AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Narzędzia do debugowania

Arkadiusz Kasprzak

Wydział Fizyki i Informatyki Stosowanej
Informatyka Stosowana

26.03.2019

Plan prezentacji

1. Wstęp teoretyczny - wyjaśnienie kilku pojęć.
2. Architektura i działanie Valgrinda
3. Narzędzia Valgrinda - Memcheck
4. Narzędzia Valgrinda - Cachegrind
5. Narzędzia Valgrinda - Massif

AGH

Wstęp teoretyczny

Po co nam narzędzia do analizy programów?

- rozwój oprogramowania sprawia, że kod coraz bardziej skomplikowany
- brak ochrony przed błędami dostarczonej ze strony samych języków (C i C++)
- debugowanie bez pomocy narzędzi czasochłonne, możliwe głownie na poziomie kodu źródłowego lub obserwacji działania programu

Kilka podstawowych pojęć

Aby zrozumieć jak działa i czym tak naprawdę jest Valgrind, musimy najpierw poznać kilka pojęć z nim związanych

- analiza programu i jej rodzaje
- instrumentacja kodu
- architektura Valgrinda

Analiza programu

Sposoby analizowania programów można podzielić na kilka sposobów, w zależności od przyjętego kryterium podziału. Jeden z możliwych podziałów:

- analiza statyczna (static analysis) - analiza kodu źródłowego lub maszynowego **bez uruchamiania programu**. Jaki znacie przykład?
- analiza dynamiczna (dynamic analysis) - polega na analizie programu **w trakcie jego wykonania**. Tym rodzajem analizy będziemy się zajmować dalej.

Jak myślicie, jakie są zalety i wady powyższych podejść?

Analiza programu

Sposoby analizowania programów można podzielić na kilka sposobów, w zależności od przyjętego kryterium podziału. Jeden z możliwych podziałów:

- analiza statyczna (static analysis) - analiza kodu źródłowego lub maszynowego **bez uruchamiania programu**. Jaki znacie przykład? kompilatory
- analiza dynamiczna (dynamic analysis) - polega na analizie programu **w trakcie jego wykonania**. Tym rodzajem analizy będziemy się zajmować dalej.

Jak myślicie, jakie są zalety i wady powyższych podejść?

Analiza programu

Statyczna analiza programu pozwala rozpatrzeć wszystkie ścieżki wykonania - tak robią np. wspomniane wcześniej kompilatory. Analiza dynamiczna natomiast bierze pod uwagę tylko jedną ścieżkę - jest to tak ścieżka, która została wybrana w trakcie wykonania programu. Zaletą dynamicznej analizy jest natomiast fakt, że na tej konkretnej ścieżce pozwala na dokładniejszą analizę i wykrywanie nowych rodzajów błędów.

Analiza programu

Drugi sposób podziału analizy programu:

- analiza kodu źródłowego - analiza programu na poziomie kodu źródłowego (funkcje, wyrażenia, zmienne). Ma więc dostęp do informacji wysokopoziomowych. Tutaj znów jako przykład można podać kompilatory. Jest niezależna od platformy, ale zależna od języka.
- analiza binarna / kodu binarnego - analiza programu na poziomie kodu maszynowego (obiektowego przed linkowaniem lub wykonywalnego po tymże). Jest to analiza na poziomie rejestrów, instrukcji itp. - czyli ma dostęp do informacji niskopoziomowych. Jest w sporym stopniu niezależna od języka, ale zależna od platformy. Ogromną zaletą tego rodzaju analizy jest fakt, iż nie musimy posiadać kodu źródłowego by ją wykonać.

Analiza programu

Przedstawione podziały pozwalają wygenerować podział analizy programów na 4 główne kategorie:

- statyczna analiza kodu źródłowego
- dynamiczna analiza kodu źródłowego
- statyczna analiza binarna
- dynamiczna analiza binarna

My zajmować będziemy się **dynamiczną analizą binarną** (dynamic binary analysis - DBA) - czyli analizą kodu maszynowego mającą miejsce w czasie wykonywania programu.

Instrumentacja

Kolejne ważne dla nas pojęcie. **Instrumentacja** polega na dodawaniu do kodu dodatkowych instrukcji (kod analizujacy) pozwalających na uzyskanie nowych informacji o działaniu aplikacji. Nas interesuje instrumentacja binarna (w odróżnieniu od instrumentacji kodu źródłowego - np. *printf-based debugging* i instrumentacji w ramach kompilacji). Podział technik instrumentacji binarnej bazuje na tym, kiedy proces ten ma miejsce:

- statyczna instrumentacja binarna (static binary instrumentation) - ma miejsce przed rozpoczęciem działania programu - kod obiektowy lub wykonywalny jest wtedy przepisywany
- dynamiczna instrumentacja binarna (dynamic binary instrumentation - **DBI**) - ma miejsce w czasie działania programu. Nas interesuje właśnie to podejście.

