Closer to the Edge: Testing Compilers More Thoroughly by Being Less Conservative About Undefined Behaviour

Karine Even-Mendoza
k.even-mendoza@imperial.ac.uk
Imperial College London

Cristian Cadar
c.cadar@imperial.ac.uk
Imperial College London

Alastair F. Donaldson
alastair.donaldson@imperial.ac.uk
Imperial College London

ABSTRACT

Randomised compiler testing techniques require a means of generating programs that are free from undefined behaviour (UB) in order to reliably reveal miscompilation bugs. Existing program generators such as Csmith heavily restrict the form of generated programs in order to achieve UB-freedom. We hypothesis that the idiomatic nature of such program limits the test coverage they can offer. Our idea is to generate less restricted programs that are still UB-free—programs that get closer to the edge of UB, but that do not quite cross the edge. We present preliminary support for our idea via a prototype tool, CsmithEdge, which uses simple dynamic analysis to determine where Csmith has been too conservative in its use of *safe math* wrappers that guarantee UB-freedom for arithmetic operations. By eliminating redundant wrappers, CsmithEdge was able to discover two new miscompilation bugs in GCC that could not be found via intensive testing using regular Csmith, and to achieve substantial differences in code coverage on GCC compared with regular Csmith.

CCS CONCEPTS

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

KEYWORDS

Compilers, fuzzing, Csmith, GCC

ACM Reference Format:
Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. 2020. Closer to the Edge: Testing Compilers More Thoroughly by Being Less Conservative About Undefined Behaviour. In *Proceedings of 35th IEEE/ACM International Conference on Automated Software Engineering (ASE 2020).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Randomised testing tools (fuzzers) have proven successful at finding bugs in mature compilers [4, 10, 14–17, 20]. To find miscompilation bugs—where a compiler silently generates wrong code—such fuzzers typically employ cross-checking: comparing the behaviour of a program after compilation by distinct compilers, or the behaviour of multiple equivalent programs built by a single compiler. Result mismatches point to miscompilation bugs that can be investigated. Cross-checking avoids the *oracle problem* [1]: the approach flags up *differences* in results that are expected to be the same, and does not require knowing which (if any) result is correct.

Avoiding the oracle problem in this manner when testing C and C++ compilers relies on a source of programs that are free from *undefined behaviour* (UB). Examples of UB in C include using uninitialised variables, accessing invalid pointers, overflow of signed integer arithmetic operations, division by zero, and *unsequenced* accesses to variables [6]. A program that exhibits UB has arbitrary semantics, so optimising compilers are free to assume that input programs are free from undefined behaviour (*UB-free*) and optimize them based on that assumption. In practice, compilers do take advantage of UB to generate efficient code [19].

If a program has UB, the results of cross-checking between compilers are meaningless because the program can legitimately yield any result when executed. Similarly, it is meaningless to cross-check a single compiler against multiple related programs if one of the programs exhibits UB.

To counter this, compiler fuzzers go to great lengths to generate UB-free programs. In the case of e.g. signed integer addition, a compiler fuzzer could (1) limit the form of generated programs so that it is feasible to track the possible values of all program variables at all program points, and use this tracking to ensure that generated instances of signed addition do not overflow, or (2) avoid limiting the overall form of generated programs, but guard every instance of a signed addition operation with an overflow check that only performs the addition if it does not overflow.

The problem with such strategies is that generated programs have a restricted, idiomatic form, which can limit the extent to which they exercise the compiler under test. For example, certain peephole optimisations on arithmetic operations may be inapplicable if every arithmetic operation is enclosed in a conservative check for potential UB.

In this idea paper, we propose investigating ways to generate programs that get "closer to the edge" of UB: methods that are less restrictive in the measures taken to avoid UB, while still achieving such avoidance. We have two ideas in this regard: (a) modifying programs post-generation to make them less restricted, guided by dynamic analysis, and (b) relaxing generation-time restrictions so that programs that exhibit UB *can* sometimes be generated, and using dynamic analysis to detect and discard those that do.

