

# EXE: Pronalaženje test primera koji dovode do pada sistema

Seminarski rad u okviru kursa  
Verifikacija softvera  
Matematički fakultet

Tatjana Radovanović, 1103/2018  
tatjana.tasa.95@gmail.com

15. januar 2020

## Sažetak

U ovom radu je predstavljen EXE, alat za otkrivanje grešaka u softveru. Objašnjeno je kako pronalazi test primere koji dovode do pada sistema i koje optimizacione tehnike koristi kako bi te primere lakše i brže pronašao. Pored toga bilo je reči i o njegovom kompajleru, ali i o tome kako ostvaruje saradnju sa STP rešavačem.

## Sadržaj

<b>1</b>	<b>Uvod</b>	<b>2</b>
<b>2</b>	<b>Osnovno o EXE</b>	<b>2</b>
<b>3</b>	<b>Generisanje test primera</b>	<b>3</b>
<b>4</b>	<b>Kompilacija</b>	<b>3</b>
4.1	CIL . . . . .	3
4.2	exe-cc kompajler . . . . .	4
<b>5</b>	<b>STP</b>	<b>4</b>
5.1	STP i EXE . . . . .	4
<b>6</b>	<b>Optimizacije</b>	<b>5</b>
6.1	Keširanje ograničenja . . . . .	5
6.2	Nezavisna ograničenja . . . . .	5
<b>7</b>	<b>Zaključak</b>	<b>6</b>
	<b>Literatura</b>	<b>6</b>

# 1 Uvod

Softver je za vrlo kratko vreme postao neraskidivi deo savremenog sveta. Prisutan je u gotovo svim sferama ljudskog života. Nažalost, greške u softveru su neizbežne. Propusti koji se javljaju prilikom njegovog razvoja i održavanja imaju svoje posledice. To može biti stres zbog toga što aplikacija ne radi kako bi trebalo ili neprijatnost koju je nepravilan rad aplikacije prouzrokovao, ali greška može biti uzrok i mnogo težih problema koji mogu imati fatalan ishod. Zbog toga je potrebno pronaći što više grešaka u kodu kako bi one bile otklonjene.

Vremenom su razvijene mnoge metode i alati za verifikaciju softvera koji pomažu da softver bude što kvalitetniji. Jedna od metoda za analizu koda je simboličko izvršavanje koje se pokazalo kao efikasan način za otkrivanje grešaka. Alat koji je imao uticaj na proboj ovakvog načina provere je EXE koji služi za analizu C koda.

U ovom radu biće predstavljene neke karakteristike ovog alata i način kako on pronalazi greške i generiše test primere koji dovode do pada sistema. U poglavlju 2 biće reči o nekim osnovnim karakteristikama ovog alata. U poglavlju 3 je objašnjeno kako se generišu test primeri. Poglavlje 4 objašnjava upotrebu exe-cc kompajlera, a poglavlje 5 govori o saradnji sa STP rešavačem. Da bi analiza bila efikasnija EXE koristi određene optimizacije, a neke od njih su prikazane u poglavlju 6.

## 2 Osnovno o EXE

EXE (EXecution generated Executions) je alat namenjen pronalasku grešaka u kodu tako što automatski generiše ulaze koji proizvode pad sistema. Konceptualno počiva i na statičkim i na dinamičkim tehnikama za pronalaženje grešaka. Cilj je postići preciznost ispitivanja sistematičnim pokrivanjem statičkih metoda.[3] Osnovna ideja na koju se EXE oslanja je ta da se kod može iskoristiti da generiše sopstvene test slučajeve. Da bi se test slučajevi pronašli program se pokreće sa simboličkim ulazima, a inicijalni ulaz može biti bilo koja vrednost [6]. Dok se program izvršava EXE prati ograničenja simboličkih memorijskih lokacija. Delove programa koji koriste simboličke vrednosti EXE ne pokreće već dodaje kao ograničenja ulaza. Kada kod uslovno proverava simbolički izraz izvršavanje se grana tako što se ograničava izraz da true bude na grani koja uzima vrednost tačno, a false na ostalim. Posledice ovoga su da se može navesti izvršavanje na bilo koju putanju u programu i pri opasnim operacijama se detektuje da li trenutna ograničenja putanje dozvoljavaju bilo koju vrednost koja uzrokuje grešku. Kada na nekoj putanji pronađe grešku, automatski generiše test slučaj i koristi rešavač ograničenja kako bi razrešio trenutna ograničenja putanja i pronašao konkretne vrednosti koje uzrokuju problem. EXE ograničenja nisu aproksimizovana i kod determinističkog koda dodavanje konkretnog ulaza u neinstrumentalizovanu verziju dovodi do toga da se sledi ista putanja i pronalazi ista greška [2].