DBI - Dynamiczna instrumentacja binarna

Jak wcześniej wspomniano ma miejsce w czasie działania programu. Do aplikacji wstrzykiwany jest dodatkowy kod analizujący (instrukcje monitorujące). Staje się on częścia wykonania programu. Zaleta: nie wymaga, by badany program był w jakikolwiek sposób przygotowany wcześniej. Największą wadą tej metody jest fakt, iż koszt jej wykonania przeniesiony jest całkowicie na czas wykonania - wpływa to w sposób znaczący na wydajność. Ponadto implementacja tego typu metody jest rzeczą trudną.

AGH

Valgrind - wstęp

Valgrind - wstęp

Czym jest Valgrind?

Valgrind - wstęp

Czym jest Valgrind?
Jest to framework do tworzenia narzędzi przeprowadzających binarną analizę dynamiczną dzięki zastosowaniu techniki DBI. Architektura valgrinda:

Valgrind core + tool plug-in = Valgrind tool

Narzędzia (np. Memcheck) są tworzone w języku C jako wtyczki (plug-in) do rdzenia Valgrinda. Zadaniem narzędzia jest instrumentacja fragmentu kodu przekazanego mu przez rdzeń. Tego typu architektura ułatwia tworzenie nowych narzędzi - najtrudniejszą część pracy wykonuje gotowy już rdzeń.

Valgrind - narzędzia

Valgrind posiada standardowy zestaw narzędzi do analizy kodu. Są to m.in.:

- Memcheck - do wykrywania błędów związanych z pamięcią, np. wycieków pamięci
- Cachegrind - symuluje interakcje programu z pamięcią cache
- Callgrind - narzędzie do profilowania pozwalające na analizę grafu wywołań funkcji w programie
- Massif - narzędzie do analizowania sterty (heap)
- Hellgrind i DRD - narzędzia do wykrywania błędów związanych z wątkami. Używają innych technik analizy.
- Inne: DHAT, SGcheck, BBV, Lackey i Nulgrind.

Valgrind - użycie

Podstawowe wywołanie Valgrinda z poziomu linii poleceń wygląda następująco:

valgrind –tool=<tool_name> <program_name>

Domyślnie Valgrind produkuje wiadomości na standardowym wyjściu błędów, ale oczywiście można je przekierować np. do pliku.

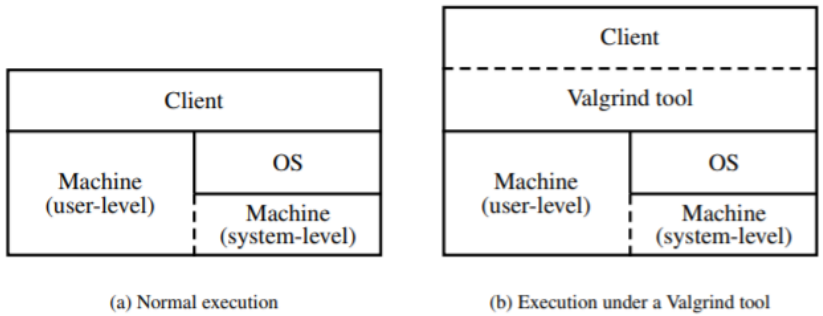
Valgrind - działanie (core)

- Valgrind podłącza się do badanego programu przy uruchomieniu, a następnie (re)kompiluje kod (jeden tzw. podstawowy blok na raz) w systemie just-in-time (JIT).
- Kompilacja polega na deasemblacji kodu maszynowego do tzw. postaci pośredniej (IR), która przekazywana jest do plug-inów (narzędzi typu Memcheck), gdzie poddawana jest instrumentacji.
- Wynik jest następnie ponownie konwertowany do postaci maszynowej. Efekt tej pracy nazywa się **translation** (ang. tłumaczenie/translacja) - co jest troche mylące, bo zarówno początkowy jak i wynikowy kod to kod maszynowy.
- Powyższy proces można w pewnym sensie porównać do działania interpretera.

Valgrind - działanie (core)

W przypadku wykonania programu bez Valgrinda program wykonuje interakcję z maszyną bezpośrednio (np. rejestry ogólnego przeznaczenia) lub za pomocą wywołań systemowych (system calls).

