Who’s Debugging the Debuggers? Exposing Debug Information Bugs in Optimized Binaries

Giuseppe Antonio Di Luna
diluna@diag.uniroma1.it
Sapienza - University of Rome - DIAG
Roma, Italy

Davide Italiano
davidino@apple.com
Apple
Cupertino, California, USA

Luca Massarelli
massarelli@diag.uniroma1.it
Sapienza - University of Rome - DIAG
Roma, Italy

Sebastian Österlund
s.osterlund@vu.nl
Department of Computer Science
Vrije Universiteit Amsterdam
Amsterdam, Netherlands

Cristiano Giuffrida
giuffrida@cs.vu.nl
Department of Computer Science
Vrije Universiteit Amsterdam
Amsterdam, Netherlands

Leonardo Querzoni
querzoni@diag.uniroma1.it
Sapienza - University of Rome - DIAG
Roma, Italy

ABSTRACT

Despite the advancements in software testing, bugs still plague deployed software and result in crashes in production. When debugging issues —sometimes caused by “heisenbugs”— there is the need to interpret core dumps and reproduce the issue offline on the same binary deployed. This requires the entire toolchain (compiler, linker, debugger) to correctly generate and use debug information. Little attention has been devoted to checking that such information is correctly preserved by modern toolchains’ optimization stages. This is particularly important as managing debug information in optimized production binaries is non-trivial, often leading to toolchain bugs that may hinder post-deployment debugging efforts.

In this paper, we present Debug2, a framework to find debug information bugs in modern toolchains. Our framework feeds random source programs to the target toolchain and surgically compares the debugging behavior of their optimized/unoptimized binary variants. Such differential analysis allows Debug2 to check invariants at each debugging step and detect bugs from invariant violations. Our invariants are based on the (in)consistency of common debug entities, such as source lines, stack frames, and function arguments. We show that, while simple, this strategy yields powerful cross-toolchain and cross-language invariants, which can pinpoint several bugs in modern toolchains. We have used Debug2 to find 23 bugs in the LLVM toolchain (clang/lldb), 8 bugs in the GNU toolchain (GCC/gdb), and 3 in the Rust toolchain (rustc/lldb)—with 14 bugs already fixed by the developers.

CCS CONCEPTS

• Software and its engineering → Compilers; Formal software verification; Correctness.

KEYWORDS

Debug Information, Optimized Binaries, Verification

ACM Reference Format:


1 INTRODUCTION

Production binaries need to be heavily optimized to maximize metrics such as speed, size, and energy consumption [22]. For this purpose, modern compilers feature optimization stages where several sophisticated transformation passes cooperate to produce the final binary. Preserving debug information in this process is important for several reasons.

First, correct debug information helps the interpretation of core dumps collected in production (e.g., to provide line information in stacks leading to a crash). More importantly, correct debug information is crucial when reproducing and debugging production issues offline on the same optimized binary. This is often necessary since compiler optimizations also alter the observability of unwanted behaviors compared to the unoptimized case. In other words, common issues such as race conditions [12], memory errors [9], and other classes of “heisenbugs” [28] may not even be reproducible without a reliable debugging process for optimized binaries [10].

However, preserving debug information for optimized production binaries is a daunting task, with no obvious mapping between source and assembly statements [7, 10] and the potential to introduce bugs at each of the several layers of modern toolchains. As we will show, efforts to provide debug-friendly optimization levels (such as -Og in modern compilers) fall short on providing a bug-free debugging experience.

Yet, despite the challenges and relevance of this task, as well as toolchain developers’ efforts to improve the debuggability of optimized binaries, little attention has been devoted to scrutinizing the full debug information lifecycle for bugs. Prior work largely focused on debugger testing [16, 25], with one recent exception
focusing on the ability to retrieve correct variable values in optimized binaries [18]. We stress that the entire toolchain (compiler, linker, debugger) must be free of debug information bugs to provide a reliable debugging experience.

In this paper, we introduce Debug$^2$, a framework to expose debug information bugs in production toolchains. The key idea is to feed random source programs to a target toolchain and compare the debugging behavior of their optimized/unoptimized binary variants to expose bugs. To achieve this, Debug$^2$ first extracts debugging traces of each binary by single-stepping its execution in the target debugger. Next, Debug$^2$ performs differential analysis of each pair of optimized/unoptimized traces to check for unexpected differences at each step. To check for (un)expected behavioral differences, Debug$^2$ relies on trace invariants empirically based on the (in)consistency of common debug elements, such as source lines, stack frames, and function arguments. Our trace invariants are explicitly designed to be generic (i.e., toolchain- and programming language-agnostic) and, while simple, can effectively pinpoint inconsistencies in the entire debug information lifecycle. As we will show, these inconsistencies can then be traced back to the underlying issue, exposing bugs in the entire toolchain; including all components of the compiler (from backend until the optimization stages) and the debugger.

Our analysis shows that after decades of development, mature toolchains still suffer from a conspicuous amount of bugs that Debug$^2$ can automatically find. Surprisingly enough, many of these bugs plague the optimization levels specifically created for a smooth debug experience (i.e., -Og).

A motivating example – Snippet 1 shows a bug Debug$^2$ exposed in the LLVM toolchain when compiling with -Og.

Debug$^2$ can expose this bug with a simple source line invariant: a line containing dead code for a given binary should be absent from its debugging trace. However, while analyzing the trace, Debug$^2$ detects the debugger eventually pointing to the dead line 8 — which is never executed, as opposed to line 7 — and flags an invariant violation. While Debug$^2$ detects this issue by line stepping in the debugger (which previous work instead assumed to be correct [18]), this is actually a compiler bug. In particular, the bug is caused by the compiler backend’s branch folding pass and it is part of a more general class of bugs in which optimization passes move instructions across basic blocks without properly updating the corresponding debug information. While these bugs clearly degrade the debugging experience by displaying misleading execution flows, they still escape state-of-the-art testing efforts and are surprisingly common.