We provide preliminary experimental evidence in support of idea (a), in the context of the Csmith compiler fuzzer [20] and the popular GCC compiler. We present a prototype tool, CsmithEdge, that uses a simple dynamic analysis to identify and eliminate redundant safety checks for UB on arithmetic operations in Csmith-generated programs. We show that CsmithEdge-generated programs can reach parts of the compiler that cannot be reached with extensive testing using Csmith. CsmithEdge has already discovered two new miscompilation bugs in GCC—a compiler that has been extremely

ASE 2020, 21 - 25 September, 2020, Online
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

well-tested by several compiler fuzzing tools for many years—which were promptly fixed by the GCC developers after we reported them.

In the remainder of the paper we describe how Csmith avoids UB (§2) and our CsmithEdge approach to avoiding redundant UB checks on arithmetic operations, with experiments showing the increased bug finding ability and more thorough compiler code coverage that this brings to GCC (§3). We conclude with future ideas for our broader vision in this area (§4).

2 CSMITH AND UB AVOIDANCE

Csmith [2, 20] generates deterministic C programs that are free from undefined and unspecified behaviour. This makes them well-suited to compiler testing approaches based on cross-checking. A Csmith program takes no input, and on termination prints a single value obtained by hashing the final values of the program's global variables. A Csmith–generated program thus has a single path; this is an important property that we take advantage of in our case study (§3).

Csmith achieves UB-freedom via a combination of *generation-time analysis* and *runtime checks*. For example, a generation-time pointer analysis is used to ensure that accesses via non-null pointers are to valid data, and runtime checks are used to avoid dereferencing null pointers.

Runtime checks are also used to avoid UB related to integer arithmetic. This is achieved via *safe math* wrappers. Instead of directly issuing an integer arithmetic expression $a \circ b$ (for some operator $\circ$), Csmith instead invokes a *safe math* wrapper for the operation. The wrapper returns $a \circ b$ if there would be no associated UB, and otherwise returns some safe value (in practice, the value a). More formally, a safe math wrapper for an operation $a \circ b$ has the form:

$$unsafe(a, b, \circ)\ ?\ a\ :\ a \circ b$$

where $unsafe(a, b, \circ)$ is an *unsafe check* that returns true if and only if the operation $a \circ b$ would trigger UB.

Csmith offers safe math wrappers in the form of *functions*, e.g. (for signed integer division):

```
int32_t safe_div(int32_t X, int32_t Y) {
  return (Y == 0 ? X : (X / Y));
}
```

and *macros*, e.g. (again for signed integer division, and simplified for readability):

```
#define int32_t safe_div(_X, _Y) \
  ({int32_t X = (_X); int32_t Y = (_Y); \
  Y == 0 ? X : X / Y;})
```

The user of Csmith can decide which to use by including an appropriate header file. The function and macro forms of these wrappers are intended to be semantically equivalent.Our understanding from talking to the Csmith developers is that the function wrappers (the default) are preferred because they are simpler to maintain. However, unless functions are fully inlined, they may inhibit compiler optimisations; a problem that the macros do not suffer from.

3 RELAXING SAFE MATH CHECKS

Using safe math wrappers eliminates arithmetic UB in a simple way. The price for this is that arithmetic operators that are potentially UB-prone *never* appear in a Csmith-generated program in a raw form. They are always enclosed in a safe wrapper, as the third argument to a conditional (ternary) operator of the form "?...:". As the conditional operator introduces control flow (due to short-circuit evaluation), the rather prescriptive program format arising from this blanket use of conditionals may bias the optimizations that a compiler applies to Csmith-generated programs, possibly reducing the extent to which Csmith can find bugs in other optimisations.

