UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Đorđe Todorović

PODRŠKA ZA NAPREDNU ANALIZU PROMENLJIVIH LOKALNIH ZA NITI POMOĆU ALATA GNU GDB

master rad

Beograd, 2018.

Mentor:

dr Mika Mikić, redovan profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Ana Anić, vanredni profesor
University of Disneyland, Nedođija

dr Laza Lazić, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Mami, tati i dedi

Sadržaj

1 Uvod **1**

2 Kako rade debageri? **2**

2.1 Prevođenje programa 3

2.2 Fajl format DWARF 3

2.3 Linux debageri 6

3 GNU GDB alat **12**

4 Datoteke jezgra **13**

5 TLS **14**

5.1 Motivacija 14

5.2 Rukovanje TLS-om tokom izvršavanja programa 14

5.3 Arhitekturalno specifične zavisnosti 14

5.4 TLS Modeli pristupa 14

6 Implementacija rešenja **15**

7 Zaključak **16**

Literatura **17**

Glava 1

Uvod

Glava 2

Kako rade debageri?

Greške su sastavni deo svakog rada koji obavlja čovek, te i programeri prave iste. Greške mogu biti hardverske i softverske. One mogu imati razne poslednice. Neke su manje važne, kao npr. korisnički interfejs aplikacije ima neočekivanu boju pozadine. Postoje i greške koje mogu imati daleko veće posledice, pa čak i ugroziti živote drugih, kao npr. greške u softveru ili hardveru uređaja i aplikacija avio industrije. Faza testiranja je veoma važna u ciklusu razvoja informacionih sistema. Nakon faze testiranja obično sledi faza analize i otklanjanja grešaka.

Debager (eng. *debagger*) je softverski alat koji koriste programeri za testiranje, analizu i otklanjanje grešaka u programima. Sam proces korišćenja takvih alata nazivamo debagovanjem (eng. *debugging*). Debageri mogu pokrenuti rad nekog procesa ili se „nakačiti" na proces koji je već u fazi rada. U oba slučaja, debager preuzima kontrolu nad procesom. To mu mogućava da izvršava proces instrukciju po instrukciju, do postavlja tačke prekida (eng. *brakpoints*) itd. Proces izvršavanja programa od strane debagera sekvencijalno, instrukciju po instrukciju ili liniju po liniju, nazivamo koračanje. Tačke prekida su mogućnost debagera da zaustavi izvršavanje programa na određenoj tački. To može biti trenutak kada program izvrši određenu funkciju, liniju koda itd. Neki debageri imaju mogućnost izvršavanja funkcija programa koji se debaguje, uz ograničenje da isti pripada istoj procesorskoj arhitekturi kao i domaćinski sistem na kojem se debager izvršava. Čak i struktura programa može biti promenjena, prateći propratne efekte.

Podršku debagerima, u opštem slučaju, daju operativni sistemi, kroz sistemske pozive koji omogućavaju tim alatima da pokrenu i preuzmu kontrolu nad nekim drugim procesom. Za neke naprednije tehnike debagovanja poželjna je podrška od strane hardvera. U radu će detaljno biti obrađen rad UNIX-olikih, posebno Linux

debagera. Windows debageri i programski prevodioci ne prate standard DWARF [2] prilikom baratanja sa debag informacijama namenjene za taj operativni sistem. Windows alati za debagovanje konsultuju standard *Majkrosoft CodeView*, više informacija možete pronaći na [5].

2.1 Prevođenje programa

Programi koji se debaguju se prevode uz pomoć opcije programskih prevodioca `-g`. Ukoliko je program koji se analizira prevoden bez optimizacija, debag informacije koje prate program su potpune. Programi koji se puštaju u produkciju, da bi bili brži i zauzimali manje memorije, se prevode uz pomoć optimizacija. Nivoi optimizacija produkcijskih programa su „-O2" i „-O3". Prilikom optimizacija se gube razne debag informacije. Neke promenljive i funkcije programa neće biti predstavljene debag informacijama. Npr. funckije koje se ne koriste će biti optimizovane i izbrisane iz mašinskog koda programa, te samim tim i debag informacije za nju će biti izostavljene. Promenljiva programa može biti živa samo u nekim određenim delovima programa, pa programski prevodioci generišu debag informacije o njenim lokacijama samo u tim određenim delovima koda.

