**Tổng quan về kiến trúc Valgrind**

**Cách sử dụng Valgrind**



**Nội Dung**

[I. Giới thiệu 4](#_Toc185567978)

[Valgrind là gì 4](#_Toc185567979)

[Một số điểm mạnh và yếu của Valgrind so với các tool debug khác 4](#_Toc185567980)

[II. Kiến trúc của Valgrind 6](#_Toc185567981)

[Valgrind Core 7](#_Toc185567982)

[Tool Plugins 10](#_Toc185567983)

[Một số thành phần trong Simulated CPU 12](#_Toc185567984)

[III. Cách sử dụng Valgrind 13](#_Toc185567985)

[Cài đặt 13](#_Toc185567986)

[Các bước tực hiện debug bằng Valgrind 13](#_Toc185567987)

[Một số tool khác của Valgrind 18](#_Toc185567988)

[IV. Valgrind GDB - VGDB 19](#_Toc185567989)

[Giới thiệu 19](#_Toc185567990)

[Các bước thực hiện debug 20](#_Toc185567991)

[V. So sánh Valgrind và Anddressanitizer 24](#_Toc185567992)

[VI. Tổng kết 26](#_Toc185567993)

# Giới thiệu

## Valgrind là gì

Valgrind, ra mắt lần đầu vào năm 2002, là một bộ công cụ mạnh mẽ được phát triển nhằm phân tích và gỡ lỗi các chương trình trên hệ điều hành Linux. Ban đầu, nó được thiết kế để phát hiện các lỗi liên quan đến quản lý bộ nhớ, nhưng sau đó đã được mở rộng để hỗ trợ tối ưu hóa hiệu năng và phân tích chi tiết hơn. Valgrind cung cấp các chức năng chính như phát hiện lỗi bộ nhớ (truy cập vùng nhớ không hợp lệ, rò rỉ bộ nhớ, và sử dụng bộ nhớ chưa được khởi tạo), tối ưu hóa hiệu năng bằng cách phân tích các điểm nghẽn trong mã nguồn, và mô phỏng CPU để kiểm tra các lỗi tiềm ẩn trên các kiến trúc phần cứng khác nhau. Công cụ này giúp các nhà phát triển cải thiện chất lượng phần mềm và tăng cường hiệu năng ứng dụng.

## Một số điểm mạnh và yếu của Valgrind so với các tool debug khác

Valgrind không yêu cầu bạn sửa đổi mã nguồn hoặc biên dịch lại với các tùy chọn đặc biệt. Chương trình có thể chạy trực tiếp thông qua Valgrind hay nói cách khác là bạn có thể sử dụng một file .bin để đưa vào Valgrind chạy.

**Đa dạng công cụ phân tích:**

* Ngoài Memcheck, Valgrind còn cung cấp các công cụ hữu ích khác:
  + **Cachegrind**: Phân tích hiệu năng liên quan đến bộ nhớ đệm (cache).
  + **Massif**: Phân tích việc sử dụng bộ nhớ heap.
  + **Helgrind**: Phát hiện các lỗi trong lập trình đa luồng (thread race conditions).
  + **Callgrind**: Tối ưu hóa chương trình bằng cách phân tích luồng thực thi (function calls).

Bên cạnh nhưng điểm mạnh của Valgrind thì vẫn còn tồn tại nhiều điểm yếu so với các tool khác như:

1. **Hiệu năng thấp**:

* Valgrind sử dụng kỹ thuật giả lập để kiểm tra chương trình, dẫn đến việc chương trình chạy chậm hơn rất nhiều (có thể giảm tốc độ từ 10 đến 50 lần so với tốc độ thực).
* Điều này khiến Valgrind không phù hợp với các ứng dụng yêu cầu kiểm thử thời gian thực hoặc các chương trình có thời gian chạy dài.

1. **Không hỗ trợ trên Windows**:

* Valgrind chỉ hoạt động tốt trên Linux và macOS, trong khi Windows chỉ có các bản port không chính thức (như Dr. Memory). Điều này hạn chế khả năng sử dụng đối với các nhà phát triển làm việc trên hệ điều hành Windows.

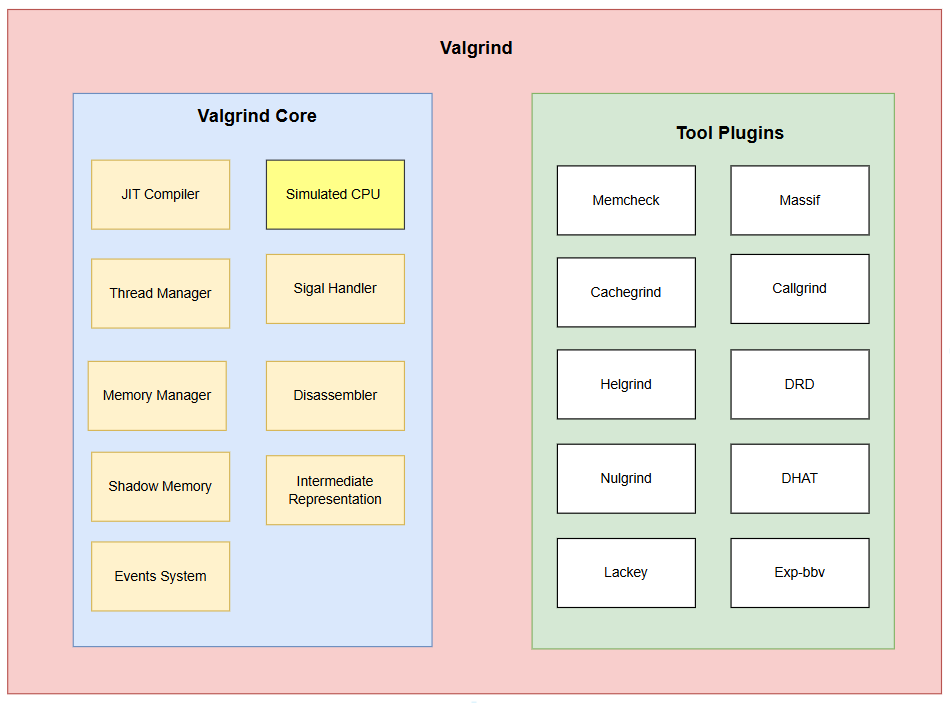
1. **Hạn chế trong việc phát hiện lỗi cụ thể**:

* So với AddressSanitizer, Valgrind có hạn chế trong việc phát hiện một số lỗi cụ thể như:
  + **Buffer overflow** hoặc **stack overflow** chi tiết như AddressSanitizer.
  + **Use-after-free** nhanh chóng và rõ ràng như ASan.
* Valgrind thiếu một số công cụ mạnh mẽ cho các lỗi đồng bộ hóa đa luồng so với ThreadSanitizer.

1. **Tiêu thụ tài nguyên lớn**:

* Valgrind tiêu thụ nhiều tài nguyên CPU và RAM khi chạy, khiến nó không phù hợp với các hệ thống có tài nguyên hạn chế hoặc các bài kiểm thử quy mô lớn.

