DTD: Comprehensive and Scalable Testing for Debuggers

HONGYI LU, Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China, Department of Computer Science and Engineering, Southern University of Science and Technology, China, and Hong Kong University of Science and Technology, China
ZHIBO LIU*, Hong Kong University of Science and Technology, China
SHUAI WANG, Hong Kong University of Science and Technology, China
FENGWEI ZHANG*, Department of Computer Science and Engineering, Southern University of Science and Technology, China and Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China

As a powerful tool for developers, interactive debuggers help locate and fix errors in software. By using debugging information included in binaries, debuggers can retrieve necessary program states about the program. Unlike `printf`-style debugging, debuggers allow for more flexible inspection and modification of program execution states. However, debuggers may incorrectly retrieve and interpret program execution, causing confusion and hindering the debugging process.

Despite the wide usage of interactive debuggers, *a scalable and comprehensive measurement of their functionality correctness* does not exist yet. Existing works either fall short in scalability or focus more on the "compiler-side" defects instead of debugger bugs. To facilitate a better assessment of debugger correctness, we first propose and advocate a set of debugger testing criteria, covering both comprehensiveness (in terms of debug information covered) and scalability (in terms of testing overhead). Moreover, we design comparative experiments to show that fulfilling these criteria is not only theoretically appealing, but also brings major improvement to debugger testing. Furthermore, based on these criteria, we present DTD, a differential testing (DT) framework for detecting bugs in interactive debuggers. DTD compares the behaviors of two mainstream debuggers when processing an identical C executable — discrepancies indicate bugs in one of the two debuggers.

DTD leverages a novel heuristic method to avoid the repetitive structures (e.g., loops) that exist in C programs, which facilitates DTD to achieve full debug information coverage efficiently. Moreover, we have also designed a Temporal Differential Filtering method to practically filter out the false positives caused by the uninitialized variables in common C programs. With these carefully designed techniques, DTD fulfills our proposed testing requirements and, therefore, achieves high scalability and testing comprehensiveness. For the first time, it offers large-scale testing for C debuggers to detect debugger behavior discrepancies when inspecting millions of program states. An empirical comparison shows that DTD finds 17× more error-triggering cases and detects 5× more bugs than the state-of-the-art debugger testing technique. We have used

*Zhibo Liu and Fengwei Zhang are the correspondent authors.

Authors' addresses: Hongyi Lu, Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China and Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China and Hong Kong University of Science and Technology, Hong Kong, China, hluaw@cse.ust.hk; Zhibo Liu, Hong Kong University of Science and Technology, Hong Kong, China, zliudc@cse.ust.hk; Shuai Wang, Hong Kong University of Science and Technology, Hong Kong, China, shuaiw@cse.ust.hk; Fengwei Zhang, Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China and Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen, China, zhangfw@sustech.edu.cn.

ACM 2994-970X/2024/7-ART53
https://doi.org/10.1145/3643779

DTD to detect 13 bugs in the LLVM toolchain (Clang/LLDB) and 5 bugs in the GNU toolchain (GCC/GDB). One of our fixes has already landed in the latest LLDB development branch.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Differential Testing, Debug Information

ACM Reference Format:
Hongyi Lu, Zhibo Liu, Shuai Wang, and Fengwei Zhang. 2024. DTD: Comprehensive and Scalable Testing for Debuggers. *Proc. ACM Softw. Eng.* 1, FSE, Article 53 (July 2024), 22 pages. https://doi.org/10.1145/3643779

1 INTRODUCTION

Debugging is a crucial step in software development — it helps developers locate and fix errors. Interactive debuggers are potent tools for this purpose, as they allow users to pause program execution, modify the program state, and track the program's execution via stepping. Interactive debuggers are especially important for large-scale software that are complex and harder to diagnose using simple `printf`-style debugging.

Software debuggers help developers to analyze and fix software defects. It is thus essential to *measure the correctness of the debugger itself*, as a faulty debugger can lead to incorrect results, wasted time, and neglect of software bugs. Fig. 1 shows a surprisingly simple debugger bug in the latest version of LLDB [LLVM 2023] — the source is compiled without any optimization (i.e., `-O0`).

In Fig. 1, the LLDB pauses the execution of the program at line 6 and prints out `g`. At this point, `bad_f` should be `1795821`, its initialized value. However, due to an underflow bug in LLDB, its value is wrongly displayed as `0`. This bug affects the latest version of LLDB (we also confirmed its presence in other LLDB versions). Overall, debugger bugs like the one in Fig. 1 are very confusing. They can lead developers to believe their code is wrong even when it is not. Worse still, debugger bugs can make it hard or impossible to identify actual software bugs. This paper aims to uncover such debugger defects automatically and efficiently.

We have noticed a line of research works [Lehmann and Pradel 2018; Tolksdorf et al. 2019] that focus on testing debuggers. Furthermore, similar works [Assaiante et al. 2023; Di Luna et al. 2021; Li et al. 2020] also detect compiler bugs that fail to correctly attach debugging symbols in executables. Overall, most existing works follow a differential testing (DT) [Chen et al. 2019, 2016; Li and Su 2023; McKeeman 1998; Rigger and Su 2020; Theodoridis et al. 2022] paradigm, such that two debuggers are tested by checking the *consistency* of their retrieved program states. Inconsistently retrieved program states indicate a bug in one of the two debuggers.

Existing approaches in testing debuggers mainly focus on JavaScript (JS) debuggers, a scenario that is generally distinct from testing C debuggers (Sec. 3.2). Despite the success of those approaches, we believe they fall short in terms of *scalability* when considering C debuggers. JavaScript programs tested in these works are only of a few hundred lines, but real-world C programs can easily have over ten thousand or even millions of lines of assembly code. In other words, a naive comparison of program states obtained by debuggers can hardly scale when launching comprehensive testing with real-world C programs. This scalability issue is often addressed by only checking a subset of debug information. For example, a recent work [Di Luna et al. 2021] in testing C debuggers only collects and compares the function inputs, providing limited coverage of program data and control states. We note that nearly all defects uncovered by us from the latest versions of GDB and LLDB (Sec. 6) are *not* related to the callsite data facts and cannot be exposed by the prior work [Di Luna et al. 2021]. Moreover, current approaches to testing C debuggers more focus on uncovering "compiler-side" defects rather than debugger bugs, making them unable to detect trivial bugs (e.g., bug in Fig. 1) that are *not* caused by compiler. In sum, existing works on testing debuggers either suffer from

```
Process 104245 stopped
* thread #1, name = 'a.out', stop reason = breakpoint 2.1
frame #0: 0x000055555555515b akane.out`main at r.c:6:13
1    #pragma pack(1)
2    struct {
3      signed f : 27;
4      unsigned bad_f : 30;
5    } g = {5070, 1795821};
-> 6    int main() {return 0;}
(lldb) p g
((unnamed struct)) $0 = (f = 5070, bad_f = 0)
```

Fig. 1. A simple bug in LLDB found by DTD.

low scalability (can only use small test cases) or insufficient comprehensiveness (can only check a subset of debug information).

In this paper, we propose a DT-based framework, DTD, to test C debuggers. DTD explores a novel and carefully calibrated balance between scalability and comprehensiveness, enabling for the first time comprehensive and scalable testing of C debuggers. DTD is equipped with various optimizations; it offers a principled guarantee to take into account full program states (including both control and data facts obtained during program execution), all DWARF entries, and all executed unique instructions. This unprecedented comprehensive and speedy approach empowers DTD to detect bugs that have long been hidden in production-level C debuggers.

We implement DTD targeting two mainstream C/C++ debuggers: GDB and LLDB. DTD uses about 56,000 input programs to test GDB/LLDB, and during approximately 27 hours of testing, DTD compares 15 billion program states and found 33,134 error-triggering programs. This promising bug detection capability comes from 100% debug information coverage. In contrast, the state-of-the-art works [Di Luna et al. 2021] in this field only reach 2.5%, and thus DTD is 40x more effective in terms of debug information coverage (see Sec. 6.3).

The discovered defects lead to incorrect debugging results and raise high confusion among users. We conduct detailed analyses of the discovered bugs and summarize their characteristics. We submitted all of our findings to the developers of GDB/LLDB. As of this writing, 13 bugs have been confirmed. In sum, we make the following contributions:

- We target a crucial yet under-explored need to measure the correctness of debuggers. This paper offers a highly scalable testing framework that, for the first time, can comprehensively test C debuggers.
- Our framework, DTD, conducts a DT-based testing campaign. DTD incorporate various design principles and optimizations to largely boost testing while preserving high coverage over program states and debug information.
- We test industry-leading debuggers, GDB and LLDB, and find over 33K error-triggering programs. The flagged defects are critical and can largely mislead the user's understanding in the debugging process. Several bugs have been promptly confirmed and fixed by the developers.

We will release and maintain the codebase of DTD at [DTD 2023] to facilitate future research.

