Debugger Toolchain Validation via Cross-Level Debugging

Yibiao Yang
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
yangyibiao@nju.edu.cn

Maolin Sun
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
merlin@smail.nju.edu.cn

Jiangchang Wu
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
jiangchangwu@smail.nju.edu.cn

Qingyang Li
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
liqingyang@smail.nju.edu.cn

Yuming Zhou
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
zhouyuming@nju.edu.cn

Abstract

Ensuring the correctness of debugger toolchains is of paramount importance, as they play a vital role in understanding and resolving programming errors during software development. Bugs hidden within these toolchains can significantly mislead developers. Unfortunately, comprehensive testing of debugger toolchains is lacking due to the absence of effective test oracles. Existing studies on debugger toolchain validation have primarily focused on validating the debug information within optimized executables by comparing the traces between debugging optimized and unoptimized executables (i.e., ***different executables***) in the debugger, under the assumption that the traces obtained from debugging unoptimized executables serve as a reliable oracle. However, these techniques suffer from inherent limitations, as compiler optimizations can drastically alter source code elements, variable representations, and instruction order, rendering the traces obtained from debugging *different executables* incomparable and failing to uncover bugs in debugger toolchains when debugging unoptimized executables. To address these limitations, we propose a novel concept called *Cross-Level Debugging* (CLD) for validating the debugger toolchain. CLD compares the traces obtained from debugging the ***same executable*** using source-level and instruction-level strategies within the same debugger. The core insight of CLD is that the execution traces obtained from different debugging levels for the *same executable* should adhere to specific relationships, regardless of whether the executable is generated with or without optimization. We formulate three key relations in CLD: *reachability preservation* of program locations, *order preservation* for reachable program locations, and *value consistency* at program locations, which apply to traces at different debugging levels. We implement Devil, a practical framework that employs these relations for debugger toolchain validation. We evaluate the effectiveness of Devil using two widely used production debugger toolchains, GDB and LLDB. Ultimately, Devil successfully identified 27 new bug reports, of which 18 have been confirmed and 12 have been fixed by developers.

CCS Concepts: • **Software and its engineering → Software verification and validation**; *Compilers.*

Keywords: Debugger, Validation, Cross-Level Debugging

ACM Reference Format:
Yibiao Yang, Maolin Sun, Jiangchang Wu, Qingyang Li, and Yuming Zhou. 2025. Debugger Toolchain Validation via Cross-Level Debugging. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3669940.3707271

ASPLOS '25, March 30-April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-0698-1/25/03
https://doi.org/10.1145/3669940.3707271

1 Introduction

Debugger toolchains serve as essential infrastructure for developers, facilitating effective code comprehension, localization, and resolution of program errors within software systems. These toolchains offer sophisticated debugging functionalities, primarily involving management of breakpoints, control of the program execution flow, and access to variable states [2, 13]. These capabilities enable developers to control program execution and thoroughly examine dynamic program states precisely. However, bugs within these debugger toolchains can significantly mislead developers, leading to erroneous assumptions regarding the execution status of

specific source lines or variable values [7]. As prior study states [3], *the entire toolchain (compiler, linker, debugger) must be free of debug information bugs to provide a reliable debugging experience*. Therefore, ensuring the correctness of debugger toolchains is of utmost importance.

Unfortunately, validating debugger toolchains presents significant challenges, primarily due to the absence of effective debug actions as test inputs and the lack of well-defined expected behaviors serving as test oracles. Debuggers allow developers to employ flexible debug actions based on the dynamic program states, making it a complex task to devise effective debug actions specifically for testing debugger toolchains. Furthermore, in the context of debugger toolchain testing, the test oracle encompasses extensive information that captures the dynamic program states executed within the debugger. Obtaining such a test oracle statically, given a test program and a sequence of debug actions, is highly challenging, and the substantial human intervention required makes it impractical.

Prior studies have proposed various techniques for validating debugger toolchains [1, 3, 8]. These techniques have demonstrated their efficacy in exposing bugs in debug information emitted by compiler optimizations. However, these existing techniques suffer from inherent limitations that significantly constrain their efficacy and applicability. They have primarily focused on uncovering bugs specifically related to compiler optimizations by identifying inconsistencies between the execution traces obtained from debugging optimized and unoptimized executables (i.e., **different executables**). This narrow focus is predicated on the assumption that the traces from debugging unoptimized executables serve as a reliable oracle, which may fail to uncover bugs in debugger toolchains that are entirely unrelated to compiler optimizations. Moreover, these techniques may result in an influx of false positives due to the drastic effects of compiler optimizations, which can aggressively optimize out source code elements and variables, as well as reorder instructions, rendering the state traces between debugging optimized and unoptimized executables fundamentally incomparable.

Approach. In this paper, we present a novel concept called *Cross-Level Debugging* (CLD) for identifying bugs in debugger toolchains. CLD compares the traces for debugging the **same executable** within the debugger respectively using *source-level* and *instruction-level* debugging strategies and identifies violations on predefined relations as potential bugs in debugger toolchains. In particular, we formalize three key predefined relations in CLD across different levels of debugging strategies: *reachability preservation* of program locations, *order preservation* for reachable program locations, and *value consistency* at program locations, which apply to traces at different debugging levels. With CLD, we implement Devil, a practical framework that employs these three predefined relations for debugger toolchain validation. We evaluate the effectiveness of Devil using two widely used production debugger toolchains, GDB and LLDB. Ultimately, Devil successfully identified 27 new bug reports, of which 18 have been confirmed and 12 have been fixed by developers. In summary, we make the following main contributions:

- **Novel Conceptual Approach**: We propose a novel concept called *Cross-Level Debugging* (CLD) to address the test oracle problem for validating debugger toolchains. CLD offers a fresh perspective by comparing traces across different levels for debugging the **same executable**, without relying on the problematic assumption that traces from debugging the unoptimized executable (i.e. a **different executable**) serve as a reliable oracle. This innovative approach enhances the effectiveness and broad applicability of debugger toolchain validation.
- **Comprehensive Systematic Framework**: We introduce a comprehensive framework for exhaustively exploring debugger behavior. This framework allows a program to be run to any specified location using various methods, after which both source-level and instruction-level debugging strategies are applied to obtain the corresponding traces. The framework formalizes three key relations for comparing these traces across different debugging levels, effectively capturing essential debugger behavior. This structured framework facilitates in-depth comparative analysis and yields valuable insights.
- **Practical Implementation**: We develop a practical prototype called Devil to validate debugger toolchains. Through comprehensive evaluations on two widely-used debuggers, GDB and LLDB, Devil successfully identified and reported 27 bugs, 18 of which have been confirmed or fixed by developers. Notably, several bugs reported by Devil have been marked as critical, and these critical bugs cannot be exposed by existing techniques.

2 Background and Motivation

This section overviews the debugger toolchain and the existing research on its validation. Then, we use a concrete example to motivate our approach.

2.1 Debugger Toolchain and Validation

The debugger toolchain is composed of a compiler and a debugger. Compilers are responsible for generating debug information along with machine code to facilitate the debugging process. The debug information plays a vital role in establishing the relationship between the source code and the resulting executable. It encompasses essential elements such as variable and function names, types, symbol locations, scopes, and function call stack frames.

Debuggers are essential tools for programmers, providing the capability to analyze the state of a running program. They heavily rely on the availability of debug information, which is crucial for a comprehensive understanding of the program's

behavior. By leveraging the debug information, debuggers offer stepping functionality, enabling developers to investigate the call stack, which consists of stack frames generated during function calls. Examining the stack helps developers identify where the program paused and how execution reached that specific point, thus enabling the inspection of variable values and program locations.