### Snippet 1: Clang bug 46009: wrong step at line 8.

```c
int a, b, c;
int main()
{
    int u11 = 5, u12 = b;
    c = u12 == 0 ?
        u11 :
    (u11 / u12);
    return 0;
}
```

We have used Debug$^2$ to expose many such toolchain bugs, in passes ranging from control-flow graph simplification to loop-invariant code motion. The reported issues sparked lively discussions among toolchain developers, evidencing the lack of much needed guidelines defining how to preserve debug information during optimization passes. This forces developers to find the best course of action on an issue-by-issue basis, slowing down the rate at which bugs can be fixed. Interestingly, our bug reports influenced an initial document published by LLVM developers on how to handle debug information during optimization passes [14].

We hope our work will serve as an inspiration to evolve the standard UNIX debugging format (DWARF), which currently lacks proper support to represent the effect of transformations on the source-to-binary mapping. Even simple optimizations (e.g., common subexpression elimination) struggle to correctly preserve debug information due to DWARF’s inability to map a single address to multiple source locations.

### Contributions

- Debug$^2$: a framework to scrutinize the debug information lifecycle for optimized binaries using trace invariants ($\S 2$).
- Four trace invariants to pinpoint debugging behavioral differences in optimized/unoptimized binaries and expose toolchain bugs. Debug$^2$’s invariants are empirically derived and designed to expose bugs across different toolchains and programming languages ($\S 3$).
- An extensive experimental evaluation of Debug$^2$ on the LLVM toolchain. We also present experiments on the GNU and Rust toolchains to confirm the generality of our approach. Debug$^2$ exposed bugs in all such toolchains ($\S 5$), specifically 23 bugs in the LLVM toolchain (clang/lldb), 8 bugs in the GNU toolchain (GCC/gdb), and 3 in the Rust toolchain (rustc/lldb). The developers have already confirmed 22 of these bugs, and 14 of these have been fixed.
- Lessons learned from our interactions with the toolchain developers, egregious bugs found, and current shortcomings in the DWARF debugging format ($\S 7$ and $\S 6$).

### 2 THE DEBUG$^2$ FRAMEWORK

The Debug$^2$ framework is based on a simple idea: consider a program that is compiled and debugged with a given toolchain; the execution traces obtained from the debugger will, in general, differ if they are obtained from versions compiled with different optimization flags, but their semantics must match. In particular, there is some behavior that must manifest coherently in all traces, no matter the optimization level used in the compilation process. For example, a line in the source that represents dead code must be absent from any execution trace.

Our framework uses a set of these intuitive rules, that we call trace invariants, to check if the traces of differently optimized binaries are coherent or not. Whenever an invariant is violated, an inconsistency is detected that points to a candidate bug in the toolchain. Candidates are then manually verified before reporting them to developers.

Debug$^2$ encompasses four main modules (see Figure 1):

- **Program Generation**: responsible for generating the source code samples used to test a specific toolchain; the output of this module is the source code $C$.
- **Tested Toolchain**: the toolchain that is being tested that typically includes at least a compiler and a debugger; the compiler is used to compile $C$ using both a desired optimization level, and with no optimization; the two resulting binaries are then
fed to the debugger that outputs the two traces $T(\text{opt})$ and $T(\text{unopt})$.

- **Trace Consistency**: takes as input the two traces and checks if any of the trace invariants are violated, indicating an inconsistency that could point to a bug in the toolchain producing the optimized trace. This module outputs any source code $C$ that violates at least one trace invariant.

- **Violation Triaging**: this module reduces the large number of violations caused by the same candidate bug by clustering them and extracts a fragment of representative code for each cluster. This simplifies the manual analysis of the candidates to remove false positives and report to developers only highly probable bugs.

2.1 Program generation

The program generation module has the sole purpose of generating source code to be fed to the tested toolchain module. The only requirement for this module is that it must output a source code $C$ that can be correctly compiled by the tested toolchain and run; moreover, to simplify the design of the trace invariants, we assume the generated programs to be well-formed, deterministic, closed (i.e., don’t take any input), and single-threaded.

There are several strategies that can be used when implementing this module. We can follow a purely generative approach, e.g., by using tools like Csmith [27] or Yarpgen [4]; or a more refined mutational approach guided by some feedback loop (e.g., using the coverage of the tested toolchain). In our implementation of the framework, we mainly used the generative approach (leveraging both Csmith and Yarpgen); we also performed an analysis implementing a guided mutational approach that prefers mutated samples increasing the internal coverage of the tested toolchain. However, we did not find it to bring improvements over the pure generative approach. Details are reported in §5.3.

2.2 Tested toolchain

This module integrates the toolchain to be tested by Debug², it takes as input the source code $C$ and it outputs the two traces $T(\text{opt})$ and $T(\text{unopt})$ that will be analyzed by the trace consistency module. More precisely, we model the toolchain’s compiler as an object that takes an input a source code $C$, and outputs a compiled program $P$. We assume that $P$ contains debug information (practically speaking $P$ has been compiled with -g flag). $P$’s code can be either optimized or not, depending on the optimization flags used to customize the compiling phase. Since we are interested in comparing the debug information of an optimized and unoptimized binary generated from the same code $C$, we will indicate with $\text{unopt}$ the unoptimized version of $P$, and with $\text{opt}$ the optimized counterpart.

Using the toolchain’s debugger, we execute $P$ repeatedly stepping over source lines until it exits, obtaining an execution trace $T(P)$ (i.e., an ordered list of elements, each being a step over a source line). We assume the following information are associated to a step $s \in T(P)$:

- **Line information** – Given $s$, the line function $l(s)$ returns either a line in the source code $C$ or a dummy line $\perp$. In practice, this information is provided by the majority of debuggers on executables compiled with debug information. The dummy line $\perp$ models the absence of line information caused by its dropping in some optimization passes; for example, clang sometimes creates line information that points to the artificial line 0, which is the equivalent of our $\perp$.

- **Variables information** – Given $s$, the variables function $V(s)$ returns the set of all global variables, local variables, and parameters visible at step $s$. Given a variable $v$ in $V(s)$ we use $\text{value}(v,s)$ to indicate its value at step $s$. We will use the special value $\perp$ to model optimized out variables; i.e., variables that have been removed by some optimization passes while producing $P$. Function $V$ can be for example implemented in lldb with command frame var.

- **Backtrace information**: Given $s$, the backtrace function $BT(s)$ returns the set of function names present in the stack before the execution of $s$ (n.b., usually this set is ordered, but in our paper, we never make use of such an ordering). An example is the output of the bt command in lldb or gdb.