2 PRELIMINARY

An interactive debugger allows developers to pause the execution of their code at specific points with breakpoints, check the program's state, and step through the code. Below is a two-step approach to how an interactive debugger works:

Preparing Debug Information. Compilers automatically generate debug information when creating executables. The debugger then queries this debug information to offer useful insights during runtime. The debug information is usually attached to the executable output with dedicated

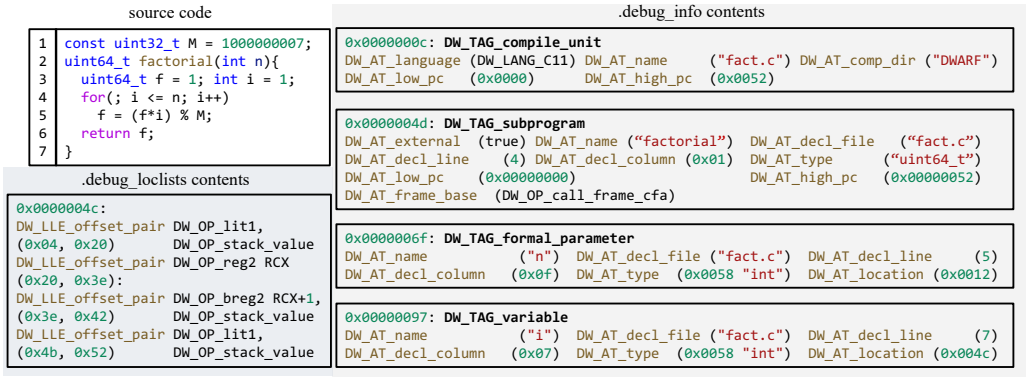

Fig. 2. Example of DWARF debug information.

sections or separate files. Depending on the OS and compiler, debug information may be stored in various formats, such as DWARF, PDB, or STABS. The DWARF format, however, is the most commonly used across compilers and debuggers.

Fig. 2 shows the DWARF debug information of an executable compiled from a simple C program that computes the factorial. The DWARF debug information contains a tree-like structure, where each node represents a source-level scope. Such debug information facilitates the debugger to map the data and instructions, from binary offset 0x00 ("`DW_AT_low_pc`") to 0x52 ("`DW_AT_high_pc`"), in the binary file to the source code (`fact.c`). Note that in such mappings, the debugger uses the *program counter* value as the index; for each unique PC value, only one set of debug information entries is used. To iterate the entire DWARF debug information, the debugger does not need to cover every *executed* instruction (which might be significantly bloated due to loops) but only instructions with unique PC values. The root node "`DW_TAG_compile_unit`" (stored in the `.debug_info` section) represents the entire program. One of its child nodes, "`DW_TAG_subprogram`," represents a function inside the program ("`factorial(int n)`"), and the "`DW_TAG_variable`" and "`DW_TAG_formal_parameter`" nodes represent variables and parameters of this function.

When compiler optimizations are enabled, DWARF debug information will be altered according to the optimizations applied. For example, if a local variable `i` is optimized and allocated to registers, such modification and the corresponding registers should be precisely recorded in the debug information (`.debug_loclists` section) as location entries ("`DW_LLE_offset_pair`"). The "`DW_AT_location`" item of the variable `i` will point to its optimized locations, which maps program address ranges to lists of DWARF operators indicating how debuggers could retrieve variable values. Note that these retrieved variable values are not *static* values but dynamic expressions of runtime values. Thus, though the value of `i` differs at each iteration, the debugger still parses the same DWARF entry and retrieves the correct value of `i` by calculating the same expression. Existing works [Assaiante et al. 2023; Di Luna et al. 2021; Li et al. 2020] show that optimizations may impede compilers from generating correct DWARF information. We show that optimizations also stress debuggers. Optimization generates corner case DWARF symbols, which are often misinterpreted due to debugger defects (Sec. 6).

Runtime Debugging. Debuggers replace certain instructions with software interrupt instructions (e.g., `int3` on x86) to inspect program runtime information. In particular, the debugger process first attaches to the target process via system calls (e.g., `ptrace` on Linux). For software breakpoints on x86, the debugger replaces the first byte of the instruction with `int3`. Then, when the CPU launches the target process and reaches the breakpoint, the CPU emits a debug exception handled by the OS and notifies the debugger process.

At each breakpoint, the debugger can inspect and modify the call stack and registers. Moreover, by reading debug information, the debugger can map the instruction address to the source code line, and likewise, it can also map the register/memory address to the variable names. This way, the debugger can provide useful information for developers to understand the process execution. As a convention, a variable is deemed "visible" at a line if it can be accessed at that line. For instance, the variable `i` is visible in line 6 in Figure 2.

3 MOTIVATION AND DTD OVERVIEW

3.1 Debugger Testing Requirements

Comprehensiveness. Debuggers rely on debugging information (often in the form of DWARF entries) in executables to retrieve program states. Therefore, to comprehensively expose possible flaws, debugger testing methods should form their testing oracles by properly considering both the program execution state and DWARF entry coverage. More precisely, we propose the following criteria to depict the comprehensiveness of a debugger testing approach.

- **P**rogram State Coverage, denoting the proportion of program states being captured by the testing oracle when executing each instruction, including source information (e.g., source line number), variable values, and register values. Ideally, the testing oracle should inspect the debugger behavior when retrieving full program states at each executed instruction.
- **E**xecuted Instruction Coverage, denoting the proportion of executed instructions examined by the testing oracle. Executing common C programs often cover millions of instructions, and ideally, the testing oracle should inspect the debugger behavior at each executed instruction.
- **D**ebug Information Coverage, denoting the proportion of debug information covered by the testing oracle. Ideally, the testing oracle should cover every DWARF entry at each executed instruction.

Scalability. Comprehensive debugger testing should achieve full coverage on both program states (**P**) and executed instructions (**E**), which consequently results in full coverage of debug information (**P+E→D**). However, this approach is too expensive and impractical for common C code (see Sec. 6). Thus, we further define a practical *scalability* requirement.

- **S**calability, requiring that the overhead of testing grows *linearly* with the test case scale, which includes state complexity M (e.g., #variables) and time complexity N (e.g., #executed instructions).

It is easy to see that "comprehensiveness" can likely conflict with "scalability." Intuitively, as a program's scale increases, it is likely that it will declare more variables and execute more instructions at the same time. Thus, to achieve **P** and **E**, the testing overhead is $O(MN)$ which is approximately quadratic to the program's scale and therefore violates scalability (**S**).

We propose a weaker version of **E** as follows. In Sec. 4.1, we explain that the following requirement **WE** can still facilitate obtaining full debug information coverage (**D**). That is, **P+WE →D**.

- **W**eak **E**xecuted Instruction Coverage, denoting the proportion of executed *unique* instructions that are taken into account by the testing oracle.

Achieving full **WE** no longer needs to inspect the program states after a repeatedly executed instruction. Note that such repeated instructions account for most executed instructions (see Sec. 4.1) but do not result in new coverage of debug information (**D**). Note that the number of *unique* instructions is almost constant with respect to the execution cost of the program. Therefore, by focusing on **WE** instead of **E**, DTD not only achieves full **D** but also satisfies **S** simultaneously.

Later in Sec. 6.3, through an empirical comparison between DTD and current state of the art in testing debugger [Di Luna et al. 2021], we show that satisfying the above testing requirements is not just theoretically appealing, but also practically enhances DTD's capability for bug detection.

3.2 Review of Existing Approaches

Table 1 reviews existing approaches in this field. In general, defects that impede the interactive debugging process may root from two sources: ① erroneous debug information emitted by the compiler during compilation and ② erroneous behaviors exhibited by the debuggers during runtime. Accordingly, a line of research [Assaiante et al. 2023; Di Luna et al. 2021; Li et al. 2020] test or verify compilers and optimization passes, aiming to check if the compiler or its optimization passes generate incorrect debug information. In contrast, DTD shares the same goal as existing approaches [Lehmann and Pradel 2018; Tolksdorf et al. 2019] that focus on testing debuggers, i.e., to check if the debuggers behave correctly during runtime.[1]

Table 1. Comparison with existing works.

