

UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Đorđe Todorović

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

master rad

Beograd, 2019.

Mentor:

dr Milena VUJOŠEVIĆ-JANIČIĆ, redovan profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Filip MARIĆ, redovan profesor
Univerzitet u Beogradu, Matematički fakultet

dr Miroslav MARIĆ, redovan profesor
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Daretu

Sadržaj

1	Uvod	1
2	Kako rade debageri?	3
2.1	Prevođenje programa	4
2.2	Fajl format DWARF	4
2.3	Fajl format ELF	8
2.4	Sistemska poziv <code>ptrace</code>	9
2.5	Realizacija osnovnih elemenata upotrebe debagera	10
3	Alat GNU GDB	14
3.1	Istorija alata	14
3.2	Šta je <i>arhitektura</i> za GNU GDB?	15
3.3	Datoteke jezgra	18
3.4	<i>Multiarch</i> GNU GDB	19
4	TLS	23
4.1	Motivacija	23
4.2	Definisanje novih podataka u ELF-u	24
4.3	Rukovanje TLS-om tokom izvršavanja programa	26
4.4	Pokretanje i izvršavanje procesa	28
4.5	TLS Modeli pristupa	31
5	Implementacija rešenja	33
5.1	Detalji implementacije	34
5.2	Alternativno rešenje	37
5.3	Unapređenje GNU GDB alata prilikom čitanja/pisanja datoteke jezgra za MIPS procesorsku arhitekturu	38
5.4	Testiranje	39

<i>SADRŽAJ</i>	v
5.5 Upotreba alata	40
6 Zaključak	44
Literatura	45

Glava 1

Uvod

Arhitektura ugrađenih uređaja se najčešće ne poklapa sa arhitekturom ličnih računara. Poznato je da se programi koji su prevedeni za jednu arhitekturu računara ne mogu izvršavati na računarima koji poseduju drugačiju arhitekturu. Greške u programima koji se izvršavaju na ugrađenim uređajima najčešće otklanjamo koristeći alate na ličnim računarima. GNU GDB alat, između ostalog, omogućava analiziranje i otklanjanje grešaka u programima koji se izvršavaju na drugim arhitekturama. Načini kroz koje se to može ostvariti jesu korišćenje GDB servera, uz korišćenje posebnih protokola, ili korišćenje datoteka jezgara. Datoteka jezgra može biti kreirana iz korisničkog nivoa, npr. baš uz pomoć GDB alata, ili prilikom neregularnog prekida izvršavanja programa samo jezgro operativnog sistema može krerati istu. Ona sadrži stanje radne memorije procesa prilikom rušenja ili prekidanja na neki drugi nestandardni način. Preciznije, one sadrže informacije o vrednostima promenljivih, vrednosti procesorskih registara, programske brojače, informacije o samom procesu, informacije o nitima itd. Tako kreiranu datoteku jezgra na gostujućoj arhitekturi učitavamo u GNU GDB alat na računaru sa arhitekturom domaćina i otklanjanje grešaka i analiza mogu da počnu. Ovaj rad, pored implementacije poboljšanja alata, opisuje i postupak korišćenja GNU GDB alata prilikom dobijanja vrednosti TLS promenljivih, koje predstavljaju promenljive lokalne za niti. Svaka nit ima svoju kopiju TLS promenljive i prilikom analiziranja programa značajno je pročitati vrednosti iz svih niti. Ključna reč, programskih jezika C i C++, koja se dodaje ispred definicije ili deklaracije promenljive je `__thread`. Jeadan od slučaja upotrebe takve promenljive je predstavljanje jedinstvenog broja koji označava grešku koju je prijavio program, pa ima smisla da je u različitim nitima ova promenljiva imala drugačiju vrednost. Različite arhitekture mogu imati različitu implementaciju

TLS-a, te je njihovo čitanje iz alata GNU GDB dodatno otežano. Glavni doprinos rada predstavlja omogućavanje dohvatanja vrednosti TLS promenljive za sve procesorske arhitekture podržane u GNU GDB alatu na arhitekturi domaćina, čime se proširuju dostupne informacije za analizu programa drugih arhitektura. Rad se pored prvog uvodnog, sastoji još iz pet poglavlja. Drugo poglavlje opisuje motivacionu ideju. Treće poglavlje pruža informacije o zahtevanim karakteristikama samog programa koji se analizira. Četvrto poglavlje prikazuje detalje implementacije. U petom poglavlju je opisan precizan postupak prevođenja GNU GDB alata, dok je u šestom poglavlju predstavljen zaključak.

Glava 2

Kako rade debageri?

Greške su sastavni deo svakog rada koji obavlja čovek, te ih i programeri prave. 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 softvera. Nakon faze testiranja obično sledi faza analize i otklanjanja grešaka.

Debager (eng. *debugger*) je softverski alat koji koriste programeri za testiranje, analizu i otklanjanje grešaka u programima. Sam proces korišćenja takvih alata nazivamo debugovanjem (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 omogućava da izvršava proces instrukciju po instrukciju, do postavlja tačke prekida (eng. *breakpoints*) 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 debuguje, uz ograničenje da program pripada istoj procesorskoj arhitekturi kao i domaćinski sistem na kojem se debager izvršava. Čak i struktura programa može biti promenjena, prateći proratne efekte.

Podršku debagerima, u opštem slučaju, daju operativni sistemi, kroz sistemске pozive koji omogućavaju tim alatima da pokrenu i preuzmu kontrolu nad nekim drugim procesom. Za neke naprednije tehnike debugovanja 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 debug informacijama namenjene za taj operativni sistem. Alati za debugovanje u okviru operativnog sistema Windows koriste standard *Microsoft CodeView*. Više informacija o ovom standardu može se pronaći u literaturi [6].

2.1 Prevođenje programa

Programi koji se debuguju se prevode uz pomoć odgovarajuće opcije programskih prevodioca (za prevodioce GCC i LLVM/Clang, to je opcija `-g`) koja obezbeđuje generisanje pomoćnih debug informacija. Ukoliko je program koji se analizira preveden bez optimizacija, debug 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 debug informacije. Neke promenljive i funkcije programa neće biti predstavljene debug informacijama. Npr. promenljiva programa može biti živa samo u nekim određenim delovima programa, pa programski prevodioci generišu debug informacije o njenim lokacijama samo u tim određenim delovima koda. Prilikom optimizacija na nivou mašinskog koda život promenljive može biti skraćen, pa čitanje vrednosti promenljive iz debagera u nekim situacijama neće biti moguće, iako gledajući izvorni kod očekujemo da je ona živa u tom trenutku.

2.2 Fajl format DWARF

DWARF je debug 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 debugovanje 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. Arhitekturno je nezavisan i predstavlja „most” između izvornog koda i izvršnog fajla. Trenutno je poslednji realizovani standard verzija 5 formata DWARF.

Debug fajl format DWARF se odnosi na Unix-olike operativne sisteme, kao što su Linux i MacOS. Generisane debug 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 izvor-

nog koda, lokacijama promenljivih i ključna debug 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 debug jedinicom DIE (eng. Debug Info Entry). Osnovna debug 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 debug jedinica je opisana određenim DWARF atributima sa prefiksom `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 slikom 2.1 je dat slikom 2.2.

```
#include <stdio.h>

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

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

    return 0;
}
```

Slika 2.1: Izvorni kôd test primera koji se analizira.

Funkcija `main` primera sa slike 2.1 na slici 2.2 je predstavljena DWARF tagom `DW_TAG_subprogram`. Atribut te debug 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. Sledeći čvor drveta predstavlja promenljivu `x` istog test primera. Ta debug jedinica je dete čvora koji predstavlja funkciju `main` i ukazuje da se promenljiva `x` nalazi unutar funkcije `main`. Promenljiva `x` je predstavljena DWARF tagom `DW_TAG_variable`. Atribut `DW_AT_name` predstavlja ime promenljive, `DW_AT_type` referiše na debug jedinicu

```

<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.

koja predstavlja tip promenljive, dok `DW_AT_location` atribut predstavlja lokaciju promenljive u memoriji programa.

Debag promenljive

Svaka promenljiva programa prevedenog sa debug 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.

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

```
<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: Debug jedinica promenljive `x` u optimizovanom kodu.

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 Fajl format ELF

ELF (eng. *Executable and Linkable Format*) [1] je format izvršnih fajlova, deljenih biblioteka, objektnih fajlova i datoteka jezgara.

ELF sadrži razne informacije o samom fajlu. Podeljen je 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*), sekcijску tabelu zaglavlja (eng. *Section header table*) i ulazne tačke prethodne dve tabele. Slika 2.5 prikazuje primer prikaza fajla u formatu ELF.

```

ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
  Class:                               ELF64
  Data:                               2's complement, little endian
  Version:                             1 (current)
  OS/ABI:                             UNIX - System V
  ABI Version:                         0
  Type:                                EXEC (Executable file)
  Machine:                             Advanced Micro Devices X86-64
  Version:                             0x1
  Entry point address:                 0x4005c0
  Start of program headers:            64 (bytes into file)
  Start of section headers:           7944 (bytes into file)
  Flags:                               0x0
  Size of this header:                 64 (bytes)
  Size of program headers:            56 (bytes)
  Number of program headers:           10
  Size of section headers:            64 (bytes)
  Number of section headers:          39
  Section header string table index: 36

Section Headers:
[Nr] Name                               Type           Address         Offset
    Size                               EntSize
[ 0]                                     NULL           0000000000000000 00000000
    0000000000000000 0000000000000000 0 0 0
[ 1] .interp                             PROGBITS       0000000000400270 00000270
    000000000000001c 0000000000000000 A 0 0 1
[ 2] .note.ABI-tag                       NOTE           000000000040028c 0000028c
    0000000000000020 0000000000000000 A 0 0 4
[ 3] .note.gnu.build-id                 NOTE           00000000004002ac 000002ac
    0000000000000024 0000000000000000 A 0 0 4
[ 4] .gnu.hash                          GNU_HASH       00000000004002d0 000002d0
    000000000000001c 0000000000000000 A 5 0 8
[ 5] .dynsym                            DYNSYM        00000000004002f0 000002f0
    00000000000000d8 0000000000000018 A 6 1 8

```

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

2.4 Sistemski poziv ptrace

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:

```
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 taj argument 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);
```

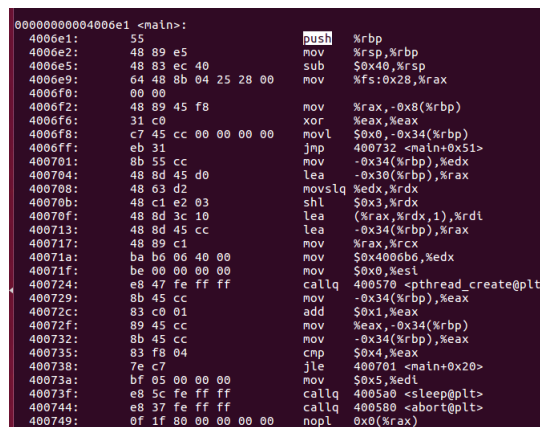
2.5 Realizacija osnovnih elemenata upotrebe debagera

Tačke prekida

Postoje dve vrste tačaka prekida (eng. *breakpoints*): softverske i hardverske [5].

