simboličko stablo izvršavanja (eng. *symbolic execution tree*), u kome svako moguće izvršavanje odgovara putanji od vrha stabla do lista koji predstavlja ciljno stanje. Na samom početku programa promenljive se označavaju kao simboličke. Umesto na primer da promenljiva x uzima konkretnu vrednost (npr. $x = 5$), posmatra se λ koje uzima sve moguće vrednosti za x. Svaki put kada je instrukcija grananja uslovljena predikatom P koji inidrektno ili direktno zavisi od x, izvršavanje se deli na dva izvršavanja E_i i E_j, kreiraju se dve kopije stanja programa. Sve promenljive u putanji E_i koje su uključene u P moraju da budu ograničene tako da P ima vrednost tačno, dok su u putanji E_j ograničene tako da P ima vrednost false.

Proces se rekurzivno ponavlja, svaka putanja se može dalje deliti. Svako izvršavanje naredbe grananja kreira novi skup dece, čime se ono

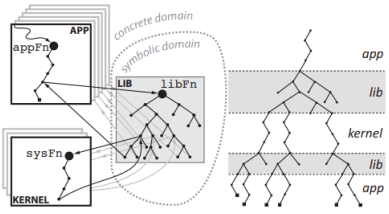
što bi normalno bilo linearno izvršavanje (ukoliko su korišćene konkretne vrednosti) pretvara u stablo izvršavanja (pošto se koriste simboličke vrednosti). Čvor s u stablu predstavlja stanje programa (skup promenljivih sa formulama koje ograničavaju vrednosti promenljivih), dok na primer ivica $s_i \rightarrow s_j$ indikuje da je s_j naslednik od s_i za svaku putanju koja zadovoljava ograničenja u s_j. Putanje u stablu se mogu istovremeno obilaziti, kako se stablo razvija. Pošto se stanje programa kopira, putanje se mogu nezavisno istraživati.

S^2E koristi opažanje da su često samo određene familije putanja od interesa. Ukoliko je potrebno istražiti sve putanje nekog programa, ali da se pri tom ne brine o putanjama biblioteka ili jezgra operativnog sistema, S^2E treba da podeli izvršavanja radi istraživanja različitih putanja. Izvršavanje više putanja može prestati kad god se pozove neki drugi deo sistema kao što je biblioteka, i može se nastaviti izvršavanje jedne putanje. Potom, kada se izvršavanje vrati u program, potrebno je nastaviti izvršavanje više putanja.

U režimu izvršavanja više putanja familiji putanja je dozvoljeno da se širi i stablo raste u širinu i dubinu, dok u režimu izvršavanja jedne putanje familija putanja se ograničava i stablo raste jedino u dubinu. Iz tog razloga se kaže da je S^2E istraživanje putanja programa elastično. Kad god postoji mogućnost za to, S^2E isključuje režim izvršavanja više putanja, kako bi se smanjila veličina izvršnog stabla i uključile jedino one putanje koje su od interesa za ciljnu analizu.

Elastičnost je ključna u in-vivo istraživanju više putanja programa unutar kompleksnih sistema, kao što je Windows. Kombinovanjem elastičnosti sa virtuelizacijom stvara se iluzija da se ceo softverski stek izvršava simbolički, dok se zapravo simbolički izvršavaju samo selektovane komponente. Na primer, kod konkretnog izvršavanja biblioteka i jezgra, S^2E dozvoljava da se istraže putanje programa bez potrebe da se vrši modelovanje njihovog okruženja. Pod ovim se podrazumeva selektivno simboličko izvršavanje.

Mešanje faza simboličkog izvršavanja sa fazama konkretnog izvršavanja mora se izvesti pažljivo, tako da prelaženje između izvršavanja više putanja (simboličkog) i izvršavanja jedne putanje (konkretnog) u oba smera očuva izvršavanje konzistentnim.



Slika 1: Simboličko/konkretno izvršavanje

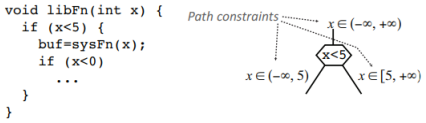
Na slici 1 predstavljen je pojednostavljen primer korišćenja S^2E, gde aplikacija *app* koristi biblioteku *lib* koja se nalazi iznad jezgra operativnog sistema (*kernel*). Ciljna analiza zahteva da se simbolički izvrši samo biblioteku *lib*. Funkcija $appFn$ iz aplikacije poziva bibliotečku funkciju $libFn$, koja eventualno poziva sistemski poziv $sysFn$. Kada se izvršavanje $sysFn$ završi, $libFn$ nastavlja da obavlja nekakva procesiranja i po završetku nastavak rada programa se vraća u $appFn$. Izvršno stablo se proširuje nakon što izvršavanje pređe iz konkretnog u simbolički domen. Pošto se izvršavanje vrati u konkretni domen, izvršno stablo se ograničava i ne

dodaju se nikakve nove putanje, dok se izvršavanje ne vrati u simbolički domen [5].

