Tehnike simboličkog izvršavanja: Eksploatacija preduslova i ulaznih karakteristika, spajanje stanja, veća primena analize programa i tehnika optimizacije

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Saša Cvetković, 1106/17
mi13171@alas.bg.ac.rs

10. decembar 2018

Sažetak

U okviru ovog rada su prikazane razne tehnike koje mogu pomoću u manjoj ili većoj meri i rešiti problem eksplozije puta koji nastaje usled simboličkog izvršavanja.

Sadržaj

1 Uvod **2**

2 Eksploatacija preduslova i ulaznih karakteristika **2**

3 Spajanje stanja **2**

4 Veća primena analize programa i tehnika optimizacije **4**
4.1 Deljenje programa 4
4.2 Analiza oštećenja 4
4.3 Fuzzing 4
4.4 Predviđanje puta 5
4.5 Provera tipa 5
4.6 Razlike programa 5
4.7 Optimizacija kompajlera 5

5 Zaključak **6**

Literatura **6**

1 Uvod

Jedan od glavnih izazova simboličkog izvršavanja jeste problem eksplozije puta: simboličko izvršavanje može otkazati novo stanje u svakom putu programa i ukupan broj stanja može lako postati eksponencijalan u putevima. Praćenje velikog broja stanja koje treba istražiti, utiču i na vreme rada i na potrebe prostora simboličkog izvršavanja. U ovom radu biće prikazane tehnike eksploatacija preduslova i ulaznih karakteristika, spajanje stanja i veće primene analize programa i tehnika optimizacije.

2 Eksploatacija preduslova i ulaznih karakteristika

Jedan od načina smanjivanja eksplozija puta jeste eksploatacija znanja o nekim ulaznim osobinama. AEG[1] predlaže preduslovno simboličko izvršavanje, kako bi se smanjio broj ispitanih stanja, usmeravajući tako istraživanje na podskup ulaza koji zadovoljavaju predikat preduslova. Ideja je da se fokusira na ulaze koji mogu dovesti do određenih ponašanja programa (npr. sužavanje istraživanja na ulaze maksimalne veličine da bi otkrili potencijalne poplave bafera). Preduslovi simboličkog izvršavanja menjaju ispravnost za performanse: dobro osmišljeni preduslovi trebaju biti ni previše specifični (oni bi propustili zanimljive putanje) niti previše generički (oni bi kompromitovali ubrzanja koja proizilaze iz redukcije prostora). Umesto da počnemo od prazne putanje bez postavljenih uslova, ovaj pristup dodaje pretpostavke početnom π tako da će u ostataku istraživanja biti preskočeni delove koji ih ne zadovoljavaju. Dodavajući više uslova na π pri inicijalizaciji potencijalno dolazi do povećavanja vremena izvršavanja, jer dolazi do potrebe da se izvrši veći broj provera na svakoj od putanja, međutim ovo može biti u velikoj meri nadmašeno dobitkom performansi zbog manjeg broja stanja.

Uobičajene vrste preduslova koji se uzimaju u simboličkom izvršavanju su: poznata dužina (tj veličina bafera je poznata), poznati prefiks (tj., bafer ima poznati prefiks) i potpuno je poznat (tj. sadržaj bafera je potpuno konkretan). Ovi preduslovi su prilično dobri kada se bave kodom koji radi sa ulazom koji ima dobru, poznatu ili unapred definisanu strukturu, kao što su to parseri stringova ili alatke za obradu paketa.

3 Spajanje stanja

Spajanje stanja je moćna tehnika koja spaja različite putanje u jedno stanje. Spojeno stanje je opisano formulom koja predstavlja disjunkciju formula kojim bi se opisivale pojedinačna stanja ako bi bila odvojena. Za razliku od drugih tehnika analize statičkih programa, kao što je apstraktno tumačenje, spajanje stanja u simboličkom izvršavanju ne dovodi do prekomerne aproksimacije.