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 kôd `main` funkcije primera izgleda kao u primeru sa slike 2.6.



```

000000004006e1 <main>:
4006e1: 55                push    %rbp
4006e2: 48 89 e5          mov     %rsp,%rbp
4006e5: 48 83 ec 40       sub     $0x40,%rsp
4006e9: 64 48 8b 04 25 28 00 mov     %fs:0x28,%rax
4006f0: 00 00
4006f2: 48 89 45 f8       mov     %rax,-0x8(%rbp)
4006f6: 31 c9            xor     %eax,%eax
4006f8: c7 45 cc 00 00 00 00 movl    $0x0,-0x34(%rbp)
4006ff: eb 31            jmp     400732 <main+0x51>
400701: 8b 55 cc          mov     -0x34(%rbp),%edx
400704: 48 8d 45 d0       lea     -0x30(%rbp),%rax
400708: 48 03 d2          movslq  %edx,%rdx
40070b: 48 c1 e2 03       shl     $0x3,%rdx
40070f: 48 8d 3c 10       lea     (%rax,%rdx,1),%rdi
400713: 48 8d 45 cc       lea     -0x34(%rbp),%rax
400717: 48 89 c1          mov     %rax,%rcx
40071a: ba b6 06 40 00    mov     $0x4006b6,%edx
40071f: be 00 00 00 00    mov     $0x0,%esi
400724: e8 47 fe ff ff    callq   400570 <pthread_create@plt>
400729: 8b 45 cc          mov     -0x34(%rbp),%eax
40072c: 83 c9 01          add     $0x1,%eax
40072f: 89 45 cc          mov     %eax,-0x34(%rbp)
400732: 8b 45 cc          mov     -0x34(%rbp),%eax
400735: 83 f8 04          cmp     $0x4,%eax
400738: 7e c7            jle     400701 <main+0x20>
40073a: bf 05 00 00 00    mov     $0x5,%edi
40073f: e8 5c fe ff ff    callq   4005a0 <sleep@plt>
400744: e8 37 fe ff ff    callq   400580 <abort@plt>
400749: 0f 1f 80 00 00 00 00 nopl    0x0(%rax)

```

Slika 2.6: Asemblerski kôd `main` funkcije test primera na x86-64 platformi.

Primeru 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
```

Ukoliko korisnik želi da nastavi dalje, instrukcija prekida se zamenjuje sa originalnom instrukcijom koja se izvršava i nastavlja se sa radom programa.

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 zaustavlja na toj tački. Debager je već upoznat od strane korisnika da je tačka prekida postavljena te on čeka na signal koji ukazuje na to da je program dostigao do instrukcije prekida. Operativni sistem prepoznaje instrukciju `int3`, poziva se specijalni obrađivač tog signala (na 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 debug simbole generišemo uz pomoć `-g` opcije kompajlera.

```
<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.

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

Koraćanje

Pod procesom koraćanja (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 debuguje [5].

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

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

Operativni sistem će poslati debageru signal `SIGTRAP` kada je korak izvršen.

Pored instrukcijskog koraćanja pomenućemo još jednu vrstu koraćanja u neku funkciju koja je pozvana `call` ili `jump` instrukcijom. Komanda debagara GNU GDB 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 koraćanje i za njih se koraćanje implementira na drugačiji način, uz pomoć emulacije instrukcija, ali u ovom radu neće biti reči o tome.

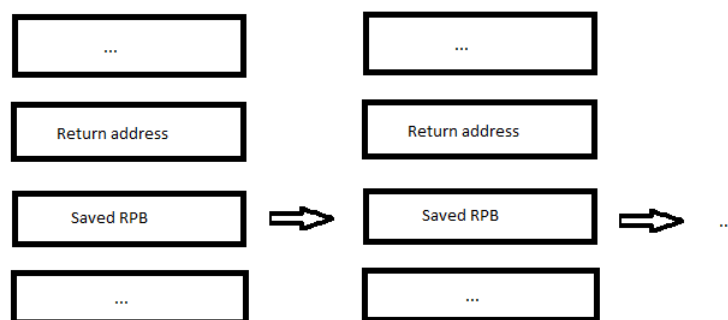
Izlistavanje pozivanih funkcija

Objasnimo komandu izlistavanje pozivanih funkcija (eng. *backtrace*) posmatrajući organizaciju stek okvira (eng. *stack frames*) na platformi Intel x86-64 [5].

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 sve okvire 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 sa debug 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



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

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 [5].

Glava 3

Alat GNU GDB

GNU GDB je alat koji omogućava uvid u događanja unutar drugog programa koji se nalazi u fazi izvršavanja, ili u slučaju neregularnog prekida izvršavanja programa uvid u to šta se dešavalo pa je do toga došlo. GDB debager takode omogućava uvid u to šta se dešavalo sa programima i na platformama koje imaju različitu arhitekturu od domaćinske arhitekture (eng. *host architecture*). Da bi se to realizovalo koriste se GDB server, što se naziva udaljeno debugovanje, ili *Multiarch* GNU GDB koji koristi biblioteke namenjene ciljanim arhitekturama. Programi koji mogu biti analizirani mogu biti napisani u raznim programskim jezicima, kao što su Ada, C, C++, Objective-C, Pascal. GNU GDB alat se može pokrenuti na najpopularnijim operativnim sistemima UNIX i Microsoft Windows varijanti. U radu se podrazumeva korišćenje UNIX-olikog operativnog sistema.

3.1 Istorija alata

Izvorni kod alata GDB je originalno napisan od strane Ričarda Stolmana 1986. godine kao deo GNU sistema [8]. GDB je slobodan i besplatan softver realizovan pod *GNU General Public License (GPL)*. Od 1990. do 1993. godine alat je održavao Džon Gilmor, a trenutno za održavanje alata je zadužena *GDB Steering Committee* grupa, odobrena od strane FSF (eng. *Free Software Foundation*) [4].

Poslednja realizovana verzija alata GNU GDB je 8.2.1.

3.2 Šta je *arhitektura* za GNU GDB?

Za alat GNU GDB arhitektura je veoma labav koncept. Može se posmatrati kao bilo koje svojstvo programa koji se debuguje, ali obično se misli na procesorsku arhitekturu. Za debugger su bitna dva svojstva procesorske arhitekture:

- Skup instrukcija (eng. *Instruction Set Architecture - ISA*) predstavlja specifičnu kombinaciju registara i mašinskih instrukcija.
- ABI (eng. *Application Binary Interface*) predstavlja spisak pravila koja propisuju pravilan način korišćenja skupa instrukcija.

Domaćinski GNU GDB

Domaćinski GNU GDB alat je preveden za istu procesorsku arhitekturu kao i računar na kome se alat izvršava. Korišćenje domaćinskog alata GNU GDB nad nekim programom ima ograničenje. Program koji se debuguje mora biti iste procesorske arhitekture kao i arhitektura domaćina.

Prevođenje domaćinskog GNU GDB alata

Preuzimanje izvornog koda alata se radi sledećim komandama:

```
mkdir gdb
cd gdb
git pull http://gnu.org/gnu/gdb.git
```

Prva komanda pravi direktorijum u kome se izgrađuje (prevodi) alat. Druga komanda vrši pozicioniranje trenutne putanje u taj direktorijum. Trećom komandom se preuzima izvorni kod alata.

Prvi korak prevođenja je konfiguracija direktorijuma u kome se prevođenje izvršava. Ovim komandama kreiramo Makefile:

```
mkdir build
cd build
../configure
```

Prva komanda pravi direktorijum u kome se izgrađuje (prevodi) alat. Druga komanda vrši pozicioniranje trenutne putanje u taj direktorijum. Trećom komandom se kreira Makefile. *configure* skripte na UNIX-olikim operativnim sistemima kao

ulaz parsiraju *Makefile.in* skriptu podešavajući okruženje (putanje do deljenih biblioteka, programski prevodilac, itd.) prilikom čega je krajnji izlaz fajl *Makefile* kojim se izgrađuje (prevodi) softver.

Nakon konfiguracije direktorijuma prevođenje alata se vrši komandom:

```
|| make
```

Pokretanje domaćinskog GNU GDB alata

GNU GDB očekuje kao argument komandne linije program koji je preveden za istu procesorsku arhitekturu. Ukoliko je trenutna putanja pozicionirana u direktorijum gde je izgrađen alat, pokretanje alata nad programom pod imenom *test* se vrši sledećom komandom:

```
|| ./gdb/gdb test
```

Korišćenje domaćinskog GNU GDB alata

Pokažimo neke od osnovnih komandi alata GNU GDB.