W przypadku korzystania z Valgrinda badany program (client) i narzędzie są częścią jednego procesu. Pełną kontrolę nad tym, co się stanie ma jednak narzędzie - prowadzi ono interakcję z maszyną i systemem.



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

Źródło: `http://valgrind.org/docs/phd2004.pdf`

Valgrind - działanie (core)

Najważniejszym krokiem w działaniu Valgrinda jest tworzenie translacji. Ma to miejsce w następujących krokach:

- deasemblacja - Valgrind reprezentuje kod w tzw. postaci pośredniej o nazwie UCode.
- optymalizacja - poprawa jakości kodu po deasemblacji
- instrumentacja - narzędzie dodaje kod analizujący.
- alokacja rejestrów - rejestry wirtualne są przypisywane do rejestów ogólnego przeznaczenia
- generowanie kodu - instrukcje UCode są konwertowane niezależnie od siebie do kodu maszynowego.

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

Valgrind - narzędzia

Memcheck

Narzędzie służące do wykrywania błędów związanych z zarządzaniem pamięcią. Często mówiąc Valgrind mamy na myśli dokładnie do narzędzie. Służy głównie do badania programów napisanych w C i C++. Wykrywa błędy takie jak:

- wycieki pamięci
- problemy ze stertą (heap) - podwójne zwalnianie pamięci, niedopasowane free/delete
- użycie niezainicjalizowanych wartości
- próba dostępu do pamięci, do której nie powinniśmy się odwoływać (np. zwolnionej lub niezaalokowanej)
- próba przekazania do funkcji typu strcpy() nakładających się obszarów pamięci

Memcheck - użycie

Aby uruchomić Memcheck, wystarczy w linii poleceń wpisać:

valgrind <nazwa_programu>

Nie trzeba (chociaż można) podawać nazwy narzędzia - Memcheck jest tym domyślnym. Memcheck wykrywa błędy w momencie ich wystąpienia - podaje numer linii w pliku źródłowym i tzw. stack trace (zrzut stosu - lista aktywnych ramek stosu). Użycie Memchecka oznacza ok. 10-30 krotnie wolniejsze działanie programu.

Memcheck - błędy na przykładach

- niepoprawne czytanie/zapis do pamięci - Przykład_1_1
- uzycie niezainicjalizowanych wartości - Przykład_1_2 i Przykład_1_3
- niepoprawne zwalnianie pamięci - Przykład_1_4
- użycie niezainicjalizowanych wartości w wywołaniu systemowym - Przykład_1_5
- niedopasowana funkcja zwalniająca pamięć - Przykład_1_6
- nakładające się zakresy w funkcjach typu strcpy() - Przykład_1_7
- niepoprawne wartości definiujące rozmiar - Przykład_1_8
- wycieki pamięci

Memcheck - wycieki pamięci

Memcheck śledzi wszystkie bloki na stercie (heap) zaalokowane za pomocą malloc/new. Dzięki temu kiedy działanie programu kończy się, Memcheck jest w stanie powiedzieć które z tych bloków nie zostały zwolnione. Memcheck wyróżnia 4 rodzaje wycieków pamięci:

- definitely lost - nie ma wskaźnika pokazującego na zaalokowany blok pamięci
- indirectly lost - oznacza, że wskaźnik pokazujący na blok nie został zgubiony, natomiast wskaźnik gwarantujący dostęp do tego wskaźnika nie istnieje. Przykład: jeśli mamy drzewo i stracimy dostęp do roota.
- possibly lost - łańcuch jednego lub więcej wskaźników do zaalokowanego bloku został znaleziony, ale przynajmniej jeden z nich nie wskazuje na początek bloku
- still reachable - istnieje wskaźnik pokazujący na początek bloku, ale blok nie został jeszcze zwolniony

Przykład_1_9

Memcheck - przydatne opcje linii poleceń

- –leak-check=<no|summary|yes|full> [default: summary] - ustawia szczegółowość raportu o wyciekach - summary pokazuje jedynie podsumowanie, yes i full pokazują każdy wyciek
- –show-leak-kinds=<set> [default: definiete, possible] - wybór rodzajów, jakie pokazać szczegółowo przy leak-check=full. Możliwe wartości: definite, indirect, possible, reachable, all oraz none.
- –track-origins=<yes|no> [default: no] - decyduje, czy Memcheck śledzi pochodzenie niezainicjalizowanych wartości
- Więcej: `http://valgrind.org/docs/manual/mc-manual.html#mc-manual.options`