Validating the debugger toolchain is of utmost importance to ensure the precise generation and effective utilization of debug information. However, due to the interactive nature of debugger toolchains, automatically detecting bugs in them presents significant challenges. Previous studies have focused on specific aspects of debugger toolchain validation. For instance, Li et al. [8] propose an approach employing actionable programs to inspect specific variable values, thereby validating the debug information produced by compilers at different optimization levels. Similarly, Di Luna et al. [3] devise a framework that establishes various trace invariants involving the hit line, backtrace, and parameters rather than the sole consideration of variables. More recently, Assaiante et al. [1] introduce the concept of the completeness problem concerning debug information, unveiling instances of incompleteness within the debug information.

While the aforementioned studies have demonstrated their effectiveness in validating debugger toolchains, they exhibit notable limitations. Primarily, these techniques predominantly focus on exposing bugs within the debug information associated with compiler optimizations, rendering them incapable of detecting issues unrelated to optimizations. Furthermore, comparing dynamic state traces between optimized and unoptimized code is often infeasible, as many source code and variables are eliminated during optimization.

2.2 Motivating Example

This paper introduces a novel concept called *Cross-Level Debugging* (CLD) and proposes a technique based on this concept for validating debugger toolchains. In Figure 1, we present a concrete example to illustrate our approach.[1]

Figure 1a presents a simplified test program that triggers bug #27151 in the GCC toolchain. In Figure 1b, the program is compiled using GCC with the `-g` flag to enable debug information at the optimization level `-O0`. Following compilation, the program is executed using GDB. A breakpoint is set at line 8 with the `break 8` command, followed by the `run` and `step` commands to initiate and continue execution at the source level. According to the GDB documentation[2], the `run` command starts the program under GDB, while the `step` command directs the program to "continue running until control reaches a different source line, then stop and return control to GDB." This functionality ensures that execution

[1]For illustrative purposes, the code and associated debugging commands in all figures within this paper are simplified from the original bug report.
[2]https://sourceware.org/gdb/current/onlinedocs/gdb.html/

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main()
6  {
7    int *p = (int *)malloc(sizeof(int)*4);
8    memset(p, 0, sizeof(p));
9    printf("hello world");
10   return 0;
11 }
```

(a) $\mathcal{P}_1$ that reveals a bug in the GDB toolchain. #27151

```
$ gcc -O0 -g a.c
$ gdb a.out
(gdb) break 8
Breakpoint at a.c:8
(gdb) run
8 memset(p, 0, sizeof(p));
(gdb) step
hello world
Process exited
```

(b) Source level

```
$ gcc -O0 -g a.c
$ gdb a.out
(gdb) break 8
Breakpoint at a.c:8
(gdb) run
8 memset(p, 0, sizeof(p));
(gdb) stepi 22
9 printf("hello world");
...
```

(c) Instruction level

Figure 1. A motivating example of CLD, derived and simplified from the GDB issue #27151 reported by Devil.

halts only at the first instruction of a source line, thereby preventing multiple interruptions within a statement.

However, in this instance, the program incorrectly exits normally, failing to pause at line 9 as expected when using the `step` command in GDB. In contrast, when the `stepi` command is utilized to step through the program at the instruction level, as illustrated in Figure 1c, the program behaves as expected and pauses at line 9. Notably, the `stepi 22` command instructs GDB to execute the next 22 instructions in the program. The value '22' is derived from the traces of the debug actions, indicating that 22 `stepi` operations are required to reach the ninth statement.

The discrepancy between source-level and instruction-level debugging reveals a bug in the GDB toolchain, initially identified by Devil and subsequently confirmed and resolved by the developers as a genuine GDB bug. Notably, compiling the test program with optimization levels such as `-O1`, `-O2`, and others exhibits the same erroneous behavior in the debugger as observed with `-O0`. This indicates that existing techniques are inadequate for detecting this issue. Thus, this case highlights the limitations of existing techniques in validating debugger toolchains and emphasizes the potential of CLD as a promising method to address these shortcomings.

3 Approach

In this paper, we propose a validation technique for debugger toolchains called *Cross-Level Debugging* (CLD). CLD involves comparing traces obtained from the debugger using different levels of debugging strategies.

3.1 Formulation

CLD utilizes two levels of debugging strategies, namely source-level debugging and instruction-level debugging, to execute the *same* target executable in the debugger step-by-step. This study assumes that the target program is well-formed, deterministic, and single-threaded.

For each strategy, we trace and record the program's location and variable values at each step as the dynamic program states. The program location includes information such as the source filename, line number, offset (i.e., the column of the source line), and the address of the corresponding machine instruction. Particularly, we record the occurrence and order of program locations being hit during execution, denoted as $\mathcal{H}(P)$ and $\mathcal{O}(P)$, respectively. Furthermore, we record the values of variables at each step, represented by $\mathcal{V}(P)$. Based on our observations of the debugger toolchain, we propose the following relations for the traces obtained from source-level and instruction-level debugging strategies:

1. R#1: **(Reachability preservation)** A program location in the source program should be hit at source-level stepping if and only if it can be hit at instruction-level stepping.
2. R#2: **(Order preservation)** If a program location l_a is hit before location l_b at the source-level stepping, then l_a should be hit before l_b at the instruction-level stepping.
3. R#3: **(Value consistency)** When a program location can be reached at both the source level and instruction level, the initial values of each variable at that program location should be consistent across different levels of debugging.

As discussed in Section 2.1, a line of source code can be associated with multiple machine instructions, while each machine instruction is typically mapped to only one line of source code. This characteristic implies that instruction-level debugging provides more fine-grained control over the program's execution compared to source-level debugging. At source-level stepping, when the program's execution reaches a specific source line in the debugger, it essentially pauses at the head-most instruction associated with that source code line. Consequently, if a program location is reached through source-level stepping, it must also be encountered at instruction-level stepping. Thus, the relation of R#1 holds true. Furthermore, if there are two reachable lines of source code, their respective hit order should remain consistent across the two different debugging strategies, ensuring the satisfaction of relation R#2. Moreover, given a specific input, the execution of a program is deterministic. Regardless of the debugging strategy employed, the dynamic program states at specific program locations under different debugging levels should not contradict each other. Therefore, the values of each variable at different debugging levels should remain consistent, i.e., the relation R#3 is upheld. The formal definitions of the relations are as follows:

$$\text{R\#1:}\forall l \in P,\ l \in \mathcal{H}_s(P) \Rightarrow l \in \mathcal{H}_i(P)$$

$$\text{R\#2:}\forall l_a, l_b \in P,\ \mathcal{O}_s^{l_a}(P) > \mathcal{O}_s^{l_b}(P) \Leftrightarrow \mathcal{O}_i^{l_a}(P) > \mathcal{O}_i^{l_b}(P)$$

$$\text{R\#3:}\forall v^l \in (\mathcal{V}_s^l(P) \wedge \mathcal{V}_i^l(P)),\ v_s^l(P) = v_i^l(P)$$

Here, P denotes a given program, and l represents a program location in P, which can be a source line or an address. $\mathcal{H}_s(P)$ and $\mathcal{H}_i(P)$ represent the set of program locations hit at source-level and instruction-level stepping, respectively. $\mathcal{O}_s^{l_a}(P)$ and $\mathcal{O}_i^{l_b}(P)$ represent the hit order of program locations l_a and l_b at source-level and instruction-level stepping, respectively. $\mathcal{V}_s^l(P)$ and $\mathcal{V}_i^l(P)$ represent the values of the variables when the execution reaches source line l at source-level and instruction-level stepping, respectively. Correspondingly, $v_s^a(P)$ and $v_i^a(P)$ represent the value of variable v when the execution reaches source line l by source-level and instruction-level stepping, respectively. If the traces violate any of the aforementioned relations, a potential bug is exposed within the debugger toolchain.

3.2 Algorithm

Based on the aforementioned formalized relations, we implemented a tool named Devil to validate debugger toolchains via CLD. Algorithm 1 outlines the main process of Devil, which consists of the following main steps:

- ***Forwarding Program:*** Running the executable in the debugger and stopping it at a randomly selected program location using any debugging strategies.
- ***Stepping Program:*** Resuming to run the program in the debugger step-by-step, utilizing source-level stepping and instruction-level stepping, respectively.
- ***Recording Traces:*** Extracting dynamic program states of the program at each step where it paused in the debugger at different levels of debugging.
- ***Comparing Traces:*** Comparing the traces between source-level and instruction-level stepping to identify violations to the predefined relations as potential bugs in the toolchain.

3.2.1 Forwarding Program. To comprehensively examine the behavior of debugger toolchains, it is insufficient to solely execute the executable step-by-step starting from the beginning within the debugger. Devil addresses this limitation with a more comprehensive approach. It initiates the executable within the debugger and runs it until it reaches a randomly selected starting location (Line 6). This starting location can be either a source line of code or an address within the target executable. Devil employs various debugging strategies to reach this starting location. For example, when a source line is specified as the starting location, Devil can set a breakpoint on that line and execute the program in the debugger until it stops at that breakpoint. Alternatively, Devil can execute the program step-by-step at the source or instruction level until reaching this specified

Algorithm 1: DEVIL's process for debugger validation

Data: The debugger D, compiler C, program p, input i
Result: reported bugs

