Narzędzia do debugowania - Valgrind

Arkadiusz Kasprzak

Spis treści

1 Podstawowe pojęcia

Po co nam narzędzia do debugowania? Przykładowe powody:

- rozwój oprogramowania sprawia, że kod jest coraz bardziej skomplikowany
- brak ochrony przed błędami dostarczanej przez języki (C/C++)
- debugowanie bez narzędzi jest czasochłonne i możliwe głównie na poziomie kodu źródłowego lub obserwacji działania programu

1.1 Analiza programu

Samo pojęcie analizy programu można formalnie podzielić na kilka sposobów. Jeden z możliwych podziałów:

- analiza statyczna (static analysis) - analiza kodu źródłowego lub maszynowego **bez uruchamiania programu**. Pozwala rozpatrzeć wszystkie ścieżki wykonania. Przykład: kompilator.
- analiza dynamiczna (dynamic analysis) - analiza programu **w trakcie jego wykonania**. Analizowana jest jedna ścieżka (ta aktualnie wykonywana), ale za to analiza jest dokładniejsza.

Istnieje również drugi sposób podziału analizy programu:

- analiza kodu źródłowego - analiza programu na poziomie funkcji, wyrażeń, zmiennych (kod źródłowy). Charakteryzuje ją dostęp do informacji wysokopoziomowych. Niezależna od platformy, zależna od języka.
- analiza binarna/kodu binarnego - analiza programu na poziomie kodu maszynowego. Analiza na poziomie rejestrów i instrukcji - czyli ma dostęp do informacji niskopoziomowych. Jest w sporym stopniu niezależna od języka, ale bardzo zależna od platformy. Zaletą jest brak konieczności posiadania kodu źródłowego.

Przedstawione podziały pozwalają wygenerować podział analizy programów na **4 kategorie**: statyczna analiza kodu źródłowego, dynamiczna analiza kodu źródłowego, statyczna analiza binarna, dynamiczna analiza binarna.
W kontekście Valgrinda interesuje nas **dynamiczna analiza binarna - DBA** - analiza kodu maszynowego mająca miejsce w czasie wykonywania programu.

1.2 Instrumentacja

Jest to dodawanie do kodu dodatkowych instrukcji (kod analizujący) pozwalających na uzyskanie nowych informacji od działaniu aplikacji. Jest kilka rodzajów, przy czym interesuje nas instrumentacja binarna (w odróżnieniu od instrumentacji kod źródłowego i instrumentacji w ramach kompilacji). Podział **technik instrumentacji binarnej**:

- statyczna instrumentacja binarna - ma miejsce przed rozpoczęciem działania programu - kod obiektowy lub wykonywalny jest wtedy przepisywany
- dynamiczna instrumentacja binarna **DBI** - ma miejsce **w czasie działania programu**.

1.3 DBI - Dynamiczna Instrumentacja Binarna

Interesujące nas podejście (na jej zasadzie działa Valgrind). Ma miejsce w czasie działania programu. Do aplikacji wstrzykiwany jest dodatkowy kod (instrukcje monitorujące). Staje się on częścią wykonania programu. Zaletą tej metody jest brak konieczności przygotowywania kodu wcześniej. Największą wadą jest oczywiście przeniesienie kosztu instrumentacji na czas wykonania programu - co mocno wpływa na wydajność. Ponadto realizacja takiego zabiegu jest trudna.

2 Valgrind

2.1 Czym jest Valgrind?

Valgrind jest **frameworkiem** do tworzenia narzędzi wykonujących **dynamiczną analizę binarną** dzięki zastosowaniu techniki **DBI**.
Architektura Valgrinda:

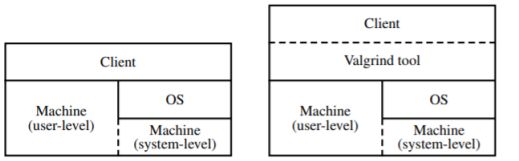
Valgrind core + tool plug-in = Valgrind tool

Narzędzia (takie jak znany wszystkim Memcheck) są tworzone w języku C jako **wtyczki** do rdzenia Valgrinda. Zadaniem narzędzia jest **instrumentacja fragmentu kodu przekazanego mu przez rdzeń**. Taka architektura ułatwią tworzenie nowych narzędzi.