	Focus	Tech. category	Target language	Comprehensive	Scalable
DBDB [Lehmann and Pradel 2018]	Debuggers	DT	JavaScript	**P**	?
Tolksdorf et al. [2019]	Debuggers	MT	JavaScript	**P**	?
Debug^2 [Di Luna et al. 2021]	Compiler/Opt.	DT	C/Rust	**E**	✓
Assaiante et al. [2023]	Compiler/Opt.	Static inference	C	**N.A.**	✓
Li et al. [2020]	Compiler/Opt.	DT	C/Rust	**N.A.**	✓
DTD	Debuggers	DT	C	**P**,**D**,**WE**	✓

Despite sharing the same scope, prior works [Lehmann and Pradel 2018; Tolksdorf et al. 2019] are limited to JS debuggers, a generally distinct scenario from testing C/C++ debuggers. Specifically, DBDB [Lehmann and Pradel 2018] adopts a DT approach by comparing two JS debuggers (Chromium and Firefox) to locate discrepancies. Tolksdorf et al. [2019], on the other hand, proposed a metamorphic testing MT-based [Chen et al. 2020a] approach which mutates debugger inputs and examines the consistency of debugger outputs. Both works randomly "skip" a fair proportion of instructions without inspecting the program states. They only obtain full program state coverage (**P**) at a limited number of instructions, thus failing **E** and **D**. Moreover, the test cases used in [Lehmann and Pradel 2018; Tolksdorf et al. 2019] are rather simple JS programs. It is unclear if this method scales well on large, complicated C/C++ programs. To avoid overclaiming, we mark their scalability as "**?**" in Table 1.

There are also works [Assaiante et al. 2023; Li et al. 2020] focusing on checking if compilers and optimization passes correctly insert debug information in C executables. While they also use DT (and static inference), these works typically only checks the debug information at specific points instead of comprehensively going through every aspects of it. For instance, Li et al. [2020] propose a testing method that checks the consistency of variables between the optimized and unoptimized executables by inserting `printf` statements. This can only be applied to specifically designed programs and fails to generalize. Assaiante et al. [2023] propose a static inference method that only checks whether the compiler spuriously optimize certain variables without checking the correctness of its value. Therefore, we mark their comprehensiveness as "**N.A.**" in Table 1 to avoid confusion.

Debug^2 [Di Luna et al. 2021], on the other hand, generalizes the idea of previous works [Li et al. 2020], inspecting not only the consistency of variable values, but also other aspects like source line number and backtraces. Specifically, it proposes invariants that the compiler should hold when optimizing an executable, and performs DT to find any violation of these invariants. However, when checking the consistency of variables, Debug^2 only cross-compares callsite data facts (i.e., function arguments). Technically speaking, its method successfully achieves **E** and **S** simultaneously, at the cost of failing **P**. Nevertheless, inspecting only callsite data facts significantly constrains the bug detection capability of Debug^2, causing it to ignore a considerable amount of

[1]Nevertheless, when investigating our findings, we did trace the root cause of some defects to the compiler; see Sec. 6.1.

debugger defects as long as those defects do not affect callsite data facts. Moreover, the invariants proposed by Debug^2 only apply to the comparison between optimized and unoptimized binaries. As a result, most defects uncovered by Debug^2 are related to the compiler instead of the debugger. In contrast, DTD launches a comprehensive and scalable testing campaign across different debugger implementations, which focuses on uncovering defects in debuggers themselves.

In Sec. 6.3, we will present a series of empirical comparisons to show how DTD outperforms Debug^2 by satisfying all three coverage requirements simultaneously (i.e., **P**, **D**, **WE**). Moreover, we will show that DTD is *not* merely an incremental work based on Debug^2, but solves its unattended challenges, leading to both higher testing coverage and superior bug detection capability.

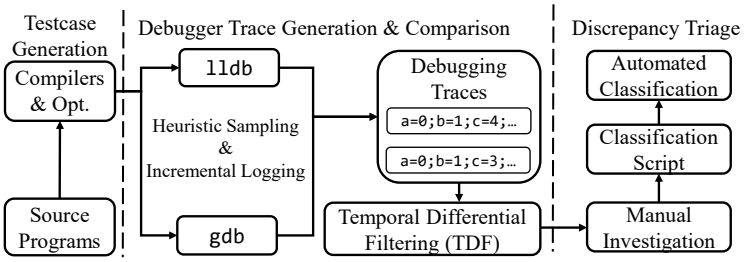

Fig. 3. Testing workflow of DTD.

3.3 DTD Overview

Fig. 3 depicts the testing workflow of DTD, which mainly encompasses three stages: test case generation, debugger trace generation, and discrepancy triage.

Test Case Generation. DTD leverages the well-known random program generator, Csmith [Yang et al. 2011], to generate a set of C programs as test inputs. These programs are then compiled using different toolchains and different configuration settings. The executables are the test inputs we use to test debuggers. Note that, unlike previous works [Di Luna et al. 2021; Li et al. 2020] focusing on the behavioral differences of the *compiler* in different optimization levels, DTD focuses on the defects of the *debugger* itself. Therefore, instead of comparing the optimized binaries against the unoptimized ones like in previous works [Di Luna et al. 2021; Li et al. 2020], DTD looks for discrepancies between two debuggers over the *same* binary.

Debugger Trace Generation & Comparison. In this work, we mainly test two mainstream C debuggers, LLDB and GDB (see Sec. 8 on extending DTD to other settings). Given an input binary B, DTD employs two debuggers to step through B to generate two traces, T_{GDB} and T_{LLDB}, respectively. To ensure comprehensively covering executed instructions (criterion **E** in Sec. 3.1), we use `stepi` (i.e., step instruction) instead of `step` (i.e., step source code line) to step through every assembly instruction. Specifically, let the input executable be $B = \{inst_n\}$, where $inst_k$ is the kth executed assembly instruction.[2] DTD collects a trace T where each element $s_k \in T$ includes the following program state information (including both control and data facts) retrieved by the debugger.

(1) Control Fact. Given an instruction $inst_k$, DTD asks the debugger to provide the program counter (PC) register `pc` and the source code line number $line_k \in \mathbb{N} \cup \{\top\}$ associated with $inst_k$. These two values form the control fact collected by DTD at $inst_k$. Here, $\top$ indicates that debugger is unable to give line number either due to optimization or a debugger bug.

(2) Data Fact—Variable. Variable values var_k denote crucial data facts a debugger can provide at $inst_k$. For a variable $v \in var_k$, we encode its value as $value(v) \in \mathbb{V} \cup \{\top, err, nil\}$. $\top$ indicates

[2]We "misuse" the concept of static binary form and dynamic execution trace to ease presentation. To clarify, each test input (Csmith programs) used in our study does not need explicit user inputs, and its execution is deterministic and always follows the same path.

that the value of v is optimized out, whereas the variable symbol is still visible. *err* means that an error is encountered when evaluating $value(v)$, despite that v is still visible. Unlike $\top$ and *err*, *nil* means that the variable is not found in the first place. Unlike previous works [Di Luna et al. 2021], which view *err*, *nil* and $\top$ as equivalent states, DTD treats them as distinct states as these "symptoms" might be caused by different bugs.

(3) Data Fact—Register. Further to variable-level data facts, DTD also collects register-level data facts. Particularly, we collect the values of all status registers and general registers (e.g., `%eax`) from both tested debuggers.

In sum, DTD fully checks the program states when testing debuggers, and thus satisfies the coverage criteria **P**. After two traces, T_{GDB} and T_{LLDB}, are generated after executing B, DTD compares these two traces and looks for discrepancies. Since these two traces are generated when debugging the *same* binary B, they should be exactly the same. For example, if a variable v is $\top$ at $s_k \in T_{GDB}$, then in $s'_k \in T_{LLDB}$, v should also be $\top$. This way, DTD flags discrepancies between T_{GDB} and T_{LLDB}.

Discrepancy Triage. Given the discrepancies DTD found, we triage them manually with the help of C-Reduce [Regehr et al. 2012]. Unlike bugs found by previous works [Assaiante et al. 2023; Di Luna et al. 2021; Li et al. 2020] mainly caused by optimization passes of the compiler, the bugs discovered by DTD are often subtle defects in the debugger that do not have a clear concept of "compiler pass" to serve as fingerprints. This makes the whole triage process challenging. To address this issue, we adopt an automated classification process for discrepancies found by DTD. Specifically, for each discrepancy, we first investigate its characteristics. Based on these characteristics, we then design scripts to automatically classify discrepancies of the same type to avoid repetitively examining them by hand. This automated classification process significantly saves the manual effort of discrepancy triage. With automated classification, it takes about ten man-hours to manually summarize the characteristics of an untriaged bug, and no manual effort is required to triage an error-triggering case to a known bug, which could take about two man-hours before. All the characteristics we summarize are discussed in Sec. 6.

We clarify one critical point in the triage. For each test program P, we only analyze the first discrepancy occurrence on $T_{GDB}(P)$ and $T_{LLDB}(P)$. If comparing $T_{GDB}(P)$ and $T_{LLDB}(P)$ yields multiple discrepancies, we only analyze the first one. Our tentative study shows that once a discrepancy is found (indicating a debugger becomes buggy) on the trace, often a number of similar discrepancies are continuously found in the following part of the trace, which are verbose to analyze. More importantly, those discrepancies are likely caused by the same root cause (because two debuggers are already "unaligned" in the first discrepancy). As a result, we only analyze the first discrepancy and ignore the rest.