2.1 Prelaz iz konkretnog u simboličko izvršavanje

Korišćenjem konkretnih argumenata $appFn$ poziva $libFn$. Kako bi se konkretni argumenti zamenili simboličkim, najjednostavnije je pozvati S^2E selektor, pa tako umesto $libFn(10)$ biće pozvana $libFn(\lambda)$. Dodatno se λ može ograničiti. S^2E potom simbolički izvršava $libFn$ ali istovremeno izvršava $libFn$ sa originalnim konkretnim argumentima. Kada se istraživanje $libFn$ završi, S^2E će funkciji $appFn$ da vrati konkretnu povratnu vrednost kao rezultat konkretnog izvršavanja, ali će se $libFn$ takođe simbolički iztražiti. Ovim je izvršavanje aplikacije app konzistentno. Konkretan domen nije svestan izvršavanja $libFn$ u režimu više putanja. Sve putanje se nezavisno izvršavaju, a na dodacima analizatora S^2E je da odluče da li pored posmatranja konkretne putanje, žele da posmatraju i simboličke putanje [5].

2.2 Prelaz iz simboličkog u konkretno izvršavanje



Slika 2: Početak izvršavanja $libFn$ funkcije

Mnogo komplikovanije je prelaženje iz funkcije $libFn$ u $sysFn$. Ukoliko bi funkcija $libFn$ imala kod prikazan kao na slici 2, i pri tom bila pozvana sa neograničenom simboličkom vrednošću za $x \in (-\infty, +\infty)$. Instrukcija if na samom početku bi izvršila grananje na dve različite putanje, gde za jednu važi uslov da $x \in (-\infty, 5)$, a za drugu važi da $x \in [5, +\infty)$. Ovi izrazi se nazivaju ograničenjima putanja (eng. *path constraints*). U okviru *then* grane poziva se funkcija $sysFn(x)$, što zahteva da x uzme konkretnu vrednost. Na primer za x se uzima vrednost 4, koja je konzistentna sa uslovom $x \in (-\infty, 5)$ i potom se izvršava $sysFn(4)$ poziv. Ograničenja putanje u simboličkom domenu se ažuriraju tako da odgovaraju uslovu da je $x = 4$. S^2E konvertuje vrednost za x iz simboličke u konkretnu na zahtev, samo prilikom konkretnog izvršavanja pri čitanju vrednosti za x. Ovo predstavlja važnu optimizaciju za izvođenje in-vivo simboličkog izvršavanja, jer se veliki broj podataka prenosi kroz slojeve softverskog steka bez konverzije.

Po završetku funkcije $sysFn$ izvršavanje prelazi u $libFn$, odnosno simbolički domen. Povratna vrednost $sysFn$ je korektna jedino pod ograničenjem da je $x = 4$, koje je dodato u ograničenja putanja pri konkretizaciji vrednosti za x. Sva izračunavanja u $sysFn$ su izvršena pod ovom pretpostavkom i izvršavanje funkcije $libFn$ se može nastaviti jedino pod ovim ograničenjem čime se čuva korektnost.

Kako ograničenje $x = 4$ familiju budućih putanja koje se mogu istražiti pozivom $libFn$ ograničava, x više ne može uzimati vrednosti iz $(-\infty, 5)$. Upravo iz tog razloga *then* grana instrukcije $if(x < 0)$... više neće biti dostižna. Ovo se naziva "$overconstraining$", odnosno ograničenje nije uvedeno kao karakteristika koda iz $libFn$ funkcije već kao rezultat konkretizacije vrednosti za x pri prelasku u konkretni domen. Na $x = 4$ se posmatra kao na slabo ograničenje (eng. *soft constraint*) postavljenog pri

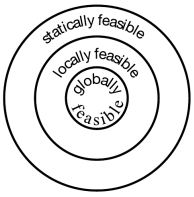
prelasku iz simboličkog u konkretno izvršavanje, dok se na $(-\infty, 5)$ posmatra kao jako ograničenje postavljenog u funkciji $libFn$. Kad je neka grana u simboličkom domenu onemogućena zbog slabog ograničenja, moguće je vratiti se unazad u stablu izvršavanja i izabrati dodatnu vrednost za konkretizaciju, odnosno dodati novo podstablo izvršavanja i ponoviti poziv funkcije $sysFn$ kako bi se omogućilo izvršavanje nove grane. S^2E može da prati uslove grananja u konkretnom domenu, što omogućava da se ponovo izvrše pozivi funkcija koji izvršavaju nove grane stabla [5].

3 Modeli konzistentnosti izvršavanja

Za stanje izvršavanja kažemo da je *konzistentno* ako postoji dostižna (eng. *feasible*) putanja od početnog do trenutnog stanja. Postoje mnoge analize za koje je pretpostavka kozistentnosti stanja sistema nepotrebno jaka[2]. Takođe, pružanje takve konzistentnosti tokom istraživanja više putanja (eng. *multi-path exploration*) je previše skupo. Zato se uvode različiti nivoi konzistentnosti izvršavanja (eng. *consistency models*).