This set of information will be used to define the four trace invariants presented in §3. However, the framework may accommodate a richer definition of trace based on further information available from the debugger. Our trace invariants assume that, at each step, one of the traces contains correct information. We also assume that the unoptimized trace steps on all the source-lines that are executed, we found this assumption to be true since the compiler, with all optimization turned off, lowers any executable source code line to at least one assembly instruction.

2.3 Trace consistency

The trace consistency module analyzes the $T(\text{unopt})$ and $T(\text{opt})$ traces to find inconsistencies that are likely due to a bug in the toolchain (usually in the compiler or in the debugger). The module takes as input two debug traces $T(\text{unopt})$ and $T(\text{opt})$ and a set of trace invariants. Each invariant takes the two traces $T(\text{Opt})$, $T(\text{Unopt})$, analyses the information contained, and raises a violation in case an inconsistency is found. For each violation, the corresponding source code $C$ is sent to the triaging module. In this paper, we present four trace invariants (see §3), but this module can easily accommodate further invariants that reason on the trace contents.
2.4 Violations triaging

The last module of Debug\(^2\) is used to cluster test cases exhibiting violations that are likely to share the same “root cause”, i.e., that are caused by the candidate bug. This is important to reduce the number of candidate bugs to be verified, as this verification must be performed manually.

Once we find that a certain source code \(C\) generates a violation on the toolchain under test, we extract a fingerprint by testing \(C\) against a set of different versions of the same toolchain (e.g., using several releases of the same compiler). Each of these versions applies a different sequence of optimization passes. We progressively apply the passes in the sequence until we find the first pass that causes the violation; the name of this pass will be included in the fingerprint for the specific toolchain version under analysis; if no violation happens, we include a default value in the fingerprint. Given a source code \(C\), we then obtain a vector \(F(C)\), where each component corresponds to the output of our procedure on a specific version of the toolchain under test. This vector is the fingerprint of \(C\).

For example, consider a source code sample \(C\) that violates a trace invariant on a given toolchain \(X\). To build its fingerprint, we test the same code on \(n\) different versions of \(X\), and check for each version which optimization pass first violates the invariant. Assume the violation happens on pass \(j\) for versions 5.0 and 6.0, while it happens on pass \(k\) for the other versions. Then the fingerprint for \(C\) will be \(F(C) = \{j, j', k'\}\).

Note that building a fingerprint only looking at the toolchain versions, without considering the specific optimization passes where the invariant is first violated, would not permit to discriminate between two bugs that appear in the exact same versions but are in two different optimization passes. For a similar reason, we cannot use a single toolchain version in the fingerprint. We cluster together source samples with the same fingerprint as they link to a single candidate bug with high probability. Then, we select a random sample for each cluster and extract from it a piece of minimal code that allows an easier manual analysis.

3 TRACE INVARIANTS

In this paper, we introduce four trace invariants, designed to detect different problems in a target trace. They are:

- **The Line Invariant (LI)** checks for misstepped lines;
- **The Backtrace Invariant (BI)** checks for spurious frames in the backtrace of a line;
- **The Scope Invariant (SI)** checks if there are out-of-scope variables;
- **The Parameters Invariant (PI)** checks the consistency of the values assumed by function parameters.

Note that our set of four trace invariants does not pretend to be complete; that is, further invariants may be defined that could possibly allow the identification of further bugs whose semantics is not captured by the invariants presented here.

For each of our invariants, we introduce it and justify the assumption behind its design; then, we give a more formal definition and present an example of a real-world bug discovered with it.

**Lines Invariant (LI)** – Lines Invariant checks whether there exists a line of \(C\) that appears to be executed in the optimized trace while the unoptimized trace never executes it. The rationale behind LI is that such a spurious line is either dead code or a glitch of the compiler/debugger (e.g., stepping over a variable declaration); this comes directly from the fact that \(C\) is closed and deterministic, see §\(\S\)2.1 and the unoptimized trace steps on all source lines that are executed.

**Formal definition** – Given a program \(P\) we indicate with \(L(P)\) all the lines in \(T(P)\) (i.e., \(L(P): \{l(s) | s \in T(P)\}\)). The line subset invariant is violated when \(L(opt) \not\subseteq L(unopt)\cup\{\perp\}\). Note that the definition excludes the case in which the line where the violation happens is \(l(s) = \perp\); this was done to model corner cases that arise when using our invariant on specific instances of programming languages, compilers, and debuggers (e.g., the aforementioned line 0 in clang).

**Violation of LI** – Snippet 2 shows an example of a violation of LI in C code compiled with clang and optimization -Og. In this case, we have \(4 \in L(opt)\) and \(4 \notin L(unopt)\); line 4 is clearly dead code, and stepping on it is a bug caused by wrong debug information.

```
int a[1];
int b(char c) { return (k > 2) || (c > 0); }
int c = 0;
int main() { return (b(c) ^ 7) - d; }
```

**Snippet 2: LI violation. Line 4 appears to be executed.**

**Backtrace Invariant (BI)** – The Backtrace Invariant is violated when the trace from optimized code contains a step over a line \(l\) for which the stack backtrace includes a function name that is not present in the stack backtrace of any step in the trace from the unoptimized code that refers to the same line \(l\). The rationale behind BI is that the optimized code cannot reach a target function \(f\) using a path on the call graph that is never used by the optimized code, since this would imply a latent violation of the LI invariant.

**Formal definition** – Given two traces \(T(opt)\) and \(T(unopt)\), the invariant is violated if \(\exists s \in T(opt)\) such that \(\forall s' \in T(unopt)\) with \(l(s) = l(s')\) we have \(BT(s) \not\subseteq BT(s')\).

```
void func_1() { func_2(); }
main() { func_1(); }
```

**Snippet 3: BI violation. Backtrace at line 8 contains func_2.**

```
static int a[1];
static int b = 0;
static int func_2() { return a; }
short int f() { return a; }
for (; g > 0; g = b(g, 2))
  *a = g;
return a;
int main() { return a; }
```

**Snippet 4: SI violation. At line 2 variables \(i, j, k\) are visible, even if out of scope.**

**Violation of BI** – Snippet 3 shows C code that violates BI when compiled with clang and -Og. In this case, despite the absence of inlining in the optimized assembly, a step on line 8 shows an inconsistent backtrace. In our formalism, \(\exists s \in T(opt)\) with \(l(s) = 8\) and
BT(s) = \{main, func_1, func_2\}, while for all steps on line 8 in T(unopt) the backtrace is \{main\}.