# Kiến trúc của Valgrind



Kiến trúc Valgrind được chia thành hai thành phần chính:

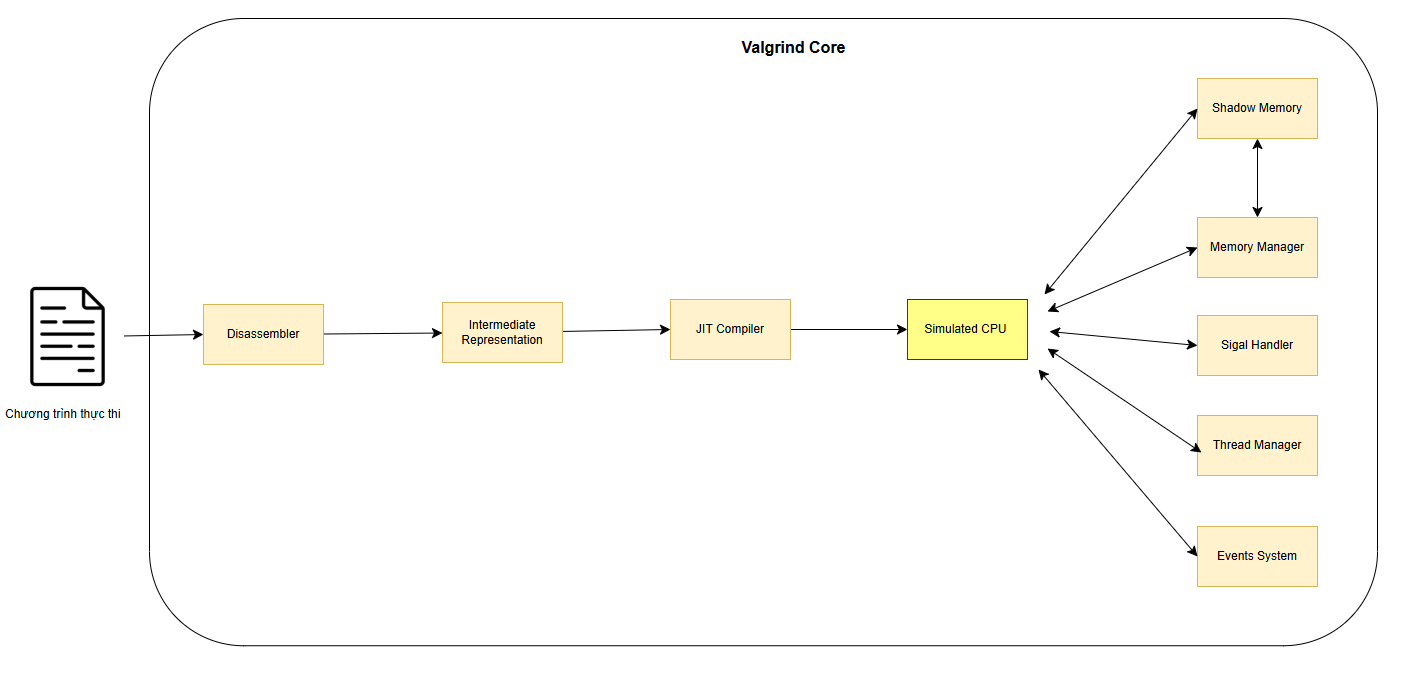
* **Valgrind Core**: Lõi hệ thống chịu trách nhiệm dịch và thực thi mã máy trong môi trường giả lập.
* **Tool Plugins**: Các plugin mở rộng được sử dụng để thực hiện các nhiệm vụ cụ thể như kiểm tra bộ nhớ, phân tích hiệu suất, hoặc phát hiện lỗi đồng thời.

## Valgrind Core

Valgrind Core cung cấp nền tảng cho việc dịch mã, kiểm tra lỗi, và thực hiện các phân tích. Các thành phần chính gồm:

* **JIT Compiler**: JIT Compiler dịch mã từ chương trình (application) sang mã nhị phân, đồng thời thêm các đoạn mã phân tích (instrumentation code).
* **Simulated CPU**: Đây là CPU ảo do Valgrind tạo ra để mô phỏng quá trình thực thi mã.
* **Disassembler**: Phân tích và dịch ngược mã máy từ chương trình thành dạng trung gian (IR) để dễ dàng kiểm tra và tối ưu hóa.
* **Intermediate Representation (IR)**: Biểu diễn trung gian của mã, cung cấp một dạng mã đơn giản và nhất quán để thực hiện các bước tối ưu hóa và kiểm tra.
* **Shadow Memory**: Một cơ chế theo dõi trạng thái bộ nhớ (như đã được khởi tạo hay chưa) để phát hiện lỗi truy cập bộ nhớ.
* **Memory Manager**: uản lý việc cấp phát và giải phóng bộ nhớ trong cả Valgrind và chương trình.
* **Thread Manager**: Quản lý các luồng xử lý (threads) của chương trình.
* **Signal Handler:** Quản lý các tín hiệu (signals) được gửi tới chương trình hoặc CPU giả lập.
* **Event System**: Ghi nhận và xử lý các sự kiện trong chương trình như gọi hàm, truy cập bộ nhớ, hoặc thực thi tín hiệu.

Luồng hoạt động của các thành phần trong Valgrind Core:



*Luồng hoạt động của các thành phần trong Valgrind*

Mô tả:

**Disassembler**:

* Khi một chương trình bắt đầu chạy trên Valgrind, mã máy của chương trình được đưa vào Disassembler.
* Disassembler phân tích và dịch ngược mã máy thành một dạng trung gian (Intermediate Representation - IR). Dạng IR này đơn giản hơn và dễ xử lý hơn so với mã máy gốc.

**Intermediate Representation (IR)**:

* Dạng IR sau đó được sử dụng làm đầu vào cho các bước tiếp theo.
* Các bước tối ưu hóa và phân tích được thực hiện trên IR để loại bỏ mã thừa hoặc tối ưu hóa hiệu suất.

**JIT Compiler (Just-In-Time Compiler)**:

* Sau khi tối ưu hóa, IR được JIT Compiler biên dịch lại thành mã nhị phân.
* Trong quá trình này, JIT Compiler cũng thêm các đoạn mã phân tích (instrumentation code) vào chương trình để theo dõi các hoạt động như truy cập bộ nhớ hoặc gọi hàm.

**Simulated CPU**:

* Simulated CPU là thành phần cốt lõi của Valgrind, chịu trách nhiệm thực thi mã nhị phân đã được biên dịch từ JIT Compiler.
* Nó giả lập quá trình thực thi của CPU thực, giúp cách ly chương trình với CPU thật, đồng thời thu thập thông tin phân tích từ các đoạn mã đã được gắn thêm.

**Shadow Memory**:

* Trong khi Simulated CPU thực thi mã, Shadow Memory theo dõi trạng thái của bộ nhớ.
* Nó kiểm tra xem các ô nhớ đã được khởi tạo hay chưa, phát hiện các lỗi như truy cập bộ nhớ chưa khởi tạo hoặc tràn bộ nhớ.