EXE precizno modeluje sve operacije sa simboličkim pokazivačima<sup>1</sup>. Može da upravlja ograničenjima koja nastaju aritmetičkim operacijama nad pokazivačima, da čita i piše u memoriju dereferencirajući simbolički pokazivač i da radi sa nizovima simboličke veličine [3].

Takođe, može da obradi sve konstrukte programskog jezika C. Radi sa unijama, poljima bitova, operatorom kastovanja, čak i sa bitovskim opera-

---

<sup>1</sup>Pokazivači čije adrese nisu konkretne već su simbolički ograničene

torima. Ovo je moguće jer svaki bit programa koji se izvršava predstavlja ili memoriju programa ili ima simboličko ograničenje koje je tačno [3].

EXE proširuje efekat pokretanja jedne putanje programa jer upotreba rešavača omogućava rezonovanje o svim mogućim ulaznim vrednostima za tu putanju, a ne o jednom skupu konkretnih vrednosti iz pojedinačnog test slučaja. Sistem ne proverava vrednosti samo na jednoj putanji već se mogu konstruisati vrednosti tako da se, u idealnom slučaju, prolazi kroz sve putanje [3].

### 3 Generisanje test primera

U ovom poglavlju biće objašnjeno kako se pronalaze test primeri. Na najvišem nivou, generisanje test slučajeva ima pet koraka [3]:

1. Korisnik označi deo memorije koji želi da obradi simbolički

```
1000 void make_symbolic(T *obj);  
void make_symbolic_bytes(void *bytes, unsigned nbytes);
```

Prva funkcija označava ceo objekat *obj* kao simbolički, a druga funkcija simbolički obeležava raspon bajtova od *bytes* do *bytes + nbytes*. U početku se sadržaj simboličke memorije tretira bez ograničenja, a ograničenja se dodaju dok se program izvršava.

2. Program se kompajlira exe-cc kompajlerom.
3. Zatim se kod kompajlira običnim kompajlerom (npr. gcc), povezuje sa EXE runtime sistemom i pokreće.
4. Izvršavanje određene putanje se prekida kada se pozove funkcija *exit()*, kada pukne program, kada padne assert ili kada EXE detektuje grešku. Kada se završi izvršavanje putanje EXE razrešava ograničenja konkretnih vrednost.
5. Konkretni ulazi se vraćaju kao test slučajevi. U ovoj fazi program se može kompajlirati uobičajenim kompajlerom bez upotrebe EXE sistema. Lažno pozitivni rezultati nisu mogući, sve što je prijavljeno kao greška je i stvarno greška.

### 4 Kompilacija

Da bi EXE mogao da analizira program potrebno je kod prevesti na određen način. Pre nego što se kod kompajlira nekim uobičajenim kompajlerom, kao što je gcc, potrebno ga je prevesti exe-cc kompajlerom. Ovaj kompajler koristi CIL frontend [5, 3].

#### 4.1 CIL

CIL je međujezik visokog nivoa koji zajedno sa setom alata omogućava jednostavnu analizu i source-to-source transformaciju C programskog koda. U poređenju sa C-om ima manje konstrukata i određene komplikovane konstrukcije C-a razgrađuje na jednostavnije. Na primer, sve konstrukcije petlji se svode na jedan oblik. Tako dobijamo reprezentaciju koja olakšava analizu i upravljanje C programima, a emituje ih u obliku koji liči na izvorni kod[5].

Pošto je CIL konceptualno blizu C-a, zaključci o CIL programu se mogu preslikati na izvorni C program. Pored toga, prevođenje sa CIL-a

na C je vrlo jednostavno. Moguće je i rekonstruisati uobičajene sintaksne idiome u C-u[5].

## 4.2 exe-cc kompajler

Pre kompilacije programa potrebno je program prevesti *exe-cc* kompajlerom da bi se kod pripremio za simboličko izvršavanje. Ovaj kompajler ima tri glavna zadatka[2].

Na početku ubacuje provere oko svake dodele, izraza i grane u programu kako bi utvrdio da li su njegovi operandi stvarni ili simbolički. Operand je konkretan ako i samo ako su svi njegovi sastavni bitovi konkretni. Ako je bilo koji operand simbolički, operacije se ne izvode već se prosleđuju EXE runtime sistemu koji ga dodaje kao ograničenje trenutne putanje. U suprotnom, tj. ako su svi konkretni, operacije se izvode normalno.