**Scope Invariant (SI)** – This invariant searches for lines of source code that are stepped over in both traces, but where there is at least one variable that is only visible in the trace from the optimized code. Essentially, when SI is violated, there is a point in the optimized trace where an out-of-scope variable is visible. This behavior is not desirable during the debugging of an optimized binary as it would provide misleading information that does not match the visibility inferred by looking at the source code.

**Formal definition** – Given traces T(opt) and T(unopt), the invariant is violated if \(\exists s \in T(opt)\) and \(\exists s' \in T(unopt)\) with \(l(s) = l(s')\) and \(V(s) \not\subseteq V(s')\).

**Violation of SI** – In Snippet 4, there is a C code that violates SI when compiled with GCC and -Og and debugged using lldb. In this case, the third time we step on line 2 the variables i, j, k are visible even if they are declared in another scope.

Snippet 5: PI violation, p_6 seems to assume value -1.

**Parameters Invariant (PI)** – With Parameters Invariant, we focus only on the variables that are also parameters of some function \(f\) in the source code C. PI is violated if, in the trace from optimized code, there is a parameter with a value that is never observed in the trace from unoptimized code.

Intuitively, there exists a step where a parameter gets a value that is not consistent with the source code C (a disagreement implies a bug, assuming that one of the traces is correct)\(^1\).

**Formal definition** – Let Par(P) be the set of all function parameters observed in trace T(P), i.e. \(Par(P) : = \{v | \exists s \in T(P) \land v \in V(s) \land v \text{ is parameter of a function } f\}\); for each \(v \in Par(P)\) let \(\text{Values}(v, P)\) be the set of all values \(\nu\) assumes in trace T(P), i.e. \(\text{Values}(v, P) : = \{v\text{al}\ | \exists s \in T(P) \land \nu \in V(s)\land v = \text{value}(v, s)\}\). Then, PI is violated if \(\exists v \in V(opt) \land V(unopt)\) such that \(\text{Values}(v, opt) \not\subseteq \text{Values}(v, unopt)\). We use the \(\subseteq\) symbol to model the special value that is assigned by debuggers to optimized out variables.

**Violation of PI** – Snippet 5 shows C code that violates PI when compiled with clang and -O3. In this case, at line 6 the parameter p_6 is observed in the trace from optimized code having value -1 while its actual value is 1; interestingly, this bug is also visible at line 14 of the same snippet where the value of variable e in the same trace is -1.

\(^1\)We do not have any source of randomness in our programs, which are deterministic. Moreover, we excluded pointers since the memory layout may change if some variables are optimized out (e.g., pointers to global strings, arrays, etc.)

---

### 4 IMPLEMENTATION

We implemented Debug\(^2\) for the LLVM, GNU, and Rust toolchains. We first detail the LLVM implementation and then highlight the differences for other toolchains.

**Program Generation** – In our default program generation module, we generate C source code using Csmith 2.4.0 [27] and use Compcert 3.7 [17] to discard sources containing undefined behavior. Moreover, we also discard generated sources for programs that do not terminate in a fixed number of steps when executed.

Sources are compiled with and without optimization flags to obtain the program pairs. The optimized program is obtained by using the default sequence of optimization passes as defined by a given optimization switch (e.g., level -O2 in clang); that is, we never use a custom order for optimization passes.

**Tested toolchain** – The toolchains under analysis are integrated in Debug\(^2\) using Python, following the design detailed in §3. The only variation is that, when checking for invariant PI, we discard all pointers because they are not guaranteed to be stable across executions (e.g., due to ASLR and non-deterministic heap allocators).

**Violations triaging** – We implemented our fingerprinting mechanism using the opt-bisect tool [2], a tool provided by LLVM to bisect the optimization stages. Given C code that violates an invariant, we consider six versions of clang (i.e., 5, 6, 7, 8, 9, trunk), and on each of them, we use opt-bisect to pinpoint the optimization pass that generates the violation. We cluster together code samples that trigger violations with the same fingerprint and select a representative test case at random. We use C-Reduce 2.10 [21] to reduce the test case to a minimal source snippet triggering the same violation. We manually analyze the reduced test case to confirm that the violation is caused by a bug before reporting it to the developers.

**Other toolchains** – The implementation for the GNU and Rust toolchains are similar to the LLVM one, but with the following notable exceptions. Since there seems to be no mature program generator for Rust, we simply collected a variety of source samples from the Rust test suite\(^2\). Our final dataset is composed by 12,616 source samples. We implemented the toolchain module for gdb using the pygdb library [23], which leverages the machine interface (gdb/mi) provided by gdb. We performed trace extraction similar to the LLVM toolchain using the specific commands for gdb. Finally, since the GNU and Rust toolchains have no equivalent of opt-bisect, in our experiments, we implemented the triaging procedure by means of manual analysis.

\(^2\)https://github.com/rust-lang/rust/tree/master/src/test
5  EVALUATION

Our experimental evaluation aims at answering the following main research questions:

RQ 1: Is there a relationship between invariant violations and optimization levels? And between violations and actual bugs?

RQ 2: Does Debug\textsuperscript{2} find real bugs in our reference toolchain (LLVM) across different optimization levels and components (e.g., compiler and debugger)?

RQ 3: Does Debug\textsuperscript{2} generalize to different toolchains and programming languages?

We also report results that shed light on other interesting aspects: the relationship between optimization levels and certain bugs, the optimization passes that are more prone to bugs, and how our framework fares when swapping the program generation module.

To answer all the questions above, we evaluated Debug\textsuperscript{2} on various target toolchains using the prototype described in §4. The main focus of our evaluation is the LLVM toolchain for the C language, namely the clang compiler and the llvm debugger. We also ran tests on the GNU toolchain (GCC and gdb) and on the Rust toolchain (rustc and lldb) aimed at investigating the generality of our framework among different toolchains and programming languages. All our experiments run on Google Cloud VMs equipped with 32GB of RAM and 8 cores each; experiments run in Docker containers with Ubuntu 18.04.