DWARF-oriented Debugger Testing. Real-world C programs can easily have over ten thousand or even millions of lines of assembly code, not to mention that the frequent use of loops in programs can further increase the number of program states at runtime. In other words, a naive comparison of program states obtained by two debuggers can hardly scale when launching comprehensive testing using a large number of C programs. However, previous works [Di Luna et al. 2021; Lehmann and Pradel 2018; Li et al. 2020; Tolksdorf et al. 2019] are essentially guided under program coverage, meaning they iterate every executed instruction in the runtime without the knowledge of the DWARF structure and thus bring unnecessary overhead due to repeatedly checking the same DWARF entries. Such overhead prevents the previous program-oriented method from obtaining full traces as it would be too costly to collect every datafact from every executed instruction. DTD, by contrast, leverages the knowledge that DWARF is, in fact, structured and referenced via the unique PC values in the executables (see Sec. 2). By collecting full datafacts at each unique PC value (see Sec. 4.1), DTD is able to parse all DWARF entries stored for that PC and avoids parsing the same

entries over and over again, which might occur in the program-oriented method as the instruction at a certain PC could be executed repeatedly due to loops. In Sec. 4, we present how DTD leverages this heuristic to achieve high scalability (the **S** requirement in Sec. 3.1) while preserving the **D** and **WE** requirements simultaneously.

Application Scope and Generalizability. As aforementioned, both compiler and debugger may introduce defects, impeding the debugging process. We clarify that DTD focuses on testing debuggers instead of compilers, whereas prior works [Assaiante et al. 2023; Di Luna et al. 2021; Li et al. 2020] have systematically tested compilers. As shown in Sec. 6, we uncover many debugger defects that are not found by previous approaches.

Furthermore, although DTD mainly focuses on testing debuggers, there are still chances that malformed DWARF information generated by the compiler leads to divergent behavior of two debuggers; DTD can also uncover such compiler bugs (see Sec. 6). To clarify, these bugs do not necessarily root from the buggy collaboration between the optimization process and the insertion of DWARF where previous works [Assaiante et al. 2023; Di Luna et al. 2021; Li et al. 2020] mainly focus, but the divergent understanding of the DWARF standard among compiler/debugger developers.

Our current testing procedure mainly focuses on two mainstream C debuggers, GDB and LLDB, and our evaluation is conducted on x86-64 architectures. Nevertheless, DTD is generalizable to other debuggers, programming languages, compilers, and architectures. We release the full codebase of DTD with documents to help others extend DTD. See Sec. 8 for more discussions on extensibility.

4 TECHNICAL PIPLINE OF DTD

The preceding section illustrates the overall workflow of DTD. Although the DT-based testing pipeline is generally straightforward, in practice, there are multiple non-trivial challenges for DTD to overcome. This section describes two key challenges and clarifies how DTD solves them in detail.

Table 2. Average execution trace lengths of Csmith test programs.

Opt. Level	O0	O1	O2	O3
Trace Length	59,036	41,806	17,493	16,064

4.1 C1: Trace Complexity

The trace T complexity originates from two main factors: the significant length of a single trace T and the intricacy of its logged program states. First, as indicated in Table 2, even though the average line of code (LoC) in our test case programs is around 255 (our test cases are Csmith programs), the span of their execution traces exceeds this number by far. Particularly, an unoptimized program's average trace length can reach approximately 200 times greater than its average source code length, as shown in the second column of Table 2. Second, the complexity of data facts within it also poses a challenge for DTD to conduct comprehensive testing. For instance, while the average number of global variables in the Csmith test case programs is about 30, composite data types like arrays and structures notably inflate this number to over 500.

In sum, these two factors render it impractical to gather every state of a trace T for comparison directly. Our tentative study shows that naively comparing the trace of a small-sized C program took over 30 minutes and necessitated approximately 1 GiB storage space, which is highly costly and not acceptable for a large-scale testing campaign (our 56,000 test cases could take months and over 50 TiB to finish). During our tentative experiments, we notice that the traces actually comprise many redundant patterns owing to control structures like loops. As noted in Sec. 2, DWARF information is organized in terms of lexical blocks such as `DW_TAG_subprogram` with its designated `pc` ranges. Therefore, even if a repetitive pattern appears several times in a trace, its associated DWARF information remains identical. As a result, most states in a trace are redundant, and thoroughly examining them brings no benefit for testing.

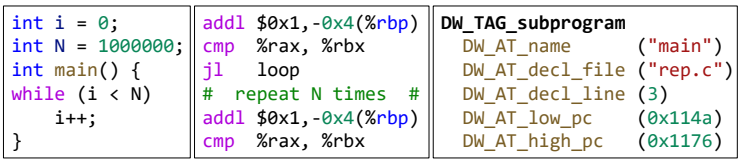

Fig. 4. Sample repeated patterns in a trace.

Fig. 4 illustrates an example where the execution trace of this program consists of one million states, though the source code only has five lines. However, even after collecting all one million states and performing differential testing, we only cover a single DWARF entry `[0x114a,0x1176]`, as displayed on the right sub-figure of Fig. 4. This observation implies the demand for a sampling method to skip redundant states, while still capturing sufficient distinct states to achieve high test coverage. Thus, a proper sampling scheme shall satisfy the following properties simultaneously.

- **(i)** Coverage. The sampled trace should retain sufficient inspection points and data/control facts to satisfy all three properties, **P**, **D**, and **WE**, as noted in Sec. 3.1.
- **(ii)** State Continuity. If two states in a trace are consecutive, they are still consecutive for at least once when sampled.[3] This ensures that all state transitions are captured.
- **(iii)** Sufficient Compression. The samples are lightweight for a practical differential comparison.

Previous works [Lehmann and Pradel 2018; Tolksdorf et al. 2019] on JavaScript randomly select states to compare. Although they successfully achieve high coverage for simple JavaScript traces, it fails to satisfy the first two properties when dealing with complex traces of C/C++ programs. Debug^2 falls short of satisfying the first and the last properties. As a result, it has to limit its scrutiny of data facts to a very narrow range (such as parameters) and does not comprehensively examine DWARF entries.

Heuristic-Based Approach. We propose a heuristic-based approach for trace sampling based on the following observations. For a normal binary, regardless of how complicated its traces may be, the pc addresses of its instructions are still well-defined and within a certain range. Furthermore, DWARF information is also organized using these pc addresses (e.g., `DW_AT_{low,high}_pc`). In other words, an intuitive approach is to scan through the trace and only collect the corresponding program state for each unique pc address. This method satisfies the first property, as every distinct pc address corresponds to a unique DWARF entry (satisfying **D** and **WE**), and we have explained why **P** is satisfied in Sec. 3.3. It also satisfies the third property, as it greatly compresses the trace. Nevertheless, it fails to maintain the continuity of states. If a function is invoked multiple times from different call sites, this method only captures the first invocation and discards all the rest due to their identical entry addresses. Thus, we improve this method by recording the tuple of two consecutive pc addresses, which captures these individual transitions.

Our heuristic trace sampling algorithm is shown in Alg. 1. It begins by generating a complete initial state of the program, which includes its current pc address, source code line number, and all of its variables/registers (Line 1). Then, for each state, if the transition $\langle pc_{k-1}, pc_k \rangle$ reaching this state has not been encountered previously, we generate a complete state with all its variables (Lines 5-7). Otherwise, if we have already observed the transition, we determine whether to generate it with or without variable information based on its frequency (i.e., how mundane it is) (Line 9).

In sum, this algorithm enables a trace generation that is concise and covers all DWARF entries. For a loop-free program, it is evident that DTD collects and compares all states on the trace.

[3]This mimics the common tactic in inter-procedural static program analysis, where we log callsite PC addresses to ensure that given different calling contexts, the same callee function is re-analyzed for better comprehensiveness.

Algorithm 1: Heuristic trace sampling.

Input: $P = \{\langle pc, line, var\rangle_n\}$
Output: $T = \{t_k\}$

1 $t_1 \leftarrow \langle pc_1, line_1, var_1\rangle$; $last_{var} \leftarrow var_1$;
2 $seen_{pc}[*] \leftarrow 0$; $k \leftarrow 2$;
3 **while** $k \leq N$ **do**
4 $C \leftarrow seen_{pc}[\langle pc_{k-1}, pc_k\rangle]$;
5 **if** $C = 0$ **or** $random(0,1) \leq \frac{1}{C}$ **then**
6 $d_k \leftarrow diff(var_k, last_{var})$; $last_{var} \leftarrow var_k$;
7 $t_k \leftarrow \langle pc_k, line_k, d_k\rangle$;
8 **else**
9 $t_k \leftarrow \langle pc_k, line_k, \emptyset\rangle$;
10 **end**
11 $seen_{pc}[\langle pc_{k-1}, pc_k\rangle] \leftarrow C + 1$;
12 $k \leftarrow k + 1$;
13 **end**

```
1 int f() {
2     int a[100][100];
3     a[0][0] = 1;
4     a[70][32] = 10;
5 }
```

C source codes

Incremental Log	Bytes
a:{{…},…,{…}}	40000
a:{0:{0:1}}	24
a:{70:{32:10}}	24
...	…

Incremental log

Fig. 5. Incremental log of a large array.