2.2 Fajl format DWARF

DWARF je debag fajl format koji se koristi od strane programskih prevodioca (kao npr. GCC ili LLVM/Clang) i debagera (kao npr. GNU GDB) da bi se omogućilo debagovanje na nivou izvornog koda. Omogućava podršku za razne programske jezike kao što su C/C++ i Fortran, ali je dizajniran tako da se lako može proširiti na ostale jezike. Arhitekturalno je nezavistan i predstavlja „most" između izvornog koda i izvršnog fajla. Trenutno je DWARF verzija 5 poslednji realizovani standard.

DWARF debug fajl format se odnosi na Unix-olike operativne sisteme, kao što su Linux i MacOS. Generisane debag informacije, prateći DWARF standard, su podeljene u nekoliko sekcija sa prefiksom `.debug_`. Neke od njih su `.debug_line`, `.debug_loc` i `.debug_info` koje redom predstavljaju informacije o linijama izvornog koda, lokacijama promenljivih i ključna debag sekcija koja sadrži debug informacije koje referišu na informacije iz ostalih debug sekcija. DWARF je predstavljen kao drvolika struktura, smeštena u .debug_info sekciju, koja razne entitete programskog jezika opisuje osnovnom debag jedinicom DIE (eng. Debug Info Entry).

Osnovna debag jedinica može opisivati lokalnu promenljivu programa, formalni parametar, funkciju itd. Svaka od njih je identifikovana DWARF tagom koji predstavlja informaciju o toj jedinici, gde je npr. tag za lokalne promenljive predstavljen sa `DW_TAG_local_variable`, ili tag za funkciju je obeležen sa `DW_TAG_subprogram`. Svaka debag jedinica je opisana određenim DWARF atributima sa prefikosm `DW_AT_`. Oni mogu ukazivati na razne informacije o entitetu kao sto su ime promenljive ili funkcije, liniju deklaracije, itd. Koren svakog DWARF stabla je predstavljen debug jedinicom, sa tagom `DW_TAG_compile_unit`, koja predstavlja kompilacionu jedinicu, tj. izvorni kod programa. Primer dela DWARF stabla za primer predstavljen 2.1 je dat slikom 2.2.

```c
#include <stdio.h>

int main()
{
    int x;
    x = 5;

    printf("The value is %d\n", x);

    return 0;
}
```

Slika 2.1: Izvorni kod test primera koji se analizira.

```
<1><73>: Abbrev Number: 4 (DW_TAG_subprogram)
   <74>   DW_AT_external    : 1
   <74>   DW_AT_name        : (indirect string, offset: 0xa6): main
   <78>   DW_AT_decl_file   : 1
   <79>   DW_AT_decl_line   : 3
   <7a>   DW_AT_type        : <0x57>
   <7e>   DW_AT_low_pc      : 0x400526
   <86>   DW_AT_high_pc     : 0x2a
   <8e>   DW_AT_frame_base  : 1 byte block: 9c        (DW_OP_call_frame_cfa)
   <90>   DW_AT_GNU_all_tail_call_sites: 1
<2><90>: Abbrev Number: 5 (DW_TAG_variable)
   <91>   DW_AT_name        : x
   <93>   DW_AT_decl_file   : 1
   <94>   DW_AT_decl_line   : 5
   <95>   DW_AT_type        : <0x57>
   <99>   DW_AT_location    : 2 byte block: 91 6c      (DW_OP_fbreg: -20)
```

Slika 2.2: Debug jedinice DWARF stabla primera sa slike 2.1.