2.2 Valgrind - działanie rdzenia

- Valgrind podłącza się do badanego programu a następnie (re)kompiluje kod, jeden podstawowy blok na raz, w systemie just-in-time (JIT).
- Kompilacja polega tutaj na deasemblacji kodu maszynowego do tzw. **postaci pośredniej**, która przekazywana jest do narzędzi takich jak Memcheck.
- W narzędziach kod poddawany jest instrumentacji - w zależności od potrzeb narzędzia może ona być różna.
- Wynik jest ponownie konwertowany do postaci maszynowej. Efekt pracy nazywa się **translation**.

W przypadku korzystania z Valgrinda badany program (client) i narzędzie **są częścią jednego procesu**. Pełną kontrolę nad tym, co się stanie, ma narzędzie, bo to ono prowadzi interakcję z systemem i maszyną. Obrazuje to poniższy rysunek:



(a) Normal execution (b) Execution under a Valgrind tool

2.3 Valgrind - etapy tworzenia translacji

- deasemblacja - Valgrind reprezentuje kod w postaci pośredniej (UCode)
- optymalizacja - poprawa jakości kodu po deasemblacji
- instrumentacja - dodanie kodu analizującego
- alokacja rejestrów - rejestry wirtualne są przypisywane do rejestrów ogólnego przeznaczenia
- generowanie kodu - UCode konwertowany powrotnie do kodu maszynowego, każda instrukcja niezależnie

Po wykonaniu translacji **jednego bloku podstawowego** jest on wykonywany i analiza przechodzi do kolejnego.

2.4 Valgrind - narzędzia

- Memcheck - do wykrywania błędów związanych z pamięcią, np. wycieków
- Cachegrind - do profilowania, symuluje interakcję programu z pamięcią cache
- Callgrind - narzędzie do profilowania pozwalające na analizę grafu wywołań funkcji w programie.
- Massif - narzędzie do analizy sterty (heap)
- Hellgrind i DRD - narzędzie do wykrywania błędów związanych z wątkami.
- inne, np Nulgrind

3 Memcheck

Narzędzie do wykrywania błędów związanych z pamięcią. Badaliśmy nim programy napisane w C i C++. Jego użycie spowalnia program od 10 do 30 razy. Wykrywa następujące **rodzaje błędów**:

- użycie niezainicjowanych wartości (również w wywołaniach systemowych)
- niepoprawne wartości definiujące rozmiar (np. ujemne)
- niedopasowanie funkcji alokującej i zwalniającej pamięć (np. alokacja za pomocą new i zwalnianie za pomocą free)
- próba dostępu do niezaalokowanej pamięci
- niepoprawne zwalnianie pamięci (np. podwójne)
- nakładające się zakresy w funkcjach typu strcpy()
- **wycieki pamięci**

Uruchomienie Memchecka:

```
valgrind <program_name>
```

lub

```
valgrind --tool=memcheck <program_name>
```

3.1 Memcheck - wycieki pamięci

Działanie Memchecka polega na śledzeniu wszystkich bloków na stercie zaalokowanych za pomocą malloc/new. Dzięki temu na zakończenie programu Memcheck jest w stanie stwierdzić, które z tych bloków nie zostały zwolnione. Memcheck wyróżnia 4 **rodzaje wycieków pamięci**:

- definitely lost - nie ma wskaźnika pokazującego na zaalokowany obszar pamięci. Programista nie jest więc w stanie zwolnić pamięci. Najpoważniejszy rodzaj wycieku.
- indirectly lost - istnieją wskaźniki do tej pamięci ale i tak nie ma tam dostępu, np. jeśli drzewie binarnym węzeł główny zostanie utracony, wszystkie jego węzły dzieci zostaną pośrednio utracone.
- possibly lost - łańcuch jednego lub więcej wskaźników do zaalokowanego bloku znaleziony, ale conajmniej jeden nie wskazuje na jego początek, ale gdzieś w środku - co może być wynikiem błędu.
- still reachable - pamięć jest osiągalna, znaleziono wskaźnik do początku zaalokowanej pamięci. Wskaźnik cały czas jest dostępny, więc w teorii programista powinien go zwolnić. Takie bloki pojawiają się często i łatwo je eliminować.