**Memory Manager**:

* Memory Manager quản lý việc cấp phát và giải phóng bộ nhớ của cả Valgrind và chương trình đang chạy.
* Nó tương tác với Shadow Memory để đảm bảo rằng bộ nhớ được sử dụng hợp lý, đồng thời phát hiện lỗi liên quan đến cấp phát và giải phóng bộ nhớ.

**Thread Manager**:

* Nếu chương trình sử dụng đa luồng, Thread Manager quản lý các luồng (threads) và đảm bảo rằng chúng được thực thi một cách an toàn trong môi trường giả lập.
* Nó cũng giúp phát hiện các lỗi như deadlocks hoặc race conditions giữa các luồng.

**Signal Handler**:

* Khi chương trình nhận các tín hiệu (signals) từ hệ thống hoặc bên ngoài, Signal Handler xử lý chúng.
* Nó đảm bảo rằng tín hiệu được xử lý chính xác, đồng thời cung cấp thông tin phân tích về cách chương trình phản hồi các tín hiệu.

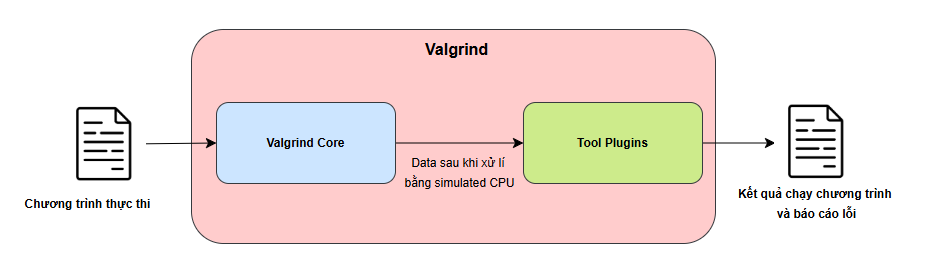
**Event System**:

* Trong quá trình thực thi, Event System ghi nhận và xử lý các sự kiện quan trọng, chẳng hạn như gọi hàm, truy cập bộ nhớ hoặc thực thi tín hiệu.
* Các sự kiện này được ghi lại và sử dụng để tạo báo cáo chi tiết về hành vi của chương trình.

## Tool Plugins

**Valgrind Tool Plugins** cung cấp các công cụ phân tích và kiểm tra khác nhau cho chương trình, mở rộng chức năng của Valgrind Core. Các công cụ chính gồm:

* **Memcheck**: Theo dõi việc sử dụng bộ nhớ trong chương trình, phát hiện các lỗi như truy cập bộ nhớ chưa khởi tạo, truy cập vượt quá giới hạn bộ nhớ được cấp phát, và rò rỉ bộ nhớ.
* **Massif**: Phân tích việc sử dụng bộ nhớ heap, cung cấp thông tin chi tiết về cách chương trình sử dụng bộ nhớ, giúp tối ưu hóa việc quản lý và cấp phát bộ nhớ.
* **Cachegrind**: Phân tích hiệu suất của bộ nhớ đệm (cache), mô phỏng các lần trượt bộ nhớ đệm (cache misses) và cung cấp thông tin để cải thiện hiệu suất truy cập bộ nhớ của chương trình.
* **Callgrind**: Theo dõi các lời gọi hàm trong chương trình, cung cấp đồ thị lời gọi (call graph) chi tiết, và phân tích hiệu suất dựa trên các hàm được gọi.
* **Helgrind**: Kiểm tra các vấn đề liên quan đến multithreading, phát hiện các lỗi như deadlock và race condition.
* **DRD**: Tương tự như Helgrind nhưng tập trung vào hiệu suất, phát hiện các lỗi đa luồng cơ bản với tốc độ nhanh hơn, nhưng cung cấp ít thông tin chi tiết hơn.
* **DHAT (Dynamic Heap Analysis Tool)**: Phân tích cách chương trình sử dụng bộ nhớ heap, đo lường thời gian sống, số lần truy cập và các mẫu truy cập bộ nhớ để giúp tối ưu hóa bộ nhớ.
* **Nulgrind**: Công cụ đơn giản nhất, không thực hiện phân tích, chủ yếu dùng để kiểm tra cơ bản khả năng hoạt động của Valgrind.
* **Lackey**: Một công cụ mẫu được thiết kế để minh hoạ cách xây dựng một plugin cho Valgrind, thường được dùng để thử nghiệm hoặc học cách tạo công cụ mới.
* **Exp-bbv (Experimental Basic Block Vector)**: Công cụ thử nghiệm, cung cấp phân tích cơ bản dựa trên các khối mã cơ bản (basic blocks)..



*Tổng quan sự liên kết giữa các thành phần trong Valgrind*

**Mô tả:**

Sau khi chương trình thực thi được xử lí thông qua Valgrind Core, data sẽ được đưa qua Tool Plugins để tiến hành phân tích lỗi, hiệu suất. Sau đó sẽ trả về cho người dùng một report đầy đủ thông tin lỗi, hiệu suất, kết quả chạy chương trình.

## Một số thành phần trong Simulated CPU

|  |  |
| --- | --- |
| **Thành phần** | **Mô tả** |
| **Thanh ghi giả lập** | %EAX, %EBX, %ECX, %EDX, %ESP, %EBP, %EFLAGS, v.v. |
| **Cờ giả lập** | %EFLAGS: Quản lý trạng thái như Zero, Sign, Carry. |
| **Bộ xử lý FPU** | Lưu trạng thái trong VG\_(baseBlock), thao tác qua frstor/fnsave. |
| **Thanh ghi tạm (TempRegs)** | Lưu giá trị trung gian khi thực thi UCode. |
| **Bộ nhớ phụ trợ (Spill)** | 24 ô nhớ trong VG\_(baseBlock) để hỗ trợ khi thiếu thanh ghi thực. |
| **Cache dịch (TC)** | Lưu mã đã dịch để tái sử dụng, giảm chi phí dịch lại. |

