Test-Case Reduction and Deduplication Almost for Free with Transformation-Based Compiler Testing

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

Paul Thomson
Google
UK
paulthomson@google.com

Vasyl Teliman
National Technical University
Ukraine
vasniktel@gmail.com

Stefano Milizia
Imperial College London
UK
stefano.milizia00@gmail.com

André Perez Maselco
Federal University of ABC
Brazil
andreperezmaselco.developer@gmail.com

Antoni Karpiński
Warsaw University of Technology
Poland
ant.karpinski@gmail.com

Abstract

Recent transformation-based approaches to compiler testing look for mismatches between the results of pairs of equivalent programs, where one program is derived from the other by randomly applying *semantics-preserving transformations*. We present a formulation of transformation-based compiler testing that provides effective test-case reduction almost for free: if transformations are designed to be as small and independent as possible, standard delta debugging can be used to shrink a bug-inducing transformation sequence to a smaller subsequence that still triggers the bug. The bug can then be reported as a delta between an original and minimally-transformed program. Minimized transformation sequences can also be used to heuristically deduplicate a set of bug-inducing tests, recommending manual investigation of those that involve disparate types of transformations and thus may have different root causes. We demonstrate the effectiveness of our approach via a new tool, spirv-fuzz, the first compiler-testing tool for the SPIR-V intermediate representation that underpins the Vulkan GPU programming model.

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

Keywords: Compilers, metamorphic testing, SPIR-V

ACM Reference Format:
Alastair F. Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. 2021. Test-Case Reduction and Deduplication Almost for Free with Transformation-Based Compiler Testing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454092

This work is licensed under a Creative Commons Attribution 4.0 International License.

PLDI '21, June 20–25, 2021, Virtual, Canada
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454092

1 Introduction

Because compilers are critical pieces of infrastructure on which practically all deployed software depends, there has been a lot of research interest in methods for *randomized compiler testing* [4], which involves feeding randomly-generated or randomly mutated programs to a compiler with the aim of provoking bugs. Using randomized testing to find *miscompilations*, where the compiler mistakenly produces wrong code, is hampered by the lack of an oracle to determine whether the result produced by a compiled program is acceptable. The oracle problem can be avoided by *cross-checking* results for a program across multiple compilers, or results for multiple equivalent programs using a single compiler.

Techniques that cross-check multiple compilers usually work by randomly generating programs and associated inputs from scratch, taking measures to ensure that generated programs are free from undefined behavior (UB) when executed on their inputs. A generated program is compiled by each compiler under test and the resulting binaries are executed. If the compilers agree on implementation-defined behavior (e.g. the widths of integer types in C), the execution results should be identical; mismatches are signs of compiler bugs. Csmith [38] and YARPGen [26] are examples of program generators that facilitate cross-checking of C compilers.

In contrast, techniques that cross-check equivalent programs with respect to a single compiler typically start with an *original* program—e.g. an existing compiler test case—and an associated input such that the program is free from UB when executed on the input. They then apply many *semantics-preserving transformations* to the program in a randomized fashion: transformations that neither change the result the program computes when executed on its input, nor introduce UB. This allows many *variants* of the original program to be

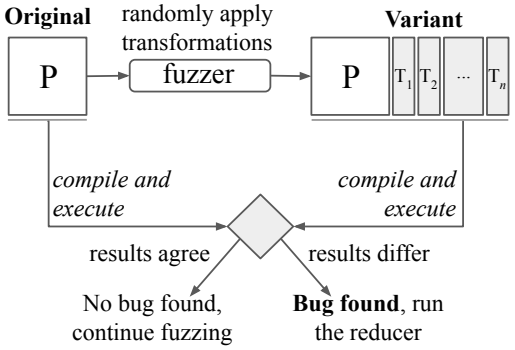


Figure 1. Transformation-based testing randomly applies semantics-preserving transformations to an original program to yield more complex *variants*