Memcheck - redefinicja operatorów alokujących

Memcheck w celu przeprowadzenia analizy pamięci zamienia standardowe funkcje zarządzające pamięcią w C i C++, takie jak malloc czy globalne operatory z rodziny new/delete. Robi to, aby móc sledzić alokacje na stercie. Zamiana globalnych operatorów z rodziny new/delete sprawia, że program uruchomiony za pomocą Valgrinda nie używa przeładowań zdefiniowanych przez użytkownika - może to mieć wpływ na sposób działania programu jeśli operatory te zawierają jakąś istotną logikę.
Przykład_2_1, Przykład_2_2 i Przykład_2_3

Memcheck - makra (Client Requests)

Valgrind posiada mechanizm pozwalający badanemu programowi (client) na przekazywanie prostych zapytań do rdzenia i aktualnie używanego narzędzia.
Pewien podzbiór tych tzw. client requests jest dostępny do użycia z poziomu kodu źródłowego programu w postaci makr - pozwala pozyskać dodatkowe informacje do debugowania lub nawet przekazać pewne informacje Valgrindowi.

Memcheck - makra (Client Requests)

W celu użycia makr należy dołączyć odpowiedni plik nagłówkowy:

- "valgrind/valgrind.h" dla makr, którymi zajmuje się rdzeń Valgrinda
- "valgrind/memcheck.h" dla makr, którymi zajmuje się narzędzie, w tym przypadku Memcheck

Makra zawarte w tych plikach generują kod inline. Kod ten jest widoczny dla Valgrinda, ale nie robi nic, jeśli program jest uruchamiany poza nim - nie jesteśmy więc zmuszeni do uruchamiania programu tylko za pomocą Valgrinda.

Memcheck - makra (Client Requests)

Proste, użyteczne makra rdzenia:

- RUNNING_ON_VALGRIND
- VALGRIND_COUNT_ERRORS
- VALGRIND_PRINTF
- VALGRIND_PRINTF_BACKTRACE

Przykład_3_1 i Przykład_3_2

Memcheck - makra (Client Requests)

Dla Memchecka:

- VALGRIND_CHECK_VALUE_IS_DEFINED
- VALGRIND_DO_LEAK_CHECK
- VALGRIND_CHECK_MEM_IS_ADDRESSABLE
- VALGRIND_CHECK_MEM_IS_DEFINED
- VALGRIND_MAKE_MEM_DEFINED i podobne

Więcej: `http://valgrind.org/docs/manual/mc-manual.html#mc-manual.clientreqs`
Przykład_3_3 i Przykład_3_4

Memcheck - filtrowanie błędów

Memcheck oprócz kodu aplikacji sprawdza również kod bibliotek, z których ona korzysta. Sprawia to, że w naszej aplikacji moga pojawić się błędy, których nie potrafimy naprawić - gdyż pochodzą one z cudzego kodu. Valgrind posiada sposób na poradzenie sobie z tym problemem - są to tzw. **suppression files**.

Memcheck - filtrowanie błędów

Filtry umieszcza się w opowiednim pliku z rozszerzeniem .supp.
Struktura pojedynczego filtru jest następujaca:
{
nazwa
tool:filtr
dodatkowe info dla niektórych filtrów
kontekst
}

Memcheck - filtrowanie błędów

Przydatne opcje:

- valgrind –gen-supressions=yes ./a.out - wyświetla informacje o błędach w sposób pozwalający łatwo tworzyć filtry (w formacie takim jak wyżej)
- valgrind -v –suppressions=<plikZFiltrami.supp> ./a.out - uruchamia Valgrinda z filtrami zdefiniowanymi w pliku <plikZFiltrami.supp>

Przykład_4_1 - proces tworzenia filtrów

Cachegrind - wstęp

Cachegrind jest drugim omawianym narzędziem Valgrinda. Przeprowadza on symulację działania pamięci cache (pamięć podręczna procesora) w czasie działania programu. Cachegrind służy do tzw. **profilowania** programu - po angielsku narzędzie takie to *profiler*.

Cachegrind - pojęcie profilowania

Profilowanie jest formą dynamicznej analizy programu. Polega ono na badaniu zachowania programu używając informacji zdobytych podczas jego wykonania. Informacje te mogą dotyczyć zużycia pamięci czy wywoływania funkcji. Profilowanie przeprowadza się, by dowiedzieć się, które części programu można zoptymalizować.