Drugi zadatak mu je da umetne kod u program kada dođe do simboličke tačke u grani tako da omogući da se istraži svaka mogućnost. Svaki put kada doda ograničenje grane, EXE poziva STP rešavač da proverí da li postoji bar jedno rešenje za trenutna ograničenja. Ako ne postoji, putanja je nemoguća i EXE zaustavlja njeno izvršavanje.

Treći zadatak je da ubaci kod koji poziva da proverí da li simbolički izraz može imati vrednost koja izaziva null ili referencu van granica memorije ili deljenje sa nulom. Ako se dogodi neki od ovih slučajeva, EXE na true grani potvrđuje da se stanje javlja, šalje test primer i prekida izvršavanje te grane. Na false grani potvrđuje da se stanje ne javlja i nastavlja sa izvršavanjem da bi na taj način pronašao još grešaka.

## 5 STP

Tokom analize programa EXE se oslanja na STP rešavač. STP je rešavač za teoriju bitvektora bez kvantifikatora. Može da reši probleme koji su generisani alatima za analizu programa kao i alatima za automatsko pronalaženje grešaka, fazerima i alatima za proveravanje modela. Primenu nalazi i u dokazivanju teorema i kriptografskim algoritmima[1].

### 5.1 STP i EXE

STP vidi memoriju kao netipizirane bajtove. Podržava tri tipa podataka, bool, bitvektore i nizove bitvektora. Izuzimajući operatore koji rade sa podacima u pokretnom zarezu, svi C operatori se mogu zameniti odgovarajućim STP operatorima koji se mogu koristiti za nametanje ograničenja bitvektorima. STP implementira sve aritmetičke operacije, operacije nad bitovima i relacione operacije. Konstrukcija if-then-else se pretvara u logičku formulu sličnu ternarnom operatoru u C-u. Pored toga, podržava konkatenciju i izdvajanje bitova. STP implementira svoje bitvektor operacije prevodeći ih u operacije na pojedinačnim bitovima. Postoje dva tipa izraza, termi (uzimaju vrednosti bitvektora) i formule (uzimaju logičke vrednosti).

EXE predstavlja simboličke blokove kao nizove osmobarbitnih bitvektora. Prilikom razrešavanja ograničenja prvo se proverava koje memorijske lokacije u označenom kodu sadrže simboličke vrednosti, a zatim se prevode na izraze ograničenja zasnovane na bitvektorima.

Na početku nema simboličkih bitova u označenom delu koda. Kada korisnik označi bajtove  $b$  kao simboličke EXE poziva STP rešavač da kreira niz  $b_{sym}$  identične dužine i beleži u tabeli da  $b$  odgovara  $b_{sym}$ . Tokom izvršavanja programa tabela koja mapira konkretne bajtove u STP bitvektore raste u narednim slučajevima.

1.  $v = e$ : gde je  $e$  simbolički izraz sa najmanje jednim operandom. EXE gradi simbolički izraz  $e_{sym}$  koje predstavlja  $e$  i beleži da ga  $\mathcal{E}v$  mapira. U ovom slučaju se ne dodeljuje nova STP promenljiva već se zamenjuje ovo mapiranje konkretnom vrednošću ili se vrši delokacija.
2.  $b[e]$ : gde je  $e$  simbolički izraz, a  $b$  niz konkretnih vrednosti. STP mora da odluči na koji skup vrednosti se  $b[e]$  može odnositi i zbog toga EXE, umesto da pozove STP za  $b$ , poziva ga za niz  $b_{sym}$  i inicijalizuje ga da ima isti sadržaj kao  $b$ . Zatim beleži da se  $b$  preslikava u  $b_{sym}$  i uklanja to mapiranje samo kada je niz delociran.

Izraz  $e$  se gradi tako što se za svako učitavanje dužine  $n$  iz lokacije za skladištenje  $l$  u  $e$  proverava da li je  $l$  konkretno. Ako jeste zamenjuje ga njegova konkretna vrednost, a ako nije EXE raščlanjuje  $l$  na njegove odgovarajuće bajtove  $b_0, \dots, b_{n-1}$ . Zatim gradi simbolički izraz iste veličine tako što nadovezuje svaki bajt. Za svaki bajt  $b_i$  proverava da li je simbolički. Ako nije koristi trenutnu vrednost koja je osmootbitna konstanta. U slučaju da je bajt simbolički EXE traži i koristi njegov simbolički izraz  $(b_i)_{sym}$ .