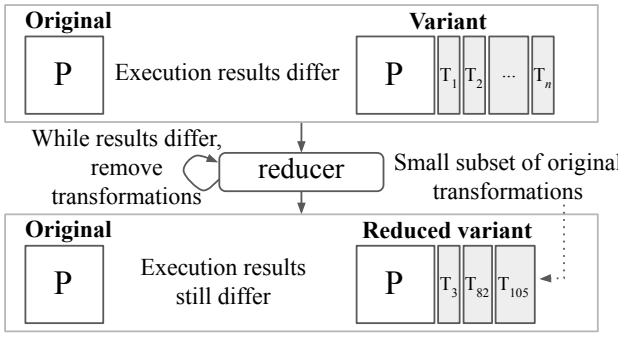


Figure 2. Test-case reduction involves searching for a *reduced variant* featuring a minimal set of bug-triggering transformations (transformations 3, 82 and 105 in this example)

```
            ...                                  ...
        OpReturnValue %83                    OpReturnValue %83
        OpFunctionEnd                        OpFunctionEnd
%18 =   OpFunction %2 None %3        %18 =   OpFunction %2 DontInline %3
%19 =   OpLabel                      %19 =   OpLabel
%86 =   OpVariable %7 Function       %86 =   OpVariable %7 Function
            ...                                  ...
```

Figure 3. The small delta between an original SPIR-V program (left) and reduced variant (right) that triggered a bug in SwiftShader. Both feature 481 instructions but differ in just one instruction.

generated, such that the original program and all variants should compute identical results for the input. Again, result mismatches are indicative of compiler bugs. As two examples, the Orion tool [22] transforms a C program by randomly deleting statements that have been shown to be dynamically unreachable, while the glsl-fuzz tool (originally called GLFuzz) [7] randomly transforms an OpenGL shader program using more general semantics-preserving transformations, such as wrapping a block of code in a single-iteration loop.

We refer to approaches that generate equivalent variants by transforming an original program as *transformation-based (compiler) testing* techniques.[1] Figure 1 illustrates visually how the process of finding bugs via transformation-based testing works. The *variant* program is depicted as the original program *P* with a number of transformations applied to it.

We present an approach to designing transformation-based testing tools that, almost for free, helps solve two important problems associated with randomized testing: automated *reduction* and *deduplication* of bug-triggering test cases. We explain why each of these problems are important and how transformation-based testing can help to solve them.

Test-case reduction. A generator of programs for compiler cross-checking should err on the side of generating large and complex programs, to increase the probability that a generated program will happen to exhibit the requisite features to trigger a bug [38]. Likewise, a transformation-based testing tool should apply many diverse transformations to an original program to generate complex variants.

However, large and complex programs are not suitable for *reporting* compiler bugs as they are difficult for human developers to understand. This has led to the design of customized tools for automatically reducing programs, such as C-Reduce [32], C-Vise [27] and `llvm-reduce` [36]. The guidelines for submitting bug reports to the LLVM compiler insist that a report should include a reduced test case and recommends using one of these tools [36]. When reducing programs that trigger miscompilation bugs, it is vital to preserve freedom from UB, otherwise the reducer tends to produce a small program that exhibits a difference in the way two compilers resolve UB instead of a program that triggers the original miscompilation bug. Program reducers rely on a variety of external static and dynamic analysis tools (such as the Clang Static Analyzer [35] and AddressSanitizer [35] in the context of C/C++) to identify and discard candidate reduced programs that trigger UB. These tools can be imprecise or expensive to run, and may not yet exist if compilers for a new language are being tested.

Transformation-based testing offers a different take on test-case reduction, illustrated in Figure 2. Having found a variant program that triggers a compiler bug (see Figure 1), instead of trying to reduce it to a small, standalone program that still triggers the bug, we can consider *simplifying* the variant so that it is as similar as possible to the original program, is still equivalent to the original program, and still triggers the bug. If the delta between the original program and the simplified variant is small enough that it is obvious to the human eye that it should have no semantic impact then the *pair* of programs provides actionable evidence for the compiler bug, suitable for inclusion in a bug report.

[1] We prefer "transformation" over "mutation" to avoid confusion with mutation-based fuzzing [28, 39].

Figure 3 illustrates the delta between an original and reduced variant for a bug found using spirv-fuzz, the new transformation-based testing tool for the SPIR-V programming language that we discuss later in the paper. The original and reduced programs both feature 481 instructions, but differ in just *one* instruction. The delta is trivial: in the reduced variant the `DontInline` attribute has been added to request that a function is not inlined. This was sufficient to provoke a bug in SwiftShader, a software rendering tool that supports SPIR-V [10]. It is immediately apparent from the delta that the underlying bug is related to the handling of function calls, providing a good starting point for debugging.