3.2 Memcheck - szczegóły działania

3.2.1 Operatory alokujące

Memcheck w celu przeprowadzenie analizy pamięci zamienia standardowe funkcje zarządzające pamięcią (np. malloc) na swoje własne. Pozwala mu to śledzić alokacje na stercie. W przypadku operatorów z rodziny new/delete zamieniane są ich **globalne wersje** - oznacza to m.in. że program uruchomiony z Memcheckiem nie używa przeładować tych operatorów zdefiniowanych przez użytkownika.

3.2.2 Jak Memcheck wykrywa nieuprawniony dostęp do pamięci?

Każdy **bajt** w pamięci ma skorelowany ze sobą tzw. **A-bit** (valid address bit). Wskazuje on, czy program może pisać lub czytać w tego miejsca w pamięci (czyli czy próba dostępu do niego jest poprawną operacją). **Uwaga:** A-bit nie wskazuje, czy dane w pamięci są poprawne (np. czy zostały zdefiniowane). Za każdym

razem, gdy program odczytuje lub zapisuje coś do pamięci, Memcheck sprawdza A-bit związany z adresem. Jeśli wskazuje on, że nie mamy prawa wykonać danej operacji, pojawia się błąd. Poniżej przykład kodu, gdzie taki błąd występuje:

```
std::vector<int> testVector { 1, 2, 3, 4, 5 };
testVector[5] = 6; // Invalid write of size 4

int tempValue = testVector[5]; // Invalid read of size 4
```

W momencie użycia malloc/new A-bity związane z zaalokowanym obszarem (i tylko z nim!) są ustawiane, by wskazywać możliwość dostępu. Przy zwalnianiu pamięci bity te są ustawiane tak, by wskazywać brak możliwości dostępu.

3.2.3 Jak Memcheck sprawdza, czy wartość jest zainicjowana?

Każdy **bit** danych zarządzany przez program (zarówno w rejestrach jak i w pamięci) jest śledzony (nie wiem jak lepiej przetłumaczyć shadowed) przez **metadane w postaci innego bitu - tzw. V-bit** (validity bit). Każdy V bit wskazuje, czy bit, który śledzi, ma aktualnie poprawnie zdefiniowaną wartość. Memcheck wykrywa więc niezainicjowane wartości z precyzją pojedynczych bitów - kosztem dwukrotnego wzrostu zużycia pamięci wynikającego tylko z tego mechanizmu.

3.3 Valgrind - makra - client request

Valgrind posiada mechanizm pozwalający badanemu programowi (client) na przekazywanie prostych zapytań **do rdzenia oraz aktualnie używanego narzędzia** (np. Memcheck). Dostępne one są w postaci makr do użycia w kodzie źródłowym aplikacji. Makra te generują kod inline. Jest on widoczny dla Valgrinda, jednak nie wpływa na wykonanie programu przy uruchomieniu bez niego. Potrzebne nagłówki:

- "valgrind/valgrind.h- dla makr, którymi zajmuje się rdzeń Valgrinda
- "valgrind/memcheck.h- dla makr, którymi zajmuje się narzędzie (tutaj Memcheck).

Przykład z makrami rdzenia:

```cpp
bool isRunningOnValgrind = RUNNING_ON_VALGRIND;
std::cout <<  std::boolalpha << isRunningOnValgrind << std::endl;

int numberOfErrors = 0;

// VALGRIND_COUNT_ERRORS macro - zwraca liczbę znalezionych do tej
// pory błędów. Nie działa dla wszystkich narzędzi - działa np. dla
// Memchecka, ale już np. dla Cachegrinda zawsze zwraca 0, ponieważ
// Cachegrind nie wyszukuje błędów.
numberOfErrors = VALGRIND_COUNT_ERRORS;

if (isRunningOnValgrind) {
    std::cout << "Number of found errors = " << numberOfErrors << std::endl;
}
```

Oraz z makrami dla Memchecka:

```cpp
// VALGRIND_CHECK_MEM_IS_DEFINED - sprawdza, czy wartości w pamięci
// są zdefiniowana - czyli czy zostały zainicjalizowane
VALGRIND_CHECK_MEM_IS_DEFINED(secondPointer, 2 * sizeof(int));

// VALGRIND_MAKE_MEM_DEFINED - sprawia, że Memcheck widzi obszar
// pamięci jako taki ze zdefiniowanymi danymi
VALGRIND_MAKE_MEM_DEFINED(secondPointer, 2 * sizeof(int));

VALGRIND_CHECK_MEM_IS_DEFINED(secondPointer, 2 * sizeof(int));
```

