Empirical Software Engineering (2022) 27: 129
https://doi.org/10.1007/s10664-022-10146-1

CsmithEdge: more effective compiler testing by handling undefined behaviour less conservatively

Karine Even-Mendoza[1] · Cristian Cadar[1] · Alastair F. Donaldson[1]

Accepted: 14 March 2022 / Published online: 8 July 2022

Abstract
Compiler fuzzing techniques require a means of generating programs that are free from undefined behaviour (UB) to reliably reveal miscompilation bugs. Existing program generators such as Csmith achieve UB-freedom by heavily restricting the form of generated programs. The idiomatic nature of the resulting programs risks limiting the test coverage they can offer, and thus the compiler bugs they can discover. We investigate the idea of adapting existing fuzzers to be less restrictive concerning UB, in the practical setting of C compiler testing via a new tool, CsmithEdge, which extends Csmith. CsmithEdge probabilistically weakens the constraints used to enforce UB-freedom, thus generated programs are no longer guaranteed to be UB-free. It then employs several off-the-shelf UB detection tools and a novel dynamic analysis to (a) detect cases where the generated program exhibits UB and (b) determine where Csmith has been too conservative in its use of *safe math* wrappers that guarantee UB-freedom for arithmetic operations, removing the use of redundant ones. The resulting UB-free programs can be used to test for miscompilation bugs via differential testing. The non-UB-free programs can still be used to check that the compiler under test does not crash or hang. Our experiments on recent versions of GCC, LLVM and the Microsoft Visual Studio Compiler show that CsmithEdge was able to discover 7 previously unknown miscompilation bugs (5 already fixed in response to our reports) that could not be found via intensive testing using Csmith, and 2 compiler-hang bugs that were fixed independently shortly before we considered reporting them.

Keywords Compilers · Fuzzing · Csmith · GCC · LLVM · MSVC

Communicated by: Dan Hao

✉ Karine Even-Mendoza
k.even-mendoza@imperial.ac.uk

Cristian Cadar
c.cadar@imperial.ac.uk

Alastair F. Donaldson
alastair.donaldson@imperial.ac.uk

1 Imperial College London, Department of Computing, London, UK

1 Introduction

The critical role of compilers, which underpin practically all deployed software, has led to a lot of interest in techniques for automated compiler testing, with a surge of interest in this topic over the last decade (see Chen et al. 2020 for a recent survey). In particular, compiler *fuzzers* have proven successful at finding bugs in mature compilers via randomised testing (Yang et al. 2011; Le et al. 2014; Nakamura and Ishiura 2016; Livinskii et al. 2020; Lidbury et al. 2015; Donaldson et al. 2017; Marcozzi et al. 2019). A popular method for finding *miscompilation* bugs—where a compiler silently generates wrong code—is *differential testing* (McKeeman 1998; Groce et al. 2007). Employed by tools such as CSMITH (Yang et al. 2011) and YARPGEN (Livinskii et al. 2020), this involves comparing the result obtained by running a program after compilation by distinct compilers or by a single compiler at different optimisation levels, with result mismatches indicating bugs. This avoids the *oracle problem* (Barr et al. 2015): it flags up differences in results that are expected to be the same, yet does not require knowing which (if any) result is correct.

Dealing with 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 (International Organization for Standardization 2018). A program that exhibits UB has arbitrary semantics, so optimising compilers are permitted to assume that input programs are free from undefined behaviour (*UB-free*) and optimise them based on that assumption. In practice, compilers *do* take advantage of UB to generate efficient code (Wang et al. 2013) (we give an example in Section 2.1), thus when a program triggers UB, the results of cross-checking the program's results between multiple compilers or optimisation levels are meaningless: the program can legitimately yield *any* result when executed.

To counter this, compiler fuzzers go to great lengths to generate UB-free programs. For instance, the CSMITH program generator (Yang et al. 2011) guards every instance of a signed arithmetic operation with an overflow check that only performs the operation if overflow will not occur, and uses conservative static analysis during generation to ensure that pointer arguments to functions always refer to valid data so that they can be dereferenced without error. 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 may be inapplicable if every arithmetic operation is enclosed in a conservative check for potential UB.

After a sufficiently long testing campaign, the idiomatic form of the programs generated by a particular compiler fuzzer leads to the compilers under test becoming "immune" to the fuzzer: the compiler bugs that remain either cannot be triggered by the kinds of programs the fuzzer generates, or can be triggered only with very low probability. For example, John Regehr, one of the authors of CSMITH, tweeted during July 2019: "I hadn't run Csmith for a while and it turns out LLVM is now amazingly resistant to it, ran a million tests overnight without finding a crash or miscompile" (Regehr 2019).

A heavyweight solution to this immunity problem is to write a brand new fuzzer, and indeed the recent YARPGEN (Livinskii et al. 2020) was able to find a large number of bugs in mainstream compilers that could not be found by CSMITH. But writing an entirely new compiler fuzzer is a large undertaking, and the compilers under test may soon become immune to the new fuzzer (as has reportedly happened with YARPGEN and LLVM Babokin 2019).

Our contribution In this work we are interested in whether a compiler's apparent immunity to a particular fuzzer might be lessened via coercing the fuzzer into generating more *diverse* programs by having it be *less conservative* about UB. We investigate the following research question:

Can a program generator that conservatively enforces UB-freedom be made less conservative such that:

a) it can still be used as a source of programs for differential testing, and
b) the resulting programs lead to more thorough compiler testing, in terms of bugs found and code covered in the compiler code base?

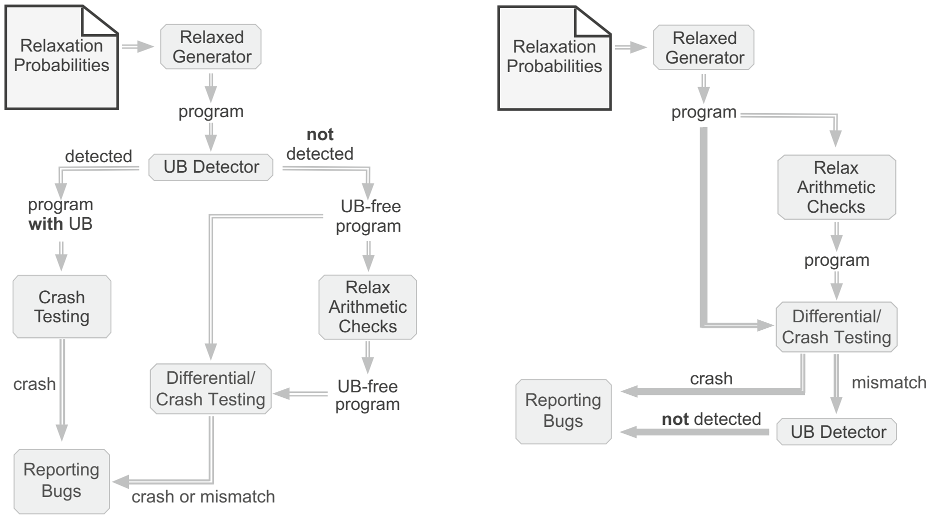
We present a new open-source tool, CSMITHEDGE (CsmithEdge - Homepage 2022), an extension of CSMITH. CSMITHEDGE employs three steps to make generated programs less restricted, which are summarised in Fig. 1a.

First, it probabilistically relaxes some of the generation-time static analyses that CSMITH uses to ensure UB-freedom. Relaxing these analyses makes them unsound, such that generated programs might actually trigger the UB that the analyses aimed to avoid.

Second, CSMITHEDGE runs each generated program using a collection of off-the-shelf UB detection tools that either show that the program does exhibit UB, or establish with high confidence that it is UB-free; we have manually tuned the range of probabilities associated with relaxation so that the rate of programs that trigger UB is not too high.

Third, if the program is judged to be UB-free, CSMITHEDGE uses a novel yet simple dynamic analysis to identify cases where safety checks for UB on arithmetic operations inserted by CSMITH are redundant, and removes these checks.

The resulting relaxed, UB-free program can then be used to test a set of compilers for miscompilation bugs via cross-checking. If, during the second step, the UB detection tools show that the generated program exhibits UB, it cannot be used for miscompilation testing,



(a) CSMITHEDGE conceptual approach (b) CSMITHEDGE lazy approach

Fig. 1 Program generation with relaxed methods during and post code generation

but can still be used to test the compilers of interest to find programs that lead to crashes or hangs.

To avoid consistently incurring the high overhead associated with UB detection, we apply a lazy approach in practice. That is, postpone UB detection to the differential testing phase, and only perform UB detection on programs that trigger a result mismatch between the compilers under test. We summarise the process with the lazy approach in Fig. 1b and discuss it in detail in Section 3.3.

Our approach aims to improve compiler testing by increasing the size of the space of test cases that can be generated. By relaxing UB-related compile-time generation constraints, CSMITHEDGE can generate test cases that are not in the vocabulary of regular CSMITH. This is in contrast to methods that improve the diversity of generated tests by changing the manner in which the test generator is configured, such as through swarm testing (Groce et al. 2012) (a technique that CSMITHEDGE also incorporates, as discussed in Section 3.4), or based on historical information (Chen et al. 2019). Such methods leave the set of *potential* test cases unchanged, but may dramatically change the probabilities associated with particular test cases being generated.

Summary of experimental findings We present a large experimental evaluation applying CSMITHEDGE to two recent versions of the GCC and LLVM compilers, both of which have been extremely well tested by several compiler fuzzing tools for many years.

To answer the part a) of our research question, we measured the rate of generated programs that could be used for differential testing. On a standard machine, CSMITHEDGE generated around 105.09 programs per hour using the flow in Fig. 1a.[1] With the optimised lazy flow of Fig. 1b, CSMITHEDGE generated 215.21 programs per hour (in this case, any terminating program that is either UB-free or triggers the same behaviours across compilers is counted). By comparison, regular CSMITH achieved 345.75 programs per hour: the degradation in performance for CSMITHEDGE comes from the overhead of running UB detection tools and the fact that a portion of programs turn out to contain UB. Overall, checking whether programs terminate is the dominating factor regardless of whether CSMITH or CSMITHEDGE is used, and in the case of CSMITHEDGE, when a program does turn out to trigger UB this is often quickly detected by a UB detection tool.

Therefore, the answer to part a) of our research question—whether relaxing restrictions on generated programs can yield a source of programs for differential testing programs—is yes, but with a 1.61 $\times$ runtime overhead with the tool's default configuration (i.e. using the lazy approach of Fig. 1b with a 50 s timeout).

We have found CSMITHEDGE to be effective in finding bugs, providing a positive answer to part b) of our research question. We found and reported 7 previously-unknown miscompilation bugs (5 in GCC, 1 in LLVM, and 1 in MSVC, the Microsoft Visual Studio Compiler), of which 5 have already been fixed in response to our reports. We also discovered 2 GCC hang bugs that turned out to have been independently discovered and fixed shortly before we considered reporting them, and several bugs in the older versions of the GCC and LLVM compilers. We also tested these compilers thoroughly with regular CSMITH and did not find any miscompilations, crashes or hangs, which provides evidence both that these compilers have become somewhat immune to the kinds of bugs that regular CSMITH can expose, and that the strategies employed by CSMITHEDGE are effective in extending the bug-finding

[1] This includes the time taken to check whether each program terminates within a 50 s time limit. The evaluation was done on a single virtual machine running Ubuntu 18.04 LTS x86_64 with two virtual CPU cores (2 Sockets, 1 Core) and 8 GB RAM.

power of CSMITH. We also measured the coverage of the GCC and LLVM code bases achieved during fuzzing campaigns using regular CSMITH and CSMITHEDGE, finding that CSMITHEDGE achieves a modest improvement in coverage for both compilers, providing further evidence that being less conservative about undefined behaviour can lead to more thorough testing.

Paper structure After discussing necessary background (Section 2), we describe the design and implementation of CSMITHEDGE, which involves two distinct parts—relaxing generation-time restrictions, and eliminating redundant safety checks post-generation (Section 3). We present our large experimental campaign, providing details of a selection of the bugs that we found in GCC, LLVM and MSVC, as well as discussing coverage results (Section 4). After discussing threats to validity (Section 5) and related work (Section 6) we conclude with a discussion of future directions (Section 7).