Modeli konzistentnosti izvršavanja se međusobno razlikuju prema putanjama izvršavanja koje prihvataju, tj. na onsnovu putanja izvršavanja čiju konzistentnost održavaju. Odabir modela konzistentnosti predstavlja kompromis između cene primene modela i koliko su putanje koje se prihvataju "realistične". Kakav kompromis je odgovarajući zavisi od toga na koji je način dostižnost različitih putanja utiče na kompletnost (eng. *completeness*) i ispravnost (eng. *soundness*) analize [1].

Prilikom definicije modela, razmatra se koje putanje taj model uključuje ili isključuje. Na slici 3 se može videti odnos tri vrste putanja. Za putanju se kaže da je *statički dostižna* (eng. *statically feasible*) ako postoji putanja u interproceduralnom grafu kontrole toka (eng. *control flow graph*) sistema[3] koja odgovara izvršavanju. Podskup tih putanja su *lokalno dostižne* (eng. *locally feasible*) putanje, čije izvršavanje je konzistentno sa sistemskim grafom kontrole toka i sa ograničenjima koja proizilaze iz jedinice. Njihov podskup su *globalno dostižne* (eng. *globally feasible*) putanje, čije izvršavanje je dodatno konzistentno sa ograničenjima kontrole toka koja slede iz okruženja.



Slika 3: Odnos statički, lokalno i globalno dostižnih putanja

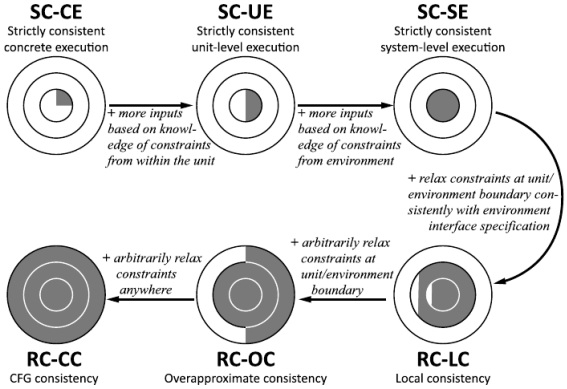
Za model se kaže da je *kompletan* (eng. *complete*) ako će svaka putanja kroz jedinicu koja odgovara nekoj globalno dostižnoj putanji kroz ceo sistem u nekom trenutku biti otkrivena pomoću tog modela. Drugim

[2]Na primer, kod jediničnog testiranja (eng. *unit testing*) [7], proveravju se sve putanje koje su u skladu sa interfejsom jedinice, bez obzira na dostižnost tih putanji u integrisanom sistemu. Ovo je jeftiniji način, koji ujedno pruža i veću pouzdanost u korektnost jedinice u budućoj upotrebi.

[3]Pod sistemom se misli na okruženje (eng. *environment*) i jedinicu (eng. *unit*) koji se analiziraju.

rečima, kompletan model će otkriti svaku putanju u jedinici kojoj odgovara neka globalno dostižna putanja, ali ne sprečava pojavljivanje putanja koje nisu globalno dostižne. Model je *konzistentan* (eng. *consistent*) ako za svaku putanju kroz jedinicu postoji odgovarajuća globalno dostižna putanja kroz ceo sistem. To znači da su sve putanje koje konzistentan model pronađe korektne, ali ne garantuje da će biti pronađene sve takve, tj. neke korektne mogu biti propuštene.

U nastavku poglavlja će biti opisani različiti modeli konzistentnosti izvršavanja. Biće opisani od onih koje krakteriše jača do onih koje karakteriše slabija konzistentnost. Grafički prikaz odnosa ovih modela se može videti na slici 4 [5].



Slika 4: Različiti modeli konzistentnosti izvršavanja pokrivaju različite skupove dostižnih putanja.

3.1 Striktna konzistentnost (SK)

Najstrožija konzistentnost prihvata samo globalno konzistentne putanje. Konkretno izvršavanje programa uvek poštuje striktno konzistentan (*SK* – eng. *strictly consistent*) model. Posmatrano iz drugog ugla, to znači da svaka putanja koja je u skladu sa SK modelom može da se mapira u neko konkretno izvršavanje. Pod SK modelom ispravne analize ne proizvode lažna prihvatanja (eng. *false positives*), a inače ne proizvode ni lažna upozorenja (eng. *false negatives*). Razlikuju se tri podkategorije modela SK na osnovu informacija koje se uzimaju u obzir prilikom istraživanja novih putanja [5].

3.1.1 Striktno konzistentno konkretno izvršavanje (SK-KI)

Striktno konzistentno konkretno izvršavanje (eng. *strictly consistent unit-level execution*) tretira ceo sistem kao crnu kutiju (eng. *black box*) [7] – unutrašnja struktura sistema se ne koristi za pronalaženje novih putanja. Jedine putanje koje se istražuju su one kojima se sistem kreće prilikom izvršavanja sa ulazom koji zadaje analiza. Dakle, nove putanje se mogu pronaći samo nasumičnim pogađanjem novih putanja. Fazing (eng. *fuzzing*) [15] potpada pod ovaj model [5].

