Testing e debugging di errori di memoria in programmi C e C++

Usando Valgrind Memcheck e AddressSanitizer

Marco Cutecchia - 2021 - Verifica e Convalida

Materiale e slide qui: https://github.com/mrkct/vec21-memory-errors

Il problema

- Linguaggi di programmazione come Java gestiscono la memoria al nostro posto
- Invece in linguaggi come C e C++ bisogna gestire la memoria manualmente
- Gestire la memoria correttamente è difficile però...
- *"The Chromium project finds that around 70% of our serious security bugs are memory safety problems"*
- *"~70% of the vulnerabilities Microsoft assigns a CVE each year continue to be memory safety issues"*

Analisi dinamica

- Eseguiamo il programma e controlliamo che non faccia cose sbagliate a runtime
- Troviamo questi errori solo se effettivamente li eseguiamo
- È importante dunque avere una suite di test che copra il programma
- Non esclude gli altri tipi di analisi! È possibile utilizzarli contemporaneamente

In questa presentazione vedremo due tool che **instrumentano** il codice binario per fare questo tipo di analisi

Instrumentazione del codice binario

Il codice macchina del programma viene modificato per introdurre degli hook che permettono al tool di analizzare il comportamento del programma

- Nei programmi C per trovare i memory leak un esempio è sostituire l'implementazione delle funzioni di allocazione e deallocazione di memoria (`malloc`, `free` etc.) e controllare a fine programma se tutta la memoria è stata rilasciata
- L'instrumentazione ha spesso un **pesantissimo** costo in termini di performance ed uso di memoria, per questo non la teniamo attiva in produzione

Shadow Memory

I tool che vedremo controllano prima di ogni accesso in memoria se l'indirizzo a cui stiamo per accedere è valido

Per fare questo devono tenere traccia di quali indirizzi sono validi

La **shadow memory** è una zona di memoria usata per tenere metadati sulla memoria stessa

- Può diventare molto grande, i tool hanno modi diversi di rappresentarla per bilanciare spazio occupato e performance in accesso

Valgrind

- Un framework di binary instrumentation sulla quale sono stati costruiti diversi tool
- Il più popolare tra questi tool è **MemCheck**
- Disponibile solo su Linux e MacOs, ma esistono tool che funzionano con lo stesso approccio per Windows (Dr.Memory)
- Per eseguirlo: `valgrind --tool=memcheck ./programma arg1 arg2 ...`

Come funziona Valgrind

1. Il programma viene diviso in blocchi di codice
2. A runtime questi blocchi vengono convertiti in un IR, modificato dal tool e ri-convertito in assembly
3. Il codice risultante viene eseguito su una CPU virtuale per poterne monitorare il comportamento

Aggiungere i simboli di debugging e disattivare le ottimizzazioni in fase di compilazione permette di avere log più precisi, ma non è necessario

Shadow memory in Valgrind[4]

Valgrind mantiene 3 metadati nella shadow memory:

- **A**ddressability: Indicano per ogni byte se è accessibile dal programma o meno
- **V**alidity: Indicano per ogni bit se sono stati inizializzati dall'utente
- Hash Blocks: Tracciano ogni blocco di memoria allocato dinamicamente

Valgrind struttura la shadow memory come una tabella a più livelli, inizializzando i livelli più bassi solo quando avviene una write.

Valgrind include una miriade di ottimizzazioni per velocizzare i casi più comuni

Demo di Memcheck

AddressSanitizer (ASan)

- Instrumentazione a compile time
- Richiede il supporto del compilatore e linker, supportato da tutti i major compiler ormai
  - `gcc programma.c -o programma -fsanitize=address -g`
  - `clang programma.c -o programma -fsanitize=address -g -fno-omit-frame-pointer`
  - Da qualche mese è anche disponibile tra le opzioni in Visual Studio
- Aggiungere i simboli di debugging e disattivare alcune ottimizzazioni permette di avere log più precisi

Come funziona AddressSanitizer

- Attorno a tutte le zone di memoria allocate dinamicamente o sullo stack vengono aggiunte delle **redzones**, zone di memoria segnate come non utilizzabili nella shadow memory
- Prima di ogni accesso in memoria viene controllata nella shadow memory se l'indirizzo è segnato come valido. Se non è così il programma viene arrestato e viene stampato un log