5.1  Trace Invariant Violation Analysis

As a first step in our experimental evaluation, we want to analyze if and how much the trace invariants defined in §3 are suitable to identify inconsistencies in the traces produced by the LLVM toolchain. Moreover, we want to study how these violations are distributed across optimization levels and if our fingerprinting mechanism can help reduce the manual effort by clustering violations sharing the same root cause.

We ran 24 hours of tests with optimization levels -O\{1, 2, 3, 4, 5\} and stopped after collecting 7,500 test cases for each level. We tested each optimization level independently, i.e., the set of source samples generated by Csmith was different from level to level. We ran experiments using LLVM version 11.0.0 commit b7397e81a7d2ed267b (initial version). We generated 1k test cases compiled using the initial version at -Og and we found 1,956 violations. From April 3 to the end of June we reported several bugs (additional details in the next section), 14 of which have been patched. We then considered the LLVM version of June 26, 2020 (containing all the patches of our bugs) as the final version. We tested the 1k test cases again on the final version and found only 7 violations. The total breakdown (initial version / final version) by invariant is LI from 31 to 6, SI from 755 to 0, BI from 9 to 0, and PI from 1,161 to 1. The total decrease from an average of 1.9 violations per test case to 7 · 10\textsuperscript{-3} shows that the vast majority of violations found in the initial version (over 99%) was caused by a bug. This confirms that an invariant violation has a pathological cause most of the time. During this experiment, we found that three bugs generate most violations: a first bug generates 1,144 violations, mapped to 17 fingerprints; a second bug generates 756 violations, mapped to 120 fingerprints; a third bug generates 41 violations, mapped to 11 fingerprints. This also shows that fingerprints effectively cluster violations from the same bug reducing the number of test case to analyze manually.

For each invariant, we then collected the unique fingerprints and discarded all duplicates (i.e., we considered a single representative test case per cluster). Looking at the violations found for optimization level -Og, we have 11 unique fingerprints for the LI violations, 1 for SI, 3 for BI, and 3 for PI. However, the total number of unique fingerprints across all violations is 16 (the total is not reported in the table), i.e., less than the sum of unique fingerprints for each invariant since a single fingerprint may incur different invariant violations.

The large gap between the total number of violations and the number of unique fingerprints can be explained by the fact that a single bug may trigger several violations. For example, by analyzing the LI violations, we found that 41 are on a ternary operator; these are all caused by a single bug (see bug https://bugs.llvm.org/show_bug.cgi?id=48895) and they map to 5 different fingerprints. For BI, we have 47 violations generated by the same bug on the same file, justifying only 3 unique fingerprints. Regarding the PI violations, we found the same behavior mentioned above for optimization levels O\{2, 3, 4\}. In general, these results show that our framework, equipped with our four trace invariants, can identify a fairly large number of violations across several optimization levels.

Relationship with bugs and fingerprints – We ran an experiment to understand the number of violations caused by a bug. We started our analysis on April 3, 2020 with LLVM commit b7397e81a7d2ed267b (initial version). We generated 1k test cases compiled using the initial version at -Og and we found 1,956 violations. From April 3 to the end of June we reported several bugs (additional details in the next section), 14 of which have been patched. We then considered the LLVM version of June 26, 2020 (containing all the patches of our bugs) as the final version. We tested the 1k test cases again on the final version and found only 7 violations. The total breakdown (initial version / final version) by invariant is LI from 31 to 6, SI from 755 to 0, BI from 9 to 0, and PI from 1,161 to 1. The total decrease from an average of 1.9 violations per test case to 7 · 10\textsuperscript{-3} shows that the vast majority of violations found in the initial version (over 99%) was caused by a bug. This confirms that an invariant violation has a pathological cause most of the time. During this experiment, we found that three bugs generate most violations: a first bug generates 1,144 violations, mapped to 17 fingerprints; a second bug generates 756 violations, mapped to 120 fingerprints; a third bug generates 41 violations, mapped to 11 fingerprints. This also shows that fingerprints effectively cluster violations from the same bug reducing the number of test case to analyze manually.

5.2  Manually analyzed and reported bugs

We present a quantitative analysis of the bugs we reported. A more qualitative analysis of selected bugs is available in §6.

Table 1: Violations found for each optimization level in LLVM.

<table>
<thead>
<tr>
<th>Opt.</th>
<th>All Violations</th>
<th>Unique Fingerprints</th>
</tr>
</thead>
<tbody>
<tr>
<td>LI</td>
<td>SI</td>
<td>BI</td>
</tr>
<tr>
<td>O1</td>
<td>54</td>
<td>3</td>
</tr>
<tr>
<td>O2</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>O3</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Og</td>
<td>54</td>
<td>1</td>
</tr>
<tr>
<td>Os</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Oz</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
LLVM toolchain
We collected several violations found by running Debug\(^2\) on the latest LLVM release from April 3, 2020. We used optimization levels -O[0,1,2,3,fast], clustered violations using their fingerprints, extracted the representative test cases’ source code from the clusters, and reduced them to smaller test cases with C-Reduce.

Analysis of bugs found – In total, we reported 23 unique bugs. As expected, our system identified bugs in the full toolchain: 16 bugs found on clang, and 7 on lldb. Table 2 presents a breakdown of the bugs by triggered invariant\(^1\). As shown in the table, the distribution follows the one we reported in §5.1.

Regarding the distribution of bugs between compiler and debugger, for LI we discovered only bugs in clang, while for BI, SI, and PI we observed mixed results: SI exposed 2 bugs in clang and 1 in lldb; PI exposed 3 bugs in lldb and 1 on clang; BI exposed 1 bug in clang and 3 bugs in lldb. This behavior stems from source-line stepping being mostly dependent on the information present in the DWARF line table, which is created entirely by the compiler, while the debugger is only responsible to parse such information. Since LI is the invariant that is more likely to find misaligned or wrong line information, most of LI violations come from compiler bugs. PI and SI find bugs in both components of the toolchain since they both assess the ability of the compiler to generate a correct DWARF table (see bug https://bugs.llvm.org/show_bug.cgi?id=45923) and the ability of the debugger to correctly parse it (see bug https://bugs.llvm.org/show_bug.cgi?id=46181). Interestingly, most of the BI bugs are in the debugger. This is sensible as the debugger does the majority of work to build a reliable stack trace using its unwinder (one of the bugs found is indeed in the lldb unwinder). Nevertheless, we also found a BI bug in clang (see bug https://bugs.llvm.org/show_bug.cgi?id=45883). Apart from the bugs above, we also reported 4 additional bugs that are duplicated (i.e., on the 27 bugs that we reported 23 had not been marked as duplicate). This low rate of duplicates is encouraging, as it shows that violation triaging works well when it is possible to identify the optimization pass that triggers the violation.