3.1 Identifying Redundant Checks

A safe math check in a Csmith-generated program is *redundant* if the program is still free from UB after removing the check. To identify and eliminate redundant safe math checks in a generated program, we temporarily replace the ith occurrence of a safe math wrapper of the form $unsafe(a, b, \circ)\ ?\ a\ :\ a \circ b$ with:

$$unsafe(a, b, \circ)\ ?\ \texttt{WARN}(i), a\ :\ a \circ b$$

where, if executed, $\texttt{WARN}(i)$ prints a message indicating that the ith safe math wrapper is truly necessary. The semantics of the C comma operator means that in the case where $\texttt{WARN}(i)$ is executed, the entire ternary expression evaluates to a, just as it would if $\texttt{WARN}(i)$ were not present.

Because a Csmith program takes no input and thus exhibits a single execution path, running the transformed program once immediately reveals the subset of safe math wrappers that are actually needed. We then prune all but these wrappers from the original program. This yields a program that is still free from UB, because all the *necessary* safe math wrappers are in place, but that may have significantly fewer safe math wrappers overall, because all the *redundant* safe math wrappers are gone, meaning that its use of arithmetic is correspondingly less constrained.

As an example, consider the following contrived program, which is similar in spirit to (though much smaller than) a program that Csmith might generate, where `safe_lshift`, `safe_add`, `safe_div` and `safe_mul` are safe math wrappers for the signed integer operators «, +, / and *, respectively:

```
int main() {
  int s = 5;
  int t = 2147483646;
  s = safe_lshift(s, 14);        // (i) redundant
  for (int k = 8; k >= -8; k--) {
    s = safe_add(s, k);          // (ii) redundant
    t = safe_div(t, k);          // (iii) necessary
  }
  t = safe_mul(safe_mul(s, t), s); // (iv) inner
                      // redundant, outer necessary
  printf(hash(s,t));
}
```

Our approach identifies the wrappers at locations (i) and (ii) to be redundant and the wrapper at (iii) to be necessary (because the divisor `k` passes through 0). The inner wrapper at location (iv) is redundant, because `s` and `t` are small enough that their product does not overflow, but multiplying this product again by `s` would lead to overflow so that the outer wrapper at (iv) is necessary.

Specifically, execution of this program with our modification lists two locations with UB: the `safe_div` call in the loop (for the iteration when `k` is 0) and the outer `safe_mul` call after the loop (when attempting to compute $81920 * 81920$). These two safe math

wrappers are thus kept, and all others are removed (e.g., location (ii) becomes `s = s + k;`).

3.2 Preliminary Evaluation

We have implemented our approach in a prototype tool called CsmithEdge, as a set of bash scripts on top of Csmith [11], and available at [12]. For a given integer seed, CsmithEdge runs Csmith to produce a C program, and uses the approach described in §3.1 to obtain a version of this program where all redundant safe math wrappers have been replaced with explicit arithmetic operations. We refer to the program that Csmith generated and the program after removal of redundant wrappers as the *original* and *relaxed* programs. Because safe math wrappers can be instantiated via functions or macros, a single integer seed yields four distinct concrete programs: **Csmith-funs:** the original program using functions for safe math wrappers; **Csmith-macros:** the original program using macros for safe math wrappers; **CsmithEdge-funs:** the relaxed program using functions for safe math wrappers; and **CsmithEdge-macros:** the relaxed program using macros for safe math wrappers.

These four programs can be used to cross-check a C compiler: they should all yield the same results when their compiled binaries are executed.[1] (Being free from UB, we could also use them for cross-checking with respect to multiple distinct compilers, or the same compiler at different optimisation levels, but we have not experimented with this yet.)

Evaluation. We evaluated CsmithEdge using a number of virtual machines running Ubuntu 18.04.4 LTS $x86_64$, each with two virtual CPU cores (2 Sockets, 1 Core) and 8 GB RAM. Each virtualisation host had two Intel Xeon CPU E5-2690 v3 CPUs per host, running at 2.6GHz, with 24 cores/48 threads per CPU.