Postavljanje tačke prekida

Postavljanje tačke prekida na funkciju programa koji se debuguje se vrši komandom **break**. Program koji se debuguje se zaustavlja kada dostigne do određene funkcije. Primer korišćenja postavljanja tačke prekida na funkciju (u ovom slučaju *fn1*):

```
|| (gdb) break fn1
Breakpoint 1 at 0x4005fe: file test.c, line 4.
(gdb) r
Starting program: /home/rtrk/fax/master/master_examples/
x86_arch/test
7
Breakpoint 1, fn1 (arg=0x7fffffffdbf4) at test.c:4
4          (*arg)++;
```

Izvršavanje programa sekvencu po sekvencu

Izvršavanje programa sekvencu po sekvencu se radi pomoću tehnike koračanja. Komanda u okviru alata GNU GDB koja nam omogućava izvršavanje instrukciju po instrukciju je **stepi**. Primer korišćenja **stepi** komande, uz

pomoć korišćenja `disassemble` komande koja nam prikazuje asemblerski kod programa koji se debuguje:

```
(gdb) disassemble
Dump of assembler code for function fn1:
0x0000000004005f6 <+0>:      push    %rbp
0x0000000004005f7 <+1>:      mov     %rsp,%rbp
0x0000000004005fa <+4>:      mov     %rdi,-0x8(%rbp)
=> 0x0000000004005fe <+8>:      mov     -0x8(%rbp),%rax
0x000000000400602 <+12>:     mov     (%rax),%eax
0x000000000400604 <+14>:     lea     0x1(%rax),%edx
0x000000000400607 <+17>:     mov     -0x8(%rbp),%rax
0x00000000040060b <+21>:     mov     %edx,(%rax)
0x00000000040060d <+23>:     mov     -0x8(%rbp),%rax
0x000000000400611 <+27>:     mov     (%rax),%eax
0x000000000400613 <+29>:     cmp     $0x5,%eax
0x000000000400616 <+32>:     jle     0x400627 <fn1+49>
0x000000000400618 <+34>:     mov     -0x8(%rbp),%rax
0x000000000400628 <+38>:     pop     %rbp
0x000000000400629 <+42>:     retq
End of assembler dump.
(gdb) stepi
0x000000000400602      4      (*arg)++;
(gdb) disassemble
Dump of assembler code for function fn1:
0x0000000004005f6 <+0>:      push    %rbp
0x0000000004005f7 <+1>:      mov     %rsp,%rbp
0x0000000004005fa <+4>:      mov     %rdi,-0x8(%rbp)
0x0000000004005fe <+8>:      mov     -0x8(%rbp),%rax
=> 0x000000000400602 <+12>:     mov     (%rax),%eax
0x000000000400604 <+14>:     lea     0x1(%rax),%edx
0x000000000400607 <+17>:     mov     -0x8(%rbp),%rax
0x00000000040060b <+21>:     mov     %edx,(%rax)
0x00000000040060d <+23>:     mov     -0x8(%rbp),%rax
0x000000000400611 <+27>:     mov     (%rax),%eax
0x000000000400613 <+29>:     cmp     $0x5,%eax
0x000000000400616 <+32>:     jle     0x400627 <fn1+49>
0x000000000400618 <+34>:     mov     -0x8(%rbp),%rax
0x000000000400628 <+38>:     pop     %rbp
0x000000000400629 <+42>:     retq
End of assembler dump.
```

Izvršavanje sledeće linije programa se radi korišćenjem komande `next`. Primer

korišćenja `next` komande:

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/rtrk/fax/master/master_examples/
    x86_arch/test
4
Breakpoint 1, fn1 (arg=0x7fffffffdbf4) at test.c:4
4          (*arg)++;
(gdb) next
5          if ((*arg) > 5)
```

3.3 Datoteke jezgra

Datoteka jezgra (eng. *core dump file*) je snimak (eng. *snapshot*) memorije programa, registara i ostalih sistemskih informacija u trenutku neočekivanog prekida rada programa. Veoma važnu ulogu ima u procesu debugovanja programa sa ugrađenih uređaja koji često pripadaju različitoj procesorskoj arhitekturi u odnosu na lični računar. Ugrađeni uređaji obično imaju ograničene resurse, pa često na takvim platformama ne postoji debager. Najčešća procedura debugovanja ovakvih programa jeste prebacivanje datoteke jezgra i programa na lični računar na kome se analiza problema odvija koristeći debager.

Struktura datoteke jezgra

Datoteke jezgra sadrže razne informacije iz memorije programa uključujući i vrednosti lokalnih promenljivih, globalnih promenljivih, podatke lokalne za niti itd. Takođe sadrži vrednosti registara u trenutku prekida programa. U to spadaju i programski brojač i stek pokazivač.

Sadržaj datoteke jezgra je organizovan sekvencijalno sledećim redosledom:

Zaglavlje. Sadrži osnovne informacije o datoteci jezgra i pomeraje (eng. *offset*) kojima se lociraju ostale informacije iz nje.

ldinfo strukture. Definiše informacije relevantne za dinamički loader.

mstsave strukture. Definiše informacije relevantne za sistemske niti.

Korisnički stek. Sadrži kopiju korisničkog steka u trenutku pucanja programa.

Segment podataka. Sadrži kopiju segmenta podataka u trenutku pucanja programa.

Memorijski mapirani regioni i vm_info strukture. Sadrži informacije o pomerajima i dužinama mapiranih regiona.

Generisanje datoteke jezgra

Datoteke jezgra se generišu ukoliko dođe do nepredviđenog prekida programa. To je podrazumevana akcija prilikom pojave signala operativnog sistema koji ukazuju na prekid rada programa. Jezgra UNIX-olikih operativnih sistema podrazumevano postavljaju dužinu datoteka jezgara na 0. To je razlog zašto na našim sistemima nemamo datoteku jezgra nakon npr. prekidanja programa uz poruku *Segmentation fault*. Da bi se datoteke jezgra generisale, potrebno je eksplicitno promeniti dužinu datoteka jezgara koristeći komandu:

```
|| ulimit -c unlimited
```

Datoteka jezgra se može generisati i iz korisničkog nivoa koristeći alat GNU GDB komandom `gcore`.

3.4 *Multiarch* GNU GDB

Multiarch GNU GDB je verzija alata koja može da debuguje programe sa platformi različitih arhitektura. Alat na korisničkom nivou emulira instrukcije i registre ciljanih platformi. Potrebne su mu i deljene biblioteke za tu ciljanu platformu koje koristi program koji se debuguje. Skup komandi alata je limitiran u odnosu na domaćinski GDB.

Prevođenje *Multiarch* GNU GDB

Prvi korak je pozicioniranje u direktorijum sa izvornim kodom alata:

```
|| cd gdb
```

Sledeći korak je konfiguracija direktorijuma u kome se prevođenje izvršava. Ovim komandama kreiramo `Makefile` kojim odobravamo debugovanje svih podržanih arhitektura u alatu GDB (kao npr. MIPS32, MIPS64, ARM, AARCH64, x86_64, i386, SPARC):


```
mkdir build_multi
cd build_multi
../configure --enable-targets=all
```

Prva komanda pravi direktorijum u kome se izgrađuje (prevodi) alat. Druga komanda vrši pozicioniranje trenutne putanje u taj direktorijum. Trećom komandom se kreira *Makefile*. Pošto je prilikom konfiguracije navedena opcija *-enable-targets=all* time je omogućeno prevodenje alata za sve podržane arhitekture. Takođe je moguće konfigurisati prevodenje za samo određene arhitekture navodeći eksplicitno opciju kojom se navode ciljane procesorske arhitekture, npr. *-enable-targets=mips-linux-gnu* za arhitekturu MIPS ili *-enable-targets=arm-linux-gnu* za arhitekturu ARM. Osnovna razlika između *Makefile*-ova domaćinske i *Multiarch* verzije debagera je ta što je za domaćinski alat potrebno okruženje (deljene biblioteke, programski prevodilac, itd.) samo za domaćinsku arhitekturu. Za *Multiarch* verziju alata potrebno je pronaći putanje do programskog prevodioca i deljenih biblioteka za ciljane platforme navedene opcijom *-enable-targets=*. Ukoliko konfiguracione skripte ne pronađu ciljano okruženje za neku od navedenih arhitektura, ta će biti ignorisana.

Pokretanje *Multiarch* GNU GDB

Ukoliko je trenutna putanja pozicionirana u direktorijum gde je izgrađen alat, pokretanje alata *Multiarch* GDB nad programom pod imenom *test* se vrši na isti način kao i domaćinska verzija alata komandom:

```
./gdb/gdb test
```

Korišćenje *Multiarch* GNU GDB

Spisak komandi koje se mogu koristiti korišćenjem *Multiarch* verzije alata GNU GDB je limitiran. To se odnosi na izvršavanje programa koji se debuguje, jer program pripada drugačijem adresnom prostoru. Neke od komandi koje mogu biti upotrebljene su izlistavanje vrednosti registara programa, izlistavanje instrukcija, analiza stek okvira pozivanih funkcija, itd. Najčešće se ova verzija alata koristi tako što se učitava datoteka jezgra koja je generisana na ciljanoj platformi kada je program koji se debuguje neočekivano prekinuo sa radom. To obično prati analiza uzroka greške.

Primer učitavanja datoteke jezgra u GNU GDB alat

Na primer ispod je prikazano učitavanje datoteke jezgra programa čije izvršavanje je prekinuto od strane jezgra operativnog sistema. Komandom alata `core-file` se učitava datoteka jezgra u debager.

Da bi se izazvalo generisanje datoteke jezgra u izvornom kodu programa napisanog u C ili C++ programskom jeziku može se koristiti `abort()` funkcija iz standardne C biblioteke. U primeru ispod je pozvana ta funkcija koja je izazvala prekid programa uz signal `SIGABRT`. Tom prilikom jezgro operativnog sistema je napravilo datoteku jezgra sa imenom `core`.