Affected versions – We tested for the presence of the 16 clang bugs on different versions of LLVM. Figure 2 presents a histogram of bugs per LLVM version. As shown in the figure, most of the bugs we found have been latent for years: for example, roughly half of the clang bugs also affect LLVM version 5.0, which was released in 2017.

Affected optimization passes and levels – As shown in Figure 2, most of the bugs plague optimization levels -Og and -O1. This can be explainable by the fact that those are the levels that drop less debug information. From our analysis, we found that -Og and -O1 share the same set of bugs (i.e., the 11 reported bugs are exactly the same) and the majority of these bugs are not present in -O[2,3,fast]. This is not surprising since, after some investigation, we found that, in LLVM, -Og is just an alias for -O1 [1]. Table 3 presents a breakdown of the 16 clang bugs by the affected optimization pass. Interestingly, the more affected passes are the ones that either move instructions across the blocks of the CFG or that change the structure of the CFG. We speculate that is due to the difficulty of forecasting the effect of such transformations on debug information and due to the lack of specific guidelines on how to preserve debug information during these passes (more details are in §7).

Other toolchains
Our analysis on LLVM has answered RQ 2. In this section, we report on the experiments we ran on other toolchains to answer RQ 3.

GNU toolchain – We performed the same experiments on the GNU toolchain (GCC and gdb). For this toolchain, we reported a total of 8 bugs, 2 of which have been confirmed (Table 4). Interestingly, we could not find a bug that generates a SI. We tested the bugs on versions 5, 6, 7, 8, 9, 10, trunk, and found that the bugs affect only version 8 and more recent versions. This is probably due to a major change in the generation of DWARF debug information. On the trunk version, the breakdown by optimization level is 6 bugs for -Og and 4 bugs for -O[1,2,3,fast]. This confirms our hypothesis that

### Table 2: Breakdown by invariant of reported bugs for LLVM.

<table>
<thead>
<tr>
<th>Triggered Invariant</th>
<th>Unconfirmed</th>
<th>Confirmed</th>
<th>Patched</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines Invariant (LI)</td>
<td>0</td>
<td>4</td>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>Backtrace Invariant (BI)</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Scope Invariant (SI)</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Parameters Invariant (PI)</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

### Table 3: LLVM bugs by affected optimization passes.

<table>
<thead>
<tr>
<th>Optimization Pass</th>
<th># bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop Strength Reduction</td>
<td>1</td>
</tr>
<tr>
<td>Machine Common Subexpression Elimination</td>
<td>1</td>
</tr>
<tr>
<td>Loop Invariant Code Motion</td>
<td>2</td>
</tr>
<tr>
<td>Control Flow Optimizer</td>
<td>1</td>
</tr>
<tr>
<td>Stack Slot Coloring</td>
<td>1</td>
</tr>
<tr>
<td>Control Flow Optimizer</td>
<td>2</td>
</tr>
<tr>
<td>Simplify the CFG</td>
<td>3</td>
</tr>
<tr>
<td>X86 DAG -&gt;DAG Instruction Selection</td>
<td>1</td>
</tr>
<tr>
<td>Branch Probability Basic Block Placement</td>
<td>3</td>
</tr>
<tr>
<td>Rotate Loops</td>
<td>1</td>
</tr>
</tbody>
</table>

### Table 4: Breakdown by invariant of reported bugs for GNU.

<table>
<thead>
<tr>
<th>Triggered Invariant</th>
<th>Unconfirmed</th>
<th>Confirmed</th>
<th>Patched</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines Invariant (LI)</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Backtrace Invariant (BI)</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Scope Invariant (SI)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Parameters Invariant (PI)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
optimization levels preserving more debug information are more vulnerable. Interestingly, we found the same set of bugs for levels -O[2, 3, fast] even if the optimization strategy differs among these levels.

**Rust toolchain** – We used the latest rust nightly build (1.46.09) for the compiler and ldad as debugger. Since our dataset has been built using the Rust test suite, we cannot guarantee that all the sources in the dataset are deterministic and without undefined behaviours. In this case, we do an a-posteriori manual analysis on the sources arising violations to check that they are deterministic and without undefined behaviours.

We compiled each sample in our dataset with the -g flag and optimization levels -O[1, 3]. We used each optimized binary as input to our trace consistency subsystem along with the non-optimized one. We manually analyzed a selected set of violations, identifying and reporting 3 bugs from 3 invariant violations: 1 LI, 1 SI, and 1 PI.

### 5.3 Other program generators

As noted earlier, the program generation component of Debug² is interchangeable by design. We show how changing the program generation component impacts the number of violations we can find, while at the same time confirming our Debug² framework is extensible by swapping components.

To this end, we ran the whole pipeline again (as described in §5), but replaced the default Csmith generator with 3 alternative approaches to program generation. We also targeted a more recent clang/LLVM version (commit 94e4e37d5564), in which numerous bugs that we reported had been fixed.

Table 5 shows the median number of unique fingerprints and violations for 20 runs of 12 hours per code generation tool. We highlight statistically significant values (Mann-Whitney \( p < 0.05 \)), as compared to the Csmith baseline. As shows in the table, yarpgen is capable of finding a large number of new invariant violations not previously found by Debug² using Csmith as the default program generation component. We also show how a simple code mutation-based approach based on MUSIC \([19]\) fares against the generation-based approach taken by Csmith and yarpgen. The mutation-based approach applies a number of (randomly selected) mutation operations on 5,000 initial seeds generated by Csmith and yarpgen. While our results suggest that a generation-based approach outperforms this simple mutation-based approach, this experiment does show Debug² generalizes to different classes of program generators.

### 6 ANALYSIS OF DISCOVERED BUGS

In this section, we detail some of the most interesting bugs we reported.