2 Background

We provide background on the CSMITH program generator, which CSMITHEDGE extends, focusing on how CSMITH ensures that generated programs are UB-free (Section 2.1), and on the tools that CSMITHEDGE employs to detect the UB that might be introduced by relaxing generation-time restrictions (Section 2.2). We also briefly discuss the C-REDUCE tool that we use for test case reduction (Section 2.3).

2.1 CSMITH and UB Avoidance

To illustrate why differential testing requires a source of UB-free programs, consider the following C example:

```
void foo(int32_t x) {
  if (x*x < 0) printf("Overflow!\n");
}
```

Someone not well-acquainted with UB in C might expect the function to print `Overflow!` when invoked with a value of x whose square cannot be represented by a 32-bit integer, since integer arithmetic has wraparound semantics on all mainstream computer architectures.

When `foo` is invoked with x=1,000,000, we find that the message `Overflow!` gets printed if the program has been compiled without optimisations using e.g. GCC 7.5.0 or LLVM 11 (targeting an x86 processor). However, no message is printed if optimisation level `-O2` or higher is used with these compilers. This is because the input x=1,000,000 triggers signed arithmetic overflow, which is UB. The optimising compiler can assume that no UB is triggered, and so may reason that the square of x cannot be negative and optimise away the entire conditional statement. (In fact, recent versions of GCC perform this optimisation even at `-O0`). This demonstrates that cross-checking a program across multiple compilers is meaningless if the program exhibits UB.

We now provide some details on how CSMITH (Csmith Homepage 2021; Yang et al. 2011) generates deterministic C programs that are free from UB. CSMITH generates a program in a top-down fashion based on the C language grammar. Global variables can have a wide variety of types, including randomly-generated structure and union types, as well as nested pointers. A program contains a series of randomly-generated functions, whose arguments and return type may be based on the structures and unions the tool has generated. Functions have a set of local variables of various types, and their bodies manipulate both local and global variables. The statements involved are often complex, involving pointers with multiple levels of indirection, nested structures and multi-dimensional arrays.

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. Each CSMITH–generated program thus has a single path; this is an important property that we take advantage of in our approach for identifying redundant arithmetic checks (Section 3.2).

CSMITH achieves UB-freedom via a combination of *generation-time restrictions* and *runtime checks*.

Generation-time restrictions CSMITH forces all variables to be initialised before use: scalars are initialised to random values and pointers to the addresses of suitable in-scope variables. Arrays are initialised via `for` loops right after their declaration or when declared via curly braces. When generating `goto` statements, CSMITH forbids a `goto` from jumping over initialisation code by limiting the possible locations of labels, and by discarding bad `goto` statements via a static analysis on the source and destination locations.

To ensure that programs do not index arrays out of bounds, CSMITH keeps track of the sizes of arrays. Array accesses via randomly-generated constant indices that respect the bounds of an array can thus be generated anywhere the array is in scope. For non-constant accesses, CSMITH generates `for` loops whose index variables iterate from a lower bound to an upper bound using a fixed stride. In such loops it exploits the combination of knowledge about the sizes of arrays and ranges of index variables to generate non-constant but in-bounds array accesses based on suitable loop index variables.

By forcing all pointers to be initialised to the addresses of existing program variables, and because it does not generate pointer arithmetic, CSMITH avoids the possibility of null or invalid pointers being dereferenced. The problem of the addresses of local variables escaping into global variables is avoided via a whole-program pointer analysis. When CSMITH detects that it has generated code that will lead to escaping pointers, it backtracks and generates a different code fragment instead. CSMITH contains options that allow it to generate programs that include null and dangling pointers with low probability—these options were introduced to allow CSMITH to be used for testing static analysers (Cuoq et al. 2012) and would usually be disabled during compiler testing; however, we make use of these options in our approach.

CSMITH applies effect analysis to avoid generating code with read/write and write/write conflicts between two sequence points. This analysis avoids the generation of an expression such as `a[i++] = i`, which would trigger UB if executed because it writes to `i` via sub-expression `i++` and reads from `i` via the sub-expression `i` on the right-hand side of =, with no sequence point determining an order for the accesses to `i` on either side of the = operator. Each variable, including unions and arrays but without distinguishing between different members or offsets, is placed by CSMITH either on a READ list or on a WRITE list to ensure an object cannot be modified more than once between two sequence points. CSMITH decides before generating code which variables are on the READ or WRITE list. This avoids read/write and write/write conflicts between sequence points, including those related to unspecified order of evaluation of function arguments.

By studying the CSMITH source code we also noticed that the tool explicitly prevents the address of a formal parameter of a function f being passed as an actual parameter to a function invoked by f. The reason for this restriction is not documented, but we do not believe it is explicitly related to UB. We found that removing the restriction has a negative impact on the time associated with static analysis, significantly increasing the overall time taken by CSMITH to generate programs, so it seems likely that the restriction is in place for performance reasons. As discussed in Section 3, we relax this restriction with low probability, in order to generate more diverse programs without making generation too slow.

Runtime checks Freedom from arithmetic-related UB is achieved through runtime arithmetic checks encoded via *safe math* wrappers. Instead of directly issuing an integer arithmetic expression $a \circ b$ (for some operator $\circ$), CSMITH 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:

$$\mathit{unsafe}(a, b, \circ)\ ?\ a\ :\ a \circ b$$

where $\mathit{unsafe}(a, b, \circ)$ 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 unsigned integer division):

```
uint32_t safe_div(uint32_t X, uint32_t Y) {
  return (Y == 0 ? X : (X / Y));
}
```