Incremental Log Compression. Besides reducing the complexity of the full trace, we propose an approach to reduce the intricacy of each individual state log. As mentioned in Sec. 4.1, the programs we tested typically contain more than five hundred global variables (including array elements/structure members) on average. However, a single instruction only modifies a fraction of these variables. Thus, recording all the variables in one state is unnecessary; instead, we compare the current state with the last state and only record the differences between them (Line 6). As displayed in Fig. 5, while the debugger returns the full content of the array `a` when stepping, DTD records the full copy of the large array `a` once on line 5 (the shaded part in Fig. 5); DTD only records the changes afterward. This approach effectively reduces the memory consumption of Alg. 1.

4.2 C2: Randomness in Debugging C Programs

Unlike more advanced languages, like JavaScript which offers the `undefined` keyword, uninitialized variables lack a precise notation in C. As shown in Fig. 6, the source code on the left declares an array `uninit` but leaves it uninitialized (Line 2). Therefore, the elements of `uninit` are random values left on the stack (marked in ?). Moreover, this specific case also reveals that the elements of `uninit` can be partially initialized (Lines 3 and 4). Only upon reaching the end of function `f`, the array `uninit` is fully determined (Line 6).

```
1 void f() {
2     int i, uninit[5];
3     uninit[0] = 8;
4     uninit[2] = 5;
5     for (i = 0; i < 5; i++)
6         uninit[i] = i;
7 }
```

C source codes

0	1	2	3	4
?	?	?	?	?
8	?	?	?	?
8	?	5	?	?
8	?	5	?	?
1	2	3	4	5
1	2	3	4	5

Elements of `uninit`

Fig. 6. Nondeterminism of uninitialized variables.

When debugging C programs, it is common to encounter uninitialized variables whose values in debuggers depend on the remaining values on the stack. These remaining values may be fixed (like local variables of previous functions), or they may be random due to pointers on the stack with random addresses allocated during runtime. Indeed, we frequently observe the latter case.

These random values make it challenging for DTD to accurately flag erroneous behaviors among debuggers. DT assumes that tested programs are expected to yield the same outputs given the same input, which are then compared for discrepancies. However, these random values break this assumption and result in false positives. Debug^2 alleviates this hurdle by restricting compared data facts to the function call parameters, ensuring initialization at the cost of reduced coverage.

Temporal Differential Filtering. To address this, we design a simple yet effective method, Temporal Differential Filtering (TDF). TDF looks for variables whose values are changed in two separate executions; TDF deems those variables as holding uninitialized values, and thus, "discrepancies" over those variables are not treated as bugs of debuggers.

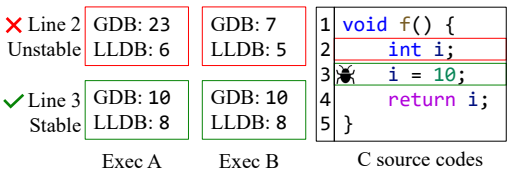

Fig. 7. Example of stable and unstable discrepancy.

To use TDF, upon analyzing the execution traces obtained from both GDB and LLDB, we observe a set of discrepancies. These discrepancies comprise not only true positives caused by bugs in the debugger but also false positives that result from uninitialized random variables. Next, we re-run the DT testing and re-collect discrepancies. After that, TDF examines these discrepancies to determine whether they retain fixed values across two DT testing. Fig. 7 shows an example of "stable" and "unstable" discrepancies. In Fig. 7, the variable `i` is declared but not initialized at Line 2. Therefore, its value is random on the stack, which varies in each execution. Such a discrepancy (A: $23 \neq 6$ and B: $7 \neq 5$) is deemed an *unstable* discrepancy since it displays different values across two executions. Unstable discrepancies are removed from consideration. Subsequently, `i` is initialized to 10 (Line 3). However, due to a hypothetical bug in LLDB, both GDB and LLDB report different values for `i` ($10 \neq 8$). Since this discrepancy remains *stable* across two executions, we view it as a potential bug and proceed to further triage it.

5 IMPLEMENTATION & EVALUATION SETUP

We have implemented DTD in approximately 2.1 K SLoC of Python: 0.5 K SLoC for interacting with the debuggers and 1.6 K SLoC for main components (e.g., heuristic sampling and log compression). We explain the evaluation setup below.

Debuggers. We test two mainstream debuggers, LLDB (ver. 15.0.7) and GDB (ver. 13.1.0). Both are the latest stable versions at the time of our study. Both debuggers offer a Python interpreter for automation. Note that we deliberately avoid using any available "binding library" for these debuggers. Instead, DTD directly operates on the Python interface provided by these debuggers. This way, we alleviate potential false positive findings due to bugs in those debugger binding libraries. Moreover, since the variables from two debuggers are provided in different formats, we implemented our own data adapters for both GDB and LLDB, such that data facts collected from them can be compared in a unified manner.

Test Inputs. As a common practice [Di Luna et al. 2021], we use Csmith (ver. 2.4.0) to generate C programs. We compile Csmith's outputs with GCC (ver. 12.2.0) and Clang/LLVM (ver. 15.0.7)

into 64-bit x86 binary code. Debuggers may encounter more obstacles when processing large and complex software. However, using large software, while could likely provoke errors, is less feasible and preferred in this research. Despite that we leverage script to help us automatically classify error-triggering cases, we still need to summarize the characteristics of untriaged bugs. This process normally takes about ten man-hours for test cases generated by Csmith. Errors in complex and large software could make manual inspection too costly or even infeasible. Nevertheless, we believe that our findings are general enough to impede debugging other large C/C++ code.

Table 3. Statistics of the C programs used in the study. Line of code (LOC) is measured with `cloc` [Danial 2021].

Total #programs generated by Csmith	10,000
Total #programs w.o. undefined behavior	7,044
Total #executables used to test debuggers	56,352
Total LOC in Csmith generated C programs	1.7 M
Total #states in DTD-analyzed traces	15 B
Total #data facts on DTD-analyzed traces	432 M

Statistics. Table 3 reports statistics of the C programs used in the study. We use Csmith to randomly generate 10,000 C programs. Among these, 7,044 pass the validation of CompCert [Leroy 2009] and are free of undefined behavior. These programs consist of about 1.7 million lines of code in total. Launching DTD over these test cases yields traces of over 15 billion states, and in total over 432 million data facts are heuristically (Sec. 4.1) collected and compared.

6 EVALUATION

Table 4 provides an overview of our findings. Out of 56,352 test cases, 33,134 (58%) tested programs behave inconsistently in LLDB and GDB. Specifically, 1,398 (2.4%) tested programs show an inconsistent value for the same variable; 3,615 (6.4%) tested programs lose track of certain variables or display out-of-scope variables during debugging, while 28,121 (49.9%) tested programs exhibit inconsistent source line numbers for the same pc.

Table 4. Inconsistencies found by DTD.

Compiler	Opt. Level	GDB	LLDB	GDB/LLDB	
		Var. Lost	Var. Lost	Var. Diff.	Ctrl. Flow
Clang	O0	0	0	151	12
	O1	70	13	251	4858
	O2	72	0	209	4518
	O3	90	0	195	2399
GCC	O0	3	0	135	0
	O1	12	2332	216	6728
	O2	4	854	116	5364
	O3	4	161	125	4242

Roughly speaking, defects reported in Table 4 can be put into six groups: (G1) variables lost tracking in GDB, (G2) out-of-scope variables displayed in LLDB, (G3) variables lost tracking in LLDB, (G4) out-of-scope variables displayed in GDB, (G5) inconsistent values between GDB and LLDB, and (G6) inconsistent control facts in GDB and LLDB. G1/G2 correspond to the third column, whereas G3/G4 correspond to the fourth column[4]. To understand the nature of these bugs, we engage two experts to check and classify the findings. Both experts are Ph.D. students with extensive experience in compiler hacking, reverse engineering, and systems programming, which ensures the credibility of our research to a great extent. Nevertheless, we admit the difficulty involved, given

[4]If a variable is *lost* in GDB, then either GDB loses track of that variable or LLDB is displaying an out-of-scope variable.

the large number and complexity of cases. We discuss bug characteristics below, accompanied by case studies of our findings in Sec. 6.1.