```
1  begin
       /* Loop each of compiler optimizations                    */
2      foreach opt ∈ D.getOptimizations() do
3          p_d ← C.compile(p, "-g", opt)
4          while no termination criterion met do
5              l ← RandLocation(p_d)
               /* S2: Run to the selected location               */
6              runToLocation(D, p_d, l)
               /* S3: Source-level debugging                     */
7              H_s, O_s, V_s ← stepping(D, p_d, "line")
               /* S3: Instruction-level debugging                */
8              H_i, O_i, V_i ← stepping(D, p_d, "inst")
               /* S4: Check predefined relations                 */
9              if isViolated(H_s, O_s, V_s, H_i, O_i, V_i) then
10                 reportBug()
   /* Run program step-by-step in debugger                       */
11 Function stepping(Debugger D, Program p_d, Level lev)
12     H, O, V ← [], [], []
13     D.start(p_d)
14     while D.processIsExit() == False do
15         if lev == "line" then
16             D.step()
17         else
18             D.stepi()
19         H, O, F, V ← D.getTrace (l)
20     return [H, O, V]
   /* Check whether relations are violated                       */
21 Function isViolated(H_s, O_s, V_s, H_i, O_i, V_i)
22     foreach l ∈ H_s do
23         if l ∉ H_i then
24             return True
25     if (l_a ∈ H_s) ∧ (l_b ∈ H_s) ∧ (l_a ∈ H_i) ∧ (l_b ∈ H_i) then
26         if (O_s^a − O_s^b) ≠ (O_i^a − O_i^b) then
27             return True
28     foreach a ∈ (V_s ∩ V_i) do
29         foreach v ∈ (V_s(a) ∩ V_i(a)) do
30             if v_s^a ≠ v_i^a then
31                 return True
32     return False
```

starting location. The primary objective of this step is to ensure the program execution successfully reaches the specified location. By employing different strategies, DEVIL aims to cover a wider range of program states, thereby increasing the possibility of identifying potential bugs.

3.2.2 Stepping Program.

Once the execution reaches the starting location of the target program, DEVIL resumes program execution and continues to execute the program step-by-step within the debugger. This step-by-step execution occurs at both the source-level and instruction-level debugging until the program exits (Lines 7-8). Taking GDB as an example, to continue running the program at the source-level debugging, DEVIL uses the debug action sequence "while(true) {`step`}" to execute the program line by line iteratively until the program exits. For instruction-level stepping, the corresponding debug action sequence is "while(true) {`stepi`}" (Lines 14-18). Notably, instruction-level stepping employs the `stepi` debug action instead of `step` to execute one instruction at each step.

3.2.3 Recording Traces.

When executing the program step-by-step within the debugger, DEVIL records traces of the dynamic program states at each step, serving as a representation of the debugger's behavior. These traces include the source line number, machine instruction address, and variable values. To extract this information, DEVIL examines the stack frame to determine the execution location, including the next source line number and the address of the subsequent machine instruction. Additionally, DEVIL records the values of individual variables at each step. For example, GDB provides commands like "`bt -frame-info`" and "`info args/locals`" to examine the stack frame and retrieve variable values. Along with stepping, the debug action sequence used to extract the dynamic states while stepping through the program in GDB is "while(true) {`bt -frame-info` → `info args/locals` → `stepi`}".

3.2.4 Comparing Traces.

By recording traces of dynamic program states during both source-level and instruction-level stepping, DEVIL compares the obtained traces to verify predefined relationships. For each test program, DEVIL examines the source lines, memory addresses, and variable values at various program locations within the traces generated by source-level and instruction-level debugging. The objective is to identify any violations of the predefined relationships within the traces. For example, if a source line s is reached during source-level stepping, DEVIL checks whether s is also reached during instruction-level stepping. Any violations identified in the traces are considered potential bugs in the debugger toolchain (Lines 9-10). It is important to note that multiple violations may arise from a single underlying bug in the debugger toolchain. To prevent duplicate or invalid bug reports, each identified violation will be manually inspected before being reported to developers.

3.3 Illustrative Example

In this section, we use three concrete examples to illustrate how DEVIL works. All these bugs are identified and reported by DEVIL and subsequently confirmed or fixed by developers.

```c
unsigned int b = 0, d = 0;
static int c[1][2]={{0, 1}};

int main() {
  for (; d<1; d++)
    for (; b<1; b++)
      c[b][d+1] = 0;

  return 0;
}
```

(a) $\mathcal{P}_2$

```
$ gcc -O0 -g small.c
$ gdb -q a.out
(gdb) b 5
Breakpoint 1 at a.c:5.
(gdb) r
Breakpoint 1 at a.c:5
5   for (; d<1; d++)
(gdb) step
7       c[b][d+1] = 0;
(gdb) step
6     for (; b<1; b++)
```

(b) Source level

```
$ gcc -O0 -g small.c
$ gdb -q a.out
(gdb) b 5
Breakpoint 1 at a.c:5.
(gdb) r
Breakpoint 1 at a.c:5
5   for (; d<1; d++)
(gdb) stepi 4
6     for (; b<1; b++)
(gdb) stepi 7
7       c[b][d+1] = 0;
```

(c) Instruction level

Figure 2. GDB bug#95360. When stepping line by line, Line #7 is hit before Line #6 as shown in (b). However, when stepping instruction by instruction, Line #6 is hit by GDB before Line #7 as shown in (c).

Violating R#1 (Reachability preservation). Figure 1, presented in the in the previous Section 2, illustrates a violation of *R#1*. As shown in Figure 1a, the process exits directly when stepping through $\mathcal{P}_1$ at the source level. However, during instruction-level stepping, Line #8 and Line #9 are encountered by the GDB debugger, as depicted in Figure 1c. It is important to note that the command "`stepi 22`" signifies executing "`stepi`" 22 times interactively within the debugger. Figure 1 clearly demonstrates that Line #8 and Line #9 can be reached during instruction-level stepping but not during source-level stepping, thereby violating the predefined relation *R#1*. This bug is identified by Devil for GDB and has been resolved by the GDB developers.

Violating R#2 (Order preservation). Figure 2 shows a concrete example demonstrating the violation of *R#2*. In this scenario, Devil compiles the input program $\mathcal{P}_2$ using GCC. When debugging the program at the source level in GDB (Figure 2b), it is observed that lines 5, 7, and 6 are hit successively, forming an ordered sequence $[\![5, 7, 6]\!]$. However, when debugging it at the instruction level (Figure 2c), the lines 5, 6, and 7 are hit successively, forming the sequence $[\![5, 6, 7]\!]$. This discrepancy between these two execution sequences indicates a bug in GDB. The bug was initially reported to the GCC Bugzilla, and with the assistance of Tom de Vries, it was transferred to the GDB Bugzilla by filing a new bug report for GDB.[3] According to the maintenance principle of GNU Bugzilla, a bug report with the status `NEW` is considered a confirmed bug report [4, 18]. Therefore, this is a confirmed GDB bug.