Razmena: Da li da se spoje ili da se ne spajaju? U principu, može biti isplativo primeniti spajanje stanja kada su u pitanju simbolička stanja koja bi procenila istu naredbu vrlo slično (tj., razlikuju se samo za nekoliko elemenata) u svojim simboličkim reprezentacijima. Neka imamo dva stanja (*stmt*, σ_1, π_1) i (*stmt*, σ_2, π_2)[2] (Odeljak 1.1), spojeno stanje se može konstruisati kao (*stmt*,σ", π_1 v π_2), gde je σ" je spojena simbolička reprezentacija između σ_1 i σ_2 izgrađena sa izrazima *ite* koji predstavljaju razlike u reprezentaciji, dok π_1 v π_2 je spoj uslova puteva iz dva spojena stanja. Strukture kontrolnog toka kao što su if-else naredbe (kao u prethodnom primeru) ili jednostavne petlje često daju prilično slično naredno stanje pa predstavljaju veoma dobre kandidate za spajanje stanja.
Rani radovi [3], [4] pokazali su da tehnike spajanja efikasno smanjuju broj puteva koje treba istražiti, ali i stavljaju opterećene na rešavače uslova, koji mogu biti ometani od strane disjunkcija. Spajanje može takođe uvesti nove simboličke izraze u kod, npr. kada se spajaju različite konkretne vrednosti od uslova u simbolički izraz preko stanja. [5] pruža odličnu diskusiju o dizajnerskom prostoru u tehnikama spajanja sa stanjima. Na jednom kraju spektra, potpuno razdvojeni putevi korišćenih u simboličkom izvršavanju na osnovu pretrage [2] (Odeljak 2.2) ne vrše spajanje. Na drugom kraju, statičko spajanje stanja kombinuje stanja u zajedničkim tačkama kontrolnog toka, u suštini predstavlja ceo program sa jedinstvenom formulom. Korišćeno je statično spajanje stanja u celom programu kao verifikacija celog programa generatorom stanja [6], [7]), koje obično menja preciznost za skalabilnost, npr., prolaz petlje samo jednom.

Spajanje heuristikom Ovo rešenje spajanja koristi heuristiku koja identifikuje stanja, čijim bi se spajanjem ubrzao proces istraživanja. Stvaranje većih simboličkih izraza i eventualno dodatnih rešavača za rešenje može prevagnuti u korist manjeg broja stanja, al to dovodi do lošijeg ukupnog učinka [4], [5]. Procena broja upita [5] oslanja se na jednostavnu statičku analizu kako bi identifikovala koliko često se svaka varijabla koristi u uslovima puta u bilo kojoj tački u CFG. Procena se koristi kao predstavnik broja rešavača upita kojim će određena varijabla biti deo. Dva stanja čine dobre kandidate za spajanje kada se očekuje da se njihove različite varijable pojavljuju retko u kasnijim upitima. Veritesting [16] primenjuje oblik spajanja heuristika zasnovan na razlikovanju između prostih i složenih naredbi, pri čemu složene uključuju indirektne skokove, sistemske pozive i druge operacije za koje je teško postići precizne statičke analize. Statičko spajanje se vrši na sekvencama prostih naredbi čiji se efekti čuvaju pomoću izraza *ite*, a simboličko istraživanje po putu se vrši kad god se nailazi na naredbu koja je teška za analizu.

Dinamičko spajanja stanja Da bi se maksimizirale prilike za spajanje, simboličko izvršavanje bi trebalo da pređe CFG tako da se kombinovano stanje za tačku programa može izračunati od njegovih prethodnika npr. ako je grafik acikličan, prateći topološki poredak. Međutim, to bi sprečilo pretraživanje strategije istraživanja koja daju prednost interesantnim stanjima. [5] uvodi dinamičko spajanje stanja koje radi bez obzira na strategiju istraživanja koje je nametnuto strategijom pretraživanja. Pret-

postavimo da je simboličko izvršavanje održava listu stanja i ograničenu istoriju svojih prethodnika. Kada izvršavanje mora da izabere sledeće stanje koje će istraživati, prvo proverava da li postoje dva stanja σ_1 i σ_2 sa liste tako da se ne podudaraju za spajanje, ali da se σ_1 i prethodnik od σ_2 podudaraju. Ako je očekivana sličnost između σ_2 i naslednika σ_1 takođe visoka, algoritam pokušava da ih spoji sa preduhitravanjem izvršavanja σ_1 za određeni broj koraka. Ovo obuhvata ideju da ako su dva stanja slična, onda će i njihovi sledbenici verovatno postati slični u nekoliko koraka. Ako spajanje ne uspe, algoritam dozvoljava da se heurističkom pretragom izabere sledeće stanje koje će se istraživati.