Funkcija `main` primera sa slike 2.1 na slici 2.2 je predstavljena DWARF tagom `DW_TAG_subprogram`. Atribut te debag jedinice predstavljen sa `DW_AT_name` ima vrednost imena funkcije. Atributi `DW_AT_low_pc` i `DW_AT_high_pc` redom predstavljaju adresu prve mašinske instrukcije te funkcije u memoriji programa i ofset na kojem se nalazi poslednja mašinska instrukcija te funkcije. Sledeci čvor drveta predstavlja promenljivu `x` istog test primera. Ta debag jedinica je dete čvora koji predstavlja `main` funkciju i ukazuje da se promenljiva `x` nalazi unutar `main` funkcije. Promenljiva `x` je predstavljena DWARF tagom `DW_TAG_variable`. Atribut `DW_AT_name` predstavlja ime promenljive, `DW_AT_type` referiše na debag jedinicu koja predstavlja tip promenljive, dok `DW_AT_location` atribut predstavlja lokaciju promenljive u memoriji programa.

Debag promenljive

Svaka promenljiva programa prevedenog sa debag informacijama, ukoliko se ne radi o optimizovanom programu, je predstavljena DWARF tagom `DW_TAG_variable`. Atribut `DW_AT_location` ukazuje na lokaciju promenljive. Lokacija može biti predstavljena DWARF izrazom, kao npr. lokacija promenljive `x` na slici 2.2. DWARF izraz te promenljive ukazuje da se ona nalazi na ofsetu -20 trenutnog stek okvira `main` funkcije. U neoptimizovanom kodu sve promenljive imaju lokacije zadate DWARF izrazom. Njihove vrednosti su dostupne debagerima u bilo kom delu koda u kom su definisane.

U optimizovanom kodu lokacija promenljive može sadržati referencu na informaciju o lokaciji u `.debug_loc` sekciji. Lokacije u toj sekciji su predstavljene listama lokacija. Jedna promenljiva u optimizovanom kodu može biti smeštena na raznim memorijskim lokacijama ili registrima. Elementi liste opisuju lokacije promenljive na mestima u kodu gde je ona živa. Ukoliko promenljiva nije živa u nekom delu koda, programski prevodioci u optimizovanom kodu neće pratiti njenu lokaciju. Slika 2.3 predstavlja primer lokacije promenljive u optimizovanom kodu.

```
<2><2cd>: Abbrev Number: 19 (DW_TAG_variable)
   <2ce>   DW_AT_name        : x
   <2d0>   DW_AT_decl_file   : 1
   <2d1>   DW_AT_decl_line   : 5
   <2d2>   DW_AT_type        : <0x5e>
   <2d6>   DW_AT_location    : 0x0 (location list)
```

Slika 2.3: Debag jedinica promenljive `x` u optimizovanom kodu.

Lokacijska lista promenljive `x` je predstavljena na slici 2.4. U ovom konkretnom primeru, promenljiva živi samo na jednom mestu. Potencijalno je mogla imati još elemenata lokacijske liste. `Offset` predstavlja informaciju gde se lokacijska lista određene promenljive nalazi u `.debug_loc` sekciji. `Begin` i `End` predstavljaju informaciju od koje do koje adrese u programu važi data lokacija, tj. od koje do koje instrukcije je određena promenljiva živa. `Expression` predstavlja DWARF izraz koji opisuje lokaciju promenljive.

```
Contents of the .debug_loc section:

    Offset   Begin            End              Expression
    00000000 0000000000400568 0000000000400583 (DW_OP_fbreg: -28)
    00000014 <End of list>
```

Slika 2.4: Lokacijska lista promenljive `x` u optimizovanom kodu.