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

```
#define uint32_t safe_div(_X, _Y) \
  ({uint32_t X = (_X); uint32_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. However, they may change the kinds of optimisations that the compiler under test performs. For example, the presence of many simple functions will frequently exercise the compiler's function inliner, but the compiler may decide *not* to inline all such function calls, which may inhibit optimisation opportunities that inlining would enable and will be possible when macros are used. Our understanding from talking to the CSMITH developers is that functions (the default) are preferred because they are simpler to maintain. This is evident from the relative simplicity of the function version above; the macro version requires a great deal of care to be taken related to casting. During our evaluation we found and fixed some discrepancies between the function and macro wrappers (GitHub 2020a; 2020b).

2.2 UB Detection

Our CSMITHEDGE tool relaxes the analyses that CSMITH uses to ensure UB-freedom, and thus risks generating programs that exhibit UB. These must be detected, as they cannot be used for miscompilation testing. The following program analysers are employed for this purpose; their role in our solution is summarised in Table 1, which we discuss in Section 3.1.

AddressSanitizer (ASAN) (Serebryany et al. 2012) is a dynamic analysis tool that uses a shadow memory approach to detect invalid memory accesses, such as buffer overflows (heap, stack, and globals), and use-after-free, use-after-scope and use-after-return accesses. It has a typical overhead of 2× (Address Sanitizer 2012).

Table 1 The CSMITH generation-time restrictions that CSMITHEDGE relaxes, together with the analysis tools that are used to detect the UB that might therefore arise, and the lower and upper bounds on the probabilities with which these relaxations are applied, both for crash testing and miscompilation testing

		Bounds $[R_l, R_u]$ for:	
Relaxation	UB detected by (in the order used)	Crash	Miscomp.
Null dereferences	UBSAN	[0, 1]	[0, 1]
Dangling pointer dereferences	ASAN, FRAMA-C	[0, 1]	[0, 1]
Variable initialisation	MSAN, FRAMA-C	[0, 1]	[0, 0.35]
Locations associated with “goto”	ASAN, MSAN, FRAMA-C	[0, 1]	[0, 1]
Array bounds	ASAN, MSAN, UBSAN, FRAMA-C	[0, 1]	[0, 0.25]
Passing the addresses of parameters	N/A	[0, 1]	[0, 0.5]
Read/write and write/write conflicts	Always rejected for miscomp. testing	[0, 1]	[0, 0]

MemorySanitizer (MSAN) (Stepanov and Serebryany 2015) is also a dynamic analysis tool based on shadow memory which can detect uninitialised memory accesses. It has a typical overhead of 3× (Memory Sanitizer 2015).

UndefinedBehaviorSanitizer (UBSAN) (Undefined Behavior Sanitizer 2017) is a dynamic analysis tool that can detect a variety of undefined behaviour, such as use of a misaligned or null pointer, out-of-bounds array indexing (if the bound can be statically determined), signed integer and floating-point types overflow, code that performs pointer arithmetic that overflows, and integer division by zero. We have found it to cause a typical slowdown of 2-3×.

FRAMA-C (Cuoq et al. 2012) is a static source-code analyser for C programs with several available algorithms (plugins). In particular, the tool has an Evolved Value Analysis (Eva) plugin, which applies abstract interpretation techniques to compute variation domains for variables to analyse program variable values (Frama-C EVA plugin 2007). Eva reports possible undefined behaviour, such as the use of uninitialised addresses, dangling pointer dereferences, out-of-bounds index accesses, and signed integer overflow. The fact that a CSMITH program has a *single* path means that this analysis is as precise as a dynamic analysis: FRAMA-C effectively acts as an interpreter that performs rigorous semantic checks. We have found it to cause a typical slowdown of 28×.

FRAMA-C has a relatively high overhead, and yet the overhead of CSMITHEDGE overall is much lower. This is related to the way we perform the UB detection: using the tools with the lower overhead first, before applying heavier tools, and terminating as soon as UB is detected in the generated program.

All of the analysis tools that we use have the potential to report either false positives or false negatives in principle. However, in practice we have found that they are mainly precise when applied to programs generated by CSMITHEDGE; we discuss some exceptions in Section 3.

2.3 Test Case Reduction

A CSMITH- or CSMITHEDGE-generated program is typically large and complex. As a result, a generated program that triggers a miscompilation bug is not directly useful for reporting the bug; a test case reducer is needed to shrink the program down to a much smaller version that still triggers the bug. In our experiments, we use the C-REDUCE tool for this purpose (Regehr et al. 2012). C-Reduce takes a C program and the path to an "interestingness test" script. The user of C-Reduce must write this script to encode the property with respect to which they want their program to be reduced: the script should return 0 if and only if a given program is deemed "interesting".

For reducing a miscompilation bug observed via a mismatch between compilers A and B, we use an interestingness test that (a) compiles and executes the program using each of the analysis tools described in Section 2.2, returning "not interesting" if any of them flags up a UB, then (b) compares the results obtained by running the program after compilation using compilers A and B, returning "interesting" if and only if the results differ. It is important to check for UB-freedom at every stage of the reduction process: even if the original program was UB-free, C-Reduce may apply transformations that introduce UB. In Section 3.3 we discuss a lazy approach to detecting UB that we also use to optimise the interestingness test during reduction.

For compiler crash bugs the interestingness test is simpler. Suppose we wish to reduce a bug where the compiler crashes with a particular error message or stack trace. We first write a regular expression characterising the output associated with the crash. The interestingness test is then set up so that "interesting" is returned if and only if the compiler under test crashes, and the compiler output matches the regular expression. Test case reduction is not effective in reducing programs that cause a compiler to hang since test case reduction involves invoking the compiler a large number of times, and detecting a hang requires using a large timeout.

3 Design and Implementation

The operation of CsmithEdge is summarised by the diagram of Fig. 1a. First, an adapted version of Csmith is invoked to generate a program; Csmith is adapted so that some of the generation-time restrictions discussed in Section 2.1 are relaxed. This means that the generated program may be less restricted than a program that Csmith typically generates, but also that it might trigger UB.

To detect possible UB, a *validation* stage invokes the analysis tools described in Section 2.2. The tools that are used depends on the specific analyses that CsmithEdge chose to relax. If a program is found to contain UB then it cannot be used for differential testing to detect miscompilations, though it can still be used to test compilers for crash and hang bugs. Programs that do not terminate within a given time bound (50 s by default) are also rejected during validation.

A program validated as UB-free and terminating will still contain a large number of *safe math* checks—the conservative checks that Csmith inserts to ensure freedom from arithmetic UB (see Section 2.1). A novel dynamic analysis is used to determine which of these checks are needed and which are redundant; the redundant checks are removed to make the program even less constrained and thus more diverse than the kinds of program regular Csmith can generate. The resulting program is then used for differential testing with the aim of detecting miscompilations.

As noted in Section 2.2, the analysis tools used during validation can in principle exhibit false positives and false negatives. False positives would lead to the rejection of programs produced by CsmithEdge that are actually valid; this might reduce the effectiveness of CsmithEdge by lowering the rate at which it can generate usable programs. False negatives could lead to programs that do exhibit UB being erroneously used for miscompilation testing, leading to compiler bug reports that turn out to be false alarms. In practice we have not found the precision of these analysers to be a major issue: our results (Section 4) show that CSMITHEDGE generates programs that pass validation at a high enough rate that it is useful for finding compiler bugs. By using a combination of analysis tools we encountered false alarm compiler bug reports extremely rarely: during our large experimental campaign we manually examined many instances of potential miscompilations flagged by CSMITHEDGE and only encountered one example of a program that actually triggered UB (this had been missed by FRAMA-C because, for performance reasons, we invoke FRAMA-C with a bound on the number of iterations of a loop that it will analyse).

We now discuss details related to relaxing generation-time constraints (Section 3.1) and detecting and removing redundant arithmetic checks (Section 3.2), and explain some further implementation details including an optimised version of the flow depicted in Fig. 1a that leads to higher performance (Section 3.3).

3.1 Relaxing Generation-Time Restrictions

CSMITH's generation-time restrictions are a key component of its guarantee that generated programs are UB-free (see Section 2.1), but they limit the form of the programs that are generated. These analyses and restrictions are all conservative: they may prevent the generation of code that would actually be UB-free. For example, much of the code that CSMITH generates will turn out to be dynamically unreachable—previous work on equivalence modulo inputs testing reported that on average 29% of lines in a CSMITH-generated program turn out to be unexecuted at runtime (Le et al. 2014). However, CSMITH does not know at generation-time whether the code it is generating will be reachable, thus it conservatively avoids generating code that would trigger UB if it were to be executed. Note that generating diverse non-idiomatic dead code is useful. This is because the compiler usually does not know at compile time whether the code will be dead or not. While compiling the dead code, some compiler optimisations may operate at the boundary of reachable and dead code and—if implemented incorrectly—may change the results computed by reachable code.

CSMITHEDGE probabilistically relaxes these restrictions in order to generate more diverse programs, using the UB-detection tools of Section 2.2 to identify any resulting programs that exhibit UB so that such programs are not used for miscompilation testing.

Table 1 summarises the generation-time restrictions that CSMITHEDGE relaxes, and the analysis tools that are used to detect the corresponding UB that might result. The table also indicates the probabilities with which relaxations are applied, which we explain further in Section 3.4. We try to use the faster analysis tools first so that if they detect UB the slower ones are skipped. The order we use is ASAN, MSAN, UBSAN, FRAMA-C (though a particular analysis tool is only invoked if a relaxation that demands the use of that tool has been applied). This order means employing the expensive FRAMA-C tool last. We acknowledge that this order might not be optimal, since the optimal order is not just a function of analysis tool speed, but also of how likely each analyser is to identify UB in generated programs in practice. We revisit this matter during our evaluation in Section 4.2.

In deciding which UB avoidance methods to relax, we were guided by (a) whether there is already a flag in CSMITH that enables the relaxation, or, if there is no such flag, (b) how tractable it was to implement the relaxation in the (complex) CSMITH code base without adversely affecting the rest of the tool. In some cases we found that introducing a relaxation would break invariants within the CSMITH and lead to assertion failures; we persevered with relaxations where it proved relatively easy to work around such problems. For example, weakening the effect analysis to allow read/write and write/write conflicts can lead to the failure of assertions that check that there are no such conflicts (an invariant that regular CSMITH maintains). We weakened the conditions of such assertions to ensure that they would not fail for conflicts introduced as a result of our relaxations.

Following the approach of *swarm testing* (Groce et al. 2012), each time CSMITHEDGE is invoked it chooses at random whether to enable each relaxation, so that only a subset of all possible relaxations is enabled during generation of a single test. For an enabled relaxation, the probability with which it will be applied is also chosen at random to be within a particular interval that we have customised for each relaxation.

We now discuss each relaxation in turn.

Null pointer dereferences We allow the generation of reads and writes from a pointer that might turn out to be null. We use CSMITH's option for testing static analysers (Cuoq et al. 2012) which allows null pointer dereferencing with a given input probability. Allowing dereferencing null pointers in code can lead to UB; however, it shall not trigger a UB in dead code (this is because dead code can never be executed and therefore cannot lead to any behaviour, including undefined one). We detect null pointer dereferences in reachable code via UBSAN, which outputs a runtime error whenever a program tries to dereference or write to a null pointer address.

Dangling pointer dereferences We allow dereferencing of pointers that may no longer refer to allocated memory with a given input probability. We use some of CSMITH's options to test static analysers (Cuoq et al. 2012) that allow dereferencing dangling pointers similarly to the null pointers option above but disable the option that generates a return statement with a dead pointer. Initially, we tried generating code with this option but received many return statements with dangling pointer dereferences; we could have lowered the probability, but this would have had a general effect on the dangling pointer dereferences instances in a program, making them unlikely to appear. As with null pointer dereferences, code that would dereference dangling pointers only leads to UB if it is actually executed. We detect dangling pointer dereferences in reachable statements via ASAN and FRAMA-C (this is part of their broader capability to detect use-after-free problems in general); we use FRAMA-C to detect local variables that escape the scope of their function, which may lead to access of a local object out of its scope.

Variable initialisation We leave global and local variables uninitialised[2] (including arrays and unions), with a given input probability. We relax the original constraints in CSMITH that force initialisation of each variable: CSMITHEDGE removes the assignment associated with a variable's declaration according to the input probability. This does not always lead to UB—e.g. a statement that initialises the variable before it is used might be generated, or the variable might appear in dead code. When UB does arise, we detect the lack of initialisation in reachable code via MSAN and FRAMA-C.

Locations associated with goto A `goto` statement can, with some input probability, jump to any label in the current context (all function's blocks) and the label's location is not restricted in the current context by CSMITHEDGE. As a result, backward and forward jumps

[2]In C, global variables are initialised to zero if no explicit initialiser is given. However, removing explicit initialisers is still a potentially interesting deviation from what CSMITH does, since CSMITH always explicitly initialises every global.

can then skip over the initialisation of variables, which triggers UB if one of those uninitialised variables is read. For example, a CSMITHEDGE-generated program can now contain this block:

```
static int a = 1;
static void func() {
  for (int i = 3; i > 0; i--) {
    if (a)
      goto lbl_forward; // jump over b's initialisation
  }
  ...
  int b = 3;            // the goto will skip this statement
lbl_forward:
  a += b;
}
```

In this example, the execution of the program jumps over the initialisation of *b* and uses it without initialising it. Similarly, a backwards jump can circumvent variable initialisation code. However, not all `goto` jumps generated by CSMITHEDGE lead to this scenario, as it is possible that the code after the label never uses the uninitialised variable or it can be assigned with a new value later on; in these cases, no UB is triggered. Like the relaxation of variable initialisation, we detect the lack of initialisation in reachable code after a goto jump via MSAN and FRAMA-C and null pointer dereferences with ASAN (in case we skipped an array or a pointer initialisation).

Array bounds CSMITHEDGE allows generation of slightly more complex array index expressions compared to CSMITH: in addition to integer constants, an array index expression can be a binary operator with a mixture of variables and constants as operands. With some input probability, we generate a constant or a binary operator expression without checking if it is within the array's bounds. Hence, CSMITHEDGE generates more interesting expressions for array accesses than CSMITH; for example, CSMITHEDGE can generate this program:

