Kompajlerski zasnovana dinamička analiza. Sanitajzeri. Svrha i mogućnosti. Alati i načini upotrebe. Primeri alata (gcc, Clang) i njihove mogućnosti.

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Nikola Nestorović, 1141/15
nikolanestorovic@hotmail.com

12. decembar 2018

Sažetak

Greške pri pristupanju memoriji uključujući prekoračenje bafera i upotreba oslobođene memorije ostaju ozbiljan problem za programske jezike. Postoji mnogo detektora grešaka u radu sa memorijom ali većina njih je spora ili može da otkrije ograničen skup bagova. U ovom radu predstavljen je Sanitajzer detektor grešaka razvijen od stane kompanije Gugl (eng. *Google*). U njemu se koristi specijalizovani alokator memorije i kodni instrumenti koji su dovoljno jednostavni za implementaciju u bilo kom kompajleru, binarnom prevodilačkom sistemu pa čak i hardveru.

Sadržaj

1 Uvod

Na raspolaganju su desetine alata za otkrivanje grešaka u memoriji koji se razlikuju u brzini, potrošnji memorije, vrstama grešaka koje mogu da detektuju, verovatnoći detektovanja greške, podržanim platformama i dr. Najmoćniji alat koji detektuje greške u toku dinamičke analize je Sanitajzer koji ima više tipova u zavisnosti od toga koji tip greške se detektuje. Adresni sanitajzer **(eng. *AddressSanitizer -ASan*)** postiže efikasnost od 73 % usporenja, 3.4 puta povećava upotrebu memorije i uvek tačno otkriva greške u trenutku pojave. Pronađeno je preko 300 ranije nepoznatih grešaka u Chrome pretraživaču i mnoštvo grešaka u drugom softveru. Ovaj alat može da otkrije sledeće vrste grešaka: pristup van opsega hipa, steka ili globala, upotrebu već oslobođene memorije, dvostruko oslobađanje memorije, curenje memorije i dr.Memorijski sanitajzer **(eng. *MemorySanitizer-MSan*)** je detektor čitanja neincijalizovane memorije. Neincijalizovane vrednosti javljaju se u trenutku kada se sadržaj memorije koja je dodeljena steku ili hipu pročita pre nego što je u nju upisana ispravna vrednost. Ovaj tip sanitajzera otkriva slučajeve u kojima takve vrednosti utiču na izvršavanje programa.Sanitajzer niti **(eng. *ThreadSanitizer-TSan*)** je dinamički detektor nekoordiniranog pristupa podacima (eng. *data races*) koji predstavlja neprijatanu vrstu greške u višenitnom programiranju [6] jer je teško pronaći je i reprodukovati.

2 Primeri upotrebe sanitajzera u gcc-u

Počevši od verzije 4.8 gcc komapjlera **AddressSanitizer** i **ThreadSanitizer** postali su deo njegovog standarda. Tokom faze kompajliranja programa AddressSanitizer podsistem izvršava dodatni kod za instrumente pristupa memoriji tokom izvršavanja programa. Ako se pristup memoriji smatra nevažećim izvršavanje programa je prekinuto i trag steka se štampa na standardnom izlazu za greške nagoveštavajući korisniku o lokaciji u izvornom kodu programa. Vrste neispravnih pristupa memorije koji su danas najčešći:

1) **pristup van opsega hipa kao i lokalne ili globalne promenljive**

2) **pristup memorijskoj lokaciji posle eksplicitnog oslobođanja tog dela memorije**

ThreadSanitizer generiše različiti kodni instrument za proveru pristupa memoriji za potencijalni nekoordinirani pristup podacima (eng. *data races*) u višenitnom kontekstu. Kada se otkrije data races trag steka se štampa na stanadardnom izlazu za greške kako bi se korisnicima pomoglo da pronađu lokaciju u izvornom delu koda. Trenutno ThreadSanitizer troši mnogo više vremena (skoro 5 puta više) od AddressSanitizer.

AddressSanitizer je trenutno dostupan za GNU/Linux na Intel, Arm i Power arihtekturi kao i za neke verzije Mac OSX. Thread sanitizer dostupan je trenutno samo za GNU/Linux x86-64 arhitekturu.

