

# Lab 2: Mini-Debugger: Technical Report

2025MCS2121 Gaurav Jain

2025MCS2122 Aman Singh

17 December 2025

## 1 Introduction

### 1.1 Problem Statement and Scope

This project entails the development of a minimal debugger for ELF binaries on the Linux operating system. The primary objective is to load, control, inspect, and step through processes using OS-level debugging primitives, specifically avoiding high-level tools like GDB. The debugger is required to support breakpoint management, single-stepping, register inspection, and status reporting.

### 1.2 Design Philosophy

The system is designed as a lightweight command-line interface (CLI) tool. It operates by forking a child process to execute the target binary while the parent process acts as the tracer. This architecture leverages the Linux `ptrace` API for process control and memory manipulation, ensuring strict adherence to the non-functional requirement of not corrupting the debugger's state.

## 2 System Architecture

The debugger is modularized into five core components, managed by a central event loop in `main.c`.

### 2.1 Process Control (`debugger.c`)

The debugger initiates the session using a standard `fork()` and `exec()` pattern.

- **Child Process:** Executes `ptrace(PTRACE_TRACEME, ...)` to allow the parent to trace it, then calls `execl()` to replace its memory image with the target program.
- **Parent Process:** Uses `waitpid()` to synchronize with the child. It issues `ptrace` commands to control execution (e.g., `PTRACE_CONT`) and intercepts signals (like `SIGTRAP`) generated by breakpoints.

### 2.2 Breakpoint Management (`breakpoints.c`)

This module manages a data structure of active breakpoints. The key challenge is safely modifying the instruction stream of the running process without permanently corrupting the binary.

- **Insertion:** The debugger reads the instruction word at the target address using `PTRACE_PEEKTEXT`. It preserves the least significant byte (LSB) in the `breakpoint` struct and replaces it with the `INT 3` opcode (`0xCC`).
- **Restoration:** To remove a breakpoint or resume execution, the original byte is restored using bitwise operations: `(data & ~0xFF) | original_byte`.

## 3 Technical Implementation

### 3.1 The Breakpoint Mechanism

The core complexity lies in handling the SIGTRAP signal when a breakpoint is hit.

1. When the CPU executes 0xCC, it halts the child process and sends SIGTRAP to the parent.
2. The parent inspects the Instruction Pointer (RIP) via PTRACE\_GETREGS.
3. Since the CPU advances RIP after fetching the instruction, the RIP will be one byte past the breakpoint address.
4. The debugger detects this condition: `if (regs.rip - 1 == bp->addr)`.
5. To resume, the debugger must:
  - Restore the original instruction byte.
  - Decrement RIP by 1 (`regs.rip -= 1`) so the original instruction is executed.
  - Update the child's registers via PTRACE\_SETREGS.

```
1 void handle_breakpoint(pid_t pid, struct breakpoint *bp)
2 {
3     struct user_regs_struct regs;
4     ptrace(PTRACE_GETREGS, pid, NULL, &regs);
5
6     if (regs.rip - 1 == (unsigned long)bp->addr)
7     {
8         // Restore original code and reset instruction pointer
9         remove_breakpoint(pid, bp);
10        regs.rip -= 1;
11        ptrace(PTRACE_SETREGS, pid, NULL, &regs);
12        printf("Hit breakpoint at %p\n", bp->addr);
13    }
14 }
```

Listing 1: Handling Breakpoint Hit

### 3.2 Single Stepping

Single-stepping is implemented via `ptrace(PTRACE_SINGLESTEP, ...)`. This places the processor in a mode where it executes one machine instruction and then raises a SIGTRAP, returning control to the debugger. This is essential for fine-grained analysis of control flow.

### 3.3 Register Inspection

Register states are retrieved using `ptrace(PTRACE_GETREGS, ...)`, which populates the `sys/user.h` defined `user_regs_struct`. The debugger outputs critical 64-bit registers (RIP, RSP, RAX, RBX, etc.) to standard output, allowing users to verify variable values and stack pointers during execution.