If transformations have been designed to be as simple and fine-grained as possible then this form of test-case reduction is easy to implement: the simple and well-known delta debugging algorithm [40] can be used to search for a short subsequence of transformations that still yields a bug-inducing variant. Because the original program is free from UB (a requirement of transformation-based testing) and transformations are semantics-preserving (thus do not introduce UB), the simplified variant program is also free from UB. External UB analysis tools are thus not required.

Test-case deduplication. Randomized testing is notorious for finding duplicate bugs. While a small number of distinct test cases that expose the same bug may be useful [41], it is not appropriate to flood the bug tracker of a system under test with a large number of reports containing many duplicates.

When applying randomized testing to a project where bugs can be rapidly fixed it may be feasible to repeat the process of: testing for a while; reporting one bug; waiting for a fix [31]. However, when conducting external testing of a system that is updated infrequently it is desirable to be able to report many distinct bugs affecting the current version of the system, so that the next update can include as many fixes as possible. Android graphics drivers fall into this category: driver updates are slow, indicated by the recent updatable drivers effort [2], which still only targets a subset of devices.

In the context of compiler testing, test cases that trigger compiler *crashes* are usually easy to deduplicate based on error messages, but deduplication is much harder for miscompilations, where incorrect code is emitted with no indication of the cause of the incorrectness.

We propose using the transformations associated with a reduced variant as a heuristic for deduplication. If two reduced variants feature radically different types of transformations then they may well turn out to trigger bugs with distinct root causes. This is based on the intuition that compiler bugs tend to be triggered by particular features of input programs, and thus two variants that have been created by applying very different transformations are likely to exhibit significantly different features. Of course, this is not guaranteed; we merely propose using this idea as a heuristic for deduplication.

Our contributions. We show that by following certain principles in the design of transformation-based compiler testing, the benefits of test-case reduction and deduplication described above can be enjoyed *almost* for free. To demonstrate this, we have designed and implemented spirv-fuzz [34], the first compiler-testing tool to directly target SPIR-V [17], the intermediate representation used by the Vulkan GPU programming model [15] and implemented by many GPU vendors. SPIR-V is a timely target: many SPIR-V compilers are under continuous development in response to evolving GPU hardware, and are thus prone to regressions that spirv-fuzz can defend against. SPIR-V features UB, but there do not yet exist mature tools for static and dynamic analysis of this UB, thus a test-case reduction approach that does not require such tools is particularly valuable. Furthermore, SPIR-V compilers are part of modern Android graphics drivers, which are infrequently updated as discussed above, motivating the need for effective deduplication techniques.

We explain our approach in general (§2) and then describe the design of spirv-fuzz (§3), using examples to illustrate why test-case reduction and deduplication come *almost* but not *completely* for free: care is required in designing transformations that are as small and independent as possible.

We report on controlled experiments to evaluate the effectiveness of spirv-fuzz (§4). We first assess its bug-finding ability by comparing it with glsl-fuzz [7], a compiler-testing tool for the OpenGL shading language that can target SPIR-V via cross-compilation [8]. Applied to 9 different SPIR-V compilers from 6 different organizations, our results show that spirv-fuzz is more effective at bug-finding than glsl-fuzz with high statistical confidence. We compare the reduction quality of spirv-fuzz with that of glsl-fuzz, which has a hand-crafted reducer, using data from hundreds of reduction runs, using the size difference between an original program and reduced variant as a measure of reduction quality (a small difference indicates that reduction has worked well). The median difference in instruction counts across all reductions is 8 for spirv-fuzz, vs. 29 for glsl-fuzz. The size differences between original and *unreduced* programs are in the order of many thousands of instructions, so our results show that both tools are effective at test-case reduction, and suggest that spirv-fuzz is slightly more effective despite the fact that its reducer is not hand-crafted. We empirically assess our deduplication heuristic using a set of 1467 reduced bug-triggering test cases that are known to trigger 78 crash bugs with distinct root causes. Our heuristic suggests creating bug reports for 49 of the reduced variants, covering 41 of the distinct bugs, thus missing 29 bugs and featuring 8 duplicates. The low rate of duplicates suggests that this approach is relatively conservative so that using it is unlikely to lead to developers being overwhelmed with duplicate bugs, while still allowing reporting of a large portion of the underlying distinct bugs.

We also discuss our experience of putting spirv-fuzz into practice, using it to test several SPIR-V compilers that are under active development, leading to us finding and reporting 74 bugs that we believe to be distinct (§5).