Primer 2.1 *Primer korišćenja AddressSanitize u otkrivanju [5]. curenje memorije u programu (eng.* memory leaking*).*

Ako naredni kod kompajliramo sa **gcc -ggdb -o s s.c -fsanitize=address** rezutat izvšavanja videćemo na slici 3

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
    char * buffer = malloc(1024);
    sprintf(buffer, "%d", argc);
    printf("%s",buffer);
}
```

Slika 1: Izvorni kod

```
$ ./s

=================================================================
==
==3911==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 1024 byte(s) in 1 object(s) allocated from:
    #0 0x7f55516b644a in malloc (/usr/lib/x86_64-linux-
gnu/libasan.so.2+0x9444a)
    #1 0x40084e in main /home/dlemire/tmp/s.c:6
    #2 0x7f555127eec4 in __libc_start_main (/lib/x86_64-linux-
gnu/libc.so.6+0x21ec4)

SUMMARY: AddressSanitizer: 1024 byte(s) leaked in 1
allocation(s).
```

Slika 2: Kao rezultat izvršavanja dobićemo poruku o curenju memorije

Postoje neke stvari koje treba uzeti u obzir prilikom korišćenja AddressSanitizer, čak i kada ASan otkrije nedozvoljen pristup memoriji on neće odmah srušiti aplikaciju. Mi možemo da primoramo ASan da sruši programa kada se desi nedozvoljen pristup na ovaj način:

ASAN_OPTIONS=abort_on_error=1

Primer 2.2 *Osnovni primer pristupa memorijskoj .[4] lokaciji van dozvoljenog opsega.*

```
int main() {
    int a[2] = {1, 0};
    int b=a[2];
}
```

Slika 3: Pristup memorijskoj lokaciji van opsega niza

Rezultat izvršavanja
==7402==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fff2971ab88 at pc 0x400904 bp 0x7fff2971ab40 sp 0x7fff2971ab30

Adresni sanitajzer će značajno usporiti izvršenje, bagovi koje pronađe su obično lakši od onih pronađenih bez njega (kao i uvek postoje izuzeci) ono što je sigurno je da će naći mnogo više grešaka.

3 Primeri upotrebe sanitajzera u Clang-u

Kako je Clang nastao kao prevodilac za programske jezike C, C++, Objective-C i Objective-C++ kao naslednik gcc-a (sa kojim je kompatabilan) od svoje 3.1 verzije ima implementiran AddressSanitizer, ThreadSanitizer, MemorySanitizer kao i DataFlowSanitizer koji za razliku od ostalih nije dizajniran da sam detektuje određenu klasu grešaka već nudi generički okvir za analizu dinamičkih podataka koji klijenti koriste da bi otkrili specifične probleme u svojoj aplikaciji.

U nastavku je prikazan učinak ThreadSanitizer-a. Prlikom kompajliranja potrebno je koristiti flag **-fsanitize=thread**, da bi se dobile razumne perfomanse može se koristiti **-O1** ili više, prilikom kompajliranja poželjno je koristiti i **-g** da bi se dobila imena datoteka i brojevi linija u porukama upozorenja.

ThreadSanitizer generalno zahteva da se svi kodovi kompajliraju sa flagom -fsanitize=thread. Ako neki kod nije kompajliran sa ovim flegom (npr. dinamičke biblioteke) to može dovesti do lažnih pozitivnih izveštaja, lažnih negativnih izveštaja ili propuštenih stekova u zavisnosti od prirode koda. Da ne bi proizvodio lažne izveštaje ThreadSanitizer mora da vidi sve sinhronizacije u programu, neke operacije sinhronizacije (atomske operacije ili statička inicijalizacija) su presretnute tokom kompajliranja i mogu se presresti jedino tokom kompajliranja. Sakuplanje tragova stekova se takođe oslanja na instrumente kompajlera jer je odlaganje steka na svaki pristup memoriji preskupo.

Primer 3.1 *Nekoordinirani pristup podacima (data conditions) se dešava kada dve niti istovremeno pristupe istoj varijabli menjajući njen sadržaj, program u tom slučaju ima nedefinisano ponašanje 4.*

```
$ cat simple_race.cc
#include <pthread.h>
#include <stdio.h>

int Global;

void *Thread1(void *x) {
  Global++;
  return NULL;
}

void *Thread2(void *x) {
  Global--;
  return NULL;
}

int main() {
  pthread_t t[2];
  pthread_create(&t[0], NULL, Thread1, NULL);
  pthread_create(&t[1], NULL, Thread2, NULL);
  pthread_join(t[0], NULL);
  pthread_join(t[1], NULL);
}
```

Slika 4: Primer koda [3] gde dve niti pristupaju istoj globalnoj promenljivoj