# Cách sử dụng Valgrind

## Cài đặt

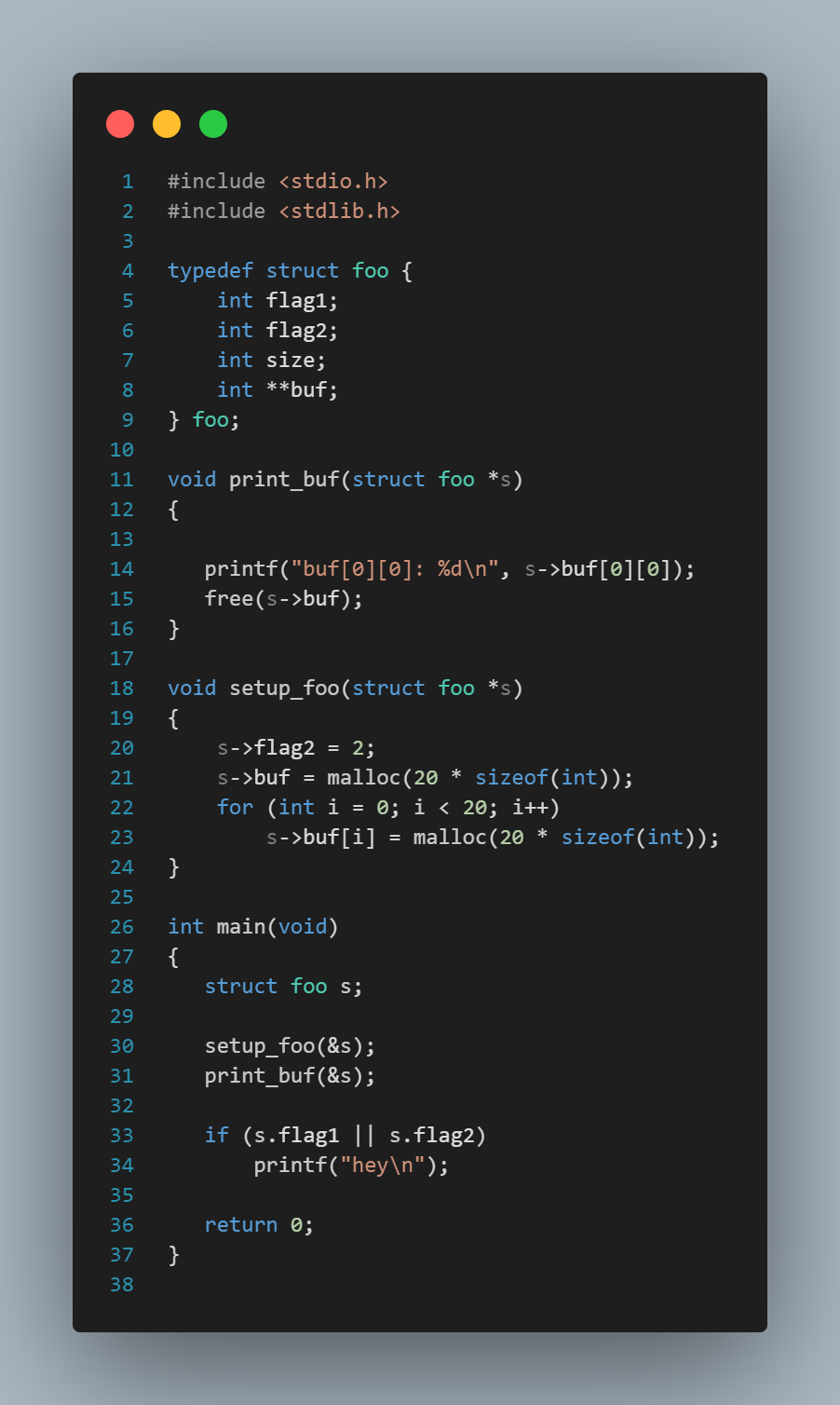
sudo apt update

sudo apt install valgrind

valgrind –version

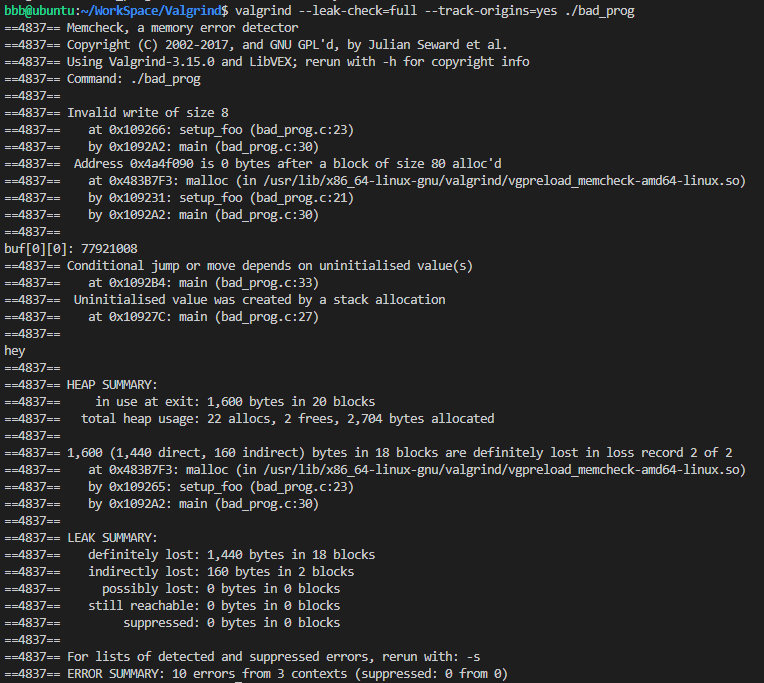
## Các bước tực hiện debug bằng Valgrind

Tạo 1 file **bad\_prog.c** để lưu code



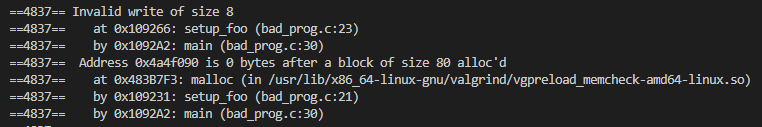
*Chương trình lỗi*

* Biên dịch file bằng tùy chọn debug: **gcc -g -o bad\_prog bad\_prog.c**
* **Khi chạy file này chúng ta sẽ thấy kết quả hiện thị bình thường và không có lỗi gì được trả về.**
* Kiểm tra bằng Valgrind: **valgrind --leak-check=full --track-origins=yes ./bad\_prog**
  + --leak-check=full: Kiểm tra đầy đủ các rò rỉ bộ nhớ.
  + --track-origins=yes: Hiển thị nguồn gốc của các giá trị không hợp lệ.
  + ./bad\_prog: Chạy chương trình thực thi.
* Phân tích kết quả:



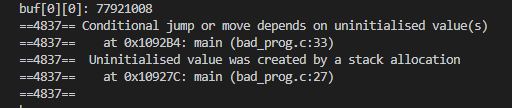
*Kết quả sau khi chạy Valgrind*

**4837** là **Process ID (PID)** của tiến trình bạn đang chạy với Valgrind



*Phân tích lỗi*

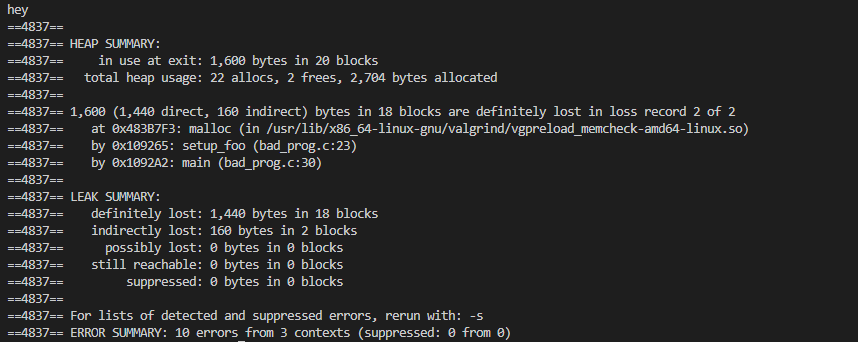
Đoạn report trên cho thấy, lỗi xảy ra khi cố ghi 8 byte vào một vùng nhớ **không hợp lệ** hoặc **nằm ngoài vùng nhớ được cấp phát**. Địa chỉ 0x4a4f090 (nơi lỗi xảy ra) nằm **ngay sau** một vùng nhớ kích thước 80 byte đã được cấp phát. s->buf thực sự là một **mảng con trỏ** (int \*\*), do đó chúng ta cần cấp phát bộ nhớ tương ứng với kích thước của các con trỏ (sizeof(int\*)), thay vì sizeof(int). **s->buf[i] = malloc(20 \* sizeof(int));** Lệnh này cố gắng ghi địa chỉ (8 byte trên hệ thống 64-bit) vào s->buf[i], nhưng bộ nhớ được cấp phát không đủ để lưu trữ 20 con trỏ. Và để nhận ra điều đó, công cụ Valgrind sử dụng là **Memcheck**