```
static int a[1][2][1] = {{{1},{2}}};
void main ()
{
  static int b = 4UL;
  int *c;
  c = a[(b%1)][b-3][3*5];
}
```

Some of the index expressions, such as $3 * 5$ in the program above, can trigger an array out-of-bounds access; any such expression in reachable code triggers UB. We use all four tools to detect UB related to out-of-bounds array accesses: while FRAMA-C can detect all such issues in principle, the sanitisers can each detect a subset of issues and incur significantly lower overhead than FRAMA-C, thus it makes sense to try them first.

Passing the addresses of parameters As discussed in Section 2.1, CSMITH does not allow the address of a parameter to the enclosing function to be passed as an argument to a function call and this appears to be for performance reasons rather than due to UB-avoidance. To increase diversity of generated programs, we allow a parameter to take an argument's address by ignoring this restriction with a very low probability when constructing an expression for a function call parameter. By using a low probability we were able to avoid performance concerns. The “N/A” entry in the second column of Table 1 indicates that no analysis was required in relation to this change, as we did not observe generated-code where the relaxation leads to additional UB.

Read/write and write/write conflicts We weaken the effect analysis in CSMITH so that read/write and write/write conflicts are allowed with some low probability. We found that this change would frequently introduce undefined or unspecified behaviour, e.g. due to the order in which function call arguments are evaluated becoming important (an unspecified behaviour in C), or due to unsequenced races akin to the illustrative example presented in Section 2.1. Furthermore, none of the UB-detectors used in this work are capable of detecting such issues.[3] We thus opted to keep this relaxation as an option for CSMITHEDGE, but to regard a generated program as automatically invalid if it uses the option, so that the program can only be used for crash/hang testing; this is indicated by the “Always rejected” entry in the second column of Table 1, and the probability range $[0, 0]$ associated with miscompilation testing.

3.2 Relaxing Arithmetic Checks

Recall from Section 2.1 that CSMITH uses safe math wrappers to eliminate potential UB arising from arithmetic operations. 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 “$e_1\,?\,e_2\ :\ e_3$”. As the conditional operator may introduce control flow (due to short-circuit evaluation), the rather prescriptive program format arising from this blanket use of conditionals may bias the optimisations that a compiler applies to CSMITH-generated programs, possibly reducing the extent to which CSMITH can find bugs in other optimisations.

An arithmetic check in a CSMITH-generated program is *redundant* if the program is still free from UB after removing the check. We use a simple yet effective dynamic analysis to successfully identify redundant arithmetic checks in a generated program that has passed validation. Our analysis temporarily instruments every arithmetic check so that it will emit a warning if it turned out to be truly necessary. To achieve this, the i^{th} occurrence of a safe math wrapper of the form $\mathit{unsafe}(a, b, \circ)\ ?\ a\ :\ a \circ b$ is replaced with:

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

If executed, $\texttt{WARN}(i)$ prints a message indicating that the i^{th} safe math wrapper was genuinely required—i.e. UB would have been triggered had it not been present. 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 arithmetic checks that are actually needed. We then prune all but these checks 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* ones are gone, meaning that its use of arithmetic is correspondingly less constrained.

[3]We believe that the CompCert compiler (Leroy 2009) may have support for detecting sensitivity to argument evaluation order, but we have not yet integrated CompCert with CSMITHEDGE.

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. The wrapper at (iii) is identified to be necessary (because the divisor `k` passes through 0). The inner wrapper at location (iv) is confirmed to be 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 flagged as 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 $81920 * 81920$; s and t's value on exiting the loop are 81920 and 1, and hence 81920 is the result of the inner multiplication in (iv)). These two safe math wrappers are thus kept, and all others are removed (e.g. location (ii) becomes `s = s + k;`).

3.3 An Optimisation: Lazy UB Checks

We first implemented CSMITHEDGE to follow the flow of Fig. 1a, which we believe remains the best way to explain conceptually how the tool works. We found that the overhead of validating each and every generated program was high.

However, when searching for miscompilations, a generated program that does *not* lead to a result mismatch between the compilers under test is *uninteresting*, regardless of whether it is UB-free. This observation leads to the following optimisation, which we call *lazy UB checks*. After generating a program in a relaxed fashion, we compile the program with one of the compilers under test and run it using a timeout. If the program exceeds the timeout then we discard it as possibly non-terminating and move on, just as we would do if testing using regular CSMITH. If the program terminates, we invoke the remaining compilers under test and look for result mismatches. If no mismatches are observed then we also discard the program. Only if a mismatch between the compilers under test is observed do we actually invoke validation to determine whether the program is UB-free. This optimisation is summarised in Fig. 1b.

This optimisation avoids the overhead associated with running the analysis tools used during validation unless it appears to be genuinely worthwhile to run them. Even though UB often does lead to a deviation in results between compilers, there is also a good chance that the compilers under test might exhibit the same behaviour for certain UB, or that some instances of UB might not lead to effects that propagate to the output of the program.

The optimisation has diminishing returns as the number of compilers simultaneously under test increases, since with a large number of compilers the time taken compiling and running the generated program multiple times may exceed the time spent during validation. However, when performing differential testing of a pair of compilers (the most common case in practice), we have found the optimisation to be effective, as we show via experiments in Section 4.2.

3.4 Implementation Details

We have implemented our approach in a prototype tool called CSMITHEDGE, as a set of bash scripts and additional C/C++ code on top of CSMITH (GitHub 2011a), and available at (GitHub 2021b).

The changes we applied on top of CSMITH were: adding a set of parameters to control the probabilities of relaxing each of the methods in Section 3.1; a mechanism for randomly relaxing the constraints themselves during the code generation; and code for processing and storing the information needed to relax these methods (e.g. to avoid backtracking if a statement we relaxed can lead to UB according to CSMITH analysis).

Functions vs. Macros Recall from Section 2.1 that safe math wrappers are available as either functions or macros. Normally a user of CSMITH would specify that one or other of these wrapper forms should be used universally. In CSMITHEDGE we add a third option, whereby both function and macro wrappers are available. In this mode, each wrapper is randomly instantiated either in its function or macro form. When generating a program, the tool chooses with equal probability whether to use functions universally, macros universally, or a random mixture of functions and macros.

Randomisation during Test Case Generation in CSMITHEDGE CSMITHEDGE consumes a configuration file with one line per relaxation.

For a relaxation R in Table 1, the file either says:

- R is disabled, or
- R is enabled with probability range $[R_l, R_u]$

When CSMITHEDGE is invoked, for each *enabled* relaxation R, the tool chooses with 50% probability whether it is enabled for the current run. If the relaxation *is* chosen to be enabled, CSMITHEDGE chooses a specific probability, R_s, randomly from the range $[R_l, R_u]$. Having chosen R_s, every time there is an opportunity to apply the relaxation during the current test case generation, CSMITHEDGE does so with probability R_s.

For **crash testing**, all relaxations are enabled, and the bounds R_l and R_u, are always 0 and 1, respectively, as shown in Table 1.

For **miscompilation testing**, read/write relaxations are disabled, all other relaxations are enabled, and the bounds R_l and R_u are shown in Table 1. The bounds were obtained by manual experimentation, to find a good balance between adventurous programs and undefined behaviour.

On each run, CSMITHEDGE also chooses randomly whether to use functions, macros or a mixture of both for safe math wrappers.

We refer to the combination of the seed with which CSMITH's internal random number generation should be initialised, the manner in which safe math wrappers should be implemented, and the specific set of relaxation probabilities, R_s, as CSMITHEDGE's seed. An example of a CSMITHEDGE seed is:

`1621474906,2,0.8,0,0,0,0.3,0,0.`

This generates a program using `1621474906` as the seed for CSMITH's random number generator, uses a mix of functions and macros for safe math wrappers,[4] allowing null pointer dereferences with a probability of 80% and array out of bound access with a probability of 30%, and disabling all other relaxations.

Our approach of trying a variety of configurations via different CSMITHEDGE seeds is an example of swarm testing (Groce et al. 2012).

4 Evaluation

We evaluate CSMITHEDGE based on its ability to find compiler bugs which standard CSMITH does not find (Section 4.1), the throughput of fuzzing and rate at which relaxations are applied (Section 4.2), and its ability to increase code coverage of the compiler under test (Section 4.3).

We evaluated CSMITHEDGE using a number of virtual machines running Ubuntu 18.04 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 (2.6GHz, 12 cores / 24 threads per CPU).

We imposed a timeout of 150 s for generating a program with the relaxed generator, 300 s for compiling the program, 50 s for a native execution of the program (remember that CSMITH-generated programs are not guaranteed to terminate in general), 600 s for each of the sanitisers, and 300 s for FRAMA-C.

We tested the most recent versions of the popular GCC and LLVM compilers at the time we conducted our experiments: GCC 10 and 11 and LLVM 10 and 11.

4.1 Compiler Bugs

Throughout our development of CSMITHEDGE, over a period of six months, we regularly ran CSMITH and CSMITHEDGE to find bugs in latest stable versions of GCC and LLVM. We also conducted a small evaluation on a Windows platform to test the Microsoft Visual Studio Compiler (MSVC). Discrepancies between Windows and Linux meant that it was difficult to run our approach natively on Windows, but we copied a set of UB-free programs pre-generated by CSMITH and CSMITHEDGE to a Windows machine and used them to conduct a 24-hour testing run.

To find compiler crashes and hangs on Linux, we compiled each concrete program with GCC and LLVM using each of the standard `-O0`, `-O1`, `-O2`, `-O3`, and `-Os` optimisation levels. We flag a program as potentially triggering a hang bug if it is compiled successfully in general, but leads to a timeout for a particular compiler and optimisation level (with our

[4] The second component of a seed is either 0, 1 or 2 indicating whether functions, macros or a mixture should be used.

300 s time limit); we confirm the hang before considering reporting the bug by checking that compilation does not terminate even after several hours. To find miscompilations on Linux, we compiled each concrete program with GCC and LLVM using each of the standard `-O0` and `-O2` levels with the differential testing approach sketched in Fig. 1b.

We tested MSVC using a randomly-chosen subset of UB-free programs arising from our Linux testing campaign. For both crashes and miscompilations we used the `/Od` (no optimisations) and `/O2` (maximise speed) optimisation settings. We used a Windows build of LLVM as a reference compiler for miscompilation testing.

We found and reported seven compiler bugs in GCC 10, GCC 11, LLVM 10, and MSVC 19.28.29915, which caused the generation of wrong code: three compiler bugs with P2 normal importance in GCC 10 (Bugzilla 2020a; 2020b) and in GCC 11 (Bugzilla 2020c) in the tree-optimisation component, a bug with P3 normal importance in GCC 11 in the tree-optimisation component (Bugzilla 2020e), a bug with P2 normal importance in GCC 11 in the middle-end component (Bugzilla 2020d), a bug in LLVM 10 in the scalar optimisations component (Bugzilla 2020f), and a bug in MSVC 19.28.29915 in an optimisation of out-of-order execution of instructions (Visual Studio Developer Community 2021). We also found two compiler hang bugs in GCC 10, which were independently fixed in the mainline before we got a chance to report them. Finally, we found evidence of several older miscompilation bugs affecting various versions of GCC from 7 through 10 and various versions of LLVM from 6 through 11. The details of all the bugs we found and investigated using CSMITHEDGE are available at GitHub (2021a). We did not find any bugs with CSMITH in any of the compilers during our evaluation.[5]

Table 2 gives an overview of the new bugs found, showing the compiler component affected, the kind of bug (miscompilation or hang), the relaxations that were applied when the bug was found, and the status of our bug report. For all the bugs we found that involved the safe math relaxation, safe math wrappers needed to be implemented as macros for the bug to trigger. However, for bugs that did not require this relaxation, the choice of implementing safe math wrappers as functions or macros had no impact.

During our testing campaign we prioritised investing manual effort into reducing and investigating bugs in the latest versions of GCC and LLVM. However, we conducted long-running background experiments using slightly older versions of these compilers, and the data from these experiments identified a number of execution result mismatches that did not trigger using the most recent compiler versions. It is *possible* that these mismatches are due to UB: although we have found the sanitizer tools used in our experiments to be adequate for our purposes, as discussed in Section 2.2, they are not perfect. However, under the assumption that these mismatches do correspond to historic miscompilation bugs, we performed some analysis to estimate the number of distinct historic miscompilations. Inspired by the "correcting commits" metric of Chen et al. (2016), we obtained results for each program triggering a mismatch for a range of compiler versions, starting with LLVM-6 and GCC-7. We assume that two test programs trigger the same miscompilation in a compiler if they trigger the mismatch for exactly the same versions of the compiler, and at identical optimization levels.

[5] We reproduced several test cases that were CSMITH-generated programs with a reachable, side-effect-free infinite loop. These programs either exceeded the time limit or terminated with a segmentation fault. We investigated one example that led to a segmentation fault in detail, and confirmed that this was due to the UB associated with the infinite loop and not due to a compiler bug.

Table 2 Compiler bugs in GCC, LLVM and MSVC found only by CSMITHEDGE (and not by CSMITH), with a note on the relaxation necessary to trigger the bugs

#Bug	Compiler component	Affected compilers	Kind	Relaxations	Status
1	Tree Optimisation	GCC 10.0.1, 9.2.1, 8.3.0	Miscompilation	Arith checks as macros	Fixed
2	Tree Optimisation	GCC 10.0.1, 9.2.1, 8.4.0, 7.5.0	Miscompilation	Arith checks as macros	Fixed
3	Tree Optimisation	GCC 11.0.0, 10.2.0, 9.3.0, 8.4.0, 7.5.0, 6.4.0, 4.8	Miscompilation	Arith checks as macros	Fixed
4	Middle-end	GCC 11.0.0, 10.2.0	Miscompilation	Arith checks as macros	Fixed
5	Tree Optimisation	GCC 11.0.0	Miscompilation	Arith checks as macros	Fixed
6	Scalar Optimisations	LLVM 6.0, 10.0.0, 11.0.0, 12.0.0, 13.0.0	Miscompilation	Arith checks as macros	Fixed
7	-	GCC 10.1.0	Compiler Hang	Null deref., variable init., goto, and array bounds	Independently fixed
8	-	GCC 10.1.0	Compiler Hang	Dangling ptr. deref., variable init., R/W and W/W conflicts, and array bounds	Independently fixed
9	Code motion optimisation	MSVC 19.28.29915	Miscompilation	Null deref., array bounds	Confirmed

Using this approach, we identified:

- One program that led to a mismatch for LLVM versions 6–11 at all optimization levels higher than `-O0`
- One program that led to a mismatch for GCC versions 7–9 at all optimisation levels higher than `-O0`
- Six programs that led to a mismatch for GCC versions 7–9 at all optimisation levels *including* `-O0`
- One program that led to a mismatch for LLVM versions 6–10 at all optimization levels higher than `-O0`, except that the mismatch does not trigger with `-O1` for LLVM version 10 (but does still trigger at the other optimization levels)

We estimate that these programs characterise four separate historic miscompilation bugs, but manual investigation would be required to confirm this.

It is important to emphasise is that none of the bugs we discovered in recent compiler versions, nor the historic bugs for which we have tentative evidence of discovery, were found by regular CSMITH. In fact, our work has been motivated by the fact that compiler fuzzing techniques like CSMITH—while having found hundreds of compiler bugs in the past—have now largely saturated (Regehr 2019; Babokin 2019), and more diverse programs, less idiomatic in their form, are needed. Our bug finding results show that it is possible to increase diversity and find additional bugs without having to resort to building a brand new fuzzing tool, with the associated manual effort that would be involved.

We now discuss three of the new bugs that we found using CSMITHEDGE.

GCC bug: Missed short-circuiting during folding (Bugzilla 2020c) In a short-circuiting operation, if the first operand is sufficient to determine the overall result, then the second operand should not be evaluated, in case it commits side effects or exhibits UB. GCC had a bug that violated this rule, as shown in the following code snippet which represents the core of the generated program:

```
int main() {
  const long ONE = 1L;
  long y = 0L;
  long x = ((long) (ONE || (y = 1L)) % 8L);
  printf("x = %ld, y = %ld\n", x, y);
}
```

This code should print `y = 0`, but prints `y = 1` with the buggy GCC versions, which generates code that incorrectly executes the `y = 1L` assignment.

This bug was promptly fixed by the developers with a patch in `gcc/fold-const.c` for GCC 11. The bug was only found by CSMITHEDGE when implementing arithmetic checks as macros and relaxing arithmetic checks.

LLVM bug: incorrectly lifting a modulo computation outside an if statement (Bugzilla 2020f) The compiler created a constant expression that was not safe to speculatively evaluate and incorrectly lifted the computation outside its block. This can lead to a miscompilation as shown by the following example, which contains the core of the generated program that exposes the bug:

```
static long r = 0;
static long *m[1] = {&r};
static int i = 1;
static int *pi = &i;
int main() {
  unsigned x = 0;
  if (i) {                          // Evaluates to true
    ++x;                            // Afterwards x == 1
    int test = pi == (int*) 1;
  }
  printf("x = %d\n", x);            // Afterwards x == 1
  if (!x) {                         // Evaluates to false
    i = 7UL % (m[0] == (long*) 1); // Unreachable
  }
}
```

The buggy LLVM version at `-O2` generates code that is incorrect: on execution it throws a "floating point exception" instead of printing "x = 1". This happens because the unreachable statement containing a modulo by 0 is incorrectly lifted and executed.

This bug, which was fixed by the developers, seems related to the `IRBuilder` files in LLVM 10. Again, this bug was only found by CSMITHEDGE when implementing arithmetic checks as macros and relaxing arithmetic checks.

MSVC bug: incorrectly moving an unreachable invalid array access into reachable code (Visual Studio Developer Community 2021) In the following program, the compiler incorrectly lifted an out-of-bounds array access to location $A[13218][231]$ before the if-statement. This led to a miscompilation of the following code with `/O2` optimisation level in MSVC version 19.28.29915 for x64 architecture on a Windows machine.