2 Transformation-Based Compiler Testing

We give an overview of our approach (§2.1) then present it more formally (§2.2), and discuss principles for maximizing the success of test-case reduction and deduplication (§2.3).

2.1 Example-Driven Overview

We illustrate our approach using a simple language of basic blocks. Every block contains zero or more instructions of the form $x := y_1$, $x := y_1 + y_2$ or $\mathsf{print}(y_1)$, where x is a variable and each of y_1, y_2 is either a variable or a literal. A block either branches unconditionally to a single successor, or conditionally to a pair of successors based on the value of a boolean variable, in which case edges are labeled with the variable or its negation. This language is only intended to provide intuition for our approach and we do not formalize it further. Figure 4 shows several "basic blocks" programs that all print the same output—the value 6—when executed on the input shown in the figure. We discuss the transformations in the figure and the meaning of **dead** in due course.

A transformation-based compiler-testing tool has two main components: a *fuzzer* and a *reducer*.

The fuzzer. The fuzzer takes an initial program and input and repeatedly modifies them in randomized fashion by applying a series of *transformations*. Each transformation is an instantiation of a template that defines a family of related ways that the program and/or its input can be transformed.

Table 1 sketches templates for a number of transformations that could be applied to "basic blocks" programs. The parameters of a template control the modifications the transformation should make. For example, the parameters of SplitBlock specify the block b that should be split, the instruction offset o at which the split should occur, and an identifier f for the new block that is introduced. A template has a *precondition* that must hold for the transformation to be safely applied, and an *effect* determining how the program and input should change. These must satisfy the following: if the precondition holds, the output of the original (program, input) pair must match the output of the (program, input) pair obtained by applying the effect. The precondition may depend on whether certain *facts* about the program and input are known to hold, and the effect can record new facts that arise as a result of how the program or input was modified.

Figure 4 shows a sequence of transformations that the fuzzer might apply to a (rather trivial) "basic blocks" program. In practice, one would start with a more realistic program and apply hundreds or thousands of transformations at random.

Transformation T_1 splits block a after its leading instruction, placing the remaining instructions in a new block b. The precondition holds because block a exists, 1 is a valid offset into a, and the identifier b is fresh.

A *dead* block c is added by T_2: a is modified to conditionally branch to its original successor b or a new block c depending on the truth of a new variable, u. The new block c branches to b. By setting u to *true* the transformation guarantees that c will not be reachable at runtime (it is dynamically *dead*). Right after T_2 has been applied this is apparent, but future transformations might obfuscate the program and make it non-obvious (especially if we had more sophisticated transformations than those of Table 1). Thus T_2 records a *fact*, "c is dead", indicated by **dead** in Figure 4, which the preconditions of future transformations can take on trust.

In particular, T_3 exploits the fact "c is dead" by adding an instruction to c that stores the value of i to existing variable s. In general, adding a store to an existing variable may completely change the program's semantics and thus would not be legitimate, but a store in a dead block has no effect.

In contrast, a load from an existing program variable into a fresh variable may be safely added at any program point; T_4 adds a load from s to new variable v.

Finally, T_5 exploits the fact that input k has the value *true* to change the assignment $u := true$ to $u := k$, obscuring from the compiler (which cannot make assumptions about the inputs on which the program will be run) the fact that control is guaranteed to pass from a directly to b.

The reducer. The example of Figure 4 is so simple that it is clear both the original and fully transformed programs should print the same value, but this will not be the case when applying transformation-based testing for real. The job of the *reducer* is to search for a small subsequence of transformations that suffice to trigger the bug. This will hopefully provide a much smaller transformed program to serve as a good starting point for debugging.

Suppose the final program of Figure 4 was found to trigger a bug in a "basic blocks" compiler. The reducer would repeatedly try applying subsequences of $T_1, \ldots, T_5$ from scratch to the original program, in each case checking whether the bug still manifests. This is where the precondition of a transformation is important: it would make no sense to apply T_2, which adds a dead block c with successor b, if T_1 had not been applied, since T_1 introduces b. Because the effect of a transformation is *guaranteed* to preserve program output when the precondition holds, the reducer can try any subsequence of transformations, skipping those whose preconditions fail. For example, applying the subsequence T_1, T_3, T_4, T_5 leads to only T_1 and T_4 being applied: T_3's precondition does not hold because block c (which would have been introduced by T_2) does not exist; T_5 cannot be applied because the assignment $u := true$, which T_5 modifies, is not present.