```c
#include <stdarg.h>

void f(int n, ...) {
  va_list ap;
  char *end;

  for(int i=0; i<2; i++) {
  va_start(ap, n);
  while (1) {
    end=va_arg(ap, char*);
    if(!end) break;
  }
  va_end(ap);
  }
}

int main() {
  f(1);
}
```

(a) $\mathcal{P}_3$

```
$ clang -O1 -g a.c
$ lldb a.out
(lldb) b main
Breakpoint 1 at a.c:18
(lldb) r
18  f(1);
(lldb) step 3
10 end=va_arg(ap, char*);
(lldb) fr var i
(int) i = 1
```

(b) Source level

```
$ clang -O1 -g a.c
$ lldb a.out
(lldb) b main
Breakpoint 1 at a.c:18
(lldb) r
18  f(1);
(lldb) stepi 19
10 end=va_arg(ap, char*);
(lldb) fr var i
(int) i = 0
```

(c) Instruction level

Figure 3. LLVM bug#46040. Inconsistent variable value between source level debugging as shown in (b) and instruction level debugging as shown in (c).

Violating R#3 (Value consistency). Figure 3 provides an example to illustrate the violation of *R#3*. For this program $\mathcal{P}_3$ (Figure 3a), Devil utilizes LLDB to execute it step-by-step in the debugger at different debugging levels. When debugging at the source level (Figure 3b), the value of variable `i` at line 10 is 1. However, when debugging at the instruction level (Figure 3c), the value of variable `i` at line 10 is 0. This discrepancy in the values of `i` at the same program location violates *R#3*, which states that the values of variables should be consistent at different debugging levels. Devil identified this bug, and it has been confirmed by the LLVM developers.

[3] https://sourceware.org/bugzilla/show_bug.cgi?id=26054

4 Experimental Setup

4.1 Research Question

In this study, we investigate the following research questions:

- ***RQ1: Effectiveness of Devil.*** *Can Devil effectively identify genuine bugs in debugger toolchains?*
- ***RQ2: Influence of Bugs.*** *What is the significance and impact of the bugs exposed by Devil?*
- ***RQ3: Contribution of Relations.*** *How effective are the predefined relations employed by Devil in exposing bugs?*
- ***RQ4: Comparative Evaluation.*** *How does the effectiveness of Devil compare to state-of-the-art (SOTA) techniques?*
- ***RQ5: Overhead of Devil.*** *What is the overhead incurred by Devil in validating debugger toolchains?*

4.2 Evaluation Setup

In this subsection, we describe the subject debugger toolchains under test, the test programs chosen for the validation, and the experimental environment.

Subject Debugger toolchains We applied Devil to the latest trunk versions of GCC/GDB and Clang/LLDB, the widely recognized debugger toolchains. We selected these toolchains as the subjects for the following reasons: (1) they have been *widely used* in various communities; (2) they are the leading compilers and debuggers, forming the most popular compiler toolchains of GNU and LLVM; and (3) these debuggers *support* the execution of programs at both the source and instruction levels. To enable debugging, we compile the source programs with the "`-g`" compilation flag, which instructs the compiler to include additional debug information to executables. Devil can be applied at any optimization level, so optimization flags such as "`-O0/-Og/-O1/-O2/-O3`" can also be optionally enabled during compilation. For instance, the command used by GCC to generate an executable with debug information enabled and optimization level "`-O1`" for a source file "`test.c`" is "`gcc -O1 -g test.c`". Similarly, when using Clang, the command is "`clang -O1 -g test.c`".

Subject Test Programs We selected the C programs from the test suite of GCC release (version 12.1.0) as the subject test programs. This test suite comprises 45,115 C source programs. We chose these test programs for the validation of C debugger toolchains due to the following reasons: (1) these test programs are openly available as open-source projects, facilitating access for research purposes; (2) these test programs cover a wide range of C semantics while strictly avoiding any undefined behaviors; and (3) there are many programs in the GCC test-suite are self-contained, independent of external libraries, and have predefined inputs that enable them to be compiled and executed independently.

Environment The experiments were conducted within a Docker container running Ubuntu 22.04, hosted on an Ubuntu 18.04 system. The underlying hardware features a 48-core Intel(R) Core(R) CPU of 2.00GHz and 125GiB of RAM.

4.3 Data Analysis Methodology

To address **RQ1**, we feed each test program from the subject test programs to Devil to uncover bugs in the debugger toolchains. We then manually inspect the identified violations and report potential bugs to the developers. We further investigate the number of bug reports that are confirmed or fixed by developers. To address **RQ2**, we investigate the significance of the bugs identified by Devil by examining their impact across different versions of the target toolchains. Although the bugs are found in the latest trunk version, they may also affect previous release versions. We select a range of official release versions to determine how many of them are affected by the corresponding bug-triggering inputs. This analysis allows us to understand the duration for which the reported bugs have existed in the subject toolchains, indicating their importance. Additionally, we review developers' discussions and feedback on the reported bugs to further evaluate their significance. To address **RQ3**, we investigate whether the state-of-the-art testing tool can detect the bugs reported by Devil. Besides, we examine the violated relations in the reported bugs. This analysis allows us to grasp the relative effectiveness of different relations in revealing bugs. This investigation enables us to understand the full extent of the contributions made by the relations in Devil.

5 Evaluation

We elaborate on the evaluation for Devil in this section with respect to each of the research questions.

5.1 *RQ1: Effectiveness of Devil.*

Our ultimate goal is to expose previously unknown bugs for the production toolchains. To this end, we applied Devil to the latest trunk of GCC/GDB (until GCC and GDB 13.1.0) and Clang/LLDB (until Clang and LLDB 16.0.0) during our experiment. For each executable generated from the seed programs, Devil executes the binary until it arrives at a predetermined program location within the debugger. Specifically, in this experimental setup, Devil sets breakpoints exclusively at the main function. Subsequently, the debugger resumes the execution of the program and proceeds to step through the remaining code using both source-level and instruction-level stepping. After that, Devil will inspect whether the predefined relationships are violated to expose bugs in the debugger toolchain. Devil regards each violation as a potential bug, and we will manually inspect each violation to determine whether it is a real bug before reporting. It is worth noting that, we exclude those uncompilable programs and those programs require more than ten seconds for compilation with respect to GCC and Clang, respectively. Besides,

Table 1. List of bugs reported by Devil.

NO.	Tool	#ID	Status	#Comment	Importance	Opt	Relation
1	GDB	25350	Fixed	8	**P2 critical**	-O0	R#1
2	GDB	25405	Fixed	3	**P2 critical**	-O0	R#1
3	GDB	25573	Fixed	1	**P2 critical**	-O0	R#1
4	GDB	26054	Confirmed	1	P2 normal	-O0	R#2
5	GDB	26061	NotABug	6	-	-O2	R#3
6	GDB	26063	Fixed	7	P2 normal	-O2	R#1
7	GDB	27151	Fixed	9	P2 normal	-O0	R#1
8	GDB	27179	Unconfirm	5	-	-O0	R#1
9	GDB	29220	Unconfirm	2	-	-O2	R#3
10	GDB	29236	Confirmed	1	P2 normal	-O2	R#1
11	GDB	90574	Fixed	8	P3 normal	-O0	R#1
12	GDB	90584	Fixed	7	P3 normal	-O0	R#1
13	GDB	90586	Confirmed	2	P3 normal	-O0	R#1
14	GDB	95414	Fixed	2	P3 normal	-O2	R#3
15	GDB	30357	Fixed	1	**P2 critical**	-O0	R#1
16	LLDB	45386	Fixed	2	enhance	-O1	R#3
17	LLDB	45387	Fixed	2	enhance	-O1	R#3
18	LLDB	45676	Confirmed	6	normal	-O0	R#1
19	LLDB	45920	Confirmed	2	normal	-O0	R#1
20	LLDB	46006	Fixed	3	normal	-O3	R#3
21	LLDB	46007	Duplicate	5	-	-O1	R#3
22	LLDB	46014	New	3	-	-O3	R#1
23	LLDB	46040	New	4	-	-O1	R#3
24	LLDB	46045	New	4	-	-O1	R#1
25	LLDB	48381	New	3	-	-O1	R#1
26	LLDB	48382	New	3	-	-O2	R#1
27	LLDB	55744	Confirmed	3	-	-O1	R#3

any program that requires more than ten seconds for the execution will also be excluded. As a result, approximately 5,000 and 3,000 test programs remain for the follow-up studies for GCC and Clang, respectively.

Bugs Found. We manually inspected the violations as well as the test programs and found that those detected violations can indeed uncover real defects in the production debuggers. We totally reported 27 valid bug reports to the bug tracking systems of GCC/GDB and LLVM. Most of them can be found under "iamanonymous.cs@gmail.com" in GDB/GCC's and LLVM's Bugzilla databases. It is important to note that the bug management guidelines of GCC/GDB and LLVM are not exactly similar [6, 14, 17]. Specifically, a reported bug in GCC/GDB Bugzilla will be initially labeled as "UNCONFIRMED," and the status will change to "NEW" if developers confirm it. However, in LLVM, a reported bug is labeled as "NEW"