```
(gdb) core-file core
[New LWP 20586]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/
libthread_db.so.1".
Core was generated by './test.core'.
Program terminated with signal SIGABRT, Aborted.
#0  0x00007fb229164428 in raise (sig=6) at raise.c:54
```

Na primeru ispod je prikazana upotreba komande alata GNU GDB `bt`. Ona se koristi za izlistavanje pozvanih funkcija programa. Stek okviri programa koji su prikazani na primeru potvrđuju da je u funkciji `main()` došlo do poziva `abort()` funkcije. Što je izazvalo prekid rada programa.

```
(gdb) bt
#0  0x00007fb229164428 in raise (sig=6) at raise.c:54
#1  0x00007fb22916602a in abort () at abort.c:89
#2  0x0000000000400605 in main () at tls.c:18
```

Analiza datoteke jezgra

Na primeru ispod je prikazan primer korišćenja alata *Multiarch* GNU GDB. U primeru se vrši učitavanje datoteke jezgra generisane na uređaju sa ugrađenim računarom arhitekture MIPS. Uz datoteku jezgra učitavaju se izvršni fajl i deljene biblioteke koje izvršni fajl koristi na ciljanoj platformi.

```
(gdb) set solib-search-path ~/fax/master/master_examples/
mips_arch/
(gdb) core-file ~/fax/master/master_examples/mips_arch/core
warning: core file may not match specified executable file.
[New LWP 21808]
```

```
[New LWP 21813]
[New LWP 21810]
[New LWP 21809]
[New LWP 21811]
[New LWP 21812]
Core was generated by 'example222'.
Program terminated with signal SIGABRT, Aborted.
#0  0x00000000 in ?? ()
[Current thread is 1 (LWP 21808)]
```

Komanda `set solib-search-path dir` alata GNU GDB korišćena u primeru na u prethodnom primeru služi za navođenje direktorijuma iz kojeg debager treba da koristi deljene biblioteke za učitani program. Ta komanda je veoma bitna za debugovanje programa sa platformi drugih arhitektura, jer ukoliko putanja nije navedena debager koristi biblioteke na domaćinskoj mašini.

Na primeru ispod je prikazana upotreba komande `info registers`. *Multiarch* GNU GDB čita informaciju o arhitekturi programa koji se debuguje iz datoteke jezgra. Nakon toga čita vrednosti registara iz nje i ispisuje ih.

```
(gdb) info registers
zero      at      v0      v1      a0      a1      a2
          a3
R0  00000000 00005530 00000000 00005530 00000000 00005530
    00000000 00000006
t0      t1      t2      t3      t4      t5      t6
    t7
R8  00000000 00000000 00000000 7fcf4ca0 00000000 00000000
    00000000 771c03f8
s0      s1      s2      s3      s4      s5      s6
    s7
R16 00000000 7719a684 00000000 00000000 00000000 00000002
    ffffffff 8013d8a8
t8      t9      k0      k1      gp      sp      s8
    ra
R24 00000000 00000000 00000000 771c6490 00000000 77160634
    00000000 7fcf4c20
sr      lo      hi      bad      cause      pc
00000000 00000000 77165000 00000000 00b24608 00000000
fsr      fir
00000000 00000000
```

Glava 4

TLS

Pisanje višenitnih programa predstavlja fudamentalan i neizbežan koncept savremenog programiranja. Gotovo nijedan kompleksan korisnički program ne može biti napisan bez korišćenja niti. Preciznije, niti podižu performanse i brzinu programa, te se sve češće koriste u pisanju softvera. Promenljive lokalne za niti su takođe važan mehanizam koji pružaju programski jezici kao što su C i C++. Prirodno je da nekada imamo potrebu da definišemo promenljivu koja će imati različitu vrednost u svakoj niti. Jedan primer je promenljiva koja identifikuje grešku. Ona može biti izazvana u svakoj posbnoj niti iz različitog razloga, te promenljiva koja je opisuje ima različitu vrednost u svakoj niti.

4.1 Motivacija

Povećanje korišćenja niti u programiranju dovelo je do potrebe programera za boljim načinom rukovanja podacima lokalnih za niti. POSIX [7], interfejs za rukovanje nitima, definiše nove interfejse koji omogućavaju smeštanje `void *` objekata posebno za svaku nit. Taj interfejs je nezgrapan za korišćenje, pa se ispostavilo da postoji potreba za efikasnijim rešenjem. Ključ za taj objekat mora da bude alocirani dinamički u vremenu izvršavanja programa. Ako se ključ ne koristi više mora biti oslobođen. Celokupan proces zahteva dosta posla programera. Pored toga je podložan greškama. Iz tih razloga postao je ozbiljan problem kada se kombinuje sa dinamičko učitanim kodom.

Da bi se odgovorilo na sve opisane probleme, odlučeno je da se programski jezici prošire i tako prepuste težak posao programskim prevodiocima.

Za jezike C i C++ ključna reč `__thread` se koristi za deklaraciju i definiciju promenljivih lokalnih za niti. Neki primeri deklaracija promenljivih lokalnih za niti:

```
__thread int j;  
  