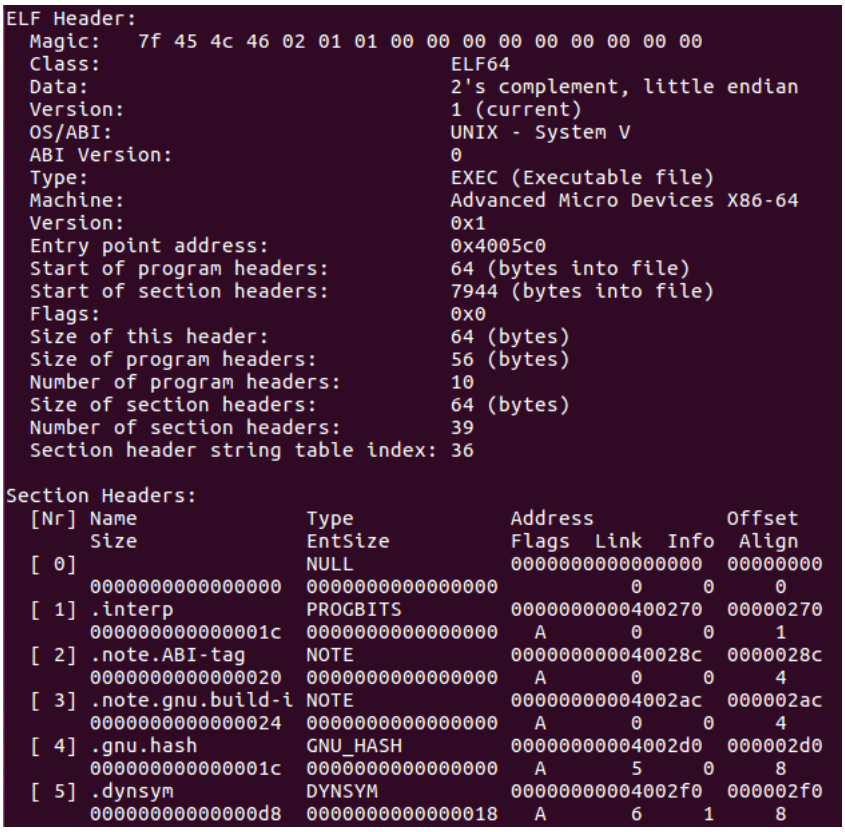
2.3 Linux debageri

Najpre definišimo neke od osnovnih pojmova Linux programiranja koje ćemo često pominjati u radu. Osvrnimo se prvo na format izvršnih fajlova, deljenih biblioteka, objektnih fajlova i datoteka jezgara ELF (eng. *Executable and Linkable Format*)[1].

ELF sadrži razne informacije o samom fajlu. Podeljen u dva dela: ELF zaglavlje i podaci fajla. ELF zaglavlje sadrži informacije o arhitekturi za koju je program preveden i definiše da li program koristi 32-bitni ili 64-bitni adresni prostor. Zaglavlje 32-bitnih programa je dužine 52 bajta, dok kod 64-bitnih programa zaglavlje je dužine 64 bajta. Podaci fajla mogu sadržati programsku tabelu zaglavlja (eng. *Program header table*), sekcijsku tabelu zaglavlja (eng. *Section header table*) i ulazne tačke prethodne dve tabele. Slika 2.5 prikazuje primer prikaza fajla u formatu ELF.

Osim fajl formata ELF, veoma bitan fajl format je DWARF, koji predstavalja format zapisa debag informacija koje debageri koriste kada analiziraju programe.

Operativni sistem GNU Linux pruža sistemski poziv `ptrace` [3] koji debagerima omogućava rad. Ovaj sistemski poziv omogućava jednom procesu kontrolu nad izvršavanjem nekog drugog procesa i menjanje memorije i registara istog. Potpis ove funkcije je:



Slika 2.5: ELF fajl format počitan alatom *objdump*.

```
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void
*data);
```