```
static int32_t a = 1L;
static int32_t b = 0;
static int64_t c = 0L;
static int64_t * volatile d = &c;
int main (int argc, char* argv[])
{
  for (b = 0; b >= 0; b--)
  {
     // Always true, so the loop is broken on its first
     // iteration
     if (a) {
       printf("breaking\n");
       break;
     }
     // Unreachable, since the loop is already broken
     uint16_t A[1][2];
     int j;
     for (j = 0; j < 2; j++)
       A[0][j] = 0;
     // This access would be out of bounds, but it is
     // unreachable
     A[13218][231] && *d;
  }
  printf("done\n");
  return 0;
}
```

The MSVC buggy version at `/O2` generates code that terminates abnormally with exit code `-1073741819` instead of printing “breaking”, “done” and returning `0` since the `if(a)` condition is always true. We performed a manual investigation of this miscompilation, comparing assembly code generated at `/O2` between the buggy and non-buggy compilers. We found that the access to array `A` in the last line of the `for` loop is erroneously computed *before* checking whether `a` is true (that is, before the conditional jump of `if(a)`).

We reported this code to Visual Studio Developer Community, and it was confirmed by a software engineer from the “Machine-Independent Codegen” Team.

This bug was only found using CSMITHEDGE, and not regular CSMITH, because it relies on the generation of code that risks accessing arrays out of bounds, which CSMITH does not generate.

4.2 Throughput and Relaxation Rate

As discussed in Section 3.4, the most natural way to apply CSMITHEDGE is as described in Fig. 1a. However, this results in a significant decrease in performance since the expensive UB detectors (sanitizers and FRAMA-C) are invoked on every program. Instead, by executing the UB detectors only when (i) we know the program terminates (similar to CSMITH), and (ii) a mismatch between compilers is detected during differential testing, our lazy UB checks mode can significantly improve performance when searching for compiler bugs (i.e. the flow of Fig. 1b). We performed our bug-finding campaign using the lazy approach and a 50 s timeout (the CSMITHEDGE default). We now present a comparison of the default settings against CSMITH and the conceptual approach of CSMITHEDGE for a selection of timeouts.

Throughput of Program Generation during Differential Testing. To quantify the performance difference between the standard and lazy UB checks modes, as well as how they compare with CSMITH, we ran an evaluation on a single virtual machine, measuring the throughput of CSMITH, and of CSMITHEDGE with and without lazy UB checks, running each configuration for 24 hours. We measured the throughput of tests comparing the behaviours of LLVM 10 and GCC 10 with `-O2`. We ran the tools in miscompilation mode for this experiment and hence relaxing read-write and write-write conflicts was disabled. We repeated the whole set of experiments presented in this section 10 times to obtain more accurate results. Accordingly, the results in the tables below are an average of these repeated experiments.

We now explain the conditions under which we deemed each tool configuration to have generated a usable test program during our experiments.

- CSMITH: Because CSMITH generates UB-free programs by construction, we regarded a generated program as usable if it did not time out on execution after compilation with LLVM.[6]
- CSMITHEDGE **without** lazy UB checks: We regarded a generated program as usable if it did not crash or time out during the dynamic analysis (Section 3.2) or UB detection stages, and was reported to be UB-free by the UB-detection stage.
- CSMITHEDGE **with** lazy UB checks: We regarded a generated program as usable if (1) it did not crash or time out during the dynamic analysis (Section 3.2) stage, (2) it

[6]We chose LLVM over GCC because, as discussed below, we sometimes detect timeouts during the UB detection phase for other tool configurations, and most of the UB detectors are LLVM-based sanitisers.

Table 3 Programs generated per hour by CSMITH and CSMITHEDGE without and with lazy UB checks

Timeout value	#programs		Csmith	CsmithEdge	CsmithEdge-Lazy
10 s	Generated		877.54	291.22	543.39
	Timed-out		108.34	29.15	51.60
	Invalid	Crash	0.00	75.10	139.10
		Sanitisers	0.00	40.57	19.58
	Usable		769.20	146.39	333.10
50 s	Generated		394.55	212.28	351.28
	Timed-out		48.80	21.30	32.85
	Invalid	Crash	0.00	55.09	90.40
		Sanitisers	0.00	30.80	12.82
	Usable		345.75	105.09	215.21
120 s	Generated		202.24	154.48	220.19
	Timed-out		25.04	15.59	20.51
	Invalid	Crash	0.00	40.25	56.32
		Sanitisers	0.00	21.87	8.05
	Usable		177.20	76.78	135.30

did not time out on execution after compilation with LLVM (since we used the LLVM sanitizers, we chose LLVM for timeout detection for consistency), and (3) either the program led to identical results after compilation and execution using the two compilers under test, or a result mismatch was detected but the program was subsequently found to be UB-free by the UB-detection stage.

Table 3 presents the average throughput of usable programs per hour for CSMITH, and CSMITHEDGE without and with lazy UB checks, respectively. Recall that in all cases, the generated program is executed at least once with a timeout to confirm that it terminates. A large portion of program generation time is due to timeouts in practice, and this will vary according to the timeout that is used. Using too short a timeout leads to discarding programs that would eventually terminate and that might be useful for bug finding, while using too high a timeout reduces the throughput of testing. We show data for three timeout values: 10 s, 50 s (the default value used in our bug-hunting experiments) and 120 s.

The *Generated* rows in Table 3 show the average rate of programs generated during a one-hour period. The generated programs are categorised into programs that time out, invalid programs and usable programs.

A program is in the timed-out category if it exceeded the time limit during the arithmetic check relaxation, UB detection, or miscompilation testing. During UB detection, we applied different time-out settings for the sanitizers, taking into account their overhead (see the beginning of Section 4 for these values).

A program is in the invalid category if it is not suitable for differential testing. Invalid programs are detected in two different ways: if the program crashes during execution (*Invalid - Crash* in the table) and if a sanitizer or FRAMA-C detects an error (*Invalid - Sanitizers* in the table). By construction, CSMITH does not generate invalid programs.[7]

[7]Actually CSMITH can generate programs with UB arising from side-effect free infinite loops, but this happens very rarely.

A program is in the usable category if it is suitable for differential testing; that is, it is UB-free and did not time out.

The rate of programs that time out is around 12% for CSMITH, around 10% for CSMITHEDGE without lazy checks, and around 9% for CSMITHEDGE with lazy checks. One possible explanation for the lower rate in CSMITHEDGE is that many of the programs CSMITHEDGE generates are invalid and quickly crash at runtime, e.g. due to a division by zero. We examined the logs of these experiments and found that majority of the invalid tests failed before reaching the validation stage (e.g. due to a segmentation fault arising during relaxation of safe math arithmetic operators): around 64% for CSMITHEDGE without lazy checks and around 87.5% for CSMITHEDGE with lazy checks.

Looking at the programs marked as invalid, we can observe that a program was relatively rarely rejected due to UB detected by sanitizers or FRAMA-C. For example, with a 50 s timeout, we generated on average 212.28 programs per hour with CSMITHEDGE, out of which 85.89 per hour were invalid but only 30.80 per hour were rejected by either the sanitizer or FRAMA-C. We did not consider any further optimisation on the order of invoking the sanitizer and FRAMA-C, as in practice, it has little effect on the performance of CSMITHEDGE with the lazy checks. However, it might be useful to obtain such order when disabling the lazy checks in CSMITHEDGE.

For each tool configuration and timeout value, the usable column in Table 3 shows the average rate of programs suitable for differential testing that could be generated during a one-hour period. Note that the results for CSMITH and CSMITHEDGE are compiler-independent: the resulting programs could be subsequently used for differential testing of other C compilers. The results for CSMITHEDGE-Lazy are tied to the pair of compilers under test, because it may be that programs on which these compilers agree actually exhibit UB.

Results for the version of CSMITHEDGE without lazy UB checks tell us the rate of UB-free programs (the usable row of Table 3 under CSMITHEDGE statistics). For example with a 50 s timeout: 105.09 programs were found to be UB-free (and thus could be used both for finding compiler crashes/hangs and miscompilations), 85.89 programs had UB detected by either the sanitisers or FRAMA-C or failed during safe arithmetic operators relaxation (and thus could only be used to find compiler crashes/hangs), and 21.30 programs hit the timeout (and hence may also contain UB and have to be discarded if used for finding miscompilations). In practice, with the lazy UB checks, CSMITHEDGE invoked these checks at a much lower rate, hence was able to generate more programs (Table 3, Usable columns of CSMITHEDGE vs. CSMITHEDGE-Lazy) and confirmed fewer programs as triggering UB after those found to exhibit a resulting mismatch (Table 3, CSMITHEDGE-Lazy's invalid rate).

By comparing the rates at which valid programs are generated for different timeout values (Table 3), it appears that a 10 s timeout (or shorter) might be a more suitable default timeout value. However, there is a hard-to-evaluate trade-off between throughput and test thoroughness. A modest number of highly complex test programs, each containing many intricate fragments of code, might have higher potential to expose miscompilation bugs compared with a larger number of more straightforward programs. Using too low a timeout value risks missing information on potential miscompilations arising from larger, more intricate programs if they are skipped due to taking too long to terminate.

Table 4 presents the exact split of time spent on test generation, execution, and external tools to check UB-freedom for CSMITH and CSMITHEDGE without and with lazy checks. We show data for three timeout values, similarly to Table 3. The Gen., Exec., and UB det.

Table 4 Average time spent on test generation, test execution, and UB detection by CSMITH and CSMITHEDGE without and with lazy UB checks

T/O value	Csmith		CsmithEdge			CsmithEdge-Lazy		
	Gen.	Exec.	Gen.	Exec.	UB det.	Gen.	Exec.	UB det.
10 s	2.107 s	1.997 s	3.761 s	0.396 s	8.308 s	3.752 s	0.550 s	2.296 s
50 s	2.148 s	6.976 s	7.513 s	0.401 s	9.291 s	7.428 s	0.586 s	2.211 s
120 s	2.138 s	15.660 s	13.941 s	0.398 s	9.249 s	13.754 s	0.596 s	1.973 s

columns present the average time in seconds spent on test generation, execution and (for CSMITHEDGE only) validation of a test case, respectively.

Because program generation using CSMITH is purely static, the time associated with generation is independent from the execution timeout value used. This is not the case for CSMITHEDGE and CSMITHEDGE-Lazy, because the safe math relaxation involves running each program as part of generation. This means that generation time increases as larger timeout values are used.

Notice that for CSMITHEDGE the amount of time spent executing a compiled program during differential testing is very small and relatively insensitive to the timeout value. This is because all timeouts are detected earlier: only programs that terminate within the timeout limit are considered for differential testing. Consequently, fewer programs have been executed with CSMITHEDGE than with CSMITHEDGE-Lazy (between 40 – 60 % fewer depending on the timeout value), which led to different average execution times between CSMITHEDGE and CSMITHEDGE-Lazy.