__thread struct state s;  
  
extern __thread char *p;
```

Prednost TLS-a nije ograničena samo na korisničke programe. Okruženje izvršavanja programa, između ostalih standardna biblioteka, takođe koristi pogodnosti ovog mehanizma. Npr. globalne promenljive `errno`, koje jednoznačno označavaju nastalu grešku, moraju biti lokalne za niti, jer u različitim nitima može doći do različite greške. Napomenimo da navođenjem `__thread` pri deklaraciji ili definiciji neke automatske promenljive nema smisla i to nije dozvoljeno, jer automatske promenljive su uvek lokalne za niti. Promenljive statičkih funkcija su takođe kandidati za korišćenje TLS promenljivih.

Promenljive lokalne za niti se ponašaju intuitivno. Npr. adresni operator vraća adresu promenljive za trenutnu nit. Memorija alocirana za promenljivu lokalnu za nit u dinamički učitanoj modulu se oslobađa kada se taj modul očisti iz memorije.

Implementacija ovog mehanizma zahteva promenu okruženja izvršavanja programa. Format izvršnih fajlova je proširen kako bi definisao promenljive lokalne za niti odvojeno od normalnih promenljivih. Dinamički punilac (eng. dynamic loader) mora biti nadograđen kako bi ispravno inicijalizovao te nove sekcije. Takođe, standardna biblioteka koja rukuje nitima je promenjena kako bi alocirala nove podatke lokalne za niti za svaku novu nit. U nastavku poglavlja će biti opisane izmene u fajl formatu ELF i detaljniji opis izvršavanja programa koji sadrže TLS promenljive.

4.2 Definisanje novih podataka u ELF-u

Izmene u fajl formatu izvršnih fajlova, potrebne za emitovanje TLS objekata su minimalni. Umesto smeštanja promenljivih u sekcije `.data` i `.bss` za inicijalizovane i neinicijalizovane promenljive redom, TLS promenljive se smeštaju u `.tdata` i `.tbss` sekcije [10]. Nove sekcije se od originalnih razlikuju u samo jednom dodatnom sekcijском flegu. Sekcijaska tabela koja opisuje nove sekcije je data slikom 4.1. Kao što se može primetiti jedina razlika u odnosu na normalne sekcije podataka jeste fleg `SHF_TLS`.

Field	.tbss	.tdata
sh_name	.tbss	.tdata
sh_type	SHT_NOBITS	SHT_PROGBITS
sh_flags	SHF_ALLOC + SHF_WRITE + SHF_TLS	SHF_ALLOC + SHF_WRITE + SHF_TLS
sh_addr	virtual address of section	virtual address of section
sh_offset	0	file offset of initialization image
sh_size	size of section	size of section
sh_link	SHN_UNDEF	SHN_UNDEF
sh_info	0	0
sh_addralign	alignment of section	alignment of section
sh_entsize	0	0

Slika 4.1: Tabela vrednosti ulaza koji opisuju nove TLS sekcije.

Imena novih sekcija, kao ni ostalih u fajl formatu ELF, nisu bitna. Linker će svaku sekciju tipa SHT_PROGBITS sa dodatnim flegom SHF_TLS tretirati kao .tdata, dok će sekcije tipa SHT_NOBITS sa dodatnim SHF_TLS tretirati kao .tbss sekciju. Odgovornost proizvođača ovakvih sekcija, obično programskih prevodioca, je da pravilno generiše sva polja prikazana na slici 4.1.

Za razliku od normalnih .data sekcija program koji se izvršava ne koristi .tdata sekciju direktno. Ta sekcija je moguće modifikovana u vreme pokretanja programa, od strane dinamičkog punilaca. Prilikom pokretanja programa, prva akcija jeste realokaciju koju izvršava dinamički punilac. Nakon toga podaci lokalni za niti se smeštaju u deo koji se naziva inicijalizaciona slika (*eng. initialization image*) i ona se ne modifikuje više nakon toga. Za svaku nit, uključujući i inicijalnu, nova memorija se alocira gde se kopira inicijalizaciona slika. Ovim se omogućava da svaka nit ima identičan početni sadržaj. Kako ne postoji samo jedna adresa koja ukazuje na simbol TLS promenljive, normalna tabela simbola ne može biti korišćena za njih. U izvršnom fajlu polje `st_value` ne sadrži apsolutnu adresu promenljive prilikom izvršavanja programa, jer apsolutna adresa nije poznata prilikom prevođenja programa. Iz tog razloga je uveden novi tip simbola (STT_TLS). Svaki simbol koji referiše na TLS ima takav tip simbola. Izvršni fajlovi u polju `st_value` imaju vrednost ofseta promenljive u TLS inicijalizacionoj slici. Nijedna realokacija ne sme pristupati simbolima tipa STT_TLS, osim onih koji su uvedene za rukovanje TLS-om. Takođe, te nove realokacije ne smeju koristiti simbole ostalih tipova.

Da bi dinamički linker mogao da izvrši inicijalizaciju inicijalizacione slike, njena

pozicija koja će biti prilikom izvršavanja programa mora biti zapisana negde. Originalno zaglavlje programa nije korisno, pa je novo zaglavlje definisano. To proširenje je prikazano na slici 4.2.

Field	Value
p_type	PT_TLS
p_offset	File offset of the TLS initialization image
p_vaddr	Virtual memory address of the TLS initialization image
p_paddr	Reserved
p_filesz	Size of the TLS initialization image
p_memsz	Total size of the TLS template
p_flags	PF_R
p_align	Alignment of the TLS template

Slika 4.2: Tabela koja predstavlja nove vrednosti programskog zaglavlja.

Svaka TLS promenljiva je identifikovana pomoću ofseta od početka TLS sekcije. U memoriji, `.tbss` sekcija je alocirana odmah nakon `.tdata` sekcije. Nijedna virtualna adresa ne može biti izračunata prilikom povezivanja (*eng. link time*).

4.3 Rukovanje TLS-om tokom izvršavanja programa

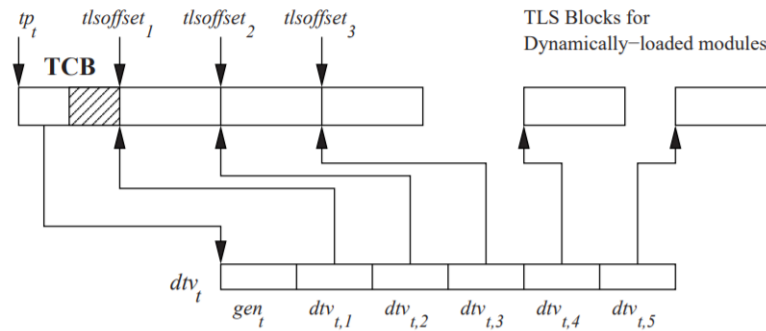
Kao što je napomenuto, obrada podataka lokalnih za niti nije prosta kao obrada normalnih podataka. Segment podataka ne može biti samo napravljen i dat procesu na korišćenje. Umesto toga, nekoliko kopija jednog istog podatka mora biti kreirano, svi inicijalizovani iz iste inicijalizacione slike.

Mehanizmi koji potpomažu izvršavanje programa bi trebalo da zaobiđu kreiranje podataka lokalnih za niti ako to nije neophodno. Npr. učitani modul može biti korišćen samo od strane jedne niti, od više kreiranih koje čine taj određeni proces. Bilo bi samo uzaludno gubljenje memorije i vremena za alociranje tih podataka za sve niti. Lenji metod je poželjan za ovakve situacije.

Nije samo alociranje memorije problem za korišćenje TLS-a. Pravila potrage za simbolima (*eng. symbol lookup*) u izvršnim fajlovima sa ELF formatom ne dozvoljavaju određivanje objekta koji sadrži korišćenu definiciju u vremenu povezivanja. I ako taj objekat nije poznat, ofset od te promenljive unutar prostora lokalnog za

nitni za taj objekat ne može biti određen takođe. Prema tome, normalan proces povezivanja ne može biti korišćen za ovakve situacije.

Promenljiva lokalna za nit je identifikovana sa referencom na objekat i ofsetom te promenljive unutar prostora lokalnog za nit. Da bi mapirali ove vrednosti u virtualne adrese, mehanizam izvršavanja programa zahteva strukturu podataka koje nisu postojale do tada. One moraju biti sposobne da mapiraju referencu objekta u neku adresu u određenom prostoru lokalnog za nit. Da bi se to omogućilo definisane su dve varijante struktura podataka. Različite procesorske arhitekture mogu odabrati jedan od ova dva pristupa, ali to mora biti propisano ABI-jem za tu arhitekturu. Jedan od razloga za korišćenje druge varijante modela je istorijski. Neke arhitekture su dizajnirale sadržaj nitne memorije na koju pokazuje nitni registar tako da nisu kompatibilne za korišćenje prve varijante TLS strukture.



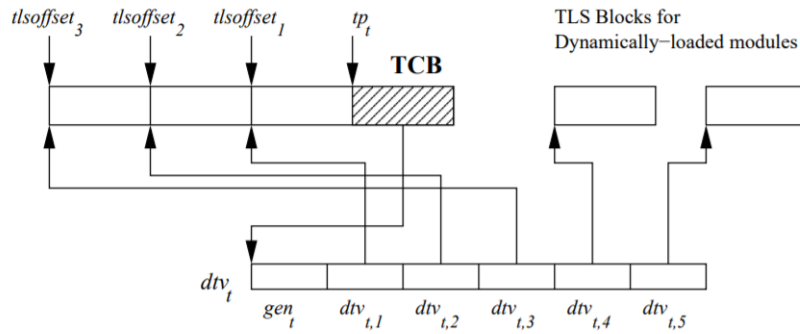
Slika 4.3: TLS struktura podataka. Varijanta 1.

Na slici 4.3 je prikazan primer prve varijante TLS strukture podataka. Nitni registar za nit t je označen sa tp_t . On pokazuje na nitni kontrolni blok TCB (*eng. Thread Control Block*), koji na ofsetu nula sadrži pokazivač na nitni dinamički vektor dtv_t za tu određenu nit.

Nitni dinamički vektor kao svoje prvo polje sadrži generacioni broj gen_t koji se koristi pri promeni veličine dtv_t i alokacije TLS blokova. Ostala polja sadrže pokazivače na TLS blokove za različite učitane module. TLS blokovi za module koji se učitavaju pri pokretanju programa su smeštena direktno nakon TCB-a i stoga ima arhitekturno specifičan, fiksni, ofset od adrese na nitni pokazivač. Za sve inicijalno dostupne module ofset svakog TLS bloka, s'tim i ofset TLS promenljive, u odnosu na TCB mora biti fiksni nakon pokretanja programa.

Druga varijanta TLS strukture podataka ima sličnu strukturu kao prva. Pri-

kazana je na slici 4.4. Jedina razlika je ta što nitni pokazivač pokazuje na nitni kontrolni blok za koji je nepoznata veličina i sadržaj. Negde svakako taj nitni kontrolni blok sadrži pokazivač na dinamički nitni vektor, ali nije navedeno gde. To je kontrolisano od strane mehanizma za izvršavanje programa. Napomenimo da je programskim prevodiocima zabranjeno da emituju kod koji direktno pristupa elementima $dtvt$ vektora.



Slika 4.4: TLS struktura podataka. Varijanta 2.

U trenutku pokretanja programa TCB, zajedno sa dinamičkim nitnim vektorom, se kreiraju za glavnu nit. Pozicija tog TLS bloka, za svaki pojedinačan modul, se računa koristeći arhitekturno specifične formule, zasnovane na veličini i poravnanju TLS bloka propisanih ABI-jem za tu specifičnu procesorsku arhitekturu.

4.4 Pokretanje i izvršavanje procesa

Za programe koji koriste TLS promenljive kod koji služi za pokretanje procesa mora podesiti memoriju za inicijalnu nit pre nego što preda kontrolu tog procesa drugim mehanizmima operativnog sistema. Podrška za TLS u statičko povezanim programima je limitirana. Neke procesorske arhitekture (kao npr. IA-64) ne definišu uopšte statičko linkvoanje (iako je podržano to je nestandardizovano). Neke druge platforme obeshrabruju korišćenje statičkog povezivanja pružanjem samo određenog proja funkcionalnosti. U oba slučaja je dinamičko učitavanje modula u statičko povezanim programima je ozbiljno limitirano ili potpuno nemoguće. Prema tome, rukovanje TLS je prosto, zbog postojanja samo jednog modula, tj. samo taj program.

Zanimljiviji je slučaj rukovanja TLS-a u dinamičko povezanom kodu. U ovom slučaju dinamički povezivač mora uključiti podršku za rukovanje takvih segmenata

podataka. U nastavku teksta je opisano učitavanje i pokretanje dinamičkog koda.

Da bi se podesila memorija za TLS, dinamički poveziivač čita sve potrebne informacije o svakom modulu, i o njegovim TLS blokovima, iz PT_TLS polja tabele TLS programskog zaglavlja prikazane na slici 4.2. Informacije o svim modulima moraju biti prikupljene. Ovaj proces se ostvaruje koristeći povezanu listu čiji element sadrži:

- pokazivač na TLS inicijalizacionu sliku
- veličinu TLS inicijalizacione slike
- TLS ofset (`tlsoffsetm`) za module `m`
- fleg koji daje informaciju o tome da li modul koristi statički TLS model

Ove informacije će biti proširene kada se učitaju dodatni dinamički moduli. Te informacije će biti korišćene od strane standardne biblioteke za rukovanje nitima kako bi podesila TLS blokove za novokreiranu nit.

Kao što je napomenuto, promenljiva unutar lokalnog prostora za nit, TLS, je identifikovana po referenci na modul i ofset u okviru tog TLS bloka. Ukoliko imamo strukturu podataka dinamički nitni vektor, možemo definisati referencu na neki modul kao ceo broj (*eng. integer*), počevši od broja 1. To može biti korišćeno kao indeks u `dtvt` nizu. Identifikacione brojeve koji svaki modul dobija određuje mehanizam izvršavanja programa, obično neki modul standardne biblioteke za rukovanje nitima. Samo izvršni fajl dobija fiksni broj, 1, a svi ostali učitani moduli dobijaju različite brojeve.

Računanje specifične adrese neke TLS promenljive je prosta operacija koja može biti izvršena ukoliko je programski prevodilac koji je preveo kod korigirao varijantu 1 TLS strukture podataka. Ali to ne može biti tako lako odrađeno u kodu koji je generisan od strane kompajlera za procesorske arhitekture koje koriste varijantnu 2 TLS strukturu podataka.

Umesto toga, definisana je funkcija `__tls_get_addr`, koja je implementirana na sledeći način:

```
void *  
__tls_get_addr (size_t m, size_t offset)  
{  
    char *tls_block = dtv[thread_id][m];
```