STP ima svoja ograničenja. Jedno od tih ograničenja je da ne podržava rad sa pokazivačima. EXE oponaša simboličke pokazivače preslikavajući ih kao referencu na niz. Posledica ovoga je da kada EXE naiđe na dvostruko dereferenciranje simboličkog pokazivača  $**p$  mora da konkretizuje prvo dereferenciranje  $*p$  fiksirajući ga na jedno od mnogo mesta za skladištenje na koje se može odnositi[2].

## 6 Optimizacije

Kako bi analiza koda bila što efikasnija EXE u svom radu koristi određene optimizacione tehnike. Najznačajnije su *keširanje ograničenja* i *nezavisna ograničenja* koje će biti predstavljene u ovom poglavlju.

### 6.1 Keširanje ograničenja

Pošto je pozivanje rešavača skupa operacija, bilo bi lepo kada bi to moglo da se izbegne. Zbog toga EXE kešira rezultate rešenja ograničenja i upita o zadovoljivosti. Ovim kešom upravljaju serverski procesi tako da više procesa može da koordinira. EXE štampa upit kao string, izračunava njegov MD4 kriptografski heš i šalje ga serveru. Server proverava keš i ako dode do pogotka keša vraća rezultat. U slučaju promašaja keša, EXE poziva STP rešavač. Kada dobije rezultat šalje par (heš, rezultat) na server gde se vrši keširanje ovog para[2].

### 6.2 Nezavisna ograničenja

Ideja za ovu optimizaciju počiva na činjenici da se skup ograničenja često može podeliti na više nezavisnih<sup>2</sup> skupova ograničenja. Ovo omogućava

---

<sup>2</sup>Dva ograničenja su nezavisna ako imaju disjunktne skupove operanada

da EXE ne šalje rešavaču sva do sada prikupljena ograničenja već samo podskup ograničenja  $s_c$  kome ograničenje  $c$  pripada i time smanji troškove pozivanja rešavača.

Još jedna prednost ove optimizacije je što povećava broj pogodaka keša. Razlog tome je taj što se dati podskup nezavisnih ograničenja možda ranije već pojavljivao i već se nalazi u kešu. S druge strane, uključivanjem svih mogućih ograničenja povećava se verovatnoća da neko ograničenje bude različito od dosadašnjih što dovodi do promašaja keša.

Nezavisni podskupovi se pronalaze tako što se konstruiše graf  $G$  čiji su čvorovi skup svih operanada koji se koriste u datom skupu ograničenja. Dva čvora se povezuju granom ako i samo ako postoji ograničenje koje sadrži oba operanda. Kada je graf konstruisan primenjuju se algoritmi za određivanje povezanosti komponenti. Za svaku povezanu komponentu kreira se odgovarajući nezavisan podskup ograničenja tako što se dodaju ograničenja koja sadrži bar jedan od čvorova u toj komponenti. Graf  $G$  se ne konstruiše eksplicitno već se koristi *union-find*<sup>3</sup>[4] struktura koja se ažurira prilikom dodavanja novih ograničenja[2].

## 7 Zaključak

U prethodnom tekstu objašnjeno je kako alat EXE pronalazi greške i generiše test primere kod kojih se te greške ispoljavaju. Pored toga predstavljena je i njegova saradnja sa STP rešavačem. Kroz prikaz rada ovog alata mogu se videti i neki osnovi simboličkog izvršavanja programa. Iako su kasnije razvijeni alati dosta više u upotrebi, značaj ovog alata se ne može zanemariti. Predstavlja jedan od prvih alata kod kojih su uspešno prevaziđeni problemi koji su se javljali pri implementaciji simboličkog izvršavanja i samim tim može se reći da ima značajne zasluge za to što je ovaj veoma moćan način analize koda zaživeo i u praksi.

## Literatura

- [1] The Simple Theorem Prover. <https://stp.github.io/>.
- [2] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2), 2008.
- [3] Cristian Cadar, Paul Twohey, Vijay Ganesh, and Dawson Engler. Exe: A system for automatically generating inputs of death using symbolic execution. In *In Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, 2006.
- [4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.
- [5] George C. Necula, Scott Mcpeak, Shree P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *In International Conference on Compiler Construction*, pages 213–228, 2002.

---

<sup>3</sup>Struktura podataka koja stapa grupe elemenata i ispituje jesu li dva elementa u istoj grupi. Svaka grupa je stablo gde je koren predstavnik grupe.

- [6] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson R. Engler. Automatically generating malicious disks using symbolic execution. In *IEEE Symposium on Security and Privacy*, pages 243–257, 2006.