Cachegrind - kilka faktów o cache

- mówimy o pamięci podręcznej procesora (CPU cache)
- krótki czas dostępu - czyli bardzo szybka
- jest częścią procesora
- pamięć statyczna (Static RAM) - może przechowywać dane bez konieczności odświeżania
- podzielona na poziomy: L1, L2 ...
- wraz z poziomem rośnie wielkość pamięci, ale maleje szybkość
- podział na cache z instrukcjami i z danymi

Cachegrind - kilka faktów o cache 2

Kiedy procesor szuka w pamięci cache danych do przeprowadzenia operacji może nastąpić jedna z dwóch sytuacji

- cache hit (trafienie) - dane zostają znalezione w pamięci podręcznej
- cache miss (chybienie) - danych nie ma w pamięci podręcznej, trzeba je pobrać z poziomu cache o wyższym numerze lub z RAM. Jest to sytuacja niekorzystna z punktu widzenia wydajności - sięganie do pamięci RAM zajmuje dużo instrukcji procesora.

Profilowanie za pomocą narzędzia Cachegrind będziemy prowadzić z uwagi właśnie na te dwa zjawiska.

Cachegrind - działanie

Cachegrind jest narzędziem przeprowadzającym symulację działania pamięci cache w czasie wykonania programu. Symulacja zakłada następujące parametry:

- Dwa poziomy cache: L1 i LL - jeśli komputer posiada większą ilość poziomów cache niż 2, Cachegrind symuluje pierwszy i ostatni poziom
- Cache na poziomie L1 podzielone na I1 - cache do instrukcji i D1 - cache na dane
- Śledzenie statystyk dla każdej linii kodu źródłowego.

Cachegrind - działanie

Po zakończeniu działania program wypisuje podsumowanie i tworzy plik:

cachegrind.out.<pid>

gdzie <pid> to ID procesu.
Aby używać Cachegrinda, kompilujemy program z flagą -g (informacje debugowe). Zwykle chcemy włączyć optymalizację - profilujemy program tak, jak będzie faktycznie wykonywany.
Mając plik out, używamy programu cg_annotate, aby uzyskać szczegółową prezentację zebranych wyników.
Przykład_5_1 i Przykład_5_2

Cachegrind - przydatne opcje

- –cachegrind-out-file=<file> - plik wyjściowy
- –I1/D1/LL=<size> - wielkość poszczególnych poziomów cache w symulacji
- –auto=<no|yes> [default: no] - dla cg_annotate

Massif - wstęp

Ostatnie z omawianych narzędzi. Służy ono do badania (profilowania) sterty (heap). Massif mierzy, jak dużo pamięci sterty używa program. Może również służyć do badania stosu, aczkolwiek nie będzie to omawiane na prezentacji. Massif pozwala zaobserwować ilość zaalokowanej pamięci, a dzięki temu - zoptymalizować je.

Massif - użycie

- kompilacja z opcją -g (informacje debugowe)
- opcje związane z optymalizacją nie mają dużego znaczenia
- uruchomienie: valgrind –tool=massif program
- produkuje podsumowanie i plik wyjściowy w formacie massif.out.<pid> - analogicznie do Cachegrind
- ms_print - przeglądanie zebranych informacji w czytelnej formie

Massif - przydatne opcje linii poleceń

- –time-unit=<i|ms|B> [default: i] - jednostka czasu użyta do profilowania. Możliwości: wykonane instrukcje (i), czas rzeczywisty (ms), ilość zaalokowanych/dealokowanych bajtów (B)
- –heap=<yes|no> [default: yes] - włączenie/wyłączenie profilowania sterty

Przykład_6

Źródła

- Valgrind User Manual: `http://valgrind.org/docs/manual/manual.html`
- Praca: `http://valgrind.org/docs/valgrind2007.pdf`
- Praca: `http://valgrind.org/docs/shadow-memory2007.pdf`
- Praca: `http://valgrind.org/docs/phd2004.pdf`
- Praca: `http://valgrind.org/docs/memcheck2005.pdf`
- Praca: `https://disconnect3d.pl/assets/about_me/disconnected_bachelor_thesis.pdf`

Źródła

- Książka: *Praktyczna inżynieria wsteczna* pod redakcją: Gynvael Coldwind, Mateusz Jurczyk: `https://ksiegarnia.pwn.pl/Praktyczna-inzynieria-wsteczna,622427233,p.html`
- Strona: `https://courses.cs.washington.edu/courses/cse326/05wi/valgrind-doc/coregrind_tools.html`
- Strona: `https://accu.org/index.php/articles/1905`
- Strona: `http://www.mateuszmidor.com/`

Szablon prezentacji

http://home.agh.edu.pl/~mszpyrka/doku.php?id=lectures:latex:aghdpl

Dziękuję za uwagę