We chose GCC version 10.0.1 (commit 613c932) as a compiler to test; GCC is one of the most widely-used C compilers, and has been extensively tested by compiler fuzzing tools in the past, so we felt it would be an adequately challenging target in which to find bugs. We used 100,000 distinct seeds to generate Csmith programs. For each seed, we obtained a 4-tuple of equivalent concrete programs as described above. We compiled each concrete program with GCC using the standard `-O2` optimisation level, executed all four compiled programs and compared their results, flagging up any result mismatches. We also collected coverage information in the GCC codebase (which was compiled without optimisation and with gcov instrumentation), separately for each of Csmith-funs, Csmith-macros, CsmithEdge-funs and CsmithEdge-macros programs, respectively. When running a program, we imposed a timeout of 500s—this is important because Csmith-generated programs are not guaranteed to terminate in general. Approximately 15k out 100k programs hit the timeout.

[1] One might ask whether it would be worth considering a fifth program variant in which *all* safe math wrappers are removed and replaced with explicit arithmetic operations. However, this would not make sense: removing any of the safe math wrappers that are necessary means that the program will trigger UB. Such programs cannot be used for finding miscompilation bugs by cross-checking compilers, because it is perfectly legitimate for two compilers to compile such programs into binaries that yield completely different results when executed, since a program that exhibits UB can behave in an arbitrary manner.

Discussion of Bugs in GCC. We found and reported two compiler bugs with normal importance (P2) in GCC-10 in the tree-optimisation component, which caused the generation of wrong code [8, 9].

GCC Bug #1: Skipping Tree-Side-Effect Evaluation of Operator's Second Argument [9]. When evaluating an expression, any side effects have to be computed even when their evaluation is not required for the evaluation of the expression. However, GCC had a bug that sometimes caused it to generate code that incorrectly ignored side effects. The following example contains the core of the generated program that exposes the bug.

Example 1. *In the following program:*

```
typedef int int32_t;
int main() {
  int a = 0;
  int32_t b = 0;
  (a > 0) * (b |= 2);
  printf("%d\n", b);
}
```

$(a > 0)$ *evaluates to 0 and as a result, the multiplication evaluates to 0. Due to bug #1,* b *is never set to* 2 *because the second argument evaluation is wrongly skipped, hence the program incorrectly prints* 0 *instead of* 2.

After our report, developers promptly fixed the bug. The fix was done in `match.pd`, which contains a series transformation patterns that GCC employs.

The bug was found only by CsmithEdge-macros-style programs. CsmithEdge found the bug in 290 out of the 100,000 generated programs (288 miscompilations and 2 runtime crashes)—we base this on the fact that these programs ceased to misbehave when recompiled with a fixed version of GCC. To understand why the bug was only found by CsmithEdge-macros, we measured the line coverage of all locations related to the patch (in files: `gimple-match.c` and `generic-match.c`, where `match.pd` is actually used). In file `gimple-match.c` the coverage was high with all four versions, but in `generic-match.c`, the coverage was 0 with all versions but CsmithEdge-macros. Therefore, we suspect that the reason regular Csmith cannot find this bug is indeed due to the extra restrictions it imposes in the generated programs. We believe CsmithEdge-funs doesn't find the bug due to its many function calls inhibiting the buggy optimisation.

GCC Bug #2: Skipping Tree-Side-Effects on Internal Calls [8]. The second bug we found was also related to incorrectly skipped side effects, but this time in a sequence of statements. The following example contains the core of the program exposing the bug.