Table 2. The optimization levels of the confirmed bugs.

Debugger	**-O0**	**-O1**	**-O2**	**-O3**
GDB	9	0	3	0
LLDB	2	3	0	1
Total	11	3	3	1

by default and will be marked as "CONFIRMED" upon developer confirmation. Additionally, many bugs identified in prior studies remain unfixed. Most violations detected by Devil in this study may be associated with these existing bugs. In order to avoid reporting duplicate bug reports, we only filed **27 new bugs reports** to the bug tracking systems of GCC/GDB and LLVM. At the end of 2021, the LLVM Foundation migrated the data from Bugzilla to the LLVM GitHub project repository.[4] In Github, if an issue is resolved, it will be closed as complete. Among the 27 bugs, 18 of them have been confirmed or fixed by the developers, as listed in Table 1. For the remaining bugs, most of them are still pending confirmation by the developers. Additionally, GDB bug #26061 has been closed as "NotABug" by the developer. Upon examination, the developer conjectured that this issue pertains to GCC rather than GDB. Two years later, after updating GCC to the latest trunk and re-evaluating this bug, we found that it had been resolved on the GCC side. We also examine the comments and annotations provided by developers to determine the related products where the bugs were located, as these bugs may reside in different components of the toolchain. Among the confirmed bugs, seven of them are diagnosed by the developers as pertaining to the debugger, while eight are attributed to compiler issues. The remaining bugs are currently pending further diagnosis.

Compilation Optimizations. Out of the 18 confirmed bugs, 11 were discovered at the "-O0" optimization level, as presented in Table 2. These bugs in the toolchains are not associated with compiler optimizations and thus are difficult to identify using existing testing tools. In addition, some bugs were exclusively identified at optimization levels (three for "-O1", three for "-O2", and one for "-O3"). Therefore, we can deduce that Devil not only uncovers bugs in the toolchains that are relevant to compiler optimizations but also detects bugs that are unrelated to compiler optimizations.

Summary on Effectiveness

Devil demonstrates its effectiveness by effectively detecting genuine bugs across various optimizations within debugger toolchains.

[4] https://github.com/llvm/llvm-project/milestone/1

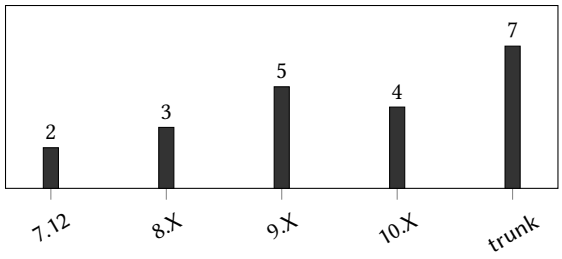

Figure 4. Confirmed and fixed bugs that affect corresponding release versions of GDB.

5.2 *RQ2: Influence of Bugs.*

We investigate the influence of bugs identified by Devil in terms of their importance, developers' feedback, and the versions they affect.

Importance of bug reports. We further examine the importance of the confirmed bugs by analyzing the importance tags assigned to the corresponding bug reports. The importance of a bug is described as the combination of its *priority* and *severity*. Specifically, the priority of a bug indicates the urgency of fixing it, with multiple levels of classification such as P1, P2, P3, and so on in GCC/GDB, while LLVM does not have priority classification. The severity of a bug, on the other hand, indicates its impact on the product, which can be classified as *critical*, *normal*, and *enhancement*. Note that the bug reports in the LLVM GitHub issue tracker typically do not have an importance tag either. Our findings are presented in the *sixth column* of Table 1. Notably, four out of the 18 confirmed bugs were assigned the importance of P2 critical, indicating a relatively high level of priority and severity in the bug management system of GCC/GDB. Moreover, as noted in Documentation of the GDB Bugzilla[5], "From the viewpoint of the GDB, how urgent is the bug fix? Select either medium or low. High is reserved." This statement implies that the 'P1' priority is reserved for the most urgent issues, suggesting that developers are unlikely to assign this highest priority to a bug without a compelling reason. To substantiate this, we reviewed all the 'P1 critical' bugs in the GDB Bugzilla over the past ten years. Only seven bugs were marked as 'P1 critical,' indicating that developers tend to reserve this highest priority for truly urgent issues. This context further emphasizes the importance of our four 'P2 critical' bugs. For LLDB, only two bugs were classified as 'P enhancement,' with the remaining bugs deemed more critical. Thus, we conclude that Devil is effective in detecting nontrivial bugs within debugger toolchains.

Feedback and discussions. The bugs reported by Devil have received positive feedback and comments from developers, reflecting their value and impact. Notably, Orlando Cazalet-Hyams, a developer of LLVM, provided feedback on our reported bug[6], acknowledging its significance and stating, *"Nice find! I get the feeling that the variable locations for 'i' are correct, but the line table is messed up."* This feedback demonstrates the engagement and recognition of the reported bugs within the developer community. Furthermore, several bugs identified by Devil have sparked extensive discussions among developers. We observed that 14 bugs received more than three comments, as indicated in the fifth column in Table 1. This is a positive indicator of the community's interest and involvement. For example, bug #90574 attracted the attention of two primary developers of GCC, who left a total of seven comments. These discussions delved into the core issue, the additional impact of the bug on the toolchains, and potential fixes. Another noteworthy example is bug #26061, where inconsistencies were observed between source-level debugging and instruction-level debugging for an argument's value. Two years after reporting the bug, we noticed that the problem no longer existed in the latest trunk version of the debugger toolchains. Consequently, we left a comment to inform the developers about the resolution. Tom Tromey, the primary GDB developer, marked this bug as "NOTABUG" but acknowledged that *"Based on this I think this is a compiler bug and not a GDB bug. There's nothing GDB can do to correct this kind of problem."* This feedback recognizes our bug report and emphasizes the importance of examining both the debugger and the compiler when inconsistencies are identified by Devil. These feedback instances highlight the significance of our bug reports and the potential for violations in the traces to uncover bugs in both the compiler and the debugger. Since both of them are essential in facilitating debugging, a thorough investigation is necessary when a violation occurs to determine whether it is a bug in the compiler or the debugger.

Table 3. Distribution of confirmed bugs across various predefined relations, categorized by debugger toolchains and optimization levels.

(a) Debugger Toolchain

Debugger	#Confirmed Bugs			
	R#1	R#2	R#3	*Total*
GDB	10	1	1	12
LLDB	2	0	4	6
Total	12	1	5	18

(b) Optimization Level

Opt.	#Confirmed Bugs			
	R#1	R#2	R#3	*Total*
-O0	10	1	0	11
-O1	0	0	3	3
-O2	2	0	1	3
-O3	0	0	1	1

Affected Versions. We conducted a thorough analysis of the affected versions of the associated products impacted by the confirmed bugs identified by Devil. We empirically investigated the affected versions of relevant products for each bug. It is important to note that a bug can affect multiple versions

[5] https://sourceware.org/gdb/bugs/

[6] https://bugs.llvm.org/show_bug.cgi?id=46040

Table 4. Distribution of confirmed bug importance across various predefined relations.

Debugger	Importance	#Confirmed bugs			
		R#1	R#2	R#3	*Total*
GDB	**P2 critical**	4	0	0	4
	P2 normal	3	1	0	4
	P3 normal	3	0	1	4
	normal	2	0	1	3
LLDB	enhancement	0	0	2	2
	normal	0	1	1	2

of a product, indicating its persistence within the toolchains over a certain period of time. As depicted in Figure 4, two out of the seven confirmed GDB bugs appeared at GDB-7.12 (released on October 7, 2016), implying that they remained dormant for over two years until our report. Furthermore, in the case of the bugs pertaining to GCC, one was found in GCC-7, also exhibiting a lifespan of approximately two years. Similarly, among the six confirmed or fixed LLVM toolchain bugs, developers have definitively identified three as originating from Clang. Upon examining these Clang bugs, we observed that all of them can be traced back to Clang-7.0, indicating that they evaded detection by other testing tools and remained concealed for around two years.

Summary on Influence

The bugs identified by Devil are regarded as important, and even critical, sparking extensive discussions. Some of these issues have remained latent for a long period, highlighting their significant influence.

5.3 *RQ3: Contribution of Predefined Relations.*