#### Table 5: Median number of fingerprints found per tool (N=20).

<table>
<thead>
<tr>
<th>Tool</th>
<th>Median fingerprints (+O)</th>
<th>Median fingerprints (+O)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Csmith</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>yarpgen</td>
<td>11 (264)</td>
<td>14 (349)</td>
</tr>
<tr>
<td>MUSIC-Csmith</td>
<td>2 (5)</td>
<td>8 (13)</td>
</tr>
<tr>
<td>MUSIC-yarpgen</td>
<td>11 (251)</td>
<td>13 (331)</td>
</tr>
</tbody>
</table>

#### Snippet 6: lldb bug 46398

**lldb bug 46398** (Snippet 6) – At line 10, lldb shows two incoherent backtraces when the analyzed binary is compiled with -Og and -O0. Specifically, it shows that variable `k` is a function in the stack trace.

The unwinder is the subsystem in lldb responsible for reconstructing the stack chains when stopped at a given address. The debugger implements unwinding as a series of plans which kick in one after another until one is successful (or the stack cannot be reconstructed, throwing an error). The wrong backtrace was caused by a bug in the unwind choosing strategy, which did not take into account the availability of additional information to pick the best strategy.

#### Snippet 7: lldb bug 46456

**lldb bug 46456** (Snippet 7) – presents a bug in the DWARF parsing and interpretation logic of lldb. This bug was discovered performing a “mix and match” testing using GCC as the compiler and lldb as the debugger. lldb shows a flow where it incorrectly appears that the function `b()` is called twice.

#### Snippet 8: clang bug 46008

**clang bug 46008** (Snippet 8) – When stepping through optimized code, lldb steps on a source line that is dead (cannot be executed). In this case, it shows the execution of line 7. This is due to the SimplifyCFG optimization pass in LLVM, which performs peephole optimizations of the control flow graph, like flattening or block merging, sometimes without proper updates to line information.
while applying the transformations. In this specific case, the pass tries to fold a dead block into a live one, wrongly dropping the location. As a result, the debug information points to a "dead" line.

```c
static int a, c, d, e;
static void dm() {
    int b = 0;
    for (; b < 56; b++)
        a = b;
}
int main() {
    dm();
    for (; c < 3; c++)
        printf("%s", c);
}
```

**Snippet 9: GCC bug 95077**

**GCC bug 95077 (Snippet 9)** – In this case, at line 14 GDB shows that the top frame of the stack trace is function `dm`, despite line 14 being in the main. This bug stems from how GCC handles debug information of inlined functions. During the optimization passes, function `dm` is inlined; the mapping between inlined assembly instructions and the `dm` symbol is kept by DWARF using a table that associates ranges of assembly instructions to inlined functions.

In this case, GCC wrongly includes an assembly instruction of the loop at line 14 in such ranges. Therefore, when GDB parses this information, it wrongly shows that line 14 belongs to inlined function `dm`.

7 DISCUSSION

In this sections, we discuss the insights that we learned during our study, as well as the limitations of our work.

The need for shared guidelines

During our investigation (see §5.2 and §6) we reported several bugs that lead to interesting discussions among the toolchain developers. During this process we realized that there has never been an effort to precisely define how to preserve debug information during the optimization passes of production compilers (we are not aware of guidelines in this sense). Therefore, several of our bugs required non-trivial fixes decided after lengthy discussions among toolchains developers.

```c
int add(int u11, int u12) {
    return u11 + u12;
}
int g_4 = 3, a;
int *b[] {
    if (g_4)
        return g_4;
    a = 0;
    return g_4;
}
int main() { b(); }
```

**Snippet 10: LSI violation in clang when compiled with -Oz. Line 2 appears to be executed.**

A prototypical example is the bug reported in the Snippet 10, where the code compiled with clang and optimization level `-Oz` shows a step on line 2. If we look at the source code, it is apparent that line 2 is never executed; and so it is in the unoptimized program. However, in the optimized binary, due to a loop rotation optimization kicking in [13], line 2 is indeed executed before reaching the end of the loop. One could argue either against or in favor of showing the step at line 2

On the one hand, one could follow the general principle of optimization transparency pioneered by Hennessy [10]: showing the step could mislead a developer, that would see a step over a location that should not be executed, this could be extremely confusing on large and complex software. On the other hand, it could be reasonable to show what is precisely happening without masking the optimization (following the line of Brooks et al. [5]): line 2 is executed even if this is done speculatively and without observable side effects.

In this specific case, clang developers finally decided that such behavior is indeed a bug and that the step at line 2 should have not been shown.

Similar design choices should derive from a broader discussion on the semantics of optimized debug information, since case by case fixes, in the long term, are likely to lead to inconsistent behaviors. Hence, our investigation is a precursor of a more in-depth process that should systematically study how to preserve debug information during optimization passes, taking specifically into account standards for debug information (such as DWARF as we will discuss in the next section).

During the production of this paper, the LLVM developers published a document containing a set of basic rules to update debug information during certain optimization passes [14]. The content of this document has clearly been influenced by the bugs we reported, and it is a first step in the direction of defining shared guidelines.

**DWARF shortcomings**

During our analysis, we encountered some bugs that highlighted a shortcoming of the DWARF standard that severely limits its expressivity, and impacts either the correctness or the completeness of the debug experience. In the following, we will show two examples, one for GCC and the other for clang, showcasing this problem.

```c
int g_4 = 3, a;
int *b[] {
    if (g_4)
        return g_4;
    a = 0;
    return g_4;
}
int main() { b(); }
```

**Snippet 11: GCC bug 95865**

**GCC bug 95865 (Snippet 11)** – In this example GDB steps on dead code; it executes line 6 instead of 4. The culprit is an optimization pass that modifies the CFG; in this case line 4 and line 6 are lowered into a single assembly instruction while the instruction `a=0` becomes guarded by the condition `g_4==0`. In DWARF, each assembly instruction is mapped to at most a single line of source code; hence, this case imposes a choice: the compiler may map the instruction to line 4 or line 6 (in this case a bug arises depending on the value of `g_4`), or completely drop the source code location on the return (impacting the amount of information shown). Therefore, we have an unavoidable tradeoff between correctness (dropping the line) and completeness (picking a line and accepting a bug in some runs).
```c
volatile int a, b, c;
int g_3[2];
int main(void) {
  for (; b > -9; b--)
    
  for (; c <= 5; c++)
    if (c)
      
  while (1)
}
```

_Snippet 12: clang bug 45523_

Clang bug 45523 (Snippet 12) — shows a case where lldb hits the body of a dead branch, stepping on line 12 despite the condition in the `if` statement is true. The loop invariant code motion pass in LLVM tries to sink load/store instructions outside of the loop, and when there are several that have different locations, just picks an arbitrary one (in general, incorrect).