Comparing the "UB det." columns for CSMITHEDGE and CSMITHEDGE-Lazy, we can see that the lazy optimisation results in significantly less time spent performing UB detection.

The average generation and validation times across different timeout values are relatively high compared with CSMITH, when only a pair of compilers are cross-checked during differential testing, due to the time spent invoking analysis tools. If a larger number of compilers, or compilers at different optimisation levels, are cross-checked, the proportion of time spent running UB analysers will be less, since more time will be spent compiling and executing programs for differential testing. The overhead of CSMITHEDGE could also be decreased by using lower timeout values for the UB analysis tools, the trade-off being that more programs would be skipped due to these analysis tools timing out.

Throughput of Generating Programs for Crash Testing. All CSMITHEDGE-generated programs are valid for crash testing since the compiler has to process programs without crashing or hanging during compilation regardless of the presence of UB. Therefore, we were also interested in the rate of raw generation. We measured the rate of generating programs per hour for CSMITH and CSMITHEDGE on a single virtual machine for 24 hours. On average, CSMITHEDGE generates 1639 programs per hour (relaxed program generator, without discarding programs that hit timeout and UB detection), while CSMITH generates 1698 programs per hour (program generator, without discarding programs that hit timeout). Note that when using CSMITHEDGE for crash testing, we do not compile and run generated programs, and therefore we do not perform relaxation of arithmetic checks. This also contributes to the high rate of generated programs, because the overhead of running the program to identify redundant arithmetic checks is removed.

Rate of Relaxations. The rate at which we applied relaxations was determined by an array of per-relaxation probabilities, described in Section 3.1. A high probability of applying a relaxation is likely to result in the majority of generated programs exhibiting UB. On the other hand, a low probability might lead to no relaxations being applied, in which case the program CSMITHEDGE generates will be identical to the program that CSMITH would have generated. CSMITHEDGE generated programs that differed from what CSMITH would have generated 84% of the time. The rate of programs out of all uniquely generated programs by CSMITHEDGE that tested to be UB-free is 40%.[8] This rate excludes the relaxation of arithmetic checks, which can still be applied to achieve diversity even when CSMITHEDGE initially generates a program identical to that which CSMITH would have generated.

4.3 Coverage of Compiler Codebases

We used 135K distinct seeds to generate CSMITHEDGE and CSMITH programs for coverage measurements. Measuring coverage requires compiling the compiler codebases without optimisations and instrumenting them with GCOV, both of which introduce large overheads.

Furthermore, we want to measure the impact of the various CSMITHEDGE components, so for each seed, we perform multiple runs. In particular, our evaluation compares four different systems: CSMITH, CSMITHEDGE, and two intermediate versions of CSMITHEDGE:

– **RELAX-GEN:** CSMITHEDGE with only the generation-time relaxations of Section 3.1 enabled.
– **RELAX-ARITH:** CSMITHEDGE with only the relaxations of arithmetic checks of Section 3.2 enabled.

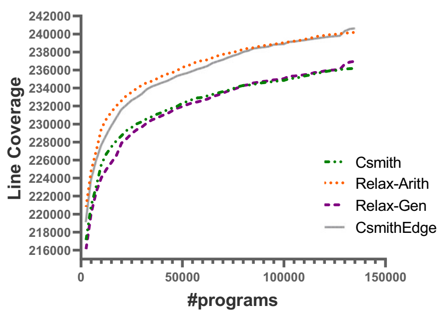
As discussed in Section 3.2, the safe math wrappers can be instantiated via functions or macros. For each CSMITHEDGE-generated program, we configured the tool to choose at random between instantiating all safe math wrappers via functions, all safe math wrappers via macros, or using a randomised mixture of functions and macros. For consistency, we configured CSMITH and the two variations of CSMITHEDGE (RELAX-GEN and RELAX-ARITH) to do the same when generating programs for our coverage experiments. To investigate whether the choice of representation for safe math macros impacted much on coverage results, we repeated the experiments described in this section twice: once forcing all tools to use function wrappers exclusively, and again forcing all tools to use macro wrappers exclusively. The results that we obtained were very similar in all cases, so we omitted the associated graphs.

During the evaluation described in this section (including the investigation above), we used a fixed set of seeds when generating the 135K programs with each of the four systems (taking the data relevant to each system); for example, if the seed for CSMITHEDGE was `1621474906,2,0.8,0,0,0,0.3,0,0` (see Section 3.4), then the seed for RELAX-GEN was the same, and the one for CSMITH was `1621474906`.

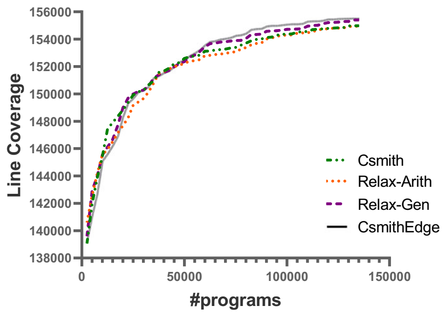
We measured the line coverage achieved on recent versions of the GCC and LLVM compilers: GCC-10.2.1 and LLVM-11.0.0, using each of the 135K programs generated by the four systems.

Figure 2a and b (best viewed in colour) report the cumulative coverage achieved by the four systems in the GCC and LLVM codebases respectively. We measured coverage every

[8]Note that for this experiment, the timeout settings had no effect. This is because we were only interested in the differences between the programs without actually running them.



(a) GCC 10.2.1



(b) LLVM 11.0.0

Fig. 2 Line coverage by the four different generation methods. The number of lines covered by a single CSMITH program is around 150K for GCC and 100K for LLVM

2,500 programs. We used the GCOV-based tool `gfauto` (GitHub 2018) to aggregate results from all machines and generate the coverage results in a human-readable format.

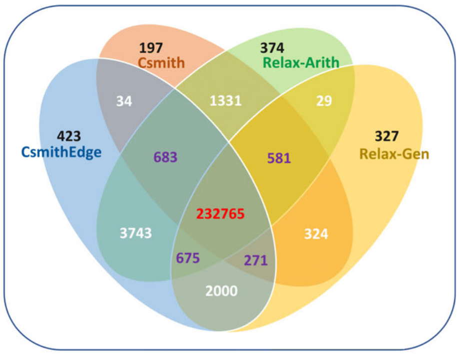
Figure 2a shows that the largest line coverage in GCC was achieved by CSMITHE DGE and the second-largest by RELAX-ARITH. After compilation of 135K programs, CSMITHEDGE covered around 4.4K, 3.6K and 0.4K more lines than CSMITH, RELAX-GEN, and RELAX-ARITH, respectively. Figure 2b shows that the largest line coverage in LLVM was also achieved by CSMITHEDGE, but the difference with CSMITH is less pronounced in this case, at only 0.5K extra lines.

Even when the difference in overall coverage is small, the systems cover different sets of lines. To demonstrate this, Fig. 3a and b present two Venn diagrams of the lines covered after compiling all 135K programs in each set. Each segment of the Venn diagram shows the number of lines covered by the tools associated with the intersection; e.g. 581 lines of GCC 10.2.1 were covered by tests generated by CSMITH, RELAX-GEN RELAX-ARITH (but not CSMITHEDGE) in Fig. 3a.

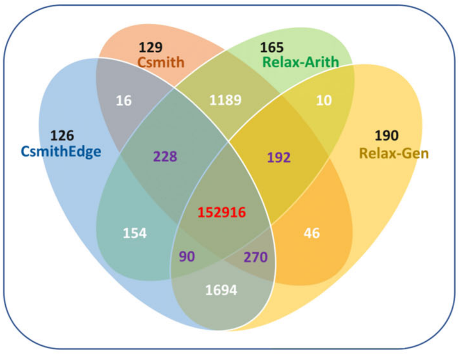
Figure 3c and d present two Venn diagrams of the lines covered by CSMITHEDGE and CSMITH. We can see that while CSMITHEDGE covered more lines of code than CSMITH, each system covers lines that the other does not. This is somewhat expected, as the restrictions in CSMITH are likely to exercise different parts of the compiler, and these restrictions are added by CSMITHEDGE with a smaller probability.

Figure 3e and f present two Venn diagrams of the lines covered by the three versions of CSMITHEDGE. These show that the generation-time and arithmetic check relaxations are complementary: in GCC, RELAX-GEN and RELAX-ARITH covers approximately 3K and 6.1K lines that the other version did not; while in LLVM, the numbers are 2.2K and 1.7K. It is also interesting to notice that RELAX-ARITH and RELAX-GEN cover lines of code which are not covered by CSMITHEDGE. Of course, CSMITHEDGE is able to eventually cover all the lines of code covered by its subcomponents. However, given a fixed time budget, configurations using only certain relaxations can be more effective. This is why we incorporated swarm testing (Groce et al. 2012) into CSMITHEDGE: during a long testing campaign, tests will be generated using many diverse configurations (Section 3.4).

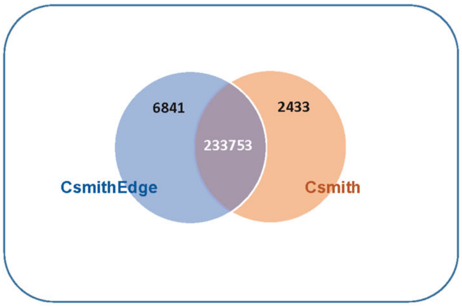
Investigation of additionally-covered code for GCC Recall from Fig. 2a that (in contrast to the results for LLVM in Fig. 2b), CSMITHEDGE provides a modest, but clearly visible, improvement in coverage over CSMITH with respect to the GCC code base, and that most



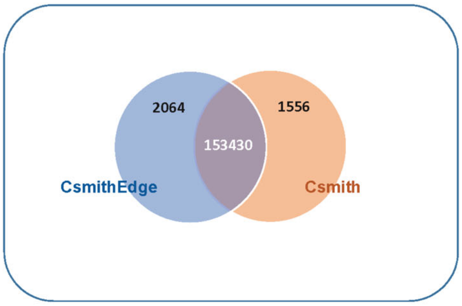
(a) GCC 10.2.1



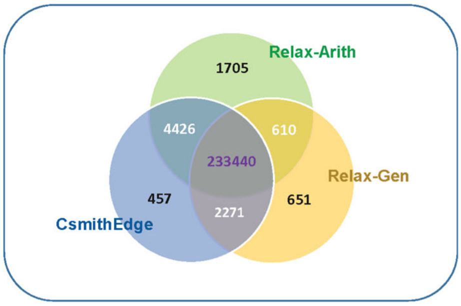
(b) LLVM 11.0.0



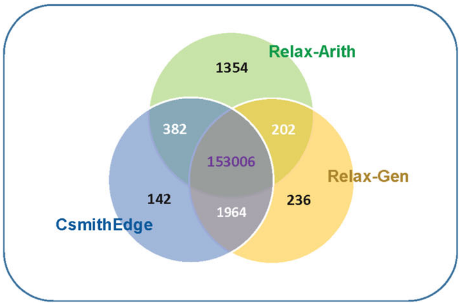
(c) GCC 10.2.1



(d) LLVM 11.0.0



(e) GCC 10.2.1



(f) LLVM 11.0.0

Fig. 3 Venn diagrams comparing the line coverage achieved by the four systems after each generates 135K programs

of this improvement appears to come from our method of relaxing arithmetic checks (see Section 3.2).

We undertook some manual investigation to gain some insights into specific components of the compiler that are covered by CSMITHEDGE but not by CSMITH. New coverage is achieved by CSMITHEDGE in more than 120 files and it was not feasible to investigate these manually. To limit our investigation we identified those files for which at least 50 new lines of code were covered by CSMITHEDGE compared with CSMITH. There were 12 such files and we manually inspected the differences between them.

Two files where CSMITHEDGE achieved very substantial additional coverage are `generic-match.c` and `gimple-match.c`. These files are generated by GCC's `genmatch` program from pattern matching rules written in a domain-specific language, and they implement optimisations that rewrite expressions. Example rewrites covered by CSMITHEDGE but not CSMITH include (from `generic-match.c`):

- `x % C` → `x & (C - 1)`, if `x` and `C` are positive
- `1 / x < y` → `__builtin_mul_overflow(x, y)`, for unsigned `x` and `y` (the original expression checks whether `x * y` would overflow, which can be replaced with a built-in check)
- `x + (x & 1)` → `(x + 1) & ~1`
- `x * C == y * C` → `x == y`, for integral types with undefined overflow, when $C \neq 0$

and (from `gimple-match.c`):

- `(x & (-y)) / y` → `x >> log2(y)`, where `y` is a power of two
- `~(~x >> y)` → `x >> y`
- `(~x & y) | x` → `x | y`
- `min(x, y) < z` → `x < z || y < z`

The vast majority of the new coverage achieved in these files is due to safe math relaxations—most of the coverage is achieved by RELAX-ARITH, and only some of it by RELAX-GEN. This is intuitive: CSMITH's default approach of guarding every potentially unsafe operation with a safe math wrapper means that many expression forms will never occur in generated programs, an issue that our safe math relaxation alleviates somewhat. For similar reasons, CSMITHEDGE achieves substantial additional coverage in `fold-const.c` and `gimple-fold.c` compared with CSMITH; these are also parts of GCC that perform expression simplification.

We found that CSMITHEDGE was able to cover a "loop versioning" transformation not exercised by CSMITH. According to comments in the GCC source code:

> This transformation given a condition and a loop, creates `if (condition) loop_copy1 else loop_copy2` , where `loop_copy1` is the loop transformed in one way, and `loop_copy2` is the loop transformed in another way (or unchanged). `COND_EXPR` may be a run time test for things that were not resolved by static analysis (overlapping ranges (anti-aliasing), alignment, etc.).

where `COND_EXPR`, mentioned in the above description, is a parameter to the transformation. Interestingly this transformation is covered by RELAX-GEN but *not* by RELAX-ARITH, indicating that it arises due to the generation-time relaxations that CSMITHEDGE employs. Code associated with the transformation spans three source files:

- `tree-loop-distribution.c` contains the analysis for determining whether a loop versioning transformation can be performed;