Tables 3 and Tables 4 present the statistics on the distribution of the detected bugs. As shown in Table 3(a), we found that all the predefined relations are able to identify inconsistencies in debuggers and detect bugs confirmed or fixed by developers. This indicates the effectiveness of these relations in exposing bugs in debuggers. Specifically, the *R#1* (Reachability preservation) rule led to the majority of confirmed bugs, accounting for approximately 67% of the total. Out of the 11 confirmed bugs discovered at the "-O0" optimization level, 10 were discovered by *R#1*, as presented in Table 3(b). Furthermore, all four P2 critical bugs were detected by *R#1*, as presented in Table 4. The *R#2* (Order preservation) and *R#3* (Value consistency) rules resulted in the detection of one and five confirmed bugs, respectively, also contributing to the overall effectiveness.

Summary on Contributions of Relations

The relation of *R#1* (Reachability preservation), to a certain extent, demonstrates a higher level of effectiveness compared to the other predefined relations.

5.4 *RQ4: Comparative Evaluation.*

To investigate whether Devil is complementary to existing techniques, we conducted a comparison with the state-of-the-art testing tool, Debug^2 [3]. Debug^2 was chosen for comparison because it is the most recent technique targeting correctness bug detection in the toolchains. We employed Debug^2 to run the bug-revealing programs corresponding to the confirmed issues reported by Devil. The results are presented in Table 5. Ultimately, we found that most of the bugs (13/18) reported by Devil cannot be detected by Debug^2. Besides, a bug related to compiler optimizations unearthed by Devil cannot be detected by Debug^2 since Devil considers a broader range of program states than Debug^2. We also examined whether Devil can expose the bugs discovered by Debug^2. We used the programs reported by Debug^2 as inputs for Devil, and observed that 5 out of the 17 reproducible bugs can be exposed by Devil. These results demonstrate that those techniques are orthogonal, and Devil is a vital complement to existing testing techniques.

Summary on Contributions of Relations

Devil is a significant complement to existing testing techniques for validating debugger toolchain.

5.5 *RQ5: Computational Overhead of Devil*

The execution overhead of testing tools is a crucial factor in assessing their practicality. To evaluate the execution overhead of Devil, we utilize the GCC release test suite (version 12.1.0) as our subject test programs. This evaluation employs GCC/GDB 13.1.0 and Clang/LLDB, specifically the commit 08d094a. Only those programs that can be compiled independently within 10 seconds, executed within 10 seconds, and debugged within 60 seconds, under both source-level stepping and instruction-level stepping, are considered for use in Devil. Table 6 presents statistics for the subject test programs that encountered errors or timeouts during the compilation, execution, and debugging stages. As shown, the majority of these programs (i.e., over 80%) cannot be compiled independently. In total, around 7,000 and 6,000 test programs (as noted in the 8th row of Table 6) meet these criteria for GCC and LLVM, respectively, and are used in Devil. Subsequently, we present the average compilation time, standalone execution time, and stepping time in Devil for both source and instruction levels in Table 7. Specifically, we find that the execution time required by Devil is significantly greater than the standalone execution time for

Table 5. Comparison between DEVIL and *Debug*^{2} (SOTA).

DEVIL					Debug^2				
Tool	#ID	Importance	Opt	SOTA	Tool	#ID	Importance	Opt	DEVIL
GDB	25350	**P2 critical**	-O0	✗	LLDB	45883	normal	-O3	✓
GDB	25405	**P2 critical**	-O0	✗	LLDB	45895	enhance	-O1	✗
GDB	25573	**P2 critical**	-O0	✗	LLDB	45902	enhance	-O3	✗
GDB	26054	P2 normal	-O0	✗	LLDB	45923	enhance	-Og	✗
GDB	26063	P2 normal	-O2	✓	LLDB	45934	enhance	-Oz	✓
GDB	27151	P2 normal	-O0	✗	LLDB	45971	enhance	Og	✗
GDB	29236	P2 normal	-O2	✗	LLDB	46002	enhance	-Og	✗
GDB	90574	P3 normal	-O0	✗	LLDB	46008	enhance	-Og	✗
GDB	90584	P3 normal	-O0	✗	LLDB	46009	enhance	-Og	✗
GDB	90586	P3 normal	-O0	✗	LLDB	46038	enhance	-Og	✗
GDB	95414	P3 normal	-O2	✓	LLDB	46074	enhance	-Og	✓
GDB	30357	**P2 critical**	-O0	✗	LLDB	46120	enhance	-Og	✗
LLDB	45386	enhance	-O1	✓	LLDB	46181	normal	-Og	✗
LLDB	45387	enhance	-O1	✓	LLDB	47093	enhance	-Og	✗
LLDB	45676	normal	-O0	✗	LLDB	47239	enhance	-Og	✗
LLDB	45920	normal	-O0	✗	LLDB	47257	normal	-Og	✓
LLDB	46006	normal	-O3	✓	LLDB	47273	enhance	-Og	✓
LLDB	55744	-	-O1	✗					
Overall				5/18	Overall				5/17

Table 6. Number of subject test programs with errors or timeouts during compilation, execution, and debugging.

Type	**Stage**	**GCC**	**LLVM**
Error	Compilation	36,241	37,543
	Execution	579	556
	Debugging	23	26
Timeout	Compilation (10s)	16	12
	Execution (10s)	35	28
	Debugging (60s)	867	899
Normal		7,354	6,051
Total		45,115	

Table 7. Average compilation, standalone execution, and stepping times in DEVIL for source and instruction levels.

Tool	**Stage**	**-O0**	**-Og**	**-O1**	**-O2**	**-O3**
GCC	Compilation	0.11s	0.12s	0.17s	0.14s	0.14s
	Standalone Execution	0.05s	0.01s	0.01s	0.01s	0.01s
	Source Level	1.70s	0.54s	0.49s	0.39s	0.36s
	Instruction Level	1.92s	1.10s	0.90s	0.71s	0.66s
Clang	Compilation	0.13s	0.15s	0.16s	0.15s	0.15s
	Standalone Execution	0.06s	0.01s	0.01s	0.01s	0.01s
	Source Level	1.78s	0.89s	0.89s	0.82s	0.82s
	Instruction Level	4.00s	2.19s	2.19s	1.82s	1.79s

each test program. Furthermore, the overhead associated with instruction-level debugging is typically higher than that of source-level debugging due to its finer granularity. Additionally, enabling higher levels of optimization leads to a reduction in execution time for DEVIL.

In general, the overhead associated with DEVIL is higher than that of standalone execution, mainly due to the methodology used in the experiments. Concretely, to comprehensively compare debugging traces between source-level and

instruction-level execution, we step through the entire program from the entry point within a debugger, which introduces overhead but remains manageable given the framework's goals. Notably, DEVIL also supports selective debugging, enabling focused analysis on specific modules or statements within test programs without exhaustive stepping.

6 Discussion

6.1 Manual Effort

In this study, we conducted a continuous bug-hunting campaign across various trunk versions of the debugger toolchains. Typically, DEVIL produces approximately 10 inconsistency reports per test program when bugs are triggered, as a single bug can result in multiple incorrect variable values or execution orders. For each test program exhibiting inconsistencies, we manually inspect only the first detected occurrence. Our analysis begins by reviewing this issue and individually submitting reports to the developers. After the developers address a reported issue, we re-run the program using the updated version of the toolchains to verify whether the remaining inconsistencies identified in the same test program persist. If these inconsistencies are resolved, it indicates they were attributable to the same underlying issue. This approach minimizes duplicate bug reports in the issue trackers. Regarding false positives, the primary scenario encounter with DEVIL arises when a variable remains uninitialized at certain points within the program. In such cases, debuggers may access this uninitialized variable, resulting in the generation of random values. Fortunately, this situation has minimal impact on the cost of manual analysis, as it is generally straightforward to identify which variables lack initialization. In summary, the manual effort required for DEVIL remains manageable.

6.2 Impact of Compiler Optimization

In practice, compiler optimizations can significantly alter the input code, such as eliminating certain variables or reordering the execution of instructions. Consequently, false positives may arise when comparing traces produced by different optimization levels. For instance, Figure 5 illustrates a bug report from prior research. Concretely, for the same program $\mathcal{P}_4$ compiled with different optimization levels, `-O1` and `-O2`, GDB displays inconsistent values for the array element `a[0][0][0]`. However, according to developers' comments, this inconsistency does not signify a genuine bug, and the developers consider this behavior a typical outcome of the optimization process. In particular, the compiler optimizes large arrays by identifying unused elements and may opt not to initialize them, leading to these inconsistencies. In contrast, DEVIL avoids this issue by comparing execution traces of identical executables generated at a fixed optimization level, while varying only the debugging levels. Thus, our method significantly decreases the incidence of such false positives induced by compiler optimizations.