## 4 Testing and Validation

The project includes a custom unit testing framework defined in `tests/test_framework.h`. The framework macros (`TEST`, `ASSERT_EQ`) facilitate automated validation of:

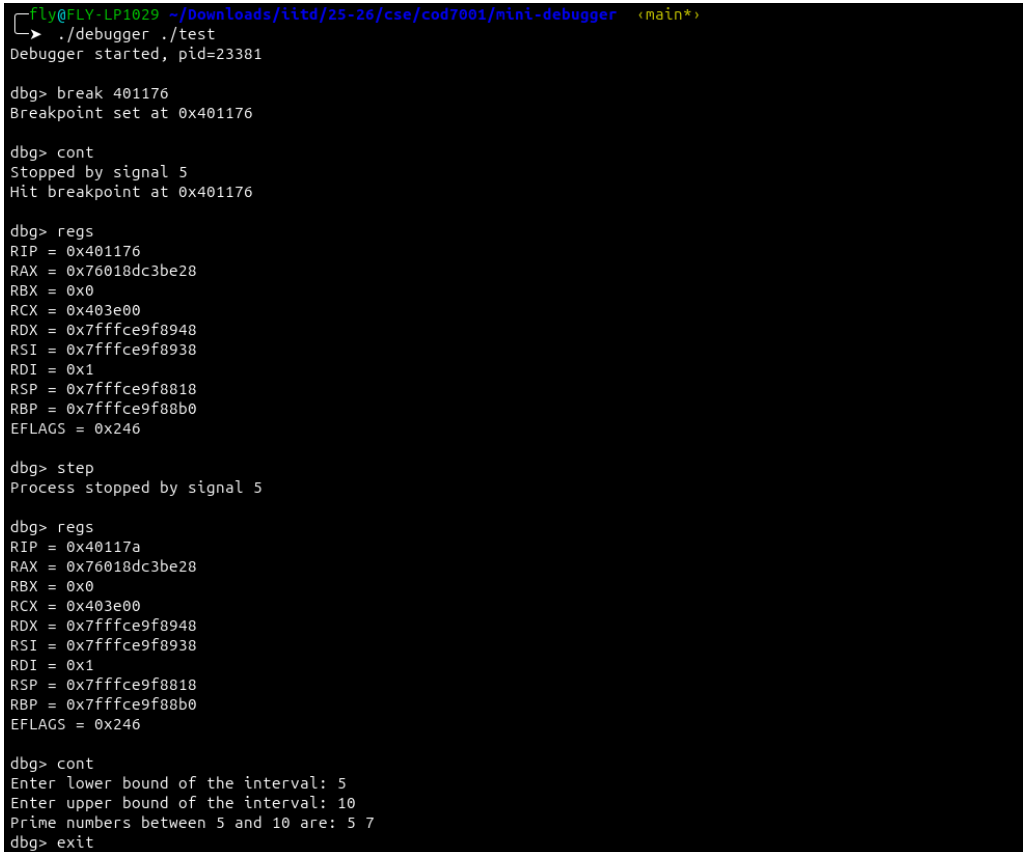
- **Data Structures:** Verifying breakpoint struct initialization and flag management.
- **Integration:** An automated test creates a dummy C program, compiles it, and verifies that the debugger can attach and control it without crashing.
- **Status Macros:** Validates correct interpretation of `WIFEXITED` and `WIFSTOPPED` macros across different OS environments.

## 5 Conclusion

The Mini-Debugger successfully meets all functional requirements set forth in Lab 2A. By directly manipulating process memory and execution flow via `ptrace`, the tool provides a robust environment for binary analysis. The implementation demonstrates a solid understanding of low-level system programming concepts, including signal handling, opcode injection, and register manipulation.

## Appendix

### A1. Demo Screenshot



```
Fly@FLY-LP1029 ~/Downloads/itd/25-26/cse/cod7001/mini-debugger (main*)
> ./debugger ./test
Debugger started, pid=23381

dbg> break 401176
Breakpoint set at 0x401176

dbg> cont
Stopped by signal 5
Hit breakpoint at 0x401176

dbg> regs
RIP = 0x401176
RAX = 0x76018dc3be28
RBX = 0x0
RCX = 0x403e00
RDX = 0x7fffce9f8948
RSI = 0x7fffce9f8938
RDI = 0x1
RSP = 0x7fffce9f8818
RBP = 0x7fffce9f88b0
EFLAGS = 0x246

dbg> step
Process stopped by signal 5

dbg> regs
RIP = 0x40117a
RAX = 0x76018dc3be28
RBX = 0x0
RCX = 0x403e00
RDX = 0x7fffce9f8948
RSI = 0x7fffce9f8938
RDI = 0x1
RSP = 0x7fffce9f8818
RBP = 0x7fffce9f88b0
EFLAGS = 0x246

dbg> cont
Enter lower bound of the interval: 5
Enter upper bound of the interval: 10
Prime numbers between 5 and 10 are: 5 7
dbg> exit
```