The reduction process can be driven by *delta debugging* [40], with the reducer terminating when a *1-minimal* sequence of transformations is found: a sequence that triggers the

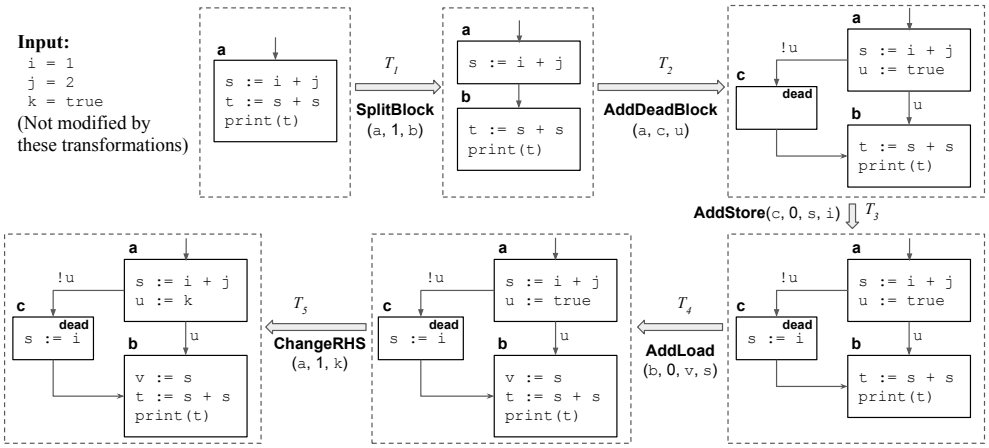


Figure 4. A series of transformations applied to a "basic blocks" program; **dead** denotes a "block is dead" fact

Table 1. Transformation templates for our basic blocks language, where b denotes a block, o an offset, x an existing variable, and f a fresh block or variable identifier. Each precondition additionally requires that any block b or variable x exists.

Transformation	Precondition	Effect
SplitBlock(b, o, f)	b has at least o instructions; f is fresh	Instructions $b[o]$ onward are placed in new block f; f's successors are b's original successors; b branches to f
AddDeadBlock(b, f_1, f_2)	b has a single successor, c; f_1 and f_2 are fresh and distinct	New block f_1 is introduced, branching to c; $f_2 := \mathit{true}$ is added to b; b is changed to branch to c if f_2 holds and to f_1 otherwise; fact "f_1 is dead" is recorded
AddLoad(b, o, f, x)	b has at least o instructions; f is fresh	$f := x$ is added to b at index o
AddStore(b, o, x_1, x_2)	Fact "b is dead" holds; b has at least o instructions	$x_1 := x_2$ is added to b at index o
ChangeRHS(b, o, x)	$b[o]$ has the form $y := z$; x and z are guaranteed to be equal at $b[o]$	z is replaced with x in $b[o]$

bug with the property that if any single transformation is removed from the sequence the bug is not triggered.

Suppose that to trigger the hypothetical bug in our example it suffices to add a dead block and obfuscate the fact that it is dead. Reduction would then find the minimized transformation sequence T_1, T_2, T_5, whose application is illustrated in Figure 5. The ticks and cross indicate that programs P_0—P_2 do not trigger the bug (if they did, the sequence would not be 1-minimal) but P_3 does. Comparing P_3 with the final program of Figure 4 we see that P_3 is simpler.

Bug reports and regression tests. Suppose we have found a 1-minimal sequence of transformations $T_1, \ldots, T_n$ that triggers a bug when applied to an initial program and input (P_0, I_0). If the program and input obtained by applying the first j transformations is denoted (P_j, I_j) then any pair of programs and inputs $((P_j, I_j), (P_n, I_n))$, for $j < n$, is suitable for illustrating the bug: the delta between P_j and P_n illustrates a sufficient modification for the bug to trigger. In practice, the cases where $j = 0$ or $j = n-1$ are the most useful. The former demonstrates the complete delta, e.g. the pair of programs (P_0, P_3) in Figure 5 (all programs in the figure operate on the same inputs). This is often attractive if the original program P_0 is well-understood. The latter demonstrates only the final transformation, e.g. the pair of programs (P_2, P_3) in Figure 5. This involves the smallest delta, but sometimes the fact that this delta is semantics-preserving can be non-obvious when it is viewed in isolation. The pair of programs and inputs used to report the bug also provides a natural regression test that can be added to the compiler's test suite or to a conformance test suite for the programming languages. Specifically the test should execute both programs on their respective inputs and check that their results are the same.