Prvi argument sistemskog poziva predstavlja informaciju kojom operativnom sistemu jedan proces, ne nužno debager, ukazuje na nameru preuzimanja kontrole drugog procesa. Ukoliko isti ima vrednost `PTRACE_TRACEME` to ukazuje na nameru praćenja (eng. *tracing*) određenig procesa, `PTRACE_PEEKDATA` i `PTRACE_POKEDATA` redom ukazuju na nameru čitanja i pisanja memorije, `PTRACE_GETREGS` i `PTRACE_SETREGS` se odnose na čitanje i pisanje registara. To su samo neki osnovni slučajevi korišćenja, za više informacija pogledati [3]. Drugi argument sistemskog poziva `pid` ukazuje na identifikacioni broj ciljanog procesa. Treći i četvrti argument se po potrebi koriste u zavisnosti od namere korišćenja sistemskog poziva `ptrace` za čitanje ili pisanje sa adrese datom trećim argumentom, pri tom baratajući podacima na adresi zadatoj četvrtim argumentom. To znači ukoliko se koristi `PTRACE_TRACEME` poslednja

tri argumenta sistemskog poziva se ignorišu. Ukoliko se koristi `PTRACE_GETREGS` sa adrese `addr` se čita jedna reč iz memorije. Navodimo par osnovnih primera korišćenja `ptrace` sistemskog poziva, a u nastavku će biti navedeno još primera.

Primer 1 *Program inicira da će biti praćen od strane roditeljskog procesa:*

```
ptrace(PTRACE_TRACEME, 0, NULL, NULL);
```

Primer 2 *Čitanje vrednosti registara procesa sa identifikatorom* `8845` *i upisivanje tih vrednosti na adresu promenljive* `regs`*:*

```
ptrace(PTRACE_GETREGS, 8845, NULL, &regs);
```

Tačke prekida

Postoje dve vrste tačaka prekida (eng. *brakpoints*): softverske i hardverske [4].

Osvrnimo se prvo na softverske tačke prekida. Postavljanje tačaka prekida predstavlja jednu od najkorišćenijih mogućnosti debagera, te stoga navedimo par smernica kako je ista realizovana, u opštem slučaju. Ali takođe treba napomenuti da ne postoji jedinstveni poziv nekog sistemskog poziva za postavljanje tačke prekida, već se ista obavlja kao kombinacija više mogućnosti sistemskog poziva `ptrace`. Opišimo ceo postupak na jednostavnom primeru. Program je preveden za procesorsku arhitekturu Intel x86-64 i asemblerski kod `main` funkcije primera izgleda kao na primeru 2.6.

Primera radi, želimo da postavimo tačku prekida na treću po redu instrukciju funkcije `main`:

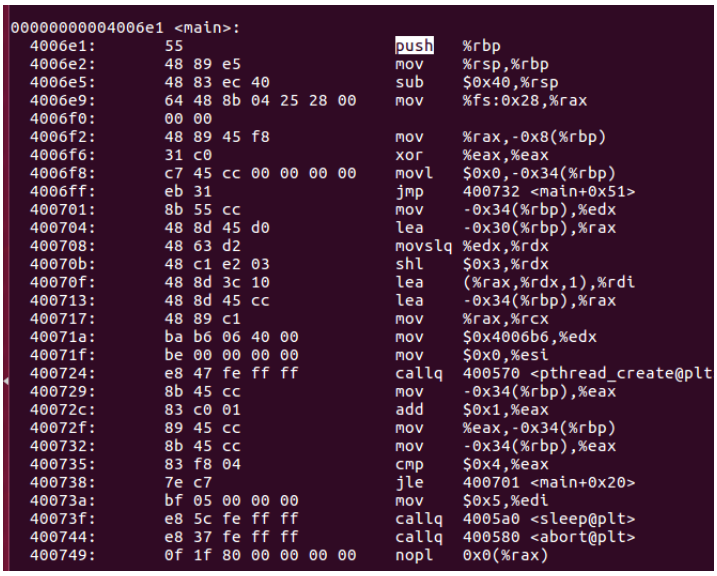
```
48 89 e5 mov %rsp,%rbp
```

Da bismo to uradili, menjamo prvi bajt instrukcije sa posebnom magičnom vrednošću, obično `0xCC`, i kada izvršavanje dostigne do tog dela koda ono će se zaustaviti na tom mestu.