Defect Characteristics. We study the characteristics of the defects reported in Table 4. In general, for issues regarding variable scopes (G1, G2, G3, and G4), an error message is displayed by the debugger when the variable is not available (e.g., "*could not evaluate* `DW_OP_entry_value`."). Therefore, we leverage these error messages to recognize and characterize these defects. As for inconsistent variable values (G5), we have to examine each case to learn their characteristics manually. As for control flow inconsistency (G6), due to the large number of cases, we manually examine 20 samples from each configuration. Note that we have eight settings in Table 4, and under `-O0` GCC does not trigger this inconsistency while Clang only outputs 12 error-triggering binaries. We presumably attribute this to the fact that compilers do not actively inline functions under `-O0`. Therefore, in total, we examined 132 samples from these configurations. We find that all of these examined cases are caused by the divergent handling of inlined functions, a common tactic adopted by compilers to improve performance. We discuss this case further in Sec. 6.2.

The 18 characteristics of our defects are summarized in Table 5. In sum, we have used DTD to find 13 bugs in the LLVM toolchain (Clang/LLDB), among which the LLDB developers have responsibly confirmed nine bugs. Similarly, we have found five bugs in the GNU toolchain (GCC/GDB), among which the GDB developers have confirmed four bugs. As of this writing, a total of 4 bugs are fixed, one of which is fixed by our pushed patch.

It should be accurate to assume that there is *no* false positive in DTD's findings on data fact discrepancies. To clarify, bugs causing variables lost tracking (G1 and G3) typically yield a clear error message that confirms their validity (e.g., Bug_2, Bug_7, Bug_8, and Bug_{10}). Meanwhile, bugs that result in inconsistent variable values exhibit distinguishable characteristics, allowing us to automatically identify them. For example, we directly search for `DW_OP_div` to pinpoint Bug_{13} automatically. Similar logic can be applied to analyze Bug_5, Bug_{11}, Bug_{14} and Bug_{16}. With this automated process, we incline that DTD reports no false positive as there is no unattributed discrepancy.

Note that control flow inconsistencies are not necessarily caused by "coding errors" but rather by GDB/LLDB's differing ways of handling inlined functions. Given that said, we still find a bug (i.e., Bug_{18}) where GDB displays an inconsistent source code and stack frame. To avoid confusion, we marked #error-triggering cases of Bug_{18} with "?", indicating the potential false negatives in these control flow inconsistencies. See Sec. 6.2 for further discussion.

Processing Time. Our experiments are launched on a machine with AMD 3970X 32-core 3.70 GHz CPU and 256 GiB memory. It takes about 27 hours to finish the entire testing process of 56,352 test cases. We report that this processing time is comparable to previous works [Di Luna et al. 2021]. See Sec. 6.3 for further details on the scalability of DTD compared with SotA.

6.1 Case Studies

We select representative bugs from each previously mentioned group (G1-G5) and detail their root causes. G6 is discussed in Sec. 6.2. All these cases are assigned with specific bug IDs.

Case 1: LLDB bug 61727 (Fig. 8(a)) — At Line 6, LLDB shows a series of incorrect values for `i` when this source code is compiled by Clang with `-O1`. Rather than displaying from 0 to 10 as expected, LLDB only displays the leading 0 and 1. The remaining values of `i` are incorrectly displayed as 0.

In this case, the variable `i` displayed incorrectly is encoded into a DWARF expression, as explained in Sec. 2, which is then evaluated to support its display. Particularly, the variable `i` is expressed using a `DW_OP_div` operator, which is expected to pop two values from the stack as operands and perform a *signed* division regardless of the type of its operands. However, LLDB neglects this subtle detail and performs an ordinary division, which implicitly promotes a signed operand to an unsigned one, resulting in arithmetic underflow.

Table 5. Characteristics of bugs found by DTD. Note that the * denotes that the characteristics of these bugs might be composited, e.g., both `DW_OP_div` and `DW_OP_deref_size` appear in the same DWARF expression (Bug_{13} and Bug_{14}). Here, "#" correlates to the number of error-triggering test cases reported in Table 4, except duplicates marked by *. Note that Bug_{13} is already fixed by our pushed patch.

	Group	Compiler	Opt. Level	#	Characteristics	Status
Bug_1	G1	Clang	O1,O2,O3	202	Clang emits incomplete DWARF information.	Confirmed
Bug_2	G1	Clang	O1,O2,O3	27	Clang emits DWARF information of incompatible types.	Confirmed
Bug_3	G1	Clang	O1	1	GDB lacks the support for multi-precision arithmetic.	Fixed
Bug_4	G1+G5	GCC	O0,O1,O2,O3	23	GDB selects out-of-scope (future scope) variable values.	Fixed
Bug_5	G2+G5	GCC	O1,O2,O3	203*	GDB selects wrong variable value at function entrypoint.	Confirmed
Bug_6	G3	Clang, GCC	O1,O2,O3	1,619*	LLDB does not show all in-scope variables in `frame variable`.	Reported
Bug_7	G3	Clang, GCC	O1,O2,O3	892	LLDB does not handle `DW_OP_bit_piece` correctly.	Comfirmed
Bug_8	G3	GCC	O1,O2,O3	1,898	LLDB fails to evaluate `DW_OP_entry_value`.	Reported
Bug_9	G3	GCC	O1,O2,O3	18	LLDB shows `<empty constant data>` for a valid entry.	Reported
Bug_{10}	G3	GCC	O1,O2,O3	3,080	LLDB lacks the support of `DW_OP_implicit_pointer`.	Reported
Bug_{11}	G3	GCC	O1,O2,O3	1,619*	LLDB does not correctly ignore "empty pc ranges".	Confirmed
Bug_{12}	G5	Clang, GCC	O0,O1,O2,O3	1,208*	LLDB underflows when evaluating bitfields.	Confirmed
Bug_{13}	G5	Clang, GCC	O1,O2,O3	203*	LLDB treats `DW_OP_div` as unsigned.	Fixed
Bug_{14}	G5	Clang, GCC	O1,O2,O3	203*	LLDB treats `DW_OP_deref_size` as unsigned.	Confirmed
Bug_{15}	G5	Clang, GCC	O0,O1,O2,O3	1,208*	LLDB displays `0` for optimized-out variables.	Confirmed
Bug_{16}	G5	Clang	O1,O2,O3	203*	Clang emits wrong DWARF information.	Confirmed
Bug_{17}	G5	Clang	O1,O2,O3	241	GDB shows `<synthetic pointer>` for non-pointer values.	Fixed
Bug_{18}	G2+G6	Clang	O1,O2,O3	4?	GDB shows inconsistent source code and stack frame.	Reported

```
1 static
2 volatile uint64_t g=0;
3 static const int f() {
4   unsigned int i;
5   for(i=0; (i!=10); i++)
6     ++g;
7 }
```

(a) LLDB bug 61727

```
1 int a; volatile int b;
2 static int func(int b) {
3   int i = 0;
4   for (; i < 10; i++)
5     { int c; continue; }
6   a = 0;
7 }
```

(b) LLDB bug 61942

Fig. 8. Case 1 and Case 2.

Intriguingly, another bug discovered by DTD is also caused by the type system inconsistency between DWARF and LLDB. According to the standard, values represented in `DW_OP_deref` are of a *generic* type with unspecified signedness. However, LLDB does not support this type and treats values as unsigned, causing all values represented by this particular expression to be wrongly evaluated if the value is signed. Overall, these two bugs advocate for a more robust type system for the DWARF expression in LLDB, which can presumably prevent similar bugs in the future.

Case 2: LLDB bug 61942 (Fig. 8(b)) — At Line 6, LLDB complains that it cannot find the variable `i` when this source is compiled by GCC with `-O1`. However, GDB correctly displays `i` with the same binary. The root cause of this issue is that LLDB fails to recognize the stack frame where `i` is located. As explained in Sec. 2, each stack frame is associated with a series of `pc` ranges in the debugging information, which indicates the lifetime of the stack frame. According to the standard, these `pc` ranges can be empty, and these empty ranges should be ignored. GCC "abuses" this corner case rule to generate a set of empty but valid ranges. However, LLDB does not accurately disregard these ranges while matching the stack frames, causing Line 6 to be prematurely matched with one of these empty ranges and resulting in the correct stack frame of `i` being discarded.

Case 3: Clang bug 61924 (Fig. 9) — When the source code is compiled by Clang with `-O1`, GDB displays a wrong value of 255 for the global variable `g`, while LLDB fails to evaluate it at all and reports that `DW_OP_piece(1)` cannot be evaluated. This issue arose due to the `global_opt` pass of Clang, which optimized out all the members of `g` except the referenced `f1`. Thus, only `f1` is present in the final debugging information, leading to the truncated value 255 for the variable `g`.