****

*Phân tích lỗi*

**Buf[0][0] là dòng in trong code**

**“Conditional jump or move depends on uninitialised value(s)”** có nghĩa là chương trình đang cố gắng thực hiện một hành động điều kiện (như nhảy tới nhánh if) dựa trên một giá trị chưa được khởi tạo (s.flag1). Lỗi này xuất hiện tại dòng 33. Sau đấy Valgrind cho biết lỗi này xuất phát từ **Stack Allocation** tại dòng 27. Đây là nơi **struct foo s;** được khai báo trên stack, Vì flag1 không được gán giá trị sau khi khai báo, nó dẫn đến lỗi này.



*Phân tích lỗi*

Tiếp đến là **Heap Summary**: 1600 bytes là dung lượng bộ nhớ được cấp phát nhưng không được giải phóng. 20 blocks tương ứng với 20 malloc được gọi. Và cuối từng là tổng heap đã sử dụng bao gồm 22 lần cấp phát, 2 lần free và 2704 byte đã được cấp phát.

Heap Summary còn trả về trong 1600 bytes không được giải phóng sẽ bao gồm 1440 bytes là từ việc cấp phát trực tiếp và 160 bytes cấp phát gián tiếp trong 18 block liên quan đến rò rỉ bộ nhớ. **s->buf = malloc(20 \* sizeof(int));** Valgrind đánh dấu đây là gián tiếp (indirect) vì các con trỏ trong mảng này trỏ đến các vùng nhớ khác với kiểu dữ liệu con trỏ trong hệ thông 64bit là 8bytes => **indirect lost = 20x8 = 160 bytes**. Vì khi cấp phát s**->buf** hệ thống xác định lỗi và chuyển thành kiểu con trỏ tương ứng với 1**60 bytes** nên khi chạy vòng lặp để cấp phát mảng con trỏ đã mặc định 2 blocks đầu tương ứng **(80bytes / 1 block)** đã được cấp phát nên chỉ có 18block tiếp theo sẽ cấp phát theo  **s->buf[i] = malloc(20 \* sizeof(int))** tương ứng **1440bytes**. Và Valgrind sẽ gán nó là direct.

Tiếp theo Valgrind xác định lỗi lại cấp phát bằng hàm malloc bằng memcheck. Cụ thể tại dòng 23

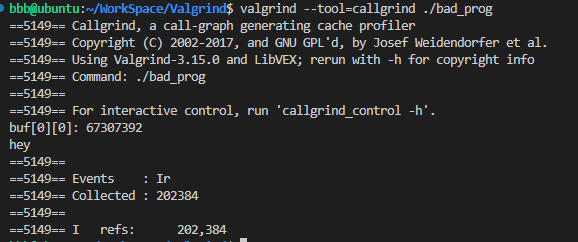
Sau đấu, Valgrind trả về LeakSummarry với ý nghĩa:

* **1,440 bytes** là tổng kích thước bộ nhớ bị rò rỉ.
* **18 blocks** là số vùng nhớ bị rò rỉ, mỗi block tương ứng với một lần gọi malloc.
* **160 bytes** là tổng kích thước bộ nhớ bị rò rỉ gián tiếp.
* **2 blocks** là số vùng nhớ bị rò rỉ gián tiếp.
* **possibly lost: 0 bytes in 0 blocks**: Không có vùng nhớ nào có thể bị rò rỉ nhưng không xác định rõ ràng.
* **still reachable: 0 bytes in 0 blocks**: Thông thường, still reachable xuất hiện khi các vùng nhớ được cấp phát nhưng không cần thiết phải giải phóng vì chúng vẫn có con trỏ trỏ đến (thường gặp trong thư viện hoặc bộ nhớ tạm).
* **suppressed: 0 bytes in 0 blocks:** Không có lỗi nào bị ẩn (suppressed) trong báo cáo này.
* **ERROR SUMMARY: 10 errors from 3 contexts (suppressed: 0 from 0) :** Tổng cộng 10 lỗi đã được phát hiện trong toàn bộ quá trình chạy chương trình. Các lỗi này thuộc về 3 bối cảnh (contexts) khác nhau trong mã nguồn. setup\_foo, print\_buf, hoặc main

Bằng cách sử dụng Memcheck thì Valgrind đã giúp người dùng xác định rõ ràng lỗi hơn.

## Một số tool khác của Valgrind

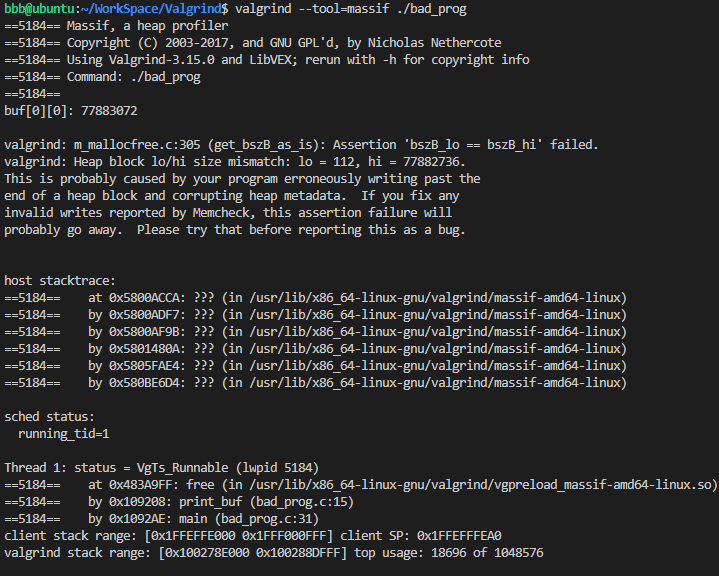
* Callgrind: **valgrind --tool=callgrind ./bad\_prog**



*Phân tích lỗi*

202,384 cho biết chương trình đã thực thi 202,384 lệnh CPU.