```c
int d, e, f, h, l;
short b, g, i, m;
void n() {
  char o;
  int p = 0, j, k;
  for (; i <= 1; i++) {
    unsigned short a[6][2][2];
    l = 0;
    for (; l < 6; l++) {
      j = 0;
      for (; j < 2; j++) {
        k = 0;
        for (; k < 2; k++)
          a[l][j][k] = 61344;
      }
    }
    for (; f <= 1; f++)
      if (d)
        ...
    if (b)
      h = a[4][1][0];
  }
}
int main() {
  n();
}
```

(a) $\mathcal{P}_4$

```
$ gcc -O1 -g a.c
$ gdb a.c:20
(gdb) b 20
Breakpoint 1 at a.c:20.
(gdb) r
20          if (b)
(gdb) p a[0][0][0]
$1 = 61344
```

(b) Compilation with `-O1`

```
$ gcc -O2 -g a.c
$ gdb a.c:20
(gdb) b 20
Breakpoint 1 at a.c:20.
(gdb) r
20          if (b)
(gdb) p a[0][0][0]
$1 = 3
```

(c) Compilation with `-O2`

Figure 5. GCC bug#92417, marked as 'WONTFIX' by developers. This issue highlights an expected inconsistency in array values between optimization levels `-O1` (b) and `-O2` (c).

6.3 Threats to Validity

One key threat to the validity of our study is the appropriateness of the subject programs used in the experiments. Specifically, employing overly complex test programs can pose substantial challenges in analyzing relevant issues, even though they may effectively stress-test the toolchains. Therefore, we selected test programs from the GCC test suite as our subjects, given their extensive coverage of C semantics [18] and their widespread use in testing compilers [4]. Moreover, these programs are of an appropriate size for thorough inspection and bug reporting. Another threat to the

validity is the exclusion of test programs with execution time exceeding ten seconds. Our objective is to expose bugs in debuggers rather than waiting for the completion of all the test programs. Therefore, this exclusion can be considered acceptable. Nevertheless, it is important to note that test programs with long execution times may potentially trigger performance bugs in the debugger, which should be taken into account for further investigation.

6.4 Future Work

In this work, we selected test programs from the GCC test suite as subjects and have identified 18 confirmed bugs in debuggers. Nevertheless, incorporating a broader range of test programs is supposed to reveal additional bugs. For instance, we could integrate fuzzing techniques to generate test programs (e.g., Csmith [16] and YARPGen [9]) or employ real-world C programs as seeds for Devil. Devil currently supports forward execution to a randomly selected program location, followed by stepping through each subsequent statement or instruction. While exhaustively forwarding execution to every program location is effective, it incurs significant overhead. To mitigate Devil's overhead, we propose exploring the following potential heuristics:

- **Input Binary Selection**: By analyzing debug information from binaries, we can identify instances where multiple instructions fail to map to specific source lines or are associated with common lines. This analysis helps determine whether exhaustive stepping is necessary and facilitates stress-testing of toolchains.
- **Important Code Identification**: This approach focuses on code segments that are likely to trigger bugs, analyzing function complexities and extracting relevant semantics from historical bug-triggering test programs. This method allows us to prioritize areas that require closer inspection.
- **Coverage-Guided Fuzzing**: This technique leverages the code coverage of debuggers to guide the fuzzing process. Specifically, if no new branches or statements are covered within the debugger during debugging, we can skip stepping for that function, reducing unnecessary overhead.

6.5 Limitation

While Devil offers several advantages, it is not without its limitations. First, manual effort is still required to inspect violations and determine whether they constitute bugs that need to be reported. To mitigate this, we can employ a duplicate cluster analysis technique, as in previous work [3], to group likely related violations and reduce redundant inspection efforts. Second, Devil can only manifest bugs, unable to pinpoint their root cause. To address this, a practical approach to locating the root cause is to identify the bug-introduced commit using `git bisect`. We plan to incorporate this function into our prototype, enhancing its capabilities and providing more valuable insights to users. Third, generalizing our approach to interpreted programming languages poses challenges due to the lack of debuggers that support various debugging levels. Specifically, debuggers such as rust-lldb and JDB for Rust and Java, respectively, offer support for both `step` and `stepi` commands, making them compatible with Devil. However, adapting CLD to interpreted programming languages like Python remains challenging. These limitations should be considered when applying Devil and exploring its applicability in different contexts.

7 Related Work

7.1 Debugger Testing

Recent research has proposed various techniques for testing and validating debuggers. Lehmann and Pradel [7] developed a differential testing approach called DBDB to identify bugs in JavaScript debuggers by comparing their behaviors. Tolksdorf et al. [15] used metamorphic testing to validate JavaScript debuggers by transforming the debugged code. Li et al. [8] and Di Luna et al. [3] proposed techniques to detect bugs related to compiler optimizations by inspecting debug information. These existing approaches, however, are limited to specific types of debugger or compiler bugs. In contrast, Devil provides a more comprehensive testing approach capable of identifying genuine correctness bugs in both debuggers and compilers, regardless of optimization.

7.2 Compiler Validation

Significant research has been conducted on ensuring the correctness of compilers [11, 12, 19]. In terms of compiler testing, generation-based approaches utilize program generators like Csmith [16] and YARPGen [9, 10] to create diverse test programs for differential testing across compilers, optimizations, and versions. Mutation-based approaches, such as Orion [4] and Athena [5], generate semantics-preserving test programs by randomly mutating non-executed code. Unlike these existing techniques, Devil focuses on exposing bugs associated with debug information, which can lead to erroneous debugging behaviors. This complements the existing compiler testing studies.

8 Conclusion

In this study, we introduce a novel concept called *Cross-Level Debugging* (CLD), which serves as a method for testing debugger toolchains. Our key insight is that traces obtained at different levels of debugging should adhere to specific constraints. To implement CLD, we developed the prototype tool, named Devil, and applied CLD to validate the GDB and LLDB toolchains. Using Devil, we successfully identified and exposed a total of 27 bugs, with 18 of them having been confirmed or addressed by the developers. These results underscore the efficacy of Devil in validating debugger toolchains. Overall, our study offers a fresh perspective on the testing of debugger toolchains.