Case 4: GDB bug 28987 (Fig. 10(a)) — At Line 7, an incorrect value of 0x0 is reported by GDB instead of the accurate value of `&a` (i.e., the memory address of `a`) for the pointer `l`. This problem arises

```c
int a, b;
union {
    short f0;
    unsigned char f1;
    int f2;
    long f3;
```

```c
} static g = {65535};
int c() {
    char *d = (char*)&g;
    *d = b;
    return 0;
}
```

Clang bug 61924

Fig. 9. Case 3.

```c
int a;
int b;
int func() {
    int *l = 0;
    int **c = &l;
    *c = &a;
    b = 0;
    return a;
}
```

(a) GDB bug 28987

```c
int a = 1;
int func() {
    int i;
    for(; i < 1; i++);
    if (a) {
        short b = 2;
        { int i; }
    }
}
```

(b) GDB bug 30315

Fig. 10. Case 4 and Case 5.

when the source code is compiled by GCC with `-O1`. Under this circumstance, GDB selects an outdated location entry for `l` at Line 7. As noted in Sec. 2, compilers like GCC produce distinct DWARF entries for a variable as it changes within the stack frame. In Fig. 10(a), two DWARF entries are created for variable `l`: one is the constant 0 for Lines 4 to 6, and the other is `&a` for Lines 7 and 8. Since all lines before Line 7 have been optimized out, the entry of constant 0 should be disregarded. Furthermore, due to this optimization, Line 7 (instead of Line 4) has become the entry of `func`. The root cause of this bug is the faulty handling of function entry inside GDB. While exploring the correct DWARF entry for a variable, GDB is supposed to search all entries exhaustively. However, if the line is at the function entry (e.g., Line 7), GDB terminates this search prematurely, leaving `l` matched with the wrong entry of constant 0.

Case 5: GDB bug 30315 (Fig. 10(b)) — This bug occurs when complied with GCC in `-O0`. At Line 6, GDB should exhibit variable `i` using its outer frame value 1 from Line 4. However, when selecting the stack frame corresponding to Line 6, GDB mistakenly chooses the inner frame at Line 7, where the variable `i` has been optimized out. In a complex program, this bug misleads developers into believing that their variables have been optimized out, creating hurdles for debugging.

6.2 Root Causes of Bugs

In this section, we summarize the common root causes of the bugs we found in LLDB and GDB.

6.2.1 Divergent Implementation of Features. During our bug-triaging process, we observed an intriguing phenomenon where different toolchains exhibited diverse implementations over the same DWARF feature. A particular feature in the latest DWARF may be well-supported in LLDB but may cause errors in GDB, and vice versa. Fig. 11 shows some DWARF expressions generated by Clang. In Line 3, the result of `DW_OP_convert` is a 72-bit long unsigned value, which is larger than the length of primitive types (64-bit). In the DWARF standard, the precision of `DW_OP_convert` is not restricted since multi-precision conversion is useful in languages like Fortran. In LLDB, this expression is successfully evaluated. However, GDB does not support multi-precision arithmetic and returns errors when encountering such expressions. Fortunately, the development branch of GDB now has support for multi-precision arithmetic to address this issue.

```
1 Range 0x461d-0x4628:
2 DW_OP_breg0 0 [$rax]
3 DW_OP_convert<DW_ATE_unsigned_72 [0x27]>
4 DW_OP_stack_value
5 [4-byte piece]
```

Fig. 11. DWARF expression of multi-precision arithmetic.

We find similar issues in the LLVM toolchain, as shown in Fig. 8(a). The bug is due to a naive implementation of the type system for DWARF expressions in LLDB. These divergent implementations can lead to various subtle problems ranging from failed evaluations to incorrect values being displayed. DTD proves to be an effective tool for detecting and locating these issues.

6.2.2 DWARF Limitation. Besides Sec. 6.2.1, some bugs arise from the limitation of DWARF itself. One typical example of this is how inlined functions are handled.

```
1 static int a = 209, b;
2 int *volatile c = &b;
3 int caller() {
4   *c = a;
5   inlined();
6 }
7 int inlined() { char l; }
```

```
mov   0x200923(%rip),%rax
mov   (%rax),%eax
xor   %ecx,%ecx
nopl  0x0(%rax)
or    %eax,-0x18(%rsp,%rcx,4)
inc   %rcx
cmp   $0x5,%rcx
```

caller instruction | border instruction | inlined instruction

Fig. 12. A example of inlined function in DWARF.

In the DWARF standard, a section named `.debug_line` records the unique line number of an instruction. This scheme works well for ordinary function invocations with an explicit prologue. However, for inlined functions, the situation becomes more complex. The program in Fig. 12 consists of two functions: `caller`, and `inlined`. The inlined function `inlined` is compiled as a part of `caller` (Line 5). As shown in the disassembled code on the right side of Fig. 12, the first three instructions (Lines 1-3) belong to the function `caller`, while the last three instructions (Lines 5-7) belong to the inlined function `inlined`, leaving the owner of instruction `nopl 0x0(%rax)` undefined (Line 4). This undefined case leads to a subtle bug in GDB; right before the instruction `nopl 0x0(%rax)` is executed, GDB enters a border state between the function `caller` and `inlined`. Specifically, while displaying the source, GDB still shows the current line number as 5 (i.e., inside `caller`, which does not contain local variable `l`). However, the stack frame of GDB has already entered the function `inlined`, with the local variable `l` visible. In contrast, LLDB shows a consistent source at line 5 without local variable `l`, which is correct.

Though this bug may not be particularly deceptive, it exposes the intricacy of debugging information. In this case, a potential solution is to define the owner of the "border instruction" explicitly in the standard. However, in other scenarios where the "border instruction" are meaningful (unlike the `nopl` here), this fix may lead to other subtle issues. Though this bug is partially caused by the limitation of the DWARF standard (i.e., each instruction can only have a unique line number), the debuggers are expected to return consistent program states during debugging.

6.3 Comparing DTD with The State-of-the-Art Work — Debug^2

Novelty Clarification. Careful readers may notice the similarity between DTD and Debug^2 to some extent. We clarify that our method is *not* merely an incremental work based on Debug^2. To begin with, DTD advocates a new DT scenario for C debuggers where different implementations of debuggers are tested rather than comparing executables before/after optimizations. Moreover, we observe that Debug^2 performs poorly in our scenarios as it only examines callsite data facts.

To support our observation, we conduct the following experiments to compare DTD and Debug^2 in terms of effectiveness (how many error-triggering cases/bugs are found) and performance (how

much time/storage used during the test). Note that since the sources of Debug^2 are not available, we did our best effort to re-implement their method [Di Luna et al. 2021] for our testing scenarios. In sum, we observe that Debug^2 leaves the following three major obstacles in testing debuggers unsolved.

Obstacle 1: Poor Effectiveness due to Limited Data Facts. Debug^2 limits its examined data facts to callsites. However, we observed that most bugs uncovered by DTD are *not* related to callsite data facts, indicating that the effectiveness of Debug^2 may primarily suffer from this limitation. To further substantiate this observation, we re-run the evaluation in Sec. 6 using Debug^2 with the same set of test cases. The experiment takes about 20 hours and we report our findings as follows.

Out of 56,352 test cases, we report that Debug^2 only detects 338 (0.5%) cases with inconsistent variable states (G1~G5), whereas DTD detects 5,013 (8.8%) of them. Moreover, among these 338 detected discrepancies, only Bug_6, Bug_8 and Bug_{11} are found (3 out of 17). This result further justifies our efforts in making DTD cover all data facts rather than just callsite data facts. It should be accurate to assume that the highly promising performance of DTD comes from its 100% debug information coverage. We also measure the debug information coverage of Debug^2, which is merely 2.5%. In terms of detecting cases with inconsistent control flow (G6), most cases found by both DTD and Debug^2 are from the divergent way of GDB/LLDB to handle inlined functions.

One may wonder the feasibility of extending Debug^2 to cover *all* data facts instead of just callsites data facts, solving the effectiveness issue. However, we show in the following paragraphs that simply extending the data facts covered brings huge overhead and high false positive rates.

Obstacle 2: Impractical Performance from Redundant Data Facts. In Sec. 4.1, we have discussed the trace complexity of seemingly simple C programs and how control structures (e.g., loops) create redundant patterns in these traces. In this paragraph, we conduct experiments to show how these complicated traces create hurdles for extending Debug^2 to collect full data facts. In this experiment, we randomly generate 20 individual sources using Csmith and compile them into 160 binaries using GCC and Clang under `-O0`, `-O1`, `-O2`, and `-O3`. Note that these cases are generated independently and *not* a part of our main experiment. The average trace length of these test cases are shown in Table 6, and there are over four million states in total in these traces. We interpret that the test cases are sufficiently large for our study usage here.

Table 6. Average trace length of test cases.

Opt. Level	O0	O1	O2	O3
Trace Length	53,156	27,532	9,400	8,580

We measure the time and storage usage of the following four setups: ① Debug^2 with full data facts, ② Debug^2, ③ DTD without log compression (Sec. 4.1), ④ DTD. As a practical setup, we limit the time for trace generation to thirty minutes and terminate the process if it exceeds the time limit.

Table 7. Trace generation time and storage size. Note that ① fails to finish in 30 minutes on all cases.