Example 2. *In the following program:*

```
int main() {
  int a = 0;
  unsigned long long one = 1;
  ((18446744073709551615UL / one) < a++, one);
  printf("%d\n", a);
}
```

The statement on line 4 contains two expressions separated by a comma, and returns the last expression. The evaluation of `a++` *is incorrectly ignored and the program incorrectly prints* 0 *instead of* 1 *when the buggy compiler is used.*

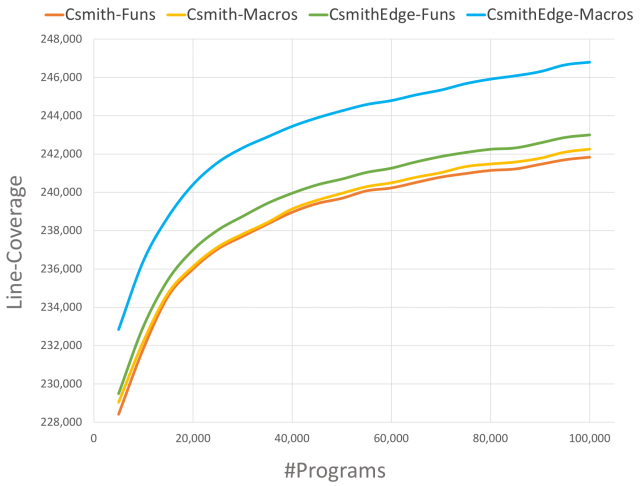

Figure 1: Line coverage in the GCC-10.0.1 compiler achieved by the four different generation methods.

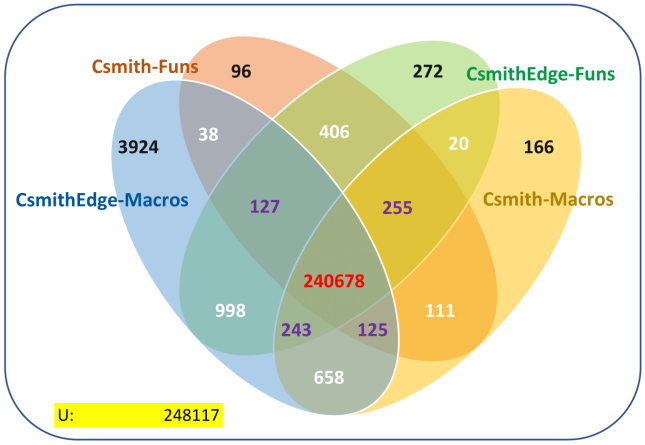

Figure 2: Venn diagram of comparison between (clock-wise, top-left) Csmith-funs, CsmithEdge-funs, Csmith-macros, and CsmithEdge-macros of line-coverage after compiling 100,000 test-cases.

This bug was also promptly fixed by the developer, with the patch in `gcc/tree.c` for GCC-10.1. The bug was similarly only found by CsmithEdge-macros, with a single program triggering it. We measured the coverage for the lines related to the patch for each method. The number of line-hits was 0 with Csmith-funs and Csmith-macros, 8 with CsmithEdge-funs, and 458 with CsmithEdge-macros. Therefore, we suspect again that the reason regular Csmith cannot find this bug is indeed due to the extra restrictions it imposes in the generated programs. Since CsmithEdge-funs also covers some lines in the patch, it is possible that it could also find it with a larger population of generated programs.

Discussion of GCC Code Coverage. Figure 1 reports the cumulative coverage achieved in the GCC codebase by the four different sets of 100K programs (Csmith-funs, Csmith-macros, CsmithEdge-funs and CsmithEdge-macros). We measured the coverage every 5,000 programs. We used the gcov-based tool `gfauto`, from the GraphicsFuzz project [5, 13], to aggregate results from all machines and generate the coverage results in a human-readable format.

Figure 1 shows that the largest line coverage was achieved by CsmithEdge-macros, and the second-largest by CsmithEdge-funs. After 100k programs are compiled, CsmithEdge-macros covers around 5K more lines than the rest.

To understand how complementary the different sets of generated programs are, Figure 2 presents a Venn diagram of the lines covered after compiling all 100k programs in each set. All sets covered a common set of 240,678 lines in the GCC codebase. Numbers with a black label show the unique lines covered by each set: 96 for Csmith-funs, 272 for CsmithEdge-funs, 166 for Csmith-macros and 3,924 for CsmithEdge-macros.[2] The other numbers show the numbers of lines covered only by two or three of the sets. Overall, we see that while CsmithEdge-macros covers most additional lines of code, there is nevertheless complementarity between the methods.