Figure 1: Debugger Interactive Session

## A2. Commit History

```
* commit 4f1a13ac369a17d3f89bd6abe895b9f587c80c1b (HEAD -> main, origin/main, origin/HEAD)
Author: Gaurav Jain <43726919+gauravjain2@users.noreply.github.com>
Date:   Wed Dec 17 12:04:01 2025 +0530

    added readme

* commit bd268fe4cf7b4651359b55c77d7c56d354e558c6
Merge: 8545593 cb52e4e
Author: Anan Singh <anansingh12647@gmail.com>
Date:   Wed Dec 17 11:52:10 2025 +0530

    Merge pull request #3 from anan-singh-12647/fixes_and_tests

    Added tests cases, make file change and fixes

* commit cb52e4e7d8b935348aa6a6b935db719b7f99499c (origin/fixes_and_tests, fixes_and_tests)
Author: Gaurav Jain <43726919+gauravjain2@users.noreply.github.com>
Date:   Wed Dec 17 11:47:43 2025 +0530

    added test cases

* commit d44d5233c8fe1bd1d4e62bfc3b554b41d938df2
Author: Gaurav Jain <43726919+gauravjain2@users.noreply.github.com>
Date:   Wed Dec 17 11:12:51 2025 +0530

    minor fixes

* commit e9694b6f7aa3f282a48e31a9150873ff0a60528b
Author: Gaurav Jain <43726919+gauravjain2@users.noreply.github.com>
Date:   Wed Dec 17 11:08:59 2025 +0530

    Integrated regs and step in main.c

* commit 8545593b68c4f0e2aa4e6b4ff84f0e6790e77bf5
Merge: 62db04f 2df3e5e
Author: Gaurav Jain <43726919+gauravjain2@users.noreply.github.com>
Date:   Wed Dec 17 10:54:44 2025 +0530

    Merge pull request #2 from anan-singh-12647/feature/step-registers

    Feature: Step and Registers

    Enhance single-step with detailed status reporting

* commit 2df3e5e56013863d7b080b9c2f5e9bb38b1e6e (origin/feature/step-registers, feature/step-registers)
Author: Gaurav Jain <43726919+gauravjain2@users.noreply.github.com>
Date:   Tue Dec 16 17:53:47 2025 +0530

    Add process status inspection helpers

* commit 683ff0ea38739cd591baf10e544d07b36cc4247c
Author: Gaurav Jain <43726919+gauravjain2@users.noreply.github.com>
Date:   Tue Dec 16 12:35:17 2025 +0530

    Implement single-step execution using PTRACE_SINGLESTEP

* commit ca1a31b342c8bc09a0bc4b3f9886d65c8211bab
Author: Gaurav Jain <43726919+gauravjain2@users.noreply.github.com>
Date:   Mon Dec 15 20:24:09 2025 +0530

    Extend register dump to general purpose registers

* commit 7e451fe4cac3df80a17704ec8117980c8121835
Author: Gaurav Jain <43726919+gauravjain2@users.noreply.github.com>
Date:   Mon Dec 15 15:14:09 2025 +0530

    Add register inspection interface using ptrace

* commit 62db04fde218dfb08a8cbb52ea3a7469bc3e537
Merge: 25202e8 fc5647e
Author: Gaurav Jain <43726919+gauravjain2@users.noreply.github.com>
Date:   Wed Dec 17 10:52:46 2025 +0530

    Merge pull request #1 from anan-singh-12647/feature/process-breakpoints

    Feature: Process Breakpoints

    Add initial Makefile for building the debugger

* commit fc5647e62b19bf81521e72a8c5468ae20e6e2ca (origin/feature/process-breakpoints, feature/process-breakpoints)
Author: Anan Singh <anan.singh2@gmail.com>
Date:   Tue Dec 16 18:46:13 2025 +0530

    Implement breakpoint management functions and integrate with main debugger loop

* commit e64a4c55c18d7b6321def4c15c8260b9ce4ce12
Author: Anan Singh <anan.singh2@gmail.com>
Date:   Tue Dec 16 18:01:54 2025 +0530

    Add debugger functionality with start_debugger and continue_execution functions

* commit 25202e8baee9980bbce2da3fdec2dd58c4541f63 (test)
Author: Anan Singh <anan.singh2@gmail.com>
Date:   Mon Dec 15 15:38:41 2025 +0530

    Initial repo setup

* commit 6afdc348264a3e9c65b854564138e64cf4413cf3
Author: Anan Singh <anansingh12647@gmail.com>
Date:   Mon Dec 15 09:08:31 2025 +0530
```

Figure 2: Project Commit History