- `tree-data-ref.c` contains logic for determining the pointer aliasing tests that need to be performed at runtime to decide which version of a loop to run (our understanding is that this logic contributes to the `COND_EXPR` parameter mentioned above);
- `cfgloopmanip.c` contains the code for applying the loop versioning transformation, once it has been deemed to be safe.

Another file with substantial new coverage exercised by RELAX-GEN but not RELAX-ARITH is `gimple-ssa-store-merging.c`. Our relaxations of generation-time checks lead to the exercising of additional code paths for determining when it is acceptable to merge store operations.

We have not determined which of our generation-time relaxations account for the loop versioning optimisation being triggered, and for the additional coverage related to merging of store operations. It would be possible to investigate by performing a search for specific tests that trigger this coverage, checking which relaxations were used in generating those tests, and identifying the relaxations common to all such tests.

The above discussion accounts for 8 of the 12 source files in which we identified non-trivial additional coverage. We found it harder to understand the code being exercised in the remaining files:

- `gimple-ssa-strength-reduction.c`: It appears that extra code paths are exercised related to determining when strength reduction (replacing an expensive operation in a loop with a cheaper one) can be applied, but that the new code paths do not lead to additional strength reduction being triggered: deeper precondition checks for strength reduction are performed, but ultimately they still return *false*
- `i386.md`: This is a machine description file for the IA-32 and x86-64 architectures. The new coverage in this large file is sparse and sporadic, and hard to understand.
- `insn-recog.c`: This is an auto-generated file, in which coverage is again sporadic and hard to understand.
- `internal-fn.c`: This file contains various internal GCC helper functions. The new coverage is related to handling of overflow. Our hypothesis is that, with fewer safe math wrappers that would explicitly check for overflow, CSMITHEDGE generates programs for which more compile-time overflow checking is required.

5 Threats to Validity

Internal validity Faults in CSMITHEDGE can lead to (1) detecting what appear to be miscompilation bugs with code that actually turns out to trigger UB, and (2) discarding valid programs. We now comment on some steps we have taken to thoroughly test CSMITHEDGE to minimise the chances of such bugs undermining our work. We carefully tested CSMITHEDGE by examining the rate of non-UB-free detected programs while considering the number of applied relaxations and different lines between CSMITHEDGE-generated and Csmith-generated programs.[9] For example, we investigated such pairs with few modifications or with modifications in unreachable code. In addition, we generated large sets of programs with a single relaxation and checked that the UB detection tool that was supposed to identify the UB this relaxation can cause was the only tool terminating reporting an error and did so consistently. We also investigated relaxations where more than half of the generated programs failed the UB validator. Last, we cross-checked programs with mismatched results with different compilers or optimisations with its compilation warnings to identify undefined and unspecified behaviours, which the UB detection

[9]Our implementation used randomisation but avoided using the very same mechanism used by CSMITH; hence the randomisation related to the UB relaxation part did not directly affect the rest of the random decisions. As a result, a pair of programs with the same seeds produced by CSMITH and CSMITHEDGE were usually identical except for the parts in the code affected by the relaxation.

tools used in this work cannot detect (e.g. reading a field in a union that was not the last one to be written to). Employing such careful testing throughout the project allowed us to fix a number of bugs in CSMITHEDGE during its development, increasing our confidence in the final results we have gathered.

We remark on the steps we have taken to ensure a fair evaluation between CSMITH and CSMITHEDGE. During the evaluation of CSMITHEDGE, we used virtual machines with the same specifications and compiler builds to evaluate the four systems. To account for differences in the priority with which different VMs are given access to underlying resources of the compute cluster, we randomly shuffled execution tasks between VMs. Thus our overall results for CSMITH vs. CSMITHEDGE should not be biased by resource allocation issues. Furthermore, when measuring coverage, we used the same parameters for test case generation with CSMITH and CSMITHEDGE (except for the additional relaxation options CSMITHEDGE takes, as discussed in Sections 3.4 and 4.3); for all experiments, we applied the swarm testing technique when choosing relaxation options for each run of CSMITHEDGE. We regularly ran CSMITH and CSMITHEDGE to find bugs with few optimisation levels; we did not find any bugs during the evaluation in the tested compilers. The reported bugs on Table 2 were confirmed, not marked as duplicated and mostly fixed promptly. We measured the program generation and differential testing rate over a relatively long period of 24 hours and repeated the differential testing rate experiments 10 times to ensure we measured the actual CSMITHEDGE's performance. We adopt part of the guidelines in Klees et al. (2018) when possible and relevant. Since the motivation of this work is the immunity compilers gained with time to existing fuzzers, we did not evaluate our tool against old buggy compilers (which did not yet reach that point).

Construct validity We depend on UB detection tools (sanitisers and static analysers) to decide whether programs are free from UB. This solution uses well-tested existing tools but can have false results: false-positive results can impair the ability to generate diverse code, and false-negative results may lead to a practically unusable approach. In the evaluation, we suggested two experiments to evaluate the effect of false results on (1) the rate of generating valid programs for differential testing (false-negative results) and (2) generating diverse test cases (false-positive results). The evaluation in Section 4.2 suggested a 1.61$\times$ overhead, and Section 4.3 presented a significant additional coverage (thousands of lines) with CSMITHEDGE comparing to CSMITH. We evaluate the additional line coverage with 135K programs; however, when observing the line coverage in Fig. 2a, we think it is worth considering repeating the analysis for GCC with a larger population because the graph does not show the saturation effect.

External validity and transferability In this work, we investigated the idea of adapting existing fuzzers to be less restrictive with respect to UB, and tested our hypotheses for C compilers by extending the CSMITH tool. Due to limited human resources, we did not exercise other fuzzers and programming languages. Our concrete results are thus restricted to this setting. We argue that C is a suitable choice of language because it is widely used in the development of critical code, and because the language features many sources of UB. We chose CSMITH because it generates C programs, is open source with an active community, and because it has been used extensively to test mainstream compilers to the extent that such compilers are largely immune to the tool. This provided a good setting in which to assess whether additional bugs could be found via our technique. With additional engineering, the approach presented in this work could be applied to other C/C++ test program generators, as well as to program mutators (e.g. to eliminate “bad” mutations every *n* pulses). Transferring our ideas to other programming languages that feature UB would require suitable sanitisers and static analysers capable of detecting the relevant UB.

Conclusion validity The bug reports in this article were all bugs that regular CSMITH did not find, in both recent and slightly older compiler versions. To understand how these reports relate to our changes, we examined the additional coverage that CSMITHEDGE has gained compared to CSMITH. According to Fig. 2a and b, with 135K test cases, it was an addition of 3.2% and 1.56% to the line coverage in GCC and LLVM, respectively. The results make it clear that large parts of these compiler code bases remain uncovered despite our extensions. For example, parts of the code bases that deal with C++ (or other language), or with back ends for different processor architectures, will not be covered by our approach, so bugs in such components are out of scope. Because we do not know the extent to which bugs remain in GCC and LLVM we cannot assess the criticality of the new bugs found by our approach compared with additional not-yet-discovered bugs.

During our six-month bug-finding campaign, we did not find any bugs in recent release versions of GCC and LLVM (including those reported in Table 2, and the slightly older compiler versions on which we performed additional testing) using CSMITH. This suggests that these compilers are now largely immunite to CSMITH.

6 Related Work

A preliminary sketch of some of the ideas in this paper, including a prototype of CSMITHEDGE, appeared in a new idea paper (Even-Mendoza et al. 2020).

Compiler fuzzing has a rich history, starting with techniques focusing on languages such as COBOL (Sauder 1962), PL/I (Hanford 1970), and FORTRAN (Burgess and Saidi 1996). More recently the C programming languages has received a lot of attention (Yang et al. 2011; Le et al. 2014; Le et al. 2015; Sun et al. 2016; Nagai et al. 2014; Nakamura and Ishiura 2016; Livinskii et al. 2020), with many bugs reported in popular compilers such as GCC and Clang/LLVM. See (Chen et al. 2020) for a recent survey of the field.

These fuzzers fall into two main categories: those that generate programs from scratch (Yang et al. 2011; Nagai et al. 2014; Livinskii et al. 2020), which are typically used to cross-check multiple compilers, and those that apply transformations to generate equivalent programs (Le et al. 2014; Le et al. 2015; Sun et al. 2016; Nakamura and Ishiura 2016; Donaldson et al. 2017; Donaldson et al. 2021), allowing the behaviour of a single compiler to be cross-checked against multiple equivalent programs (a form of metamorphic testing (Segura et al. 2016; Chen et al. 1998)). Fuzzers in the first category ensure UB-freedom during program generation. We have already discussed at length the kind of restrictions CSMITH imposes to ensure UB-freedom, and other fuzzers from this category similarly take measures to enforce UB-freedom: for instance, the YARPGEN (Livinskii et al. 2020) tool accurately detects and avoids undefined behaviour by tracking variable types, alignments and value ranges during program generation. Fuzzers from the second category rely on a starting set of UB-free programs (which are often obtained by using fuzzers from the first category) and a set of transformations that preserve UB-freedom. For instance, EMI (Le et al. 2014) creates new programs by randomly deleting program statements that are not executed by a certain input (and then compares the original program with the reduced one on that particular input).

As discussed in the Section 2, CSMITHEDGE uses two existing options from CSMITH that relax certain UB-free constraints, namely those for ensuring absence of null and dangling pointer dereferences. However, these options were introduced to enable testing of static analysers (Cuoq et al. 2012; GitHub 2011b), and cannot be reliably used to find miscompilations without the extra measures described in this paper.