Method	①	②	③	④
Avg. Time	> 30 mins	10.1 s	26.8 s	18.1 s
Max. Time	> 30 mins	135.3 s	158.2 s	148.6 s
Avg. Size	> 1 GiB	2.9 MiB	12.5 MiB	1.5 MiB
Max. Size	> 1 GiB	34 MiB	128 MiB	12 MiB

Table 7 shows that collecting the full data facts brings non-trivial performance overhead (> 30 mins) to Debug^2. Spending over 30 minutes analyzing *one* trace is clearly impractical. Even with a 64-core processor, a testing campaign described in Sec. 6 could take months. In terms of storage cost, our preliminary study shows collecting the full trace of a simple C program (in accordance with setting ①) takes at least 1 GiB storage, requiring over 50 TiB to store all traces from our

main experiment. Moreover, this experiments also demonstrate that our log compression design effectively cuts the final trace size by over 90%.

Obstacle 3: False Positives from Uninitialized Variables. Apart from the performance, extending Debug^2 to collect full data facts faces a more principled issue — nondeterminism of uninitialized variables. In Sec. 4.2, we have discussed the existence of these uninitialized variables and how they may cause false positives for debugger testing. Unfortunately, all C programs contain these uninitialized variables. In the previous experiment, even if we enable Debug^2 to collect all data facts with our heuristic method, *all* the tested programs exhibit discrepancies due to these uninitialized variables, making it impossible to distinguish actual bugs from these false positives. We presume this is why Debug^2 limits its examined data facts to callsites, which guarantees initialization. DTD, on the other hand, leverages TDF in Sec. 4.2 to eliminate these spurious discrepancies.

7 RELATED WORK

We have reviewed some highly relevant works in Sec. 3.2. We now review other relevant works on compiler testing. The correctness of compilers has received continuous attention over the past decades [Chen et al. 2020b]. To capture bugs from complicated modern compilers, one of the earlier works proposed the effective random C program generator, Csmith [Yang et al. 2011], which greatly promotes C compiler testing research. The following works further propose "equivalence modulo inputs" test oracles [Le et al. 2014, 2015; Sun et al. 2016a] to uncover deep functionality bugs in compiler optimizations. More recent studies specifically [Assaiante et al. 2023; Di Luna et al. 2021; Li et al. 2020; Wang et al. 2023] focus on the completeness and correctness of debug information generated by compilers. To facilitate bug root cause analysis, C-Reduce [Regehr et al. 2012] downsizes test cases while still triggering the compiler bugs. Sun et al. [2016b] provided a systematic study on the characteristics of compiler bugs. Beyond finding functionality bugs, recent work also explores locating missed optimization opportunities with DT to improve compiler infrastructures [Barany 2018; Liu et al. 2023; Theodoridis et al. 2022]. Besides C compiler, the compiler testing in general, as an emerging field, has been getting more attention from academia recently. MT-DLComp [Xiao et al. 2022] presented the first work specifically aimed at finding functionality bugs in DL compilers. Xiao et al. [2023] proposed metamorphic shader fusion for testing graphic shader compilers. Aside from new heuristics, there are also works [Cui et al. 2018; Lu and Zhang 2022; Ning and Zhang 2017; Zhang et al. 2015, 2023] that utilize hardware features to assist testing. Zhang et al. [2023] propose using Arm hardware tracer to facilitate the diagnosis of concurrency bugs. Lu and Zhang [2022] leverage RISC-V security features to conduct transparent testing and debugging.

8 DISCUSSION

Other Languages and Architectures. Although DTD is mainly designed to test debuggers of C/C++ executables, it is orthogonal to specific features in C/C++ debuggers and applicable to other platforms or languages. Also, GDB/LLDB are already ported to support debugging of executables in other languages (e.g., Rust and Go) and architectures (e.g., Arm and RISC-V). In general, DTD can be easily ported to other setups as long as 1) the debug symbols (e.g., in DWARF format) are bundled with the executable, and 2) at least two debuggers with aligned functionality exist.

If the debug information is not present in the executable, DTD can still detect bugs in the debuggers by comparing the low-level program states (e.g., registers, memory values, and pc) between two debuggers. Technically, in case only one debugger is available, DTD may still test the divergence of two different versions of the same debugger [Jung et al. 2019].

Limitations. Although DTD has encouraging bug-finding capabilities, limitations still exist. Since DTD is DT-based, it suffers from the inherent limitation of DT that bugs existing in both debuggers

cannot be detected. As clarified above, DTD does not assume any specific features of debuggers. Thus, by including other debuggers [Radare 2023], the chances of finding bugs shared by debuggers increase.

Future Work. For test case generation, DTD follows the common practice of existing works [Di Luna et al. 2021], using Csmith to generate random C programs. This leaves the doubt that whether the debug information bundled in these programs has sufficient diversity to stress the debuggers. Nonetheless, DTD has uncovered a considerable amount of bugs in popular debuggers and outperformed the state-of-the-art debugger testing tools. Moreover, DTD does not rely on specific properties in test cases. Thus, it can process test cases from more sophisticated test case generators. We leave it as one future work to design test case generators that produce more diverse debug information.

9 CONCLUSION

We have presented DTD, a differential testing framework to locate discrepancies in interactive C/C++ debuggers. DTD implements a set of optimizations to offer scalable testing and comprehensive coverage. DTD effectively uncovers discrepancies in recent GDB and LLDB, and representative findings have been confirmed and fixed by developers.

ACKNOWLEDGEMENTS

This work is partly supported by the National Natural Science Foundation of China under Grant No.62372218 and Shenzhen Science and Technology Program under Grant No.SGDX20201103095408029. We are grateful to the anonymous reviewers for their valuable comments.

REFERENCES

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

Gergö Barany. 2018. Finding missed compiler optimizations by differential testing. In *Proceedings of the 27th international conference on compiler construction*. 82–92.

Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020b. A survey of compiler testing. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–36.

Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 2020a. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543* (2020).

Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1257–1268.

Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 85–99.

Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse Debugging of Failures in Deployed Software. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 17–32. https://www.usenix.org/conference/osdi18/presentation/weidong

Albert Danial. 2021. cloc. https://github.com/AlDanial/cloc

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

DTD. 2023. DTD: Supplementary Website. https://sites.google.com/view/dtd-supplementary/

Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2019. Apollo: Automatic detection and diagnosis of performance regressions in database systems. *Proceedings of the VLDB Endowment* 13, 1 (2019), 57–70.

Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226.

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*. 386–399.

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

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (jul 2009), 107–115. https://doi.org/10.1145/1538788.1538814

Shaohua Li and Zhendong Su. 2023. Finding Unstable Code via Compiler-Driven Differential Testing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 238–251.

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

Zhibo Liu, Dongwei Xiao, Zongjie Li, Shuai Wang, and Wei Meng. 2023. Exploring Missed Optimizations in WebAssembly Optimizers. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*.

LLVM. 2023. The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. https://github.com/llvm/llvm-project

Hongyi Lu and Fengwei Zhang. 2022. Raven: a novel kernel debugging tool on RISC-V. In *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*. 1039–1044. https://doi.org/10.1145/3489517.3530583

William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

Zhenyu Ning and Fengwei Zhang. 2017. Ninja: Towards Transparent Tracing and Debugging on ARM. In *Proceedings of The 26th USENIX Security Symposium (USENIX-Security'17)*.

Radare. 2023. Radare2: UNIX-like reverse engineering framework and command-line toolset. https://github.com/radareorg/radare2

John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 335–346.

Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1140–1152.

Chengnian Sun, Vu Le, and Zhendong Su. 2016a. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications*. 849–863.

Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016b. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th international symposium on software testing and analysis*. 294–305.

Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding missed optimizations through the lens of dead code elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 697–709.

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

Theodore Luo Wang, Yongqiang Tian, Yiwen Dong, Zhenyang Xu, and Chengnian Sun. 2023. Compilation Consistency Modulo Debug Information. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 146–158.

Dongwei Xiao, Zhibo Liu, and Shuai Wang. 2023. Metamorphic Shader Fusion for Testing Graphics Shader Compilers. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2400–2412. https://doi.org/10.1109/ICSE48619.2023.00201

Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Metamorphic testing of deep learning compilers. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 1 (2022), 1–28.

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*. 283–294.

Fengwei Zhang, Kevin Leach, Angelos Stavrou, Haining Wang, and Kun Sun. 2015. Using Hardware Features for Increased Debugging Transparency. In *Proceedings of The 36th IEEE Symposium on Security and Privacy (SP'15)*.

Yiming Zhang, Yuxin Hu, Haonan Li, Wenxuan Shi, Zhenyu Ning, Xiapu Luo, and Fengwei Zhang. 2023. Alligator in Vest: A Practical Failure-Diagnosis Framework via Arm Hardware Features. 917–928. https://doi.org/10.1145/3597926.3598106

Received 2023-09-27; accepted 2024-01-23