```

    return tls_block + offset;
}

```

To kako je vektor `dtv[thread_id]` smešten u memoriji je arhitekturno specifično. `m` is identifikacioni broj modula, koji mu je dodeljen od strane dinamičkog punionca kada je učitavan. Korišćenje `__tls_get_addr` funkcije ima i dodatne prednosti kako bi se olakšala implementacija dinamičkog modela, gde je alokacija nekog TLS bloka odložena do njegove prve upotrebe. Da bi se to podržalo potrebno je popuniti `dtv[thread_id]` vektor sa specijalnom vrednošću kojom će biti prepoznate situacije kada je taj vektor trenutno prazan, tj. da u datom trenutku određeni blok nije u upotrebi. Da bi se to podržalo, potrebna je mala promena u izvornom kodu `__tls_get_addr` funkcije:

```

void *
__tls_get_addr (size_t m, size_t offset)
{
    char *tls_block = dtv[thread_id][m];

    if (tls_block == UNALLOCATED_BLOCK)
        tls_block = dtv[thread_id][m] = allocate_tls(m);

    return tls_block + offset;
}

```

Funkcija `allocate_tls` mora da odredi memorijske zahteve za TLS modula `m` i inicijalizuje ga ispravno. Kao što je i napomenuto, postoje dva tipa podataka: inicijalizovani i neinicijalizovani. Inicijalizovani podaci moraju biti kopirani iz inicijalizacione nitne slike, podešenih prilikom učitavanja modula `m`. Neinicijalizovani podaci se postavljaju na vredosti 0. Jedan primer implementacije može biti:

```

void *
allocate_tls (size_t m)
{
    void *mem = malloc (tlssize[m]);
    memset (memcpy (mem, tlsinit_img[m], tlsinit_size[m]), '\0',
            tlssize[m] - tlsinit_size[m]);
    return mem;
}

```

`tlssize[m]`, `tlsinit_size[m]` i `tlsinit_img[m]` su poznati nakon učitavanja modula `m`. Primetimo da se ista inicijalizaciona slika `tlsinit_img[m]` koristi za sve

nitima modula, bilo kada da se one kreiraju. Novonapravljena nit ne nasleđuje podatke od svog oca (*eng. parent*), već dobija samo kopiju inicijalnih podataka.

4.5 TLS Modeli pristupa

Svako referisanje TLS promenljive prati jedan od dva modela pristupa: dinamički i statički. Različite arhitekture ABI-jem propisuju koji od modela pristupa će koristiti kao podrazumevani. Razni modeli pristupa se mogu sresti, ali tri najpoznatija pristupa su opisana u nastavku teksta.

Generalni dinamički TLS model

Generalni model pristupa TLS promenljivoj dozvoljava referisanje svih TLS promenljivih, bilo da je to iz deljene biblioteke ili dinamičkog izvršnog fajla. Ovaj model takođe podržava odloženo alociranje TLS bloka do trenutka kada se prvi put taj blok referiše iz specifične niti.

Lokalni dinamički TLS model

Lokalni dinamički model pristupa TLS promenljivoj predstavlja optimizaciju generalnog dinamičkog modela. Programski prevodilac može odrediti da je neka promenljiva definisana samo lokalno, ili zaštićeno (*eng. protected*) u objektu koji je napravljen u programu. U tom slučaju, programski prevodilac daje instrukcije linkeru da statički poveže dinamički TLS ofset i da korsiti ovaj model. Iz tog razloga predstavlja hibridnu kombinaciju statičkog i dinamičkog modela pristupa, te prema tome predstavlja model koji diže performanse u odnosu na generalni dinamički model. Samo jedan poziv `tls_get_addr()` po funkciji je potreban za određivanje adrese `dtv0,m`.

Statički TLS model sa dodeljenim ofsetima

Ovaj model pristupa dopušta referisanje samo na one TLS promenljive koje su dostupne kao deo inicijalnog statičkog TLS šablona (*eng. template*). Ovaj šablon je sastavljen od svih TLS blokova koji su sačinjeni prilikom pokretanja procesa. U ovom modelu, relativni ofset pokazivač na nit date promenljive `x` je smešten u polju GOT-a (*eng. Global offset table*) za promenljivu `x`.

Deljene biblioteke uobičajeno koriste dinamički model pristupa, jer statičkim modelom mogu referisati samo na određeni broj TLS promenljivih.

Statički TLS model

Ovaj model pristupa dopušta referisanje samo na one TLS promenljive koje su dostupne kao deo TLS bloka od tog dinamičkog izvršnog fajla. Linker računa relativni ofset pokazivača na nit statički, bez potrebe za dinamičkim realokacijama ili dodatnih informacija iz GOT-a. Ovaj model pristupa ne dopušta referisanje TLS promenljivih izvan tog dinamičko povezanog izvršnog fajla.

Glava 5

Implementacija rešenja

Kao što je napomenuto, TLS može biti različito implementiran za različite procesorske arhitekture. Može se smestati u posebne registre, u određene delove memorije itd. GNU GDB alat mora poznavati sve arhitekturne razlike i anulirati ih na neki način. Implementacija ovog proširenja alata se upravo i zasniva na prethodnoj pretpostavci, te upravo to i realizuje. Istražene su implementacije TLS-a raznih arhitektura, kako u funkcijama GNU C biblioteke (poznatije kao *glib*) [9], tako i karakteristike koje su ABI-jem propisane. Poboljšanje alata koje je uspešno realizovano se odnosi na verziju *Multiarch* GNU GDB-a. Ukoliko je program koji je učitao u *Multiarch* GNU GDB alat iste arhitekture kao arhitektura domaćina, zadržava se način dohvaćanja vrednosti TLS promenljive kao u slučaju GNU GDB alata arhitekture domaćina, jer je ta funkcionalnost već implementirana. Treba napomenuti da je u razvoju i alternativno rešenje od strane programera iz kompanije *Red Hat*[REF NA RedHat] u vidu projekta *Infinity*[REF NA INFINITY]. Taj projekat ima ideju da obuhvati rešenja za razne probleme o debugovanju niti, ne samo obradu TLS-a. Prednost rešenja koji se predstavlja u ovom master radu jeste svakako ta što je cela funkcionalnost prepuštena samom GNU GDB alatu, te nema potrebe za instalacijom ili prevođenjem eksternih projekata. Sa tim u vidu se svakako dobija na uštedi korisničkog napora pri korišćenju debagera. Izvorni kod alata i instrukcije za prevođenje i upotrebu debagera sa poboljšanjem za dohvaćanje vrednosti TLS promenljive se može pronaći na strani [REF NA MOJ GITHUB gdbtls].

5.1 Detalji implementacije

Izmena sistema izgradnje alata

Fajlovi *gdb/configure* i *gdb/Makefile.in* su zaduženi za izradnju alata. U najopštijem slučaju na UNIX-olikim operativnim sistemima, *configure* skripta je zadužena za pravljenje konačnog *Makefile* od ulaznog *gdb/Makefile.in*.

Izvorni kod koji u sebi sadrži funkcije za čitanje TLS-a (*gdb/linux-thread-db.c*) se prevodio samo za domaćinsku verziju GNU GDB alata, pa stoga u proširenju za *Multiarch* GNU GDB takođe treba uvrstiti pomenuti fajl prilikom prevođenja. Pored toga, postojeće funkcije koje su zadužene za dohvaćanje vrednosti TLS promenljivih je trebalo izmeniti na neki način te je u projekat dodat direktorijum *gdb/glibc-dep/* koji sadrži funkcije koje barataju sa TLS-om različite procesorske arhitekture od arhitekture domaćina. Da bi se ti fajlovi prevodili samo u slučaju *Multiarch* verzije definisan je makro pod imenom `CROSS_GDB` koji se definiše prilikom kreiranja *Makefile*. Da bi se ispratila konvencija pisanja skripti za izgradnju alata, u direktorijum *gdb/config/* je dodat fajl *glibc.mh* koji definiše spisak objektnih fajlova koji služe sa dohvaćanje vrednosti TLS promenljive u tom slučaju i takođe u tom fajlu se definiše gore pomenuti makro `CROSS_GDB`.

Sadržaj *gdb/config/glibc.mh* fajla izgleda na sledeći način:

```
# GLIBC fragment comes in here
GLIBCFILES = td_symbol_list.o \
              fetch-value.o gdb_td_ta_new.o \
              td_thr_tlsbase.o td_thr_tls_get_addr.o \
              td_ta_map_lwp2thr.o native_check.o
INTERNAL_CFLAGS += -DCROSS_GDB
```

Da bi sadržaj iz novokreiranog *gdb/config/glibc.mh* fajla bio ispisan u krajnji *Makefile* trebalo je uvesti promenljivu koja proverava da li je u prilikom navođenja opcija *gdb/configure* skripti navedena opcija `-enable_targets`, koja zapravo označava da korisnik ima nameru da alat prevede kao *Multiarch* verziju. Vrednost promenljive `cross_makefile_frag` u *gdb/configure* skripti postaje putanja do *gdb/config/glibc.mh* fajla ukoliko je pomenuta opcija navedena, a u suprotnom ona ostaje prazna. Deo koda u *gdb/configure* skripti kojim se to postiže je:

```
if test "${gdb_native}" = "no" || test "${enable_targets}" != ""
; then
    cross_makefile_frag=${srcdir}/config/glibc.mh
else
```