3.1.2 Striktno konzistentno izvršavanje na nivou jedinice (SK-IJ)

Model striktno konzistentnog izvršavanja na nivou jedinice za pronalaženje novih putanja može koristiti interne informacije iz jedinice. Okruženje se tretira kao crna kutija, tj. ograničenja putanje koja definiše okruženje se ne prate, i zbog toga neće sve globalno dostižne putanje biti istražene. Međutim, time se štedi vreme, jer se ne analizira okruženje, koje može da bude nekoliko redova veličine veća od jedinice [5]. Ovaj model se često koristi u alatima za simboličko [10] i konkoličko izvršavanje [14].

3.1.3 Striktno konzistentno izvršavanje na nivou sistema (SK-IS)

Model striktno konzistentnog izvršavanja na nivou sistema (eng. *strictly consistent system-level execution*) koristi informacije o svim delovima sistema za pronalaženje novih putanja kroz jedinicu. Ovakvo istraživanje putanja je i kompletno i ispravno. Drugim rečima, svaka putanja kroz jedinicu moguća u konkretnom izvršavanju će u nekom trenutku biti pronađena pomoću SK-IS modela, što čini ovaj model jedinim koji je i striktan i kompletan. Njegova loša strana je implementacija, zbog problema eksplozije putanja [3] – broj globalno dostižnih putanja je približno eksponencijalan po veličini sistema [5].

3.2 Lokalna konzistentnost (LK)

Lokalno konzistentan (eng. *locally consistent*) model je kombinacija performansi SK-IJ i kompletnosti SK-IS. Pronalaženje svih putanja kroz okruženje se u ovom modelu izbegava, ali se istražuju svi odgovarajući segmenti putanja kroz jedinicu tako što se rezultati nekih poziva ka okruženju zamene simboličkim vrednostima koje odgovaraju mogućim validnim rezultatima izvršavanja, tj. onim koje su u skladu sa specifikacijom okruženja. Ovakvo ponašanje dovodi do globalne nekonzistentnosti, ali se očuvava interna konzistentnost jedinice time što se vodi računa o propagiranju nekonzistentnosti kroz okruženje. Svaka ispravna analiza koja uzima u obzir samo interno stanje jedinice ne proizvodi lažna prihvatanja pod modelom LK. U praksi, što jedinica manje interaguje sa okruženjem, to je manje putanja koje će biti zaustavljene ili propuštene zbog propagacije nekonzistentnosti [5].

3.3 Relaksirana konzistentnost (RK)

Pod modelom relaksirane konzistentnosti (eng. *relaxed consistency*) prihvataju se sve putanje kroz jedinicu, čak i one koje nisu dozvoljene u SK i LK modelima. RK je nekonzistentan u opštem slučaju. Glavna prednost su performanse. Prihvatanjem dodatnih nedostižnih putanja, analiza velikih delova sistema, koji nisu cilj analize, može biti izbegnuta. Prihvatanjem lokalno nedostižnih putanja, analiza postaje sklona prijavljivanju lažnih upozorenja (eng. *false negatives*), jer neke putanje koje se na ovaj način analiziraju nikad neće biti izvršene u konkretnom pokretanju [5].

3.3.1 Precenjujuća konzistentnost (RK-PK)

Model precenjujuće konzistentnosti (eng. *overapproximate consistency*) ne prati ograničenja postavljena od strane okruženja i jedinice. To znači da pretpostavlja da se mogu javiti vrednosti i van opsega propisanih specifikacijom okruženja i jedinice. Ovaj model nije konzistentan, ali jeste kompletan. Koristan je za obrnuto inženjerstvo (eng. *reverse engineering*) [8]. Osim što istražuje sva moguća ponašanja jedinice u validnom okruženju, dodatno pretražuje ponašanja koja su moguća samo ako se okruženje ponaša van svoje specifikacije. Takvo precenjivanje povećava kvalitet obrnutog inženjerstva [5].

3.3.2 Konzistentnost grafa kontrole toka (RK-KK)

Konzistentnost grafa kontrole toka (eng. *control flow graph consistency*) podrazumeva da je dozvoljena izmena bilo kojih delova stanja sistema. Potrebno je samo da pronađene putanje izvršavanja odgovaraju putanjama u interproceduralnom grafu kontrole toka jedinice. Ovaj model je kompletan [5].

3.4 Implementacija modela konzistentnosti

U nastavku će biti opisano kako modeli konzistentnosti mogu biti implementirani pomoću motora (eng. *engine*) selektivnog simboličkog izvršavanja. Biće ukratko opisane specifičnosti konverzije iz simboličkog u konkretno i obrnuto.