Shadow Memory in AddressSanitizer[3]

- Inizia ad un indirizzo prefissato in memoria supposto libero
- Ogni byte nella shadow memory contiene le informazioni per un blocco di 8 byte
- Il valore 0 indica che tutti gli 8 byte del blocco sono accessibili
- Un valore compreso tra 1 e 7 indica quanti byte contigui a partire dall'inizio del blocco sono accessibili
- I rimanenti valori sono utilizzati per indicare perchè non è accessibile la zona di memoria
    - 0xfd: Zona di heap recentemente rilasciata
- Viene stampata una legenda quando troviamo un fault

Demo di address sanitizer

Il brutto di questi tool

- Controllare ogni accesso in memoria ha un costo significativo in termini di performance
  - Valgrind: dalle 10 alle 50 volte più lento [5]
  - AddressSanitizer: Circa il 73% più lento [3]
    - Alcuni compilatori permettono di segnalare alcune funzioni da NON instrumentare
- Aumento significativo dell'uso di memoria
  - AddressSanitizer: 3.4x in media [3]
- AddressSanitizer aumenta di molto anche i tempi di compilazione

Falsi negativi e positivi

AddressSanitizer e Valgrind non riescono a trovare tutti i problemi

- Per alcuni problemi semplicemente non sono stati implementati i check
- Altri problemi sono proprio difficili da trovare:
  - E se sforassimo talmente tanto da un buffer che saltiamo le redzones e l'indirizzo in cui finiamo è un'altra zona di heap?

In alcuni casi di indecisione Valgrind potrebbe segnalare dei falsi positivi (segnalandoci che non è certo), AddressSanitizer preferisce stare zitto

Per concludere

- I tool di analisi dinamica della memoria sono ormai semplicissimi da usare
- Quando si scrive codice e nella esecuzione dei test possono dare un grande aiuto a trovare bug anche subdoli
- Non trovano tutti i problemi però, non si può abbassare la guardia

Qualche extra riguardo AddressSanitizer

- AddressSanitizer + fuzzing è una coppia interessante
  - Qui una presentazione Google di come applicano questa cosa per Chrome:
  - https://www.usenix.org/sites/default/files/conference/protected-files/enigma_slides_serebryany.pdf
- AddressSanitizer non è l'unico "Sanitizer": date un occhiata anche ad *ThreadSanitizer*, *UBSan*, *SafeStack* ed altri
  - Non sono disponibili in tutti i compilatori, ed alcuni vanno in conflitto tra loro: guardate il manuale del vostro compilatore

Qualche extra riguardo Valgrind

- Valgrind è un framework sulla quale sono stati costruiti diversi tool
- Se state sviluppando un'applicazione dove le performance sono importanti potrebbero interessarvi:
  - Cachegrind: profiling della cache
  - Callgrind: profiling delle chiamate a funzione
  - Massif: profiling dell'heap

Hardware Assisted Address Sanitizer [7]

- Evoluzione di ASan che rende l'overhead sulla CPU (quasi) 0
- Ha bisogno di supporto hardware e anche dal sistema operativo
- Salva alcune informazioni nei puntatori stessi e lascia all'hardware il compito di controllare nella shadow memory
- Al momento il supporto è veramente basso, solo architetture SPARC e parzialmente AArch64 (ARM) su un kernel linux patchato
- Forse tra qualche anno ci sarà più supporto...

References e link utili

- [1] https://github.com/google/sanitizers/wiki/AddressSanitizer
- [2] https://valgrind.org/info/tools.html#memcheck
- [3] AddressSanitizer: A Fast Address Sanity Checker, 2012 USENIX ATC, Konstantin Serebryany and Derek Bruening and Alexander Potapenko and Dmitriy Vyukov
- [4] How to Shadow Every Byte of Memory Used by a Program, ACM SIGPLAN/SIGOPS VEE 2007, Nicholas Nethercote and Julian Seward
- [5] https://www.valgrind.org/docs/manual/manual-core.html
- [6] Memory Tagging and how it improves C/C++ memory safety, Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, Dmitry Vyukov