```

|| cross_makefile_frag=/dev/null
|| fi

```

Implementacija funkcija za dohvaćanje TLS-a

U direktorijumu *gdb/glibc-dep/* se nalazi srž funkcionalnosti dohvaćanja vrednosti TLS promenljive iz datoteke jezgara sa drugih platformi. Kako standardna C biblioteka već ima implementirane svaku arhitekturnu zavisnost u vezi TLS-a za svaku procesorsku arhitekturu podržanu u alatu GNU GDB, potrebno je na neki način izopštiti tu funkcionalnost iz same biblioteke unutar alata. Preciznije za baratanjem nitima, GDB koristi *libthread_db* biblioteku iz standardne biblioteke. Funkcije *td_ta_new()*, *td_thr_tls_get_addr()* i *td_th_tlsbase()*, koje se služe za dohvaćanje TLS promenljivih, očekuju da se na arhitekturi domaćina prirodno barata sa programima iste arhitekture. Izbegavanje modifikacije „*libthread_db*“ biblioteke moguće je izbeći tako što se pomenute funkcije implementiraju u GNU GDB-u, na isti način kao u „*glibc*“ biblioteci, s' tim što se sve arhitekturno zavisne vrednosti promenljivih i makroa koje primaju te funkcije postavljaju na vrednosti koje bi trebalo da imaju u slučaju da je ta ciljna arhitektura zapravo arhitektura domaćina.

Prvi korak ovog dela implementacije je izmeštanje funkcija iz standardne biblioteke unutar GDB projekta. Unutar *gdb/glibc-dep/* je kreiran direktorijum *nptl_db* koji sadrži pomenute funkcije. Zapravo, ukoliko alat koristi bilo koju verziju standardne biblioteke do 2.22 funkcije iz *nptl_db* nisu ni potrebne za dohvaćanje vrednosti TLS promenljive. Sve potrebne informacije za dohvaćanje vrednosti TLS promenljive se mogu pročitati iz same datoteke jezgra. Ukoliko alat koristi verziju standardne biblioteke 2.22 i ili veću, funkcije iz *nptl_db* direktorijuma su neophodne kako bi se anulirale novonastale izmene prilikom baratanja TLS-om. Ta funkcionalnost je dodata u funkciji *gdb_td_thr_tlsbase()* koja je dodata u fajl *gdb/glibc-dep/nptl_db/td_thr_tlsbase.c*. Sve funkcije koje bi trebalo da emuliraju ponašanje TLS funkcija standardne biblioteke koje su dodate u projekat GNU GDB imaju prefiks u imenu *gdb_*. Npr. GDB pandam funkciji *td_ta_new()* je *gdb_td_ta_new()*. Pomenuta funkcija je promenjena da bi alat GNU GDB imao informaciju o tačnoj verziji standardne biblioteke koju je program koji se debuguje koristio na platformi na kojoj se izvršavao, jer kao što je napomenuto od verzije 2.22 TLS se drugačije dohvata. Nije novo da za debugovanje višenitnih programa GDB alatom, čak i upotrebom domaćinske verzije, je neophodno da GDB koristi

istu verziju *libthread_db* biblioteke kao i program koji se debuguje. To znači ako je program preveden sa verzijom 2.x standardnom bibliotekom, alat mora korsiti istu verziju 2.x tokom svog debugovanja. To se postiže eksplicitnim navođenjem putanje do *libthread_db* biblioteke komandom `set libthread-db-search-path`. Ukoliko se verzije *libthread_db* biblioteke ne poklapaju debager će ispisati upozorenje.

Takođe u `gdb_tda_new()` funkciji se poziva funkcija `init_target_dep_constants()`, zadužena za inicijalizaciju arhitekturno zavisnih osobina o TLS-u. Deo kojim se različite osobine inicijalizuju za arhitekture MIPS i ARM u funkciji `init_target_dep_constants()` izgleda:

```
case bfd_arch_mips:
    tls_tcb_at_tp = 0;
    tls_dtv_at_tp = 1;
    forced_dynamic_tls_offset = -2;
    no_tls_offset = -1;
    tcb_alignment = 16;
    break;
case bfd_arch_arm:
    tls_tcb_at_tp = 0;
    tls_dtv_at_tp = 1;
    forced_dynamic_tls_offset = -2;
    no_tls_offset = -1;
    tcb_alignment = 0;
    break;
```

U ovom konkretnom slučaju sve TLS osobine su identične, osim poravnanja nitnog kontrolnog bloka.

Funkcija `native_check()` definisana u fajlu *gdb/glibc-dep/native-check.c* daje informaciju da li se *Multiarch* verzijom alata debuguje program domaćinske arhitekture. Ako je to slučaj, dohvatanje vrednosti TLS promenljive se vrši kao do sada uz pomoć funkcija iz standardne C biblioteke. Ukoliko to nije slučaj, pozovaju se novokreirane funkcije iz direktorijuma *gdb/glibc-dep/*.

Cela funkcionalnost se zapravo dodaje u *gdb/linux-thread-db.c* fajl u funkciju koja je zadužena za dohvatanje vrednosti TLS promenljive. Izmena nije direktno u funkciji `thread_db_get_thread_local_address` koja vraća adresu TLS promenljive. Naime, promena je izvedena u funkciji `try_thread_db_load_1()` koja uspostavlja konekciju između alata GDB i *libthread_db* biblioteke su promenjeni pokazivači na TLS funkcije u slučaju nedomaćinskih datoteka jezgara. Deo koda koji implementira pomenutu funkcionalnost je:

```

#ifdef CROSS_GDB
if (native_check(arch) != 0){
    info->td_thr_tls_get_addr_p=gdb_td_thr_tls_get_addr;
    info->td_thr_tlsbase_p=gdb_td_thr_tlsbase;
}else{
    //if it's host we want to keep old way of counting tls address
    TDB_DLSYM (info, td_thr_tls_get_addr);
    TDB_DLSYM (info, td_thr_tlsbase);
}
#else
    TDB_DLSYM (info, td_thr_tls_get_addr);
    TDB_DLSYM (info, td_thr_tlsbase);
#endif

```

Kao što je i napomenuto, iako je u pitanju *Multiarch* verzija alata, ako je datoteka jezgra kreirana na domaćinskoj platformi zadržava se normalno dohvaćanje TLS-a.

5.2 Alternativno rešenje

Drugi način implementacije je modifikacija *libthread_db* funkcija eksplicitno u standardnoj biblioteci, modifikujući ih tako da rade sa različitim arhitekturama. Naime, formula koja služi za dohvaćanje TLS promenljive se računala u vremenu prevođenja biblioteke, sada se računa u vremenu izvršavanja programa, i u zavisnosti od arhitekture promenljive koje učestvuju u pomenutoj formuli će dobiti odgovarajuće vrednosti. Ukoliko se radi o programu arhitekture domaćina u funkciji *td_ta_new()* promenljive dobijaju vrednosti koje odgovaraju arhitekturi domaćina, dok ako se radi o programu ciljne arhitekture, tj. različite od arhitekture domaćina, iz GNU GDB-a će biti pozvana nova funkcija u standardnoj biblioteci *td_ta_init_target_consts()* koja će postaviti vrednosti promenljivih koje odgovaraju ciljnoj arhitekturi. Ideja je upisati ove vrednosti u datoteku jezgra na ciljnoj arhitekturi i kasnije ih pročitati na arhitekturi domaćina korišćenjem GNU GDB alata. U ovom slučaju izmena u samom GNU GDB-u bi bila minorna, te zahteva samo proveru da li je učitani program arhitekture domaćina ili ne, i u zavisnosti od toga se poziva *td_ta_init_target_consts()* ili ne.

5.3 Unapređenje GNU GDB alata prilikom čitanja/pisanja datoteke jezgra za MIPS procesorsku arhitekturu

Kako bi funkcionalnost bila moguća za svaku podržanu procesorsku arhitekturu, potrebno je da alat GNU GDB ima potpunu podršku za čitanje informacija iz datoteka jezgara. Podrška je za sve arhitekture potpuna, ali u trenutku razvoja ove funkcionalnosti uočeno je da alat nema potpunu podršku za čitanje identifikacionog broja procesa (eng. *Process ID*) iz datoteke jezgra za MIPS arhitekture, neophodnog za dohvaćanje bazičnih informacija o nitima. Kao što je napomenuto, koristeći komandu alata `gcore` moguće je kreirati datoteku jezgra i sa korisničkog nivoa, gde je takođe uočeno da alat nema potpunu podršku za ispisivanje nekih informacija o procesu za MIPS procesorsku arhitekturu. Te izmene su odrađene i prihvaćene od strane GNU zajednice. Takođe treba napomenuti da su te izmene ušle u izvršnu verziju alata.

Deo izvornog koda (u fajlu `bfd/elf32-mips.c`) izmene za čitanje informacija o procesu iz datoteke jezgra:

```
case 128:                                /* Linux/MIPS elf_prpsinfo */
    elf_tdata (abfd)->core->pid
    = bfd_get_32 (abfd, note->descdata + 16);
    elf_tdata (abfd)->core->program
    = _bfd_elfcore_strndup (abfd, note->descdata + 32, 16);
```

Deo izvornog koda (u fajlu `bfd/elf32-mips.c`) izmene za pisanje informacija o procesu u datoteku jezgra:

```
/* Write Linux core PRSTATUS note into core file. */

static char *
elf32_mips_write_core_note (bfd *abfd, char *buf, int *bufsiz,
    int note_type,
    ...)
{
    switch (note_type)
    {
        default:
            return NULL;
    }
}
```