[2]See [3, 12] for a breakdown of the lines covered uniquely by CsmithEdge-macros.

4 FUTURE DIRECTIONS

In this idea paper, we presented promising empirical evidence that lifting some of the restrictions introduced by compiler fuzzers to generate UB-free programs can find compiler bugs that would be otherwise out of reach.

Immediate future work could include relaxing more runtime checks (e.g., for null-pointer dereferences), trying different compilers (e.g., LLVM or Microsoft Visual Studio), and crosschecking the generated programs across multiple compilers. We would also like to further understand the impact of using function calls vs. macros for the safe math checks. Given the complementary compiler coverage achieved by the function and macro-based versions, it would be interesting to explore a version mixing both functions and macros.

We also plan to explore the idea of relaxing the methods used during program generation so that programs exhibiting UB can sometimes be generated, and then using dynamic analysis to detect and discard such programs. For instance, with Csmith, array indexes are forced to be in bounds, and gotos are forbidden from spanning initialisation code. These and other checks could be relaxed or omitted with low probability during generation, in the hope that the generated program still has a high chance of being UB-free. Program analysers such as ASan [18] or FramaC [7] could be used to discard programs that do turn out to exhibit UB, and the programs that remain could be used for cross-checking multiple compilers, their less idiomatic form perhaps leading to the discovery of bugs in previously under-tested parts of the compiler under test.

ACKNOWLEDGEMENTS

We would like to thank John Regehr for answering our questions about Csmith, Paul Thomson for guidance in using `gfauto` for coverage analysis, and to Michaël Marcozzi for guidance in using `gcov` and sharing data. This work was supported by EPSRC projects EP/R011605/1 and EP/R006865/1.

REFERENCES

[1] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering (TSE)* 41, 5 (2015).

[2] Csmith Homepage. Date Accessed March 09, 2020. https://embed.cs.utah.edu/csmith/.

[3] CsmithEdge. Date Accessed July 27, 2020. https://srg.doc.ic.ac.uk/projects/CsmithEdge/.

[4] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. In *Proc. of the ACM on Programming Languages (OOPSLA'17)*.

[5] Alastair F. Donaldson, Hugues Evrard, and Paul Thomson. 2020. Putting Randomized Compiler Testing into Production. In *Proc. of the 34th European Conference on Object-Oriented Programming (ECOOP'20)*.

[6] International Organization for Standardization. 2018. *ISO/IEC 9899:2018: Programming Languages—C*.

[7] Frama-C. Date Accessed July 29, 2020. https://frama-c.com.

[8] GCC Bugzilla. Date Accessed April 28, 2020. Bug 94809. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=94809.

[9] GCC Bugzilla. Date Accessed February 15, 2020. Bug 93744. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=93744.

[10] GitHub. 2018. Git Repository of Yarpgen. https://github.com/intel/yarpgen.

[11] GitHub. Date Accessed April 13, 2020. Git Repository of Csmith. https://github.com/csmith-project/csmith.git.

[12] GitHub. Date Accessed July 13, 2020. Git repository of CsmithEdge. https://github.com/karineek/CsmithEdge.git.

[13] GitHub. Date Accessed May 15, 2020. Git Repository of gfauto. https://github.com/google/graphicsfuzz.git.

[14] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'14)*.

[15] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core compiler fuzzing. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'15)*.

[16] Alastair Donaldson Cristian Cadar Michael Marcozzi, Qiyi Tang. 2019. Compiler Fuzzing: How Much Does It Matter?. In *Proc. of the ACM on Programming Languages (OOPSLA'19)*.

[17] Kazuhiro Nakamura and Nagisa Ishiura. 2016. Random testing of C compilers based on test program generation by equivalence transformation. In *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*.

[18] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proc. of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*.

[19] Xi Wang, Nickolai Zeldovich, Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior. In *Proc. of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*.

[20] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'11)*.