- **SK-KI** dozvoljava samo konkretne ulaze u sistem. Izvršava se samo jedna putanja kroz jedinicu i okruženje. Nema potrebe za praćenjem ili rešavanjem ograničenja, zato što nema simboličkih podataka.
- **SK-IJ** prevodi sve simboličke podatke u konkretne kada jedinica poziva neku funkciju iz okruženja, i to je jedina konverzija koja se vrši. Ona je konzistentna sa trenutnim skupom ograničenja putanje u jedinici. Dobijene putanje su globalno dostižne, ali istraživanje nije kompletno. Prevođenje u konkretne vrednosti se tretira kao jako ograničenje u daljem kodu, što može skratiti globalno dostižne putanje.
- **SK-IS** ceo sistem izvršava simbolički. To čuva globalnu konzistentnost izvršavanja. Eksplozija putanja može činiti SK-IS teškim za korišćenje u praksi. Problem nastaje zbog veličine okruženja u odnosu na jedinicu. Takođe, putanja kroz okruženje može biti toliko više da može odložiti ciljnu analizu. Postoje heuristike za prioretizovanje putanja za istraživanje, ili se može koristiti inkrementalno simboličko izvršavanje (eng. *incremental symbolic execution*)[4].
- **LK** na prelazu jedinica-okruženje konvertuje vrednosti generisane od strane okruženja u simboličke vrednosti koje zadovoljavaju ograničenja koja slede iz specifikacije okruženje. Ovo omogućava izvršavanje više putanja u jedinici. Ako se neki podaci koriste kao ulaz u okruženje, njihova propagacija se mora pratiti. Ako neka grana u okruženju zavisi od tih podataka, putanja se prekida – ovo je neophodno zato što su ti podaci mogli biti "simbolizovani" na osnovu konkretnih

[4]Izvršavaju se delovi okruženja onoliko koliko je potrebno za brže otkrivanje zanimljivih putanja u jedinici. [5]

podataka iz okruženja, a koje okruženje nije moglo da proizvede u trenutnom stanju.

- **RK-PK** prevodi konkretne vrednosti na prelazu između jedinice i okruženja u *neograničene* simboličke vrednosti, ne vodeći pritom računa o uslovima postavljenim tim prelazom. Ovo obezbeđuje kompletnost po cenu suštinskog precenjivanja. Nijedna dostižna putanja neće biti isključena iz simboličkog izvršavanja, ali mogu postojati lokalno nedostižne putanje stvorene postojanjem neograničenih vrednosti iz okruženja.
- **RK-KK** može biti implementirano tako što se prate svi izlazi iz svih grana, bez obzira na ograničenja putanje – tako se obilaze sve ivice interproceduralnog grafa kontrole toka jedinice. Istraživanje je brzo, jer dostižnost grana ne mora da se proverava pomoću rešavača ograničenja (eng. *constraint solver*). Implementacija RK-KK može zahtevati znanje specifično za jezik koji se koristi [5].

3.5 Modeli konzistentnosti i postojeći alati

Većina alata *dinamičke analize* [12] koriste SK-KI, npr. Valgrind [16] i Eraser [13]. Oni izvršavaju program kroz jednu putanju određenu ulazom koji je zadao korisnik. Značajno su brži od analize više putanja i korisni za objašnjavanje ponašanja programa na nekom malom skupu putanja (test slučajevi).

Alati za *dinamičko generisanje test slučajeva* [11] obično koriste SK-IJ ili SK-IS. Na primer, DART [9] koristi SK-IJ – konkretno izvršava program sa proizvoljnim ulazom, zatim instrumentalizuje kod za skupljanje ograničenja putanje prilikom svakog izvršavanja. Ta prikupljena ograničenja se koriste za generisanje novih konkretnih ulaza koji će aktivirati druge putanje. Alat *KLEE* [4] koristi ili SK-IS ili neki oblik SK-IJ, zavisno od toga da li je okruženje modelirano ili nije. Ako jeste, i jedinica i okruženje se izvršavaju simbolički, a inače, kada jedinica poziva okruženje, ono se izvršava sa konkretnim argumentima, ali se ne prate bočni efekti takvog izvršavanja, već se dopušta njihova propagacija.

Alati *statičke analize* [12] obično koriste neke oblike model RK. Na primer, SDV alat [2] prevodi program u binarni oblik, što je precenjivanje originalnog programa. Kao posledica, biće pronađene sve dostižne putanje, ali i neke nedostižne [5].

4 Sistemska analiza sa S^2E platformom

S^2E je platforma za brzo pravljenje prototipova sistemske analize prilagođene zahtevima. Pruža dva ključna interfejsa:

- interfejs za izbor (eng. *selection interface*) – navodi istraživanje putanja izvršavanja
- interfejs za analizu (eng. *analysis interface*) – prikupljanje događaja i proveravanje svojstava putanja izvršavanja

Ispod haube, S^2E se sastoji od prilagođene virtuelne mašine, dinamičkog binarnog prevodioca (DBP) i ugrađenog motora simboličkog izvršavanja. DBP odlučuje o tome koje instrukcije izvršavati konkretno na fizičkom procesoru, a koje simbolički pomoću ugrađenog motora simboličkog izvršavanja [5].