The comprehensibility of these bug reports and regression tests depends on a reasonably small original program. In our experiments (§4) we have found that a set of existing manually-written test cases works well.

Deduplicating bug reports. Suppose we ran "basic blocks" fuzzing over a weekend and returned to find a set of 100 minimized miscompilation bug reports. As randomized testing is infamous for producing duplicate bug reports, it is likely that the reports all stem from a small number of distinct miscompilation bugs, but we have no way of knowing for sure without investigating the root cause of each bug in turn.

Suppose that: 35 of the minimized reports (set A) involve all three of SplitBlock, AddDeadBlock and ReplaceRHS, potentially multiple times, and no other transformations; 42

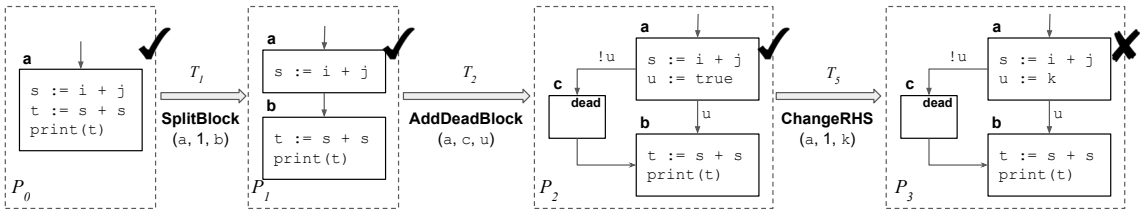


Figure 5. A minimized transformation sequence obtained by applying test-case reduction to the transformations of Figure 4

Input: *Tests*, a set of reduced test cases
Output: *ToInvestigate*, a subset of *Tests* for investigation

```
ToInvestigate ⇐ ∅
i ⇐ 1
while Tests ≠ ∅ do
  if ∃t ∈ Tests . |types(t)| = i then
    ToInvestigate ⇐ ToInvestigate ∪ {t}
    Tests ⇐ {t′ ∈ Tests | types(t) ∩ types(t′) = ∅}
  else
    i ⇐ i + 1
  end if
end while
```

Figure 6. Deduplicating reduced test cases; $types(t)$ denotes the set of transformation types associated with test case t

of the reports (set B) feature all of, and only, AddStore and AddLoad; and the remaining 23 reports feature at least four of the five types of transformations. It seems reasonable to report two bugs, one from each of A and B. Being triggered by totally different transformations they have a good chance of being distinct.

This deduplication approach is formalized by the algorithm of Figure 6. In the algorithm, $types(t)$ returns the (unordered, duplicate-free) set of transformation types associated with the transformations involved in test t. The algorithm takes a set of reduced test cases (i.e. their transformation sequences have been minimized) and returns a subset *ToInvestigate* such that no two tests in *ToInvestigate* share a common transformation type. Our hypothesis is that they thus have a good chance of triggering distinct bugs. If a transformation-based testing tool features certain basic transformations whose main purpose is to support more interesting future transformations then it may be worthwhile ignoring these basic transformations when deciding whether two transformation sequences have any types in common. We discuss this adaptation in the context of our spirv-fuzz tool in §3.5.

2.2 Formal Treatment

We now present transformation-based testing more precisely, formalizing the notion of the precondition and effect of a transformation and the role of facts. We deliberately leave details of the programming language of interest abstract.

Definition 2.1. A program P is *well-defined* with respect to an input I if, according to the semantics of the programming language, P is guaranteed to terminate normally when executed on I yielding a deterministic result. In this case we use $\text{Semantics}(P, I)$ to denote the result.

This definition restricts us to testing correct compilation of programs that compute a *deterministic* final result, a restriction shared with most prior work on compiler testing.

Definition 2.2. When applied to a program P and input I, an *implementation* Impl of the programming language of interest either *faults* or yields a value $\text{Impl}(P, I)$. Implementation Impl is *correct* if, whenever a program P is well-defined with respect to an input I, Impl does not fault when applied to P and I, and satisfies $\text{Impl}(P, I) = \text{Semantics}(P, I)$.