Pošto `0x48` menjamo sa `0xcc` i na tom mestu u kodu dobijamo instrukciju:

```
cc 89 e5 int3
```

Instrukcija `int3` je posebna instrukcija procesorske arhitekture Intel x86-64, koja izazva softverski prekid. Kada registar programski brojač (eng. *CPU register pc*) stigne do `int3` instrukcije izvršavanje se zaustavalja na toj tački. Debager je već upoznat od strane korisnika i svestan je da je tačka prekida postavljena te isti čeka na signal koji ukazuje na to da je program dostigao do instrukcije prekida. Operativni sistem prepoznaje `int3` instrukciju, poziva se specijalni obrađivač tog signala (na



Slika 2.6: Asemblerski kod `main` fukcije test primera na x86-64 platformi.

Linux sistemima `do_int3()`), koji dalje obaveštava debager šaljući mu signal sa kodom `SIGTRAP` koji on obrađuje na željeni način. Treba napomenuti da ovo važi za procesorsku arhitekturu Intel x86-64, instrukcija prekida za arhitekture kao što su ARM, MIPS, PPC itd., se drugačije kodira, ali postupak implementacije tačaka prekida je isti.

Ukoliko želimo da stavimo tačku prekida eksplicitno na funkciju `main`, za to koristimo posrednika u vidu DWARF debug informacija. U tom slučaju debager traži element DWARF stabla koji ukazuje na informacije o `main` funkciji, i odatle dohvata informaciju na kojoj adresi u memoriji se nalazi prva mašinska instrukcija date funkcije. Na slici 2.7 vidimo DWARF element koji opisuje funkciju uz pomoć atributa. Debager će pročitati `DW_AT_low_pc` atribut i na tu adresu postaviti `int3` instrukciju. Napomenimo da DWARF debag simbole generišemo uz pomoć `-g` opcije kompajlera.

Hardverske tačke prekida su direktno povezane sa hardverom u vidu specijalnih registara. Postavlja se na određenu adresu i harverski *watchpoint* montri na zadatu adresu i može signalizirati razne promene na istoj, npr. čitanje, pisanje ili izvršavanje, što im daje prednost u odnosu na softverske tačke prekida. Mane u odnosu na softverske tačke prekida su performanse, gde su iste neuporedivo sporije, i takođe neophodna hardverska podrška za korišćenje hardverskih tačaka prekida.

```
<1><bd>: Abbrev Number: 6 (DW_TAG_subprogram)
   <be>   DW_AT_external    : 1
   <be>   DW_AT_name        : (indirect string, offset: 0x21): main
   <c2>   DW_AT_decl_file   : 1
   <c3>   DW_AT_decl_line   : 15
   <c4>   DW_AT_prototyped  : 1
   <c4>   DW_AT_type        : <0x57>
   <c8>   DW_AT_low_pc      : 0x4006e1
   <d0>   DW_AT_high_pc     : 0x68
   <d8>   DW_AT_frame_base  : 1 byte block: 9c        (DW_OP_call_frame_cfa)
   <da>   DW_AT_GNU_all_tail_call_sites: 1
   <da>   DW_AT_sibling     : <0xfa>
```

Slika 2.7: `main` funkcija prikazana DWARF potprogramom.

Koračanje

Pod procesom koračanja (eng. *stepping*) kroz program podrazumevamo izvršavanje programa sekvencu po sekvencu. Sekvenca može biti jedna procesorska instrukcija, linija koda ili pak neka funkcija programa koji se debaguje [4].

Instrukcijsko koračanje na platformi Intel x86-64 je direktno omogućeno kroz sistemski poziv `ptrace`:

```
ptrace(PTRACE_SINGLESTEP, debuggee_pid, nullptr, nullptr);
```

Operativni sistem će dati signal kada je korak izvršen.