3.4 Memcheck - filtrowanie błędów

Pliki z filtrami pozwalają wybrać nam błędy, które nie powinny być wyświetlane, gdy korzystamy z Memchecka. Pozwala to wyeliminować raporty o błędach, na które nie mamy bezpośredniego wpływu (np. pochodzące z bibliotek, których używamy). Przykład filtru poniżej:

```
{
   Condition_1
   Memcheck:Cond
   fun:_Z3funv
   fun:main
}
```

Pierwsza linia to nazwa filtra (wybieramy sobie ją sami). Potem mamy nazwę narzędzia i po dwukropku typ błędu, który chcemy filtrować. Następnie może pojawić się linia z dodatkowymi informacjami (tylko dla niektórych filtrów, tutaj jej nie ma). Pozostałe linie to kontekst - ścieżka wywołań funkcji prowadząca do błędu. Tego typu informację łatwo wygenerować uruchamiając Memchecka z opcją

```
--gen-suppressions=yes
```

4 Cachegrind

Cachegrind to narzędzie przeprowadzające symulację działania pamięci Cache (pamięć podręczna procesora) w czasie działania programu. Służy do **profilowania** programu.

4.1 Profilowanie

Forma dynamicznej analizy programu. Badanie zachowania programu używając informacji zdobytych podczas jego wykonania. Mogą one dotyczyć np. zużycia pamięci czy wywołań funkcji. Profilowanie pozwala nam dowiedzieć się, które części programu można by zoptymalizować.

4.2 Teoria o cache

- pamięć podręczna procesora
- szybka
- często jest częścią procesora
- pamięć statyczna - przechowuje dane bez przeładowywania (SRAM)
- podzielona na poziomy: L1, L2...
- wraz z poziomem rośnie wielkość pamięci, ale maleje szybkość
- podzielone na pamięć z instrukcjami i z danymi

Kiedy procesor szuka w pamięci cache danych do przeprowadzenia operacji, może być jedna z dwóch sytuacji:

- cache hit - dane znalezione w pamięci podręcznej
- cache miss - danych nie ma w pamięci podręcznej, trzeba je dopiero pobrać. Niekorzystne wydajnościowo.

Za pomocą Cachegrind możemy badać program ze względu na te dwa zjawiska

4.3 Działanie

Cachegrind symuluje szczegółowo dwa poziomy pamięci cache - pierwszy i ostatni. Na poziomie pierwszym wprowadza dodatkowy podział na I1 - cache do instrukcji, i D1 - cache do danych. Cachegrind pozwala na śledzenie statystyk dla **każdej linii kodu źródłowego**. Cachegrind bardzo znacząco spowalnia badany program. Do analizy danych wyprodukowanych przez Cachegrind służy program cg_annotate.

5 Massif

Narzędzie służące do profilowania sterty. Massif mierzy, jak dużo pamięci na stercie zużywa program. Poza tym posiada również opcje do badania stosu. Massif gromadzi dane nie tylko o rozmiarach bloków pamięci przydzielonych przez program, ale także zbiera dane o tym, ile pamięci jest używane do przechowywania informacji pomocniczych. Po zakończeniu programu wypisywane jest krótkie podsumowanie - szczegółowe dane trafiają natomiast do osobnego pliku. Analizy tego pliku można dokonać za pomocą programu ms_print.
Użycie:

```
valgrind --tool=massif <program>
```

Ciekawą cecha Massifa jest możliwość wybrania jednostki czasu użytej przy profilowaniu (czas rzeczywisty, ilość zaalokowanych i dealokowanych bajtów, oraz ilość instrukcji).

6 Callgrind

Narzędzie służące do profilowania, które rejestruje historię wywołań funkcji. Zebrane dane składają się z liczby wykonanych instrukcji, ich związku z liniami źródłowymi, oraz liczby wywołań. Dane profilu zapisywane są do pliku przy zakończeniu program. Aby zaprezentować wygenerowane dane można wykorzystać dwa narzędzia wiersza poleceń:

- callgrind_annotate
- callgrind_control