4.1 Korisnički interfejs

Izbor putanja: Prvi korak prilikom korišćenja S^2E alata je odlučivanje o pristupu izvršavanja programa – koje delove programa izvršavati u režimu više putanji (simbolički), a koje u režimu jedne putanje (konkretno). S^2E pruža podrazumevani skup selektora putanji za najčešće tipove izbora – na osnovu podataka, koda i prioriteta. Na primer, postoji dodatak *CodeSelector*, koji prima listu opsega brojača programa *program counter*, u kojoj svaki opseg može biti obeležen za simboličko ili konkretno izvršavanje.

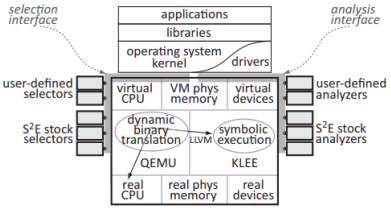
Analiza putanja: Kada su familije putanja definisane, S^2E ih izvršava i svaku od njih propušta kroz dodatke (eng. *plugins*) za analizu. Jedna klasa takvih dodataka su nalazači bagova, koji traže odgovarajuće uslove za pojavljivanje bagova i vraćaju putanju izvršavanja koja do baga dovodi. Postoje i dodaci koji selektivno beleže instrukcije na nekoj putanji sa nekim dodatnim informacijama, koje omogućavaju praćenje pokrivenosti sa unapred poznatim okruženjem (eng. *offline*). Većina dodataka ne zavisi od operativnog sistema, ali S^2E pruža i skup dodataka za analizu događaja specifičnih za operativni sistem Windows. Na primer, *WinBugCheck* dodatak hvata događaje plavog ekrana smrti (eng. *blue screen of death*) i padove kernela [5].

4.2 Interfejs za razvijanje

S^2E pruža interfejs koji se može koristiti za pisanje novih dodataka ili proširivanje starih. Isti interfejs se koristi za selektore i dodatke za analizu. Razlika je što su dodaci za analizu pasivni posmatrači, a selektori utiču na samo izvršavanje. Dodaci međusobno komuniciraju putem mehanizma događaja. Događaje generiše S^2E platforma ili neki dodatak. Dodatak se registruje na klasu događaja i definiše svoj odgovor (eng. *callback*), koji će se pozivati prilikom svake pojave događaja.

Definišu se bazni događaji, koji odgovaraju najnižem nivou apstrakcije izvršavanja, a to su: prevođenje instrukcije, izvršavanje, pristup memoriji, račvanje stanja i izuzeci. Na primer, tokom prevođenja, dodatak može obeležiti određene instrukcije, za koje se podiže događaj izvršavanja, na koji dodatak odgovara definisanim ponašanjem. Definišu se i opkodovi (eng. *opcodes*) – prilagođene instrukcije koje ova platforma direktno interpretira. Oni predstavljaju najjednostavniji način za početak korišćenja S^2E platforme, jer ne zahtevaju korišćenje dodataka. Pomoću njih se mogu, na primer, praviti simboličke vrednosti ili uključivati i isključivati izvršavanje više putanji [5].

5 Prototip S^2E



Slika 5: Arhitektura S^2E

Prototip S^2E (slika 5) platforme koristi delove QEMU virtuelne mašine, KLEE mašinu za simboličko izvršavanje i niz LLVM alata. S^2E se može pokrenuti na Mac OS X, Windows i Linux operativnim sistemima. Takođe može da izvršava bilo koji gostujući operativni sistem koji podržava x86 arhitekturu.

Istražuje putanje ciljnog sistema tako što ga pokreće u virtuelnoj mašini i selektivno simbolički izvršava njegove delove. U zavisnosti od želejnih putanja, neke od mašinskih instrukcija sistema su dinamički prevedene unutar virtuelne mašine u međukod reprezentaciju pogodnu za simboličko izvršavanje, dok su ostale prevedene u host skup instrukcija. S^2E konvertuje podatke u oba smera između konkretnog i simboličkog domena u zavisnosti od izvršavanja, kako bi se pružila iluzija da se ceo sistem izvršava u režimu više putanja.

U istoj putanji kombinuju se konkretno i simboličko izvršavanje korišćenjem reprezentacije stanja mašine koje je deljeno između virtuelne mašine i ugrađene mašine za simboličko izvršavanje. Stanje se deli tako što se vrši preusmeravanje čitanja i pisanja iz QEMU i KLEE u zajedničko stanje mašine pod kojim se podrazumeva stanje fizičke memorije virtuelne mašine, stanje virtuelnog CPU i stanje virtuelnih uređaja. Na taj način se transparentno mogu konvertovati podaci između konkretnog i simboličkog domena, i pri tom se obezbeđuju različite kopije celokupnog stanja mašine za različite putanje.

Za postizanje transparentnog mešanja simboličkog i konkretnog izvršavanja, izvršena je modifikacija QEMU dinamičkog binarnog prevodioca, tako da se instrukcije koje zavise od simboličkih podataka prevode u LLVM i šalju u KLEE.