* **Massif: valgrind --tool=massif ./bad\_prog**



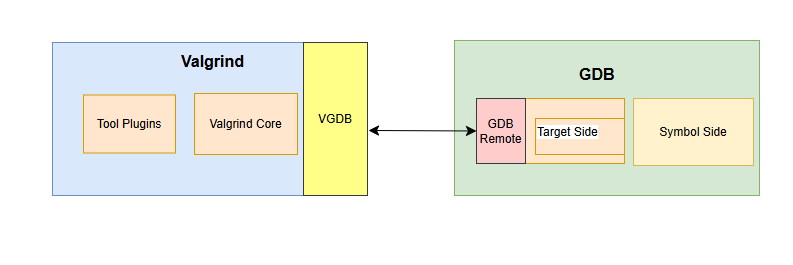
*Phân tích lỗi*

Báo cáo từ Massif chỉ ra rằng chương trình gặp lỗi nghiêm trọng do ghi vượt quá phạm vi bộ nhớ được cấp phát, dẫn đến việc hỏng metadata của heap. Cụ thể, block bộ nhớ được cấp phát với kích thước 112 bytes nhưng bị ghi đè ngoài phạm vi, khiến kích thước metadata hiển thị không khớp (112 vs 77882736). Lỗi xảy ra tại dòng 15 của hàm print\_buf khi gọi free(s->buf) và xuất hiện do s->buf không được cấp phát đủ kích thước trong hàm setup\_foo. Để khắc phục, cần đảm bảo kích thước cấp phát chính xác và giải phóng đúng cách từng block bộ nhớ.

# Valgrind GDB - VGDB

## Giới thiệu

**VGDB** là một công cụ tích hợp trong Valgrind, được thiết kế để hỗ trợ gỡ lỗi thông qua việc kết nối với GDB (GNU Debugger). Nó cho phép người dùng phân tích chi tiết trạng thái của chương trình đang chạy trong môi trường mô phỏng của Valgrind, giúp phát hiện lỗi logic, lỗi bộ nhớ hoặc các vấn đề liên quan đến hiệu năng. VGDB hoạt động như một cầu nối giữa Valgrind và GDB, cho phép kiểm tra từng bước chương trình, truy vấn các trạng thái của bộ nhớ, và kiểm tra các biến trong quá trình thực thi. Công cụ này rất hữu ích khi cần gỡ lỗi các lỗi phức tạp trong các ứng dụng lớn.

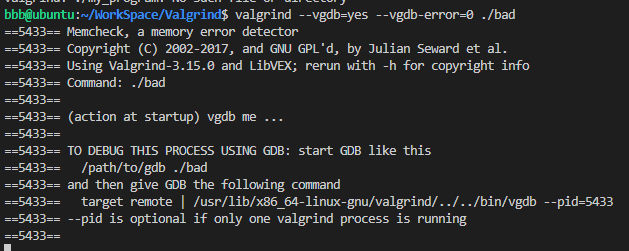


Tổng quan kiến trúc

VGDB sẽ kết nối đến GDB bằng VGDB thông qua phương thức GDB Remote. Các lệnh thao tác từ GDB sẽ được Valgrind xử lí và thực thi trong Simulated CPU.

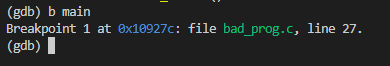
## Các bước thực hiện debug

* Biên dịch chương trình với tùy chọn debug**: gcc -g -o bad bad\_prog.c**
* Khởi động chương trình với VGDB:  **valgrind --vgdb=yes --vgdb-error=0 ./bad**
  + vgdb-error = 0 là Valgrind sẽ dừng ngay sau khi khởi chạy chương trình, trước khi bất kỳ lệnh nào của chương trình được thực thi.

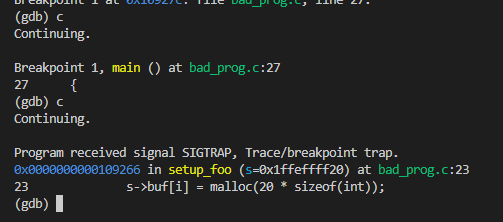
****

Valgrind sẽ hiển thị hướng dẫn các command bên GDB

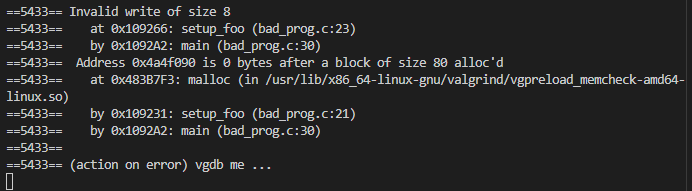
* Mở terminal khác, chạy **gdb ./bad**
* Remote đến VGDB: target remote | vgdb --pid=5433
* Thực hiện các lệnh GDB bình thường như đặt breakpoint**: b main**



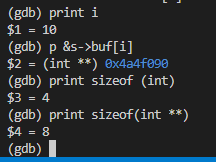
* Sau đó chạy: **continue** để tiếp tục chương trình chạy, chương trình sẽ dùng tại breakpoint main. Tiếp tục chạy **continue** đến khi gặp lỗi do Valgind phát hiện.

  
  
Tín SIGTRAP thông báo cho chúng ta biết có điều gì đó không ổn. Valgrind mô phỏng "lệnh bẫy" bằng cách tạo ra tín hiệu SIGTRAP. SIGTRAP này chỉ được Valgrind gửi khi được gắn vào GDB, do đó GDB có thể chặn tín hiệu. Nếu không, Valgrind sẽ chỉ in ra lỗi.

* Khi này, chúng ta sẽ thấy bên terminal Valgrind xuất hiện thôn tin lỗi.

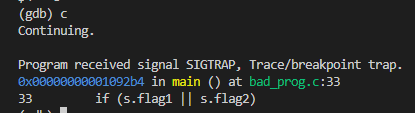


* Để tìm ra lỗi, chúng ta quay lại terminal GDB.

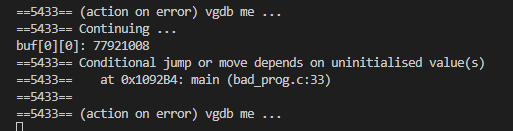


Ở đây chúng ta có thể thấy rằng địa chỉ của &s->buf[10] là 0x4a4f090. Lỗi này đã được phân tích ở trên.

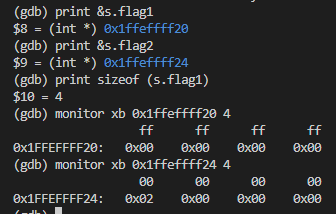
* Tiếp theo chạy continue để tiếp tục chương trình.