Acknowledgments

We are grateful to the anonymous reviewers and our shepherd, Santosh Nagarakatte, for their insightful suggestions. We thank the GCC and LLVM developers, particularly Tom de Vries, for inspecting and fixing our reported bugs. We appreciate Yanyan Jiang and Zhiqiang Zuo for their valuable feedback. Yuming Zhou and Maolin Sun are the corresponding authors. This work is partially supported by the National Natural Science Foundation of China (Grants 62072194, 624B2067, and 62172205), the Jiangsu Natural Science Foundation under Grant BK20231402, the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Fundamental Research Funds for the Central Universities (14380121).

A Artifact Appendix

A.1 Abstract

The artifact contains the code and datasets we used for our experiments, as well as scripts to generate the numbers and tables of our evaluation. Specifically, it includes (a) links and bug-triggering test cases of each reported bug; (b) seed programs from the GCC test suite used for evaluation; (c) scripts for analyzing distribution of test programs in dataset; (d) scripts for analyzing running overhead of Devil; and (e) detailed instruction documentation for using Devil. Everything is packaged and pre-built as a docker image.

A.2 Artifact check-list (meta-information)

- **Run-time environment:** Linux
- **Hardware:** X86
- **How much disk space required (approximately)?:** 20GB
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes to download and import the Docker image.
- **How much time is needed to complete experiments (approximately)?:** 2~3 hours (~30 processes in parallel)
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache 2.0
- **Archived (provide DOI)?:** Yes

A.3 Description

The artifact can be downloaded from the following link: https://doi.org/10.5281/zenodo.14053328

A.3.1 Hardware dependencies.

A standard X86 machine.

A.3.2 Software dependencies.

Docker

A.4 Installation

The Docker image is provided in a pre-configured format, obviating the need for any installation. The following commands can be employed to extract the artifact archive and import it into Docker:

```
$ gunzip -c devil.tar.gz > devil.tar
$ cat devil.tar | docker import - devil
```

A.5 Experiment workflow

1. **Read the documentation:** Documentation is available online and can be accessed at this link.
2. **Start the docker container as instructed:** A pre-configured Docker image is provided on the Zenodo repository. Alternatively, the image can be downloaded from Docker Hub. Detailed instructions for setup are included in the online documentation.
3. **Check bug reports:** The documented bugs identified in the GDB and LLDB toolchains can be reviewed here.
4. **Reproduce experimental results:** The reproduced results primarily pertain to Section 5.5. Other results presented in the paper are derived through manual analysis and therefore cannot be reproduced with scripts.

A.6 Evaluation and expected results

We provide the necessary dataset and scripts to reproduce the evaluation results presented in Section 5. Specifically, the results shown in Tables 6 and 7 can be reproduced using the provided resources. For further details, please refer to the accompanying website or archive.

References

[1] Cristian Assaiante, Daniele Cono D'Elia, Giuseppe Antonio Di Luna, and Leonardo Querzoni. 2023. Where Did My Variable Go? Poking Holes in Incomplete Debug Information. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 935–947. https://doi.org/10.1145/3575693.3575720

[2] Sangeeta Chowdhary, Jay P. Lim, and Santosh Nagarakatte. 2020. Debugging and detecting numerical errors in computation with posits. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 731–746. https://doi.org/10.1145/3385412.3386004

[3] Giuseppe Antonio Di Luna, Davide Italiano, Luca Massarelli, Sebastian Österlund, Cristiano Giuffrida, and Leonardo Querzoni. 2021. Who's Debugging the Debuggers? Exposing Debug Information Bugs in Optimized Binaries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 1034–1045. https://doi.org/10.1145/3445814.3446695

[4] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 216–226. https://doi.org/10.1145/2594291.2594334

[5] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 386–399. https://doi.org/10.1145/2814270.2814319

[6] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Randomized Stress-Testing of Link-Time Optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 327–337. https://doi.org/10.1145/2771783.2771785

[7] Daniel Lehmann and Michael Pradel. 2018. Feedback-Directed Differential Testing of Interactive Debuggers. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 610–620. https://doi.org/10.1145/3236024.3236037

[8] Yuanbo Li, Shuo Ding, Qirun Zhang, and Davide Italiano. 2020. Debug Information Validation for Optimized Code. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1052–1065. https://doi.org/10.1145/3385412.3386020

[9] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (Nov. 2020), 25 pages. https://doi.org/10.1145/3428264

[10] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages. *Proc. ACM Program. Lang.* 7, PLDI, Article 181 (June 2023), 22 pages. https://doi.org/10.1145/3591295

[11] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 22–32. https://doi.org/10.1145/2737924.2737965

[12] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2018. Practical verification of peephole optimizations with Alive. *Commun. ACM* 61, 2 (Jan. 2018), 84–91. https://doi.org/10.1145/3166064

[13] Y. N. Srikant and Priti Shankar. 2002. *Compiler Design Handbook: Optimizations and Machine Code Generation.* CRC Press, Inc., USA.

[14] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and Analyzing Compiler Warning Defects. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 203–213. https://doi.org/10.1145/2884781.2884879

[15] Sandro Tolksdorf, Daniel Lehmann, and Michael Pradel. 2019. Interactive metamorphic testing of debuggers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 273–283. https://doi.org/10.1145/3293882.3330567

[16] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[17] Yibiao Yang, Yanyan Jiang, Zhiqiang Zuo, Yang Wang, Hao Sun, Hongmin Lu, Yuming Zhou, and Baowen Xu. 2020. Automatic self-validation for code coverage profilers. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 79–90. https://doi.org/10.1109/ASE.2019.00018

[18] Yibiao Yang, Yuming Zhou, Hao Sun, Zhendong Su, Zhiqiang Zuo, Lei Xu, and Baowen Xu. 2019. Hunting for bugs in code coverage tools via randomized differential testing. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 488–499. https://doi.org/10.1109/ICSE.2019.00061

[19] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2013. Formal verification of SSA-based optimizations for LLVM. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 175–186. https://doi.org/10.1145/2491956.2462164