The problem of improving diversity during randomised testing has also been approached by adapting the manner in which a test case generator is configured, rather than fundamentally changing the way the generator works. The swarm testing technique (Groce et al. 2012) and HiCOND (Chen et al. 2019) have improved the performance of compiler testing by aiming to produce diverse fuzzing tool test configurations. Although we use the swarm testing technique to choose relaxation options for each invocation of CSMITHEDGE, the main contribution of this work lies in the way we deal with UB. We increase the approach's expressiveness by postponing the decision to discard code that might exhibit UB to the post-generation phase and applying dynamic techniques. This enlarges the vocabulary of our test case generation tool, making it possible to generate programs that were previously out of scope. In contrast, swarm testing and HiCOND aim to improve the way generation configurations are chosen, but are still stuck with the expressiveness of the original technique.

Marcozzi et al. (2019) study how compiler bugs found by fuzzers such as CSMITH compare with compiler bugs reported manually and find that they appear to have a similar impact, which motivates further work on compiler fuzzing.

7 Conclusions and Future Work

We have presented the design and implementation of CSMITHEDGE, and presented a large experimental campaign which shows that, by relaxing the constraints on undefined behaviour imposed by CSMITH, CSMITHEDGE can find previously unknown miscompilation bugs in GCC and LLVM that regular CSMITH does not detect. Our coverage results also demonstrate the extra thoroughness afforded by testing using CSMITHEDGE.

Practical avenues for taking CSMITHEDGE forward include investigating sanitiser support for detecting undefined and unspecified behaviour arising from read-write and write-write conflicts, and applying the tool to additional C compilers (e.g. commercial compilers from Intel and Microsoft, extending the limited amount of testing of the Microsoft Visual Studio Compiler reported here). With appropriate sanitiser support for the OPENCL programming model it would also be possible to apply the techniques in CSMITHEDGE to the CLSMITH compiler fuzzing tool (Lidbury et al. 2015)—an extension of CSMITH that targets OPENCL. It would also be interesting to apply similar ideas to other compiler fuzzing tools, such as YARPGEN (Livinskii et al. 2020), to see whether relaxing their enforcement of UB-freedom can enable discovery of more bugs. Finally, the additional relaxations we have implemented in CSMITHEDGE might prove complementary to the existing flags for allowing null and invalid pointers, in the context of testing static analysis tools.

Acknowledgements We would like to thank John Regehr for answering our questions about Csmith, Paul Thomson for his guidance in using `gfauto` for coverage analysis, and Michaël Marcozzi for sharing data, assisting with `gcov` and proofreading the text. We are also grateful to the anonymous reviewers for their very detailed feedback which helped us to improve the paper substantially.

Funding This project has received funding from the UK Engineering and Physical Sciences Research Council through grants EP/R011605/1 and EP/R006865/1, and from the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement 819141).

Code Availability The code for this work can be found at GitHub (2021b) tag EMSE2021.

References

Address Sanitizer (2012) https://clang.llvm.org/docs/AddressSanitizer.html

Babokin D (2019) Comment on running one million Yarpgen programs https://twitter.com/DmitryBabokin/status/1134907976085516290 [Online; accessed 24-February-2022]

Barr ET, Harman M, McMinn P, Shahbaz M, Yoo S (2015) The oracle problem in software testing: A survey. IEEE Transactions on Software Engineering (TSE) 41(5)

Bugzilla GCC (2020a) Bug 93744 https://gcc.gnu.org/bugzilla/show_bug.cgi?id=93744 [Online; accessed 24-February-2022]

Bugzilla GCC (2020b) Bug 94809 https://gcc.gnu.org/bugzilla/show_bug.cgi?id=94809 [Online; accessed 24-February-2022]

Bugzilla GCC (2020c) Bug 96369 https://gcc.gnu.org/bugzilla/show_bug.cgi?id=96369 [Online; accessed 24-February-2022]

Bugzilla GCC (2020d) Bug 96549 https://gcc.gnu.org/bugzilla/show_bug.cgi?id=96549 [Online; accessed 24-February-2022]

Bugzilla GCC (2020e) Bug 96760 https://gcc.gnu.org/bugzilla/show_bug.cgi?id=96760 [Online; accessed 24-February-2022]

Bugzilla LLVM (2020f) Bug 47578 https://bugs.llvm.org/show_bug.cgi?id=47578 [Online; accessed 24-February-2022]

Burgess C, Saidi M (1996) The automatic generation of test cases for optimizing Fortran compilers. Information and Software Technology (IST) 38:111–119

Chen J, Hu W, Hao D, Xiong Y, Zhang H, Zhang L, Xie B (2016) An empirical comparison of compiler testing techniques. In: Proc. of the 38th international conference on software engineering (ICSE'16)

Chen J, Patra J, Pradel M, Xiong Y, Zhang H, Hao D, Zhang L (2020) A survey of compiler testing. ACM Computing Surveys 53(1):4:1–4:36

Chen J, Wang G, Hao D, Xiong Y, Zhang H, Zhang L (2019) History-guided configuration diversification for compiler test-program generation. In: Proc. of the 34th IEEE international conference on automated software engineering (ASE'19), pp 305–316

Chen T, Cheung S, Yiu S (1998) Metamorphic testing: A new approach for generating next test cases. Tech. Rep. HKUST-CS98-01 Hong Kong University of Science and Technology

Csmith Homepage (2021) https://srg.doc.ic.ac.uk/projects/CsmithEdge/ [Online; accessed 24-February-2022]

Cuoq P, Kirchner F, Kosmatov N, Prevosto V, Signoles J, Yakobowski B (2012) Frama-C: A software analysis perspective. In: Proc. of the 10th international conference on software engineering and formal methods (SEFM'12)

Cuoq P, Monate B, Pacalet A, Prevosto V, Regehr J, Yakobowski B, Yang X (2012) Testing static analyzers with randomly generated programs. In: Proc. of the 4th international conference on NASA formal methods (NFM'12)

Donaldson AF, Evrard H, Lascu A, Thomson P (2017) Automated testing of graphics shader compilers. In: Proc. of the ACM on programming languages (OOPSLA'17)

Donaldson AF, Thomson P, Teliman V, Milizia S, Maselco AP, Karpinski A (2021) Test-case reduction and deduplication almost for free with transformation-based compiler testing. In: Proc. of the conference on programing language design and implementation (PLDI'21)

CsmithEdge - Homepage (2022) https://srg.doc.ic.ac.uk/projects/CsmithEdge/ (Date Accessed February 24)

Even-Mendoza K, Cadar C, Donaldson A (2020) Closer to the edge: Testing compilers more thoroughly by being less conservative about undefined behaviour. In: Proc. of the 35th IEEE international conference on automated software engineering, new ideas and emerging results (ASE NIER'20)

Frama-C EVA plugin (2007) https://frama-c.com/fc-plugins/eva.html
GitHub Y (2018) Git repository of Yarpgen https://github.com/intel/yarpgen [Online; accessed 24-February-2022]
GitHub (2020a) Csmith pull request 86 https://github.com/csmith-project/csmith/pull/86 [Online; accessed 24-February-2022]
GitHub (2020b) Csmith pull request 88 https://github.com/csmith-project/csmith/pull/88 [Online; accessed 24-February-2022]
GitHub (2021a) CSMITHEDGE bugs details. https://github.com/karineek/CsmithEdge/tree/master/results/bugs [Online; accessed 24-February-2022]
GitHub (2021b) CSMITHEDGE repository. https://github.com/karineek/CsmithEdge.git [Online; accessed 24-February-2022]
GitHub (2011a) Git repository of Csmith https://github.com/csmith-project/csmith.git [Online; accessed 24-February-2022]
GitHub (2011b) Git repository of Csmith, commit 7e33250, Csmith's options for testing static analyzers. https://github.com/csmith-project/csmith/commit/7e3325060b56cc5813b8701087b5206fb394c047, [Online; accessed 24-February-2022]
GitHub (2018) Git repository of gfauto https://github.com/google/graphicsfuzz.git [Online; accessed 24-February-2022]
Groce A, Holzmann GJ, Joshi R (2007) Randomized differential testing as a prelude to formal verification. In: Proc. of the 29th international conference on software engineering (ICSE'07). IEEE Computer Society, pp 621–631
Groce A, Zhang C, Eide E, Chen Y, Regehr J (2012) Swarm testing. In: Proc. of the international symposium on software testing and analysis (ISSTA'12), pp 78–88
Hanford K (1970) Automatic generation of test cases. IBM Syst J 9:242–257
International Organization for Standardization (2018) ISO/IEC 9899:2018: Programming Languages—C
Klees G, Ruef A, Cooper B, Wei S, Hicks M (2018) Evaluating fuzz testing. In: Proc. of the 24th ACM conference on computer and communications security (CCS'18), p 2123–2138
Le V, Afshari M, Su Z (2014) Compiler validation via equivalence modulo inputs. In: Proc. of the conference on programing language design and implementation (PLDI'14)
Le V, Sun C, Su Z (2015) Finding deep compiler bugs via guided stochastic program mutation. In: Proc. of the 30th annual conference on object-oriented programming systems, languages and applications (OOPSLA'15)
Leroy X (2009) Formal verification of a realistic compiler. Communications of the Association for Computing Machinery (CACM) 52(7):107–115
Lidbury C, Lascu A, Chong N, Donaldson AF (2015) Many-core compiler fuzzing. In: Proc. of the conference on programing language design and implementation (PLDI'15)
Livinskii V, Babokin D, Regehr J (2020) Random testing for C and C++ compilers with YARPGen. In: Proc. of the ACM on programming languages (OOPSLA'20), vol 4, pp 196:1–196:25
Marcozzi M, Tang Q, Donaldson A, Cadar C (2019) Compiler fuzzing: How much does it matter? In: Proc. of the ACM on programming languages (OOPSLA'19)
McKeeman WM (1998) Differential testing for software. Digit Tech J 10:100–107
Memory Sanitizer (2015) https://clang.llvm.org/docs/MemorySanitizer.html
Nagai E, Hashimoto A, Ishiura N (2014) Reinforcing random testing of arithmetic optimization of C compilers by scaling up size and number of expressions. IPSJ Transactions on System LSI Design Methodology 7:91–100
Nakamura K, Ishiura N (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)
Regehr J (2019) Comment on running one million Csmith programs https://twitter.com/johnregehr/status/1134866965028196352 [Online; accessed 24-February-2022]
Regehr J, Chen Y, Cuoq P, Eide E, Ellison C, Yang X (2012) Test-case reduction for C compiler bugs. In: Proc. of the conference on programing language design and implementation (PLDI'12)
Sauder RL (1962) A general test data generator for COBOL. In: Proc. of the 1962 spring joint computer conference (AIEE-IRE'62 spring)
Segura S, Fraser G, Sanchez A, Ruiz-cortés A (2016) A survey on metamorphic testing
Serebryany K, Bruening D, Potapenko A, Vyukov D (2012) Addresssanitizer: A fast address sanity checker. In: Proc. of the 2012 USENIX annual technical conference (USENIX ATC'12)
Stepanov E, Serebryany K (2015) Memorysanitizer: fast detector of uninitialized memory use in C++. In: Proc. of the international symposium on code generation and optimization (CGO'15)
Sun C, Le V, Su Z (2016) Finding compiler bugs via live code mutation. In: Proc. of the 31st annual conference on object-oriented programming systems, languages and applications (OOPSLA'16)
Undefined Behavior Sanitizer (2017) https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

Visual Studio Developer Community (2021) Bug 1485361: Msvc miscompiles program with unreachable out of bounds access at /o2 https://developercommunity.visualstudio.com/t/msvc-miscompiles-program-with-unreachable-out-of-b/1485361 [Online; accessed 24-February-2022]

Wang X, Zeldovich N, Kaashoek F, Solar-Lezama A (2013) Towards optimization-safe systems: Analyzing the impact of undefined behavior. In: Proc. of the 24th ACM symposium on operating systems principles (SOSP'13)

Yang X, Chen Y, Eide E, Regehr J (2011) Finding and understanding bugs in C compilers. In: Proc. of the conference on programing language design and implementation (PLDI'11)