For simplicity we do not get into details of program termination, and regard a non-terminating program as faulting.

A *fact* about a program P and input I is any property over P and I that holds (recall e.g. the "block b is dead" fact).

Definition 2.3. A *transformation context* C (called a *context* for brevity) is a tuple (P, I, F) where P is a program, I an input for P such that P is well-defined with respect to I, and F a set of facts about P and I. The set of all contexts is denoted by Contexts.

Definition 2.4. A *transformation* is a tuple $(Type, Pre, Effect)$ where:

- *Type* is drawn from some finite set of identifiers and indicates the type of the transformation.
- *Pre*, the *precondition* of the transformation, is a predicate over Contexts.
- *Effect*, the *effect* of the transformation, is a partial function from Contexts to Contexts.
- For every $C \in$ Contexts:
  - If $Pre(C)$ does not hold then $Effect(C)$ is not defined.
  - Otherwise $Effect(C)$ is defined as (P', I', F'), say, and $\text{Semantics}(P, I) = \text{Semantics}(P', I')$.

The *Type* component of a transformation is useful for type-based deduplication of bug reports, as shown in the algorithm of Figure 6.

By Definition 2.3, the program associated with a context is well-defined with respect to the associated input. Hence Definition 2.4 precisely captures the notion that when a transformation is applicable it produces a valid program-input pair

that yields the same output as the original program-input pair. Likewise, because the facts associated with a context hold for the program and input, the definition captures the idea that a transformation's precondition can depend on facts, and its effect can add new facts.

A sequence of transformations can be applied to a context, skipping transformations whose preconditions fail:

Definition 2.5. Let C be a context and $[T_1, \ldots T_n]$ a sequence of transformations. Define $\mathsf{Apply}(C, []) = C$ and:

$$\mathsf{Apply}(C, [T_1, \ldots, T_n]) = \mathsf{Apply}(C', [T_2, \ldots, T_n]),$$

with $C' = T_1.\mathit{Effect}(C)$ if $\mathit{Pre}(C)$ holds and $C' = C$ otherwise.

The following theorem, which follows by induction on the fact that individual transformations preserve results, shows that mismatches found by transformation-based testing do indeed point to compiler bugs:

Theorem 2.6. *Let P be a program, I an input such that P is well-defined with respect to I. Let $\vec{T}$ be a sequence of transformations with* $\mathsf{Apply}((P, I, \{\}), \vec{T}) = (P', I', F')$. *Suppose that* Impl *faults when applied to P' and I', or that* $\mathsf{Impl}(P, I) \neq \mathsf{Impl}(P', I')$. *Then* Impl *is not correct.*

2.3 Design Principles

Regarding the "almost for free" part of the paper title: whether the benefits of test-case reduction and deduplication associated with our approach work *well* depends on how carefully the transformations have been designed. We discuss three principles that should be followed.

Maximize independence between transformations. If a transformation T_j's precondition only holds after some other transformation T_i has been applied then any reduced test case that features T_j must also feature T_i. This is fine if T_i is designed to enable T_j, but otherwise it should be avoided as it can limit reduction quality.

SplitBlock (Table 1) violates this principle. If a program contains distinct instructions s and t then the operations of splitting a block before s vs. before t should be independent. But suppose s and t are in block b at offsets 3 and 5 respectively. Transformations $T_1 = \mathsf{SplitBlock}(b, 3, f_1)$ and $T_2 = \mathsf{SplitBlock}(f_1, 2, f_2)$, applied in order, have the effect of splitting before s and again before t, introducing new blocks f_1 and f_2 in the process. If these transformations are part of a bug-triggering sequence, and if the bug requires the split before t but *not* the split before s in order to trigger, reduction will be sub-optional. It will not be possible to eliminate T_1 (which splits before s) because T_1 introduces block f_1 and T_2 (which splits before t) requires f_1 to exist. A better design would involve SplitBlock taking an instruction identifier instead of a block and offset. The transformation could deduce in which block the instruction resides and split that block at the appropriate point. Our "basic blocks" language does not have syntax for uniquely identifying instructions—nor do most languages—but a transformation-based testing tool can introduce such identifiers in its internal representation.

Favor simple transformations. A conceptually-large transformation should be expressed as a sequence of smaller transformations if possible. This lets the reducer strip away excess transformations if only part of the conceptually-large transformation is required in order to trigger a bug.