4 Veća primena analize programa i tehnika optimizacije

Dublje razumevanje ponašanja programa može pomoći simboličkoj tehnici da optimizuje svoju analizu i da se fokusira na kvalitetnija stanja, npr. skraćivanje nezanimljivih delova stabla računanja. Postoji nekoliko klasičnih tehnika analize programa.

4.1 Deljenje programa

Ova analiza, počinje od podskupa ponašanja program i izvlači iz programa minimalni niz instrukcija koje verno opisuje njegovo ponašanje [8]. Ove informacije mogu pomoći simboličkoj tehnici na nekoliko načina: npr. [9] rekurzivno poziva deljenje programa da bi ograničio simboličko istraživanje prema određenoj ciljnoj tački u programu.

4.2 Analiza oštećenja

S Ova tehnika [10] pokušava proveriti koje varijable programa mogu imati vrednosti koje potiču od potencijalno opasnih spoljašnjih izvora kao što je korisnički ulaz. Analiza se može izvoditi i statički i dinamički, s tim da dinamičku pristup daje tačnije rezultate. U kontekstu simboličkog izvršavanja, analiza oštećenja može pomoći simboličkoj tehnici tako što otkriva koji putevi zavise od oštećenih vrednosti. Npr.,[11] fokusira svoju analizu na puteve gde je instrukcija skoka oštećena i koristi simboličko izvršavanje da generiše eksplotaciju.

4.3 Fuzzing

Ovaj pristup testiranja softvera proizvoljno uvećava broj kreiranih test ulaza korisnika, kako bi doveo do pada programa ili propusta, eventualno pronalaska potencijalnih curenja memorije. Fuzzing se može povećati sa simboličkim izvršavanjem da bi se prikupili uslovi za ulaz i isti negirali, tako da bi se generisali novi ulazi. S druge strane, simboličko izvršavanje može se dodatno povećati fuzzing da bi istraživanjem došao do dubljih stanja brže i efikasnije. Predstavljena su dva značajna ostvarenja ove ideje *hybrid concolic testing* [12] i Driller [13].

4.4 Predviđanje puta

Ovo je strategija za ublažavanje kaznenih mera koje su predviđene, za loše predviđanje toka izvršavanja, tako što se izbegavaju skokovi na veoma male segmente koda: npr. kontrola toka programa kao što je ternarni operator, može se zameniti sa prethodno odabranom instrukcijom. [19] izveštava eksponencijalno smanjenje broja grana za istraživanje od usvajanja ove strategije kada istovremeno vrši proveru dve implementacije programa koristeći simboličko izvršavanje.

4.5 Provera tipa

Simbolička analiza se može često mešati sa proverom unosa [17]: npr. provera tipa može odrediti povratni tip funkcije koju je teško simboličko analizirati: takve informacije može potencijalno iskoristiti izvršavanje da obriše određene grane.

4.6 Razlike programa

Analizom zavisnosti se mogu utvrditi putevi i tok podataka pod uticajem promene koda. Direktno inkrementalno simboličko izvršavanje [18] statički identifikuje CFG čvorove na koje utiču promene, i koristi takve informacije da istraži samo one puteve koji se izvršavaju otkrivenim sekvencama ugroženih čvorova.

4.7 Optimizacija kompajlera

[14] tvrdi da bi tehnike optimizacije programa trebale biti prvi deo praktičnih implementacija simboličkog izvršavanja, zajedno sa široko prihvaćenim rešenjima kao što su pretraživanja heuristikom, spajanje stanja i optimizacija rešavanja uslova. Zapravo, promene u programu mogu uticati i na složenost uslova koji se generišu tokom istraživanja puta i samog istraživanja. Npr. preračunavanje rezultata funkcije pomoću tabele za pregled dovodi do većeg broja uslova u uslovima puta usled pristupa memoriji, dok primena smanjenja snage za umnožavanje može dovesti do lanca dodatnih operacija koje su skuplje za rešavanje uslova. Takođe, način na koji su kompajlirane složene naredbe poput switch-a - mogu značajno uticati na performanse istraživanja putanje, dok se pribegavanje uslovnim instrukcijama kao što je odabir u LLVM ili setcc i cmov u x86 može izbeći skupo stanje tako što se umesto njega daje jednostavni *ite* izraz. Iako se efekti optimizacije kompajlera mogu obično predvideti po broju ili veličini instrukcija izvršenih u toku rada, slično smanjenje nije očigledno u simboličkom izvršavanju [15], uglavnom zato što se kao rešenje uslova tipično koristi crno kutija .