```
$ clang++ simple_race.cc -fsanitize=thread -fPIE -pie -g
$ ./a.out
==================
WARNING: ThreadSanitizer: data race (pid=26327)
  Write of size 4 at 0x7f89554701d0 by thread T1:
    #0 Thread1(void*) simple_race.cc:8 (exe+0x000000006e66)

  Previous write of size 4 at 0x7f89554701d0 by thread T2:
    #0 Thread2(void*) simple_race.cc:13 (exe+0x000000006ed6)

  Thread T1 (tid=26328, running) created at:
    #0 pthread_create tsan_interceptors.cc:683 (exe+0x00000001108b)
    #1 main simple_race.cc:19 (exe+0x000000006f39)

  Thread T2 (tid=26329, running) created at:
    #0 pthread_create tsan_interceptors.cc:683 (exe+0x00000001108b)
    #1 main simple_race.cc:20 (exe+0x000000006f63)
==================
ThreadSanitizer: reported 1 warnings
```

Slika 5: Kao rezultat izvršavanja dobićemo poruku o nekooridniranom pristupu podacima

Primer 3.2 *Sledeći primer na jednostavnom kodu ilustruje upotreba neinicijalizovanih podataka i grešku 8 prilikom izvršavanja programa koju prijavljuje [2] MemorySanitizer.*

```
% cat umr.cc
#include <stdio.h>

int main(int argc, char** argv) {
  int* a = new int[10];
  a[5] = 0;
  if (a[argc])
    printf("xx\n");
  return 0;
}
```

Slika 6: Jednostavan primer koda upotreba neinicijalizovanih podataka

```
% ./a.out
WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x7f45944b418a in main umr.cc:6
    #1 0x7f45938b676c in __libc_start_main libc-start.c:226
```

Slika 7: Kao rezultat izvršavanja dobićemo poruku o korišćenju neinicijalizovane vrednosti

Primer 3.3 *Sledeći primer islustruje upotrebu već oslobođene memorije u programu.[1]*

```
% cat example_UseAfterFree.cc
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc];  // BOOM
}

# Compile and link
% clang++ -O1 -g -fsanitize=address -fno-omit-frame-pointer example_UseAfterFree.cc
```

Slika 8: Primer koda i poziv adresnog sanitajzera

Rezultat izvršavanja programa:

==9442== ERROR: AddressSanitizer heap-use-after-free on address 0x7f7ddab8c084 at pc 0x403c8c bp 0x7fff87fb82d0 sp 0x7fff87fb82c8 READ of size 4 at 0x7f7ddab8c084 thread T0

4 Zaključak

Sanitajzeri su alati koji pomoću instrumentalizacije (ubacivanja novog koda u već napisan program) pokušavaju da otkriju složenije nepravilnosti u radu programa. Za razliku od ostalih alata (npr. AddrCheck zasnovan na Valgrindu) adresni sanitajzer, memorijski sanitajzer i sanitajzer niti zahvaljujući svojim algoritmima i načinu implementacije postižu veliku efikasnost u otkrivanju grešaka u programu što rezultuje prekidom programa i ispisom odgovarajuće greške na standardni izlaz. Otkrivanjem i upotrebom sanitajzera umnogome se olakšava proces verifikacije izvornog koda i eliminisanje do sad najčešćih grešaka u radu sa memorijom i nitima.

Literatura

[1] clang.llvm.org. AddressSanitizer –Clang. on-line at: https://clang.llvm.org/docs/AddressSanitizer.html.

[2] clang.llvm.org. MemorySanitizer –Clang. on-line at: https://clang.llvm.org/docs/MemorySanitizer.html.

[3] clang.llvm.org. ThreadSanitizer –Clang. on-line at: https://clang.llvm.org/docs/AddressSanitizer.html.

[4] fuzzing project.org. Find more Bugs with Address Sanitizer. on-line at: https://fuzzing-project.org/tutorial2.html.

[5] Alexander Potapenko Konstantin Serebryany, Derek Bruening. AddressSanitizer: A Fast Address Sanity Checker, 2012. on-line at: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/37752.pdf/.

[6] Timur Iskhodzhanov Konstantin Serebryany. ThreadSanitizer – data race detection in practice, 2012. on-line at: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/35604.pdf.