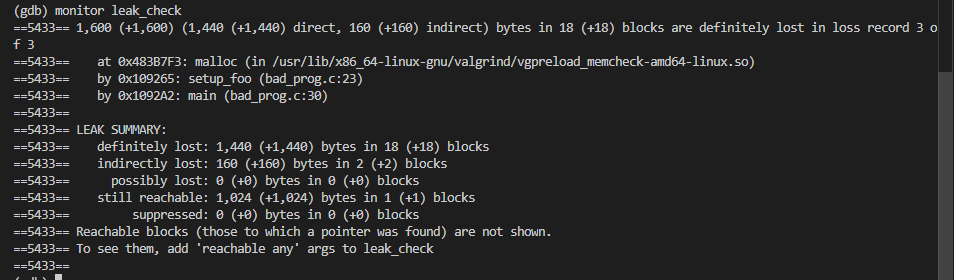
* Tiếp tục chúng ta sẽ gặp SIG TRAP. Bên giao diện Valgrind sẽ hiển thị thông tin lỗi

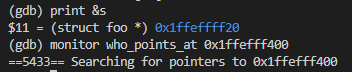


* Valgrind có hỗ trợ người dùng lệnh giám sát Monitor. Để biết thêm thông tin về lệnh, bên giao diện GDB hãy nhập: monitor help

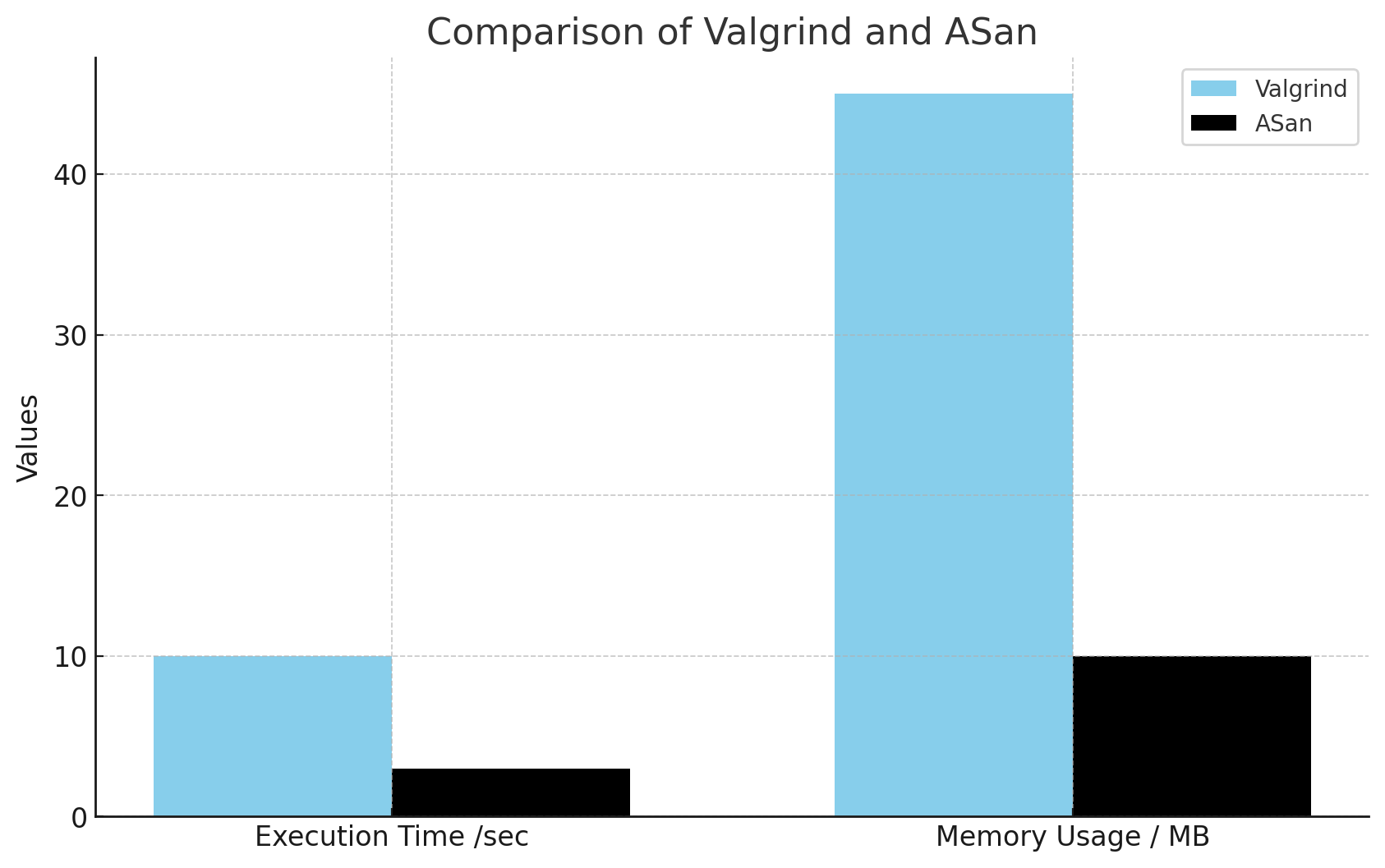


* Trong quá trình debug, GDB cho thấy s.flag2 nằm tại địa chỉ 0x1ffeffff24 và s.flag1 tại 0x1ffeffff20. Kích thước của s.flag1 là 4 byte. Khi kiểm tra bộ nhớ với lệnh monitor xb, tại địa chỉ 0x1ffeffff20 (nơi chứa s.flag1), giá trị là 0x00 0x00 0x00 0x00, nghĩa là giá trị chưa được khởi tạo hoặc mặc định là 0. Trong khi đó, tại địa chỉ 0x1ffeffff24 (nơi chứa s.flag2), giá trị là 0x02 0x00 0x00 0x00, tương ứng với giá trị được thiết lập trong chương trình (s->flag2 = 2). Điều này xác nhận rằng s.flag1 chưa được khởi tạo, dẫn đến các lỗi liên quan đến giá trị chưa xác định trong chương trình.
* Để kiểm tra rò rỉ bộ nhớ, chúng ta sử dụng lệnh: **monitor leak\_check**