5 Zaključak

Tehnike simboličkog izvršavanja su se značajno razvile u poslednjoj deceniji, sa značajnim dostignućima u izazivanje problema iz nekoliko domena poput testiranja softvera (npr. generisanje testnih ulaza, regresijska ispitivanja), sigurnost (npr. generisanje eksploatacije, autentifikacije) i analiza koda (npr. dinamičko ažuriranje softvera). Ovaj trend nije poboljšao samo postojeća rešenja, već je doveo do novih ideja i, u nekim slučajevima, velikim praktičnim novinama.

Literatura

[1] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2011. AEG: Automatic Exploit Generation. In Proc. Network and Distributed System Security Symp. (NDSS'11).

[2] A survey of symbolic execution techniques, ROBERTO BALDONI, EMILIO COPPA, DANIELE CONO D'ELIA, CAMIL DEMETRESCU, and IRENE FINOCCHI,Sapienza University of Rome

[3] Patrice Godefroid. 2007. Compositional Dynamic Test Generation. In Proc. 34th ACM SIGPLAN-SIGACT Symp. on Principles of Progr. Lang. (POPL'07). 47–54.

[4] Trevor Hansen, Peter Schachte, and Harald Søndergaard. 2009. Runtime Verification. Chapter State Joining and Splitting for the Symbolic Execution of Binaries, 76–92.

[5] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient State Merging in Symbolic Execution. In Proc. 33rd ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI'12). ACM, 193–204

[6] Domagoj Babic and Alan J. Hu. 2008. Calysto: Scalable and Precise Extended Static Checking. In Proc. 30th Int. Conf. on Software Engineering (ICSE'08). ACM, 211–220.

[7] Yichen Xie and Alex Aiken. 2005. Scalable Error Detection Using Boolean Satisfiability. In Proc. 32nd ACM SIGPLANSIGACT Symp. on Principles of Progr. Lang. (POPL'05). ACM, 351–363

[8] Mark Weiser. 1984. Program Slicing. IEEE Trans. on Software Engineering SE-10, 4 (1984), 352–357

[9] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In 22nd Annual Network and Distributed System Security Symp. (NDSS'15).

[10] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In Proc. 2010 IEEE Symp. on Security and Privacy (SP'10). IEEE Computer Society, 317–331

[11] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In Proc. 2012 IEEE Symp. on Sec. and Privacy (SP'12). IEEE Comp. Society, 380–394

[12] Rupak Majumdar and Koushik Sen. 2007. Hybrid Concolic Testing. In Proc. 29th Int. Conf. on Software Engineering (ICSE'07). IEEE Computer Society, 416–426

[13] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In 23nd Annual Network and Distr. System Sec. Symp. (NDSS'16)

[14] Cristian Cadar. 2015. Targeted Program Transformations for Symbolic Execution. In Proc. 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15). ACM, 906–909

[15] Shiyu Dong, Oswaldo Olivo, Lingming Zhang, and Sarfraz Khurshid. 2015. Studying the Influence of Standard Compiler Optimizations on Symbolic Execution. In Proc. 2015 IEEE 26th Int. Symp. on Software Reliability Engineering. 205–215

[16] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing Symbolic Execution with Veritesting. In Proc. 36th Int. Conf. on Software Engineering (ICSE'14). ACM, 1083–1094.

[17] Yit Phang Khoo, Bor-Yuh Evan Chang, and Jeffrey S. Foster. 2010. Mixing Type Checking and Symbolic Execution. In Proc. 31st ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI'10). 436–447.

[18] Guowei Yang, Suzette Person, Neha Rungta, and Sarfraz Khurshid. 2014. Directed Incremental Symbolic Execution. ACM Trans. on Software Engineering and Methodology (TOSEM) 24, 1, Article 3 (2014)

[19] Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. 2011. Symbolic Crosschecking of Floating-point and SIMD Code. In Proc. Sixth Conf. on Computer Systems (EuroSys'11). ACM, 315–328.