Za automatsko čuvanje i obnavljanje stanja virtuelnih uređaja i CPU-a, koristi se QEMU mehanizam za snimanje prilikom prelaska između stanja izvršavanja. Deljeno predstavljanje stanja memorije i uređaja između konkretnog i simboličkog domena dozvoljava S^2E da se konkretizacija podataka koja se čuva simbolički izvrši na zahtev [5].

6 Evaluacija

Glavni cilj S^2E jeste da omogući brzo pravljenje korisnih alata za sistemsku analizu. Stoga treba da se odgovori na tri ključna pitanja: Da li je S^2E zaista opšta platforma za izgradnju različitih alata za analizu (6.1)? Da li S^2E izvodi ove analize sa razumnim performansama (6.2)? Kakav kompromis je dozvoljen u biranju različitih modela konzistentnosti izvršavanja za izvršne programe koji se mogu izvršavati bilo u sistemskom ili korisničkom režimu (6.3)?

6.1 Tri slučaja upotrebe

Korišćenjem S^2E izgrađena su tri različita alata: automatski tester za drajvere zatvorenog koda (eng. *automated tester for proprietary device drivers*), alat za obrnuti inženjering binarnih drajvera (eng. *reverse engineering tool for binary drivers*) i in-vivo profajler performansi više putanja (eng. *multi-path in-vivo performance profiler*).

Korišćenjem S^2E izgrađen je alat DDT^+ koji ima svrhu testiranja drajvera za Windows čiji kod nije dostupan. Ovaj alat predstavlja reimplementaciju DDT alata sa izvršenim promenama za QEMU i KLEE i napisanim anotacijama za interfejse koje služe za implementaciju LK mo-

dela. Spaja zajedno nekoliko dodataka za istraživanje i analizu. DDT^+ implementira lokalnu konzistenciju putem anotacija interfejsa koje određuju gde je potrebno dodati simboličke vrednosti, a da se pri tom ispoštuje lokalna konzistentnost. U odsustvu anotacija, DDT^+ se vraća na strogu konzistentnost *SK-IS* gde jedini simbolički unos dolazi od strane hardvera.

Napravljen je i alat REV^+, koji predstavlja reimplementaciju $RevNIC$ alata. REV^+ služi za obrnuti inženjering binarnih drajvera za Windows. Ovaj alat uzima binarni drajver koji je zatvorenog koda, prati njegovo izvršavanje, potom dovodi izvršavanje do offline komponente koja ima mogućnost da primerni obrnuti inženjering nad logikom drajvera, i da na taj način napravi kod drajvera koji implementira potpuno isti hardverski protokol kao i originalni drajver. Koristi *RK-PK* model, i na taj način žrtvuje konzistentnost kako bi se brže izvela pokrivenost.

Kako bi se dodatno ilustrovala opštost S^2E, korišćen je za razvijanje $PROF_S$ alata, koji predstavlja in-vivo profajler performansi više putanja i debager. Ovo je prvi alat koji koristi simboličko izvršavanje za analzu performansi. Kako bi se dobili realni profili, analiza performansi se može uraditi pod lokalnom konzistentnošću ili bilo kojim strožijim modelom konzistentnosti. $PROOF_S$ dozvoljava korisnicima da izmere broj izvršenih instrukcija, promašaje keša, TLB promašaje itd. On poseduje naskup funkcionalnosti koje nudi Valgrind. Istraživanje putanja u $PROOF_S$ je moguće podesiti, dozvoljavajući korisniku da izabere bilo koji model konzistentnosti. Takođe može meriti uticaj jezgra operativnog sistema na ponašanje keša programa i obnuto, ne samo programa u izolaciji [5].

6.2 Troškovi implementacije

Usled provere pristupa simboličkoj memoriji nastaje većina troškova konkretnog režima, dok troškovi simboličkog režima nastaju usled LLVM interpretacije i razrešavanja simboličkih ograničenja. Simbolički domen zbog primene selektivnog simboličkog izvršavanja je mnogo manji od konkretnog domena, što ublažava njegove troškove. Pored toga, S^2E može u okviru simboličkog domena da razlikuje instrukcije koje se mogu konkretno izvršavati i potom ih pokrenuti na normalan način. Drugi izvor troškova predstavljaju simbolički pokazivači. Jedan od načina kojim se ovo može rešiti jeste da se doda podrška za direktno izvršavanje LLVM izvršnih fajlova unutar S^2E, čime bi se značajno smanjili troškovi koji nastaju prevođenjem sa x86 mašinskog koda u LLVM i na taj način bi se smanjili troškovi simboličkih pokazivača [5].