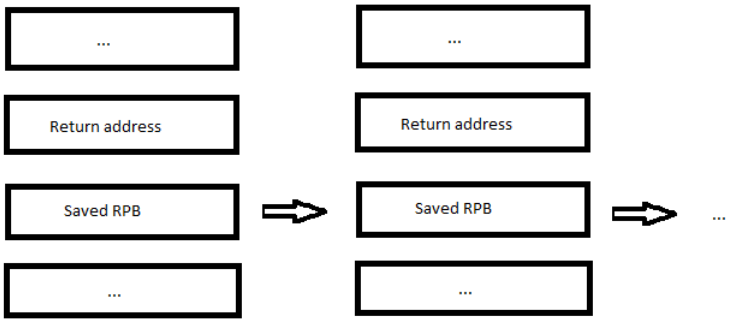
Pored instrukcijkog koračanja pomenućemo još jednu vrstu zakoračavanja u neku funkciju koja je pozvana `call` ili `jump` instrukcijom. Komanda koja nam to omogućava jeste `step in`.

Treba napomenuti da postoje arhitekture za koje ovo ne važi, kao npr. platforma ARM, koja nema hardversku podršku za instrukcijsko koračanje i za njih se koračanje implementira na drugačiji način, uz pomoć emulacije instrukcija, ali u ovom radu neće biti opširno o tome.

Izlistavanje pozivanih funkcija

Objasnimo komandu izlistavanje pozivanih funkcija (eng. *backtrace*) posmatrajući organizaciju stek okvira (eng. *stack frames*) na platformi Intel x86-64 [4].

Na slici 2.8 navedeni su stek okviri za dva funkcijska poziva. Pre povratne vrednosti funkcije obično se ređaju argumenti funkcije. Sačuvana adresa u registru `RBP` jeste adresa stek okvira svog pozivaoca. Prateći iste kao elemente povezane liste dolazimo do svih pozivanih funkcija do zadate tačke. Ako se pitamo kako debager ima informaciju o imenu funkcije odgovor je u tome što pretražuje DWARF stablo



Slika 2.8: Primer ređanja stek okvira na x86-64 platformi.

sa debag informacijama, tražeći `DW_TAG_subprogram` sa odgovarajućom povratnom adresom, pritom čitajući `DW_AT_name` atribut tog elementa.

Čitanje vrednosti promenljivih

Za čitanje vrednosti promenljivih u programu, debager pretražuje DWARF stablo tražeći promenljivu sa zadatim imenom. U slučaju lokalnih promenljivih, traži se `DW_TAG_variable` element čiji `DW_AT_name` odgovara navedenoj promenljivoj. Kada se ista pronađe konsultuje se `DW_AT_location`, koji ukazuje na lokaciju gde se vrednost promenljive nalazi. Ukoliko ovaj atribut nije naveden debager će vrednost takve promenljive smatrati kao optimizovanu prijavljujući informaciju o tome [4].

Glava 3

GNU GDB alat

Glava 4

Datoteke jezgra

Glava 5

TLS

5.1 Motivacija

5.2 Rukovanje TLS-om tokom izvršavanja programa

5.3 Arhitekturalno specifične zavisnosti

5.4 TLS Modeli pristupa

Glava 6

Implementacija rešenja

Glava 7

Zaključak

Literatura

[1] *ELF Format*. on-line at: http://www.cs.cmu.edu/afs/cs/academic/class/15213-s00/doc/elf.pdf. 1992.

[2] Free Software Foundation. *DWARF Format*. on-line at: http://dwarfstd.org/. 1992.

[3] Linux Foundation. *ptrace*. on-line at: http://man7.org/linux/man-pages/man2/ptrace.2.html.

[4] *GNU GDB*. on-line at: https://www.gnu.org/software/gdb/.

[5] Reid Kleckner. *CodeView*. on-line at: https://llvm.org/devmtg/2016-11/Slides/Kleckner-CodeViewInLLVM.pdf.

Biografija autora