```

case NT_PRPSINFO:
    BFD_FAIL ();
    return NULL;

case NT_PRSTATUS:
{
    char data[256];
    va_list ap;
    long pid;
    int cursig;
    const void *greg;

    va_start (ap, note_type);
    memset (data, 0, 72);
    pid = va_arg (ap, long);
    bfd_put_32 (abfd, pid, data + 24);
    cursig = va_arg (ap, int);
    bfd_put_16 (abfd, cursig, data + 12);
    greg = va_arg (ap, const void *);
    memcpy (data + 72, greg, 180);
    memset (data + 252, 0, 4);
    va_end (ap);
    return elfcore_write_note (abfd, buf, bufsiz,
                               "CORE", note_type, data,
                               sizeof (data));
}
}
}

```

Primeri izvornog koda izmena su prikazani samo za MIPS 32-bitnu verziju. Napomenimo da izmena obuhvata i MIPS 64-bitnu verziju procesorske arhitekture.

5.4 Testiranje

Faza testiranja je jako važan deo životnog ciklusa razvoja softvera, te je veliki značaj pridodat ovoj sekvenci. Testiranje je izvršeno na raznim verzijama platformi ARM i MIPS procesorskih arhitektura.

Takođe, implementirani su *DejeGNU*(REF NA DEJAGNU) testovi koji testiraju dohvaćanje vrednosti TLS promenljive iz datoteka jezgara raznih arhitektura. Zbog prirode problema koji se testira datoteke jezgra su generisane na različitim

platformama i kompresovane zajedno sa izvršnim fajlom u *gdb/testsuite/gdb.multi/*, te se prilikom izvršavanja konkretnih testova ti fajlovi raspakuju i učitavaju u debager. Testovi obuhvataju izvršne fajlove sa TLS-om generisanih sa 2.19 i 2.22 verzijama standardne biblioteke.

Komanda alata kojom pokrećemo testove, ukoliko smo pozicionirani u direktorijumu gde je alat izrađen:

```
|| make check
```

Pre izmene koja dodaje TLS funkcionalnost, rezultati izvršavanja testova *Multiaarch* GNU GDB verzije alata:

```
||
|| # of expected passes          33715
|| # of expected failures        196
|| # of known failures           66
|| # of unresolved testcases     5
|| # of untested testcases       67
|| # of unsupported tests        219
```

Posle izmene koja dodaje TLS funkcionalnost, rezultati izvršavanja testova *Multiaarch* GNU GDB verzije alata:

```
||
|| === gdb Summary ===
||
|| # of expected passes          33718
|| # of expected failures        196
|| # of known failures           66
|| # of unresolved testcases     5
|| # of untested testcases       67
|| # of unsupported tests        219
```

Izvršavanje samo novo kreiranih testova koji dohvataju vrednost TLS promenljive iz datoteke jezgra generisane na nekoj nedomaćinskoj platformi:

5.5 Upotreba alata

Kao što je i napomenuto, da bi GNU GDB uspešno koristio naprednije tehnike analize višenitnih programa, uključujući i TLS, potrebno je da mu se prosledi putanja do *libthread_db* koja pripada istoj verziji standardne biblioteke kao i program koji se debuguje. To isto važi i za *Multiaarch* GNU GDB verziju alata. Nakon toga potrebno je dati putanju do biblioteka koje je program koristio na tom na uređaju sa ugrađenim računarom na kome se program izvršavao. Primetimo da te biblioteke

pripadaju toj ciljanoj platformi. Debager neće izvršavati program koji se debuguje, već će samo rekonstruisati sliku stanja tog procesa kada je neočekivano prekinuo sa radom, čitajući informacije iz datoteke jezgra i deljenih biblioteka.

Izvorni kod višenitnog programa koji se debuguje u primeru je napisan u programskom jeziku C i izgleda:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

__thread int foo=0xdeadbeef;
pthread_t threads[5];

void* thread(void *e) {
    int *i = (int*)e;
    foo+=*i;
    printf("foo is %x\n", foo);
    sleep(10);
    return (void*)0;
}

int main()
{
    printf("init %x\n", foo);
    int i;
    for (i=0; i<5; i++) {
        pthread_create(&threads[i], NULL, thread, &i);
    }

    for (i=0; i<5; i++) {
        pthread_join(threads[i], NULL);
    }

    sleep(5);
    abort();
}
```

Prikaz korišćenja alata pre implementacije dohvaćanja vrednosti TLS promenljive iz datoteke jezgra generisane na nekom uređaju sa ugrađenim računarom je prikazan na slici 5.1.

Prvom i drugom komandom alata se podešavaju putanje do standardne biblioteke

```

GNU gdb (GDB) 7.12
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) add-auto-load-safe-path /home/rtrk/glibc/build_22/INSTALL/lib
(gdb) set libthread-db-search-path /home/rtrk/glibc/build_22/INSTALL/lib
(gdb) set solib-search-path ~/fax/master/master_examples/mips_arch
(gdb) file ~/fax/master/master_examples/mips_arch/example222
Reading symbols from ~/fax/master/master_examples/mips_arch/example222...done.
(gdb) core-file ~/fax/master/master_examples/mips_arch/core2.22
warning: core file may not match specified executable file.
[New LWP 21808]
[New LWP 21813]
[New LWP 21810]
[New LWP 21809]
[New LWP 21811]
[New LWP 21812]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/home/rtrk/glibc/build_22/INSTALL/lib/libthread_db.so.1".
Core was generated by './example222'.
Program terminated with signal SIGABRT, Aborted.
#0  0x00000000 in ?? ()
[Current thread is 1 (LWP 21808)]
(gdb) p/x foo
Cannot find user-level thread for LWP 21808: generic error
(gdb)

```

Slika 5.1: Korišćenje alata prikom analize TLS promenljive pre implementacije poboljšanja.

za debugovanje niti koja ima istu verziju kao i izvršni fajl koji se debuguje. Trećom komandom se zadaje putanja do deljenih biblioteka koje je izvršni fajl koji se debuguje korsitio na uređaju sa ugrađenim računarom. Četvrta komanda učitava izvršni fajl koji se debuguje. Petom komandom se učitava datoteka jezgra koja je generisana prilikom neočekivanog prekidanja izvršavanja fajla koji se debuguje. Poslednjom komandom se pokušava čitanje vrednosti TLS promenljive.

Prikaz korišćenja alata nakon implementacije dohvaćanja vrednosti TLS promenljive iz datoteke jezgra generisane na nekom uređaju sa ugrađenim računarom je prikazan na slici 5.2.

Prvom i drugom komandom alata se podešavaju putanje do standardne biblioteke za debugovanje niti koja ima istu verziju kao i izvršni fajl koji se debuguje. Trećom komandom se zadaje putanja do deljenih biblioteka koje je izvršni fajl koji se debuguje korsitio na uređaju sa ugrađenim računarom. Četvrta komanda učitava izvršni fajl koji se debuguje. Petom komandom se učitava datoteka jezgra koja je generisana

```
GNU gdb (GDB) 7.12
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) add-auto-load-safe-path /home/rtrk/glibc/build_22/INSTALL/lib
(gdb) set libthread-db-search-path /home/rtrk/glibc/build_22/INSTALL/lib
(gdb) set solib-search-path ~/fax/master/master_examples/mips_arch
(gdb) file ~/fax/master/master_examples/mips_arch/example222
Reading symbols from ~/fax/master/master_examples/mips_arch/example222...done.
(gdb) core-file ~/fax/master/master_examples/mips_arch/core2.22
warning: core file may not match specified executable file.
[New LWP 21808]
[New LWP 21813]
[New LWP 21810]
[New LWP 21809]
[New LWP 21811]
[New LWP 21812]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/home/rtrk/glibc/build_22/INSTALL/lib/libthread_db.so.1".
Core was generated by './example222'.
Program terminated with signal SIGABRT, Aborted.
#0  0x00000000 in ?? ()
[Current thread is 1 (Thread 0x771bf000 (LWP 21808))]
(gdb) p/x foo
$1 = 0xdeadbeef
(gdb)
```

Slika 5.2: Korišćenje alta prikom analize TLS promenljive nakon implementacije poboljšanja.

prilikom neočekivanog prekidanja izvršavanja fajla koji se debuguje. Poslednjom komandom se uspešno čita vrednosti TLS promenljive.

Glava 6

Zaključak

Oba predloga implementacije su poslata GNU zajednici i u toku je pregled i analiza implementacije, čija se diskusija može videti na [REF NA diskusiju]. I prvi i drugi način imaju svoje prednosti i mane. Mana prvog načina jeste ta što svaka promena arhitekturno zavisnih vrednosti promenljivih koje učestvuju u računanju lokacije TLS promenljive u *glibc*-u zahteva izmenu i u GNU GDB-u što dodatno otežava održavanje koda. Drugi način zahteva izmene i u izvornom kodu GNU GDB-a i *glibc*-a, ali ukoliko želimo da na sistemima arhitekture domaćina baratamo sa programima koji su prevedeni i izvršavani na nekim ciljnim arhitektura, što je za arhitekturu domaćina neprirodno, kompromisi jesu neophodni.

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] *Free Software Foundation*). on-line at: <https://www.fsf.org/>.
- [5] *GNU GDB*. on-line at: <https://www.gnu.org/software/gdb/>.
- [6] Reid Kleckner. *CodeView*. on-line at: <https://llvm.org/devmtg/2016-11/Slides/Kleckner-CodeViewInLLVM.pdf>.
- [7] *POSIX*. on-line at: <http://www.csc.villanova.edu/~mdamian/threads/posixthreads.html>.
- [8] Richard Stallman. *The GNU GDB documentation*). on-line at: <https://www.gnu.org/software/gdb/documentation/>.
- [9] *The GNU C Library (glibc)*. on-line at: <https://www.gnu.org/software/libc/>.
- [10] Red Hat Inc. Ulrich Drepper. *ELF Handling For Thread-Local Storage*. on-line at: <https://www.uclibc.org/docs/tls.pdf>.

Biografija autora

Dorđe Todorović, (*Užice, 12. Avgust 1993.*) Rođen sam u Užicu. Završio sam Gimnaziju u Požegi, Informatički smer, 2012. godine i iste godine upisao Matematički fakultet u Beogradu. 2016. godine sam završio osnovne studije Matematički fakulteta i iste upisao master studije. Položio sam sve ispite sa master studija u septembru 2018.godine. Od novembra 2015. pa do sada radim kao inženjer u Naučno-istraivačkom institutu RT-RK. Do sada sam objavio nekoliko naučnih radova iz oblasti debagera i generisanja debug informacija od strane programskih prevodi-laca. Na ovu temu opisanu u master radu sam objavio rad za naučnu konferenciju TELFOR [LINK KA TELFORU]. Trenutno radim na LLVM projektu, tačnije na poboljšanju tog prevodioca prilikom generisanja debug informacija.