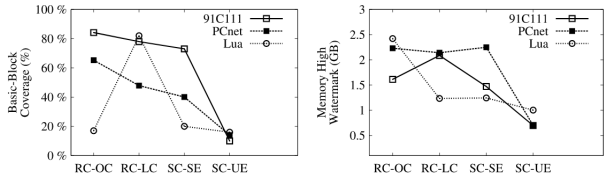
6.3 Kompromisi modela konzistentnosti izvršavanja

S^2E se može koristiti kao platforma za pravljenje moćnih alata za analizu. Međutim, potrebno je voditi računa kako odabir modela utiče na ukupno vreme izvršavanja, upotrebu memorije i pokrivenost putanja. Autori S^2E platforme su analizirali rad modela konzistentnosti upotrebljenim na različitim binarnim fajlovima. Rezultati analize se mogu videti na tabeli 1 i slici 6.

Na istoj mašini upoređivan je rad modela sa različitim binarnim fajlovima – drajverima za mrežu SMSC 91C111 (veličine 19KB) i AMD PCnet (veličine 35KB), i interpreterom za jezik Lua (12700 linija koda). Iz tabele 1 se može zaključiti da se nijedan model ne ističe kao najbrži ili najsporiji u opštem slučaju. Pokazalo se, takođe, da modeli slabije konzistentnosti (tj. "relaksiranije") pomažu dostizanje veće pokrivenosti osnovnih blo-

Konzistentnost	91C111 drajver	PCnet drajver	Lua
RC-OC	1400	3300	1103
LC	1600	3200	114
SC-SE	1700	1300	1148
SC-UE	5	7	-

Tabela 1: Vreme izvršavanja (u sekundama) istraživanja putanji za dva drajvera i Lua interpreter pod razlčitim modelima konzistentnosti.



Slika 6: Uticaj različitih modela konzistentnosti na pokrivenost osnovnih blokova (levo) i na potrošnju mememorije (desno).

kova. Na slici 6 levo se može uporediti pokrivenost pomenutih binarnih fajlova za vreme izvršavanja dato u tabeli 1. Što je model striktniji, manje je izbora simboličkih vrednosti, pa je i manje putanja koje se mogu istraživati i osnovnih blokova koji se mogu otkriti u datom vremenu. Na slici 6 desno može se videti uticaj modela na iskorišćenost memorije. Modeli SK-IS i SK-IJ (na slici SC-SE i SC-UE) konkretizuju podešavanja registara, što značajno smanjuje broj putanji koji se ispituje u odnosu na druge modele. Kada je u pitanju vreme potrebno za rešavanje ograničenja, ono zavisi od strukture sistema koji se analizira. U opštem slučaju, što je putanja dublja, složenija su odgovarajuća ograničenja putanje. U ovom slučaju, vreme razrešavanja se smanjuje što je konzistentnost striktnija, jer striktniji modeli ograničavaju količinu simboličkih podataka.

Evaluacija ja pokazala da je S^2E platforma opšte namene pomoću koje se mogu vršiti različite i zanimljive sistemske analize. Različiti modeli konzistentnosti pružaju fleksibilnost kompromisa između performansi, kompletnosti i ispravnosti analize [5].

7 Zaključak

Iako S^2E obuhvata ideje koje su istraživane i ranije, ona omogućava pisanje različitih analiza, odabir i mogućnost definisanja modela konzistentnosti, automatsku konverziju iz simboličkog u konkretni domen i obrnuto, i ne zahteva modeliranje ili modifikaciju okruženja. Upotrebom selektivnog simboličkog izvršavanja i modela relaksirane konzistentnosti izvršavanja, S^2E ima mogućnost skaliranja na velike sisteme. Selektivnost ograničava analizu više putanji na module koji su cilj analize, što minimizuje količinu simbolički izvršenog koda. Time se sprečava eksplozija putanja van tog modula. Modeli konzistentnosti omogućavaju donošenje odluka o prioritetu performansi u odnosu na tačnost tokom analize. S^2E omogućava brzo pravljenje prototipova alata za analizu uz malo napora.

Literatura

[1] José Bacelar Almeida, Maria Frade, Jorge Pinto, and Simão Sousa. *Rigorous Software Development. An Introduction to Program Verification*. 01 2011.

[2] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, April 2006.

[3] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. Rwset: Attacking path explosion in constraint-based test generation. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[4] 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, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[5] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *SIGPLAN Not.*, 47(4):265–278, March 2011.

[6] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The s2e platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49, February 2012.

[7] Lee Copeland. A practitioner's guide to software test design. 01 2004.

[8] Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. John Wiley & Sons, Inc., New York, NY, USA, 2005.

[9] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[10] James C. King. A new approach to program testing. In *Proceedings of the International Conference on Reliable Software*, pages 228–233, New York, NY, USA, 1975. ACM.

[11] B. Korel. A dynamic approach of test data generation. In *Proceedings. Conference on Software Maintenance 1990*, pages 311–317, Nov 1990.

[12] Janusz Laski and William Stanley. *Software verification and analysis. An integrated, hands-on approach.* London: Springer, 2009.

[13] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, November 1997.

[14] Koushik Sen. Concolic testing. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 571–572, New York, NY, USA, 2007. ACM.

[15] Ari Takanen, Jared DeMott, and Charlie Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., Norwood, MA, USA, 1 edition, 2008.

[16] Valgrind. Online at: http://www.valgrind.org/.