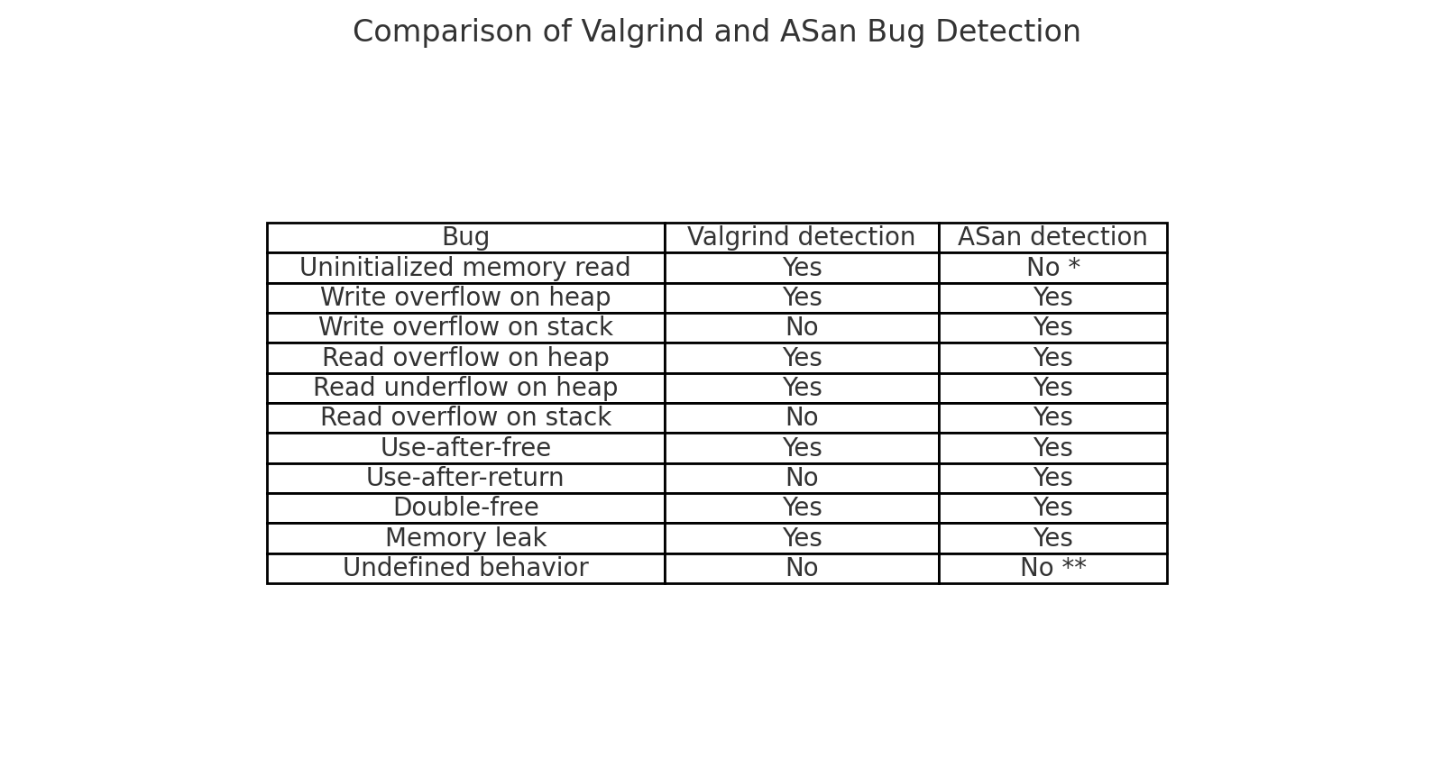
* Chúng ta cũng có thể yêu cầu thêm thông tin về các con trỏ tới bộ nhớ mà chúng ta nghĩ có thể bị rò rỉ. Ví dụ, biến s được phân bổ trong main, vì vậy bạn sẽ mong đợi nó nằm trên stack. Và **montitor who\_points\_at** sẽ cho thấy cho thấy con trỏ stack RSP trỏ tới biến đó:  
  
* Ngoài ra còn rất nhiều option hữu ích của công cụ monitor được cung cấp.
* Để kết thúc debug bằng GDB, chạy lệnh **kill.**

# So sánh Valgrind và Anddressanitizer

  
Biểu đồ so sánh Valgrind và Asan dựa trên ET và MU

Hình ảnh này so sánh hiệu suất của Valgrind và AddressSanitizer (ASan) dựa trên hai tiêu chí: thời gian thực thi (Execution Time) và mức sử dụng bộ nhớ (Memory Usage). AddressSanitizer cho thấy thời gian thực thi thấp hơn đáng kể so với Valgrind, chứng minh rằng ASan nhanh hơn trong việc xử lý. Về mức sử dụng bộ nhớ, Valgrind tiêu thụ một lượng bộ nhớ lớn hơn nhiều so với ASan, lên đến hơn 40 MB, trong khi ASan chỉ sử dụng khoảng vài MB. Điều này chỉ ra rằng Valgrind tốn nhiều tài nguyên hơn để phân tích.

Sự khác biệt về thời gian thực thi và mức sử dụng bộ nhớ giữa Valgrind và AddressSanitizer (ASan) bắt nguồn từ cơ chế hoạt động của từng công cụ. Valgrind sử dụng mô phỏng CPU và dịch mã nhị phân để theo dõi thao tác bộ nhớ, mang lại độ chính xác cao nhưng làm chậm thời gian thực thi và tăng mức tiêu thụ bộ nhớ đáng kể do phải duy trì trạng thái chi tiết của chương trình. Ngược lại, ASan chèn mã kiểm tra vào chương trình trong quá trình biên dịch, cho phép phát hiện lỗi bộ nhớ nhanh chóng mà không cần mô phỏng, dẫn đến hiệu năng vượt trội và mức sử dụng bộ nhớ thấp hơn. Do đó, Valgrind phù hợp cho việc gỡ lỗi chi tiết, còn ASan hiệu quả hơn cho các ứng dụng cần tốc độ.



Bảng So sánh khả năng phát hiện lỗi

Bảng so sánh cho thấy Valgrind và AddressSanitizer có khả năng phát hiện các lỗi khác nhau trong quản lý bộ nhớ. Valgrind nổi bật với khả năng phát hiện toàn diện, bao gồm lỗi uninitialized memory read, write overflow on heap, read underflow on heap, use-after-free, và memory leaks. Tuy nhiên, nó không hỗ trợ phát hiện lỗi write overflow on stack, read overflow on stack, hoặc undefined behavior. Ngược lại, AddressSanitizer hiệu quả hơn trong việc phát hiện lỗi write overflow on stack, read overflow on stack, và undefined behavior nhưng gặp hạn chế trong việc phát hiện uninitialized memory read. Sự khác biệt này phản ánh chiến lược thiết kế: Valgrind ưu tiên độ chi tiết và kiểm tra toàn diện, trong khi AddressSanitizer tập trung vào tốc độ và các lỗi phổ biến.

# Tổng kết

Báo cáo này cung cấp một cái nhìn tổng quan và chi tiết về Valgrind, một công cụ mạnh mẽ hỗ trợ phân tích và gỡ lỗi các chương trình trên hệ điều hành Linux. Qua các phần, chúng ta đã tìm hiểu Valgrind là gì, các điểm mạnh và yếu của nó khi so sánh với AddressSanitizer, và sự khác biệt nổi bật giữa các công cụ debug hiện có. Kiến trúc Valgrind được phân tích kỹ lưỡng, bao gồm Valgrind Core, Tool Plugins, và các thành phần trong Simulated CPU, làm rõ cách Valgrind mô phỏng CPU để phát hiện và xử lý lỗi.

Phần hướng dẫn sử dụng Valgrind minh họa các bước cài đặt, debug cơ bản, cũng như khai thác các tool khác để tối ưu hóa chương trình. Ngoài ra, VGDB được giới thiệu như một cầu nối giữa Valgrind và GDB, hỗ trợ gỡ lỗi nâng cao cho các lỗi phức tạp. Cuối cùng, sự so sánh giữa Valgrind và AddressSanitizer chỉ ra các ưu nhược điểm của mỗi công cụ, giúp người dùng lựa chọn giải pháp phù hợp theo nhu cầu cụ thể.

Valgrind không chỉ là công cụ phát hiện lỗi bộ nhớ mạnh mẽ mà còn cung cấp khả năng phân tích hiệu năng và gỡ lỗi chi tiết, là lựa chọn hàng đầu cho các nhà phát triển phần mềm. Báo cáo này hy vọng sẽ giúp người đọc hiểu rõ hơn về Valgrind, cách sử dụng nó hiệu quả, và ứng dụng trong việc phát triển và tối ưu hóa các ứng dụng phức tạp.

# Tài liệu tham khảo

1. <https://valgrind.org/>

2. <https://valgrind.org/docs/shadow-memory2007.pdf>

3. <https://stackoverflow.com/questions/5134891/how-do-i-use-valgrind-to-find-memory-leaks>