This bug was fixed by collecting all the locations, and merging them in a single one, to be shown in the debugger, as long as they agree. In case there are different locations, the final location of the sunk instructions is dropped. For the reasons explained in the previous bug, this fix is inherently an approximation of the best behavior.

A possible future proposal for Dwarf is to allow a single instruction to be mapped to multiple locations, letting the consumers (e.g., the debugger) decide the policy to adopt.

8 LIMITATIONS, AND THREATS TO VALIDITY

In our system, we have to generate programs that are free from undefined behaviors to avoid spurious violations of an invariant. The presence of undefined behavior could result in a non-deterministic program artificially inflating the number of violations. We use CompCert to filter programs with undefined behavior. While this is not provably perfect and does not rule out the possibility that undefined behavior is still present, we have never observed one during our tests. This is probably also because our code generation leverages Clsmith that, as remarked in previous work [18], uses several techniques to avoid the generation of source code containing undefined behavior [27].

Similarly to all previous works using differential testing, Debug2 cannot detect a bug that is present in both analyzed traces. This issue could be solved by designing invariants that work on a single trace, based on assumptions on the code behavior to detect inconsistencies. Future work could investigate this aspect.

**Required manual effort** — Whenever a violation is found, we must manually check if the root cause is a bug. While Debug2 does not yet provide full automatic triaging, we found that our triaging module significantly speeds up the analysis of violations. The analysis of a bug is made up of time spent running Debug2, clustering and fingerprinting, extracting a minimal sample and manually analyzing and reporting. Most of this time is spent by running the software; on average we spent between 1 and 3 man-hours to triage each bug and report it to the developers.

Incomplete debug information — Debug2 is not designed to operate in environments actively discarding all debug information. In a hypothetical extreme case in which all the debug information were dropped we would not have any invariant violation. Fortunately, modern toolchains strive to retain as much information as possible, also at high optimization levels. For example, -Og is designed to provide a debug friendly experience, while studies show that -O2 (common in production) preserves 50% of line information and 75% of variable information (see [20]). Debug2 also found bugs at -O3. Therefore, at least with modern and popular compilers, the drop of debug information is not a practical concern.

9 RELATED WORK

The investigation on the correctness of debug information started with the work of Yuanbo Li et al. [18]. The paper focuses on validating the correctness of the values of the variables shown by a debugger for optimized code for statically compiled languages. Their idea is to derive from a certain source code a modified program, built with optimizations, in which a debugger can stop at a predetermined line and print the value of a specific variable without triggering undefined behavior. Once stopped, they compare this value to the one printed, at the same program point, by an unoptimized binary.

Our work generalizes their: they are only looking at one aspect of debug information, consistency of variables information, while they neglect other aspects (e.g., they assume line information to be correct). Apart from [18], the majority of the other works in this area look at loosely related problems, and they can mainly be partitioned in the works on improving the experience of debugging optimized binaries, the ones that focus on testing the correctness of debuggers, and the massive amount of work devoted to compilers testing.

**Debugging of optimized binaries** — Generating debug information for optimized binaries is a long-studied problem. The seminal work of Hennessy [10] proposed a categorization of the effect of optimizations on the variables of a program. It also introduced an algorithm to identify endangered variables (variables whose values are possibly not correct in the optimized program). The main idea behind his approach was to make optimizations transparent and provide the user with the expected behavior. Building on top of [10], [3, 8, 26] proposed better approaches for identifying endangered variables.

Conversely, from [10], Brooks et al. [5] proposed an alternative principle to follow: try to depict what is happening in the optimized code, without striving for the illusion of transparency. To reach this goal, they aim to provide the user with visual-feedback highlighting pieces of source code during interactive debugging. Finally, to identify possible bugs in optimized code, Jaramillo et al. [11], proposed comparison-checking between unoptimized and optimized code after executing them under the same input. We remark that they are interested in finding miscompiles introduced by the optimization passes, and not in wrong debug information.

**Testing of debuggers** — Several works investigate the problem of finding debugger bugs. A differential testing approach was taken by Lehmann et al. [16]; they proposed to record debug traces for multiple debuggers and compare them to identify bugs. Such an approach can be used only when multiple mature debuggers are available. Considering a non-differential approach, Tolksdorf et al.
[25] showed how it is possible to identify bugs in Chromium debugger using metamorphic testing. We point out that none of the aforementioned works is able to find bugs in the compiler.

Compilers testing – The problem of compiler testing received a lot of attention in the past decade [15, 24, 27] and it mainly focused on finding miscompiles in optimizing compilers. For a recent survey on compiler testing techniques, see Chen et al. [6]. This research line is complementary to our since it is devoted to test the correctness of machine code generated by the compiler without caring about debug information.

10 CONCLUSIONS
This paper introduced Debug2, a framework to expose debug information bugs in production toolchains. Debug2 fills an important gap in the literature, where little attention has been devoted to scrutinizing the full debug information lifecycle for bugs. This is particularly problematic when debugging happens on production software, where binaries have been heavily optimized, and debug information bugs may hinder post-deployment debugging efforts. Debug2 relies on trace invariants to perform differential analysis on debug traces of optimized and unoptimized programs, and expose inconsistencies whose root cause may be a bug in the compiler or in the debugger. We evaluated Debug2 on three different toolchains (namely LLVM, GNU, and Rust), finding 34 new bugs. Most bugs have already been fixed by the developers. Furthermore, our findings have already sparked an interesting discussion among developers and fostered further investigation on the semantics of debug information for optimized programs.

Acknowledgments – We would like to thank our shepherd, James Bornholt, and the anonymous reviewers for their comments. The VU Amsterdam researchers were supported by the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 786669 (ReAct) and by Cisco Systems, Inc. through grant #1138109. Giuseppe Di Luna was supported by the AXA Postdoctoral Fellowship. Sapienza researchers are partially supported by the SAFE project under the Italian PNRM program and by Sapienza grant num. RM11916B75A3293D. The authors would like to thank Daniele Cono D’Elia for constructive criticism and insightful comments on a previous version of this manuscript.

REFERENCES