The $\mathsf{AddDeadBlock}(b, f_1, f_2)$ transformation of Table 1 adds a statement $f_2 := \mathit{true}$ to an existing block *and* adds a conditional branch on f_2 to a new block f_1. If triggering some compiler bug only hinges on the new statement $f_2 := \mathit{true}$ being added then test-case reduction will be sub-optimal because the reducer will apply AddDeadBlock in its entirety. A better design would be for AddDeadBlock not to add the *true*-valued variable. Instead it could *require*—via a fact—the existence of a variable v for which $v = \mathit{true}$ holds at the end of block b. Another transformation, AddTrueVariable say, could be used to introduce *true*-valued variables and associated facts. The effect of the original AddDeadBlock could be achieved by applying AddTrueVariable followed by the simpler version of AddDeadBlock. This would allow the reducer to keep only the AddTrueVariable transformation if it turns out that adding the dead block is unnecessary.

As another example, $\mathsf{AddStore}(c, 0, s, i)$ in Figure 4 adds the store $s := i$ to the start of block c, where i is a program input. If this triggered a bug, a compiler developer might think that storing the value of a program input, specifically, is important. If in fact the stored value is irrelevant then using the statement $s := 0$ would make the test case slightly simpler. An alternative design would be for AddStore to always use 0 as the stored value, but to add a fact stating that this use of 0 is *irrelevant*—i.e. the use can be changed to any other value without affecting the program's output. Another transformation, ChangeIrrelevantValue, could then be employed to change the use of 0 to something more interesting (such as the input i). If it turns out that the simple value, 0, suffices to trigger a bug then the reducer will be able to eliminate the instance of ChangeIrrelevantValue.

Use the same type for similar transformations. Our approach to deduplication depends on transformations with different types having substantially different effects. The AddLoad and AddStore transformations of Table 1 have very similar effects: they both introduce a new assignment. This might lead to ineffective deduplication if a bug can be triggered by adding a particular form of assignment and if it is possible for this form to be achieved by either AddLoad or AddStore. Instead these could be captured by a single type, $\mathsf{AddAssignment}(b, o, x_1, x_2)$, with a precondition requiring either (a) x_1 is a fresh variable identifier, or (b) fact "b is dead" holds. AddAssignment behaves like AddLoad in case (a) and AddStore in case (b). Using a single type avoids duplicate bugs being suggested in this scenario.

3 Design and Implementation of spirv-fuzz

We have implemented the ideas of §2 in an open source tool, spirv-fuzz [34], targeting compilers for SPIR-V [17], an LLVM-like intermediate representation designed with GPU computing in mind, and used by the Vulkan and OpenCL programming models. After providing background on SPIR-V (§3.1) we describe the tool and its transformations (§3.2), discuss the design principles behind a selection of transformations (§3.3), and explain how we have implemented easy test-case reduction (§3.4) and deduplication (§3.5).

3.1 SPIR-V Background

A SPIR-V module comprises a series of instructions defining types, constants and global variables, followed by a number of functions, one of which is designated as an *entry point* from which execution will commence. Types include integer, boolean and floating-point scalars, vectors of these types, floating-point matrices, structures, arrays and pointers.

A function is made up of a series of *basic blocks*, with a designated entry block. Each block has a unique *result id* and contains a sequence of instructions ending with a *block terminator*: an unconditional or conditional branch to another block or blocks or a return instruction. Every instruction that generates a result also has a unique result id; i.e. SPIR-V uses static single-assignment form. Special ϕ instructions at the start of a block can be used to select a value based on the predecessor block from which control passes at runtime.

The syntactic order in which blocks appear in a function is irrelevant, except that the entry block must appear first, and a block must appear before all blocks that it dominates. An instruction i can only use the result id of an instruction j if j is *available* at i: it either appears earlier in the same block, or in a dominating block.

We focus on the Vulkan subset of SPIR-V, which includes support for writing *fragment shaders*: programs that will be invoked by thousands of GPU threads simultaneously to render an image. Roughly speaking, each invocation (called a *fragment*) is responsible for coloring a single pixel.

3.2 Overview of the Tool and Transformations

As input, spirv-fuzz takes a SPIR-V binary module, a file describing the inputs on which the module will be executed, and a directory containing a set of *donor* SPIR-V binaries that are used in the construction of certain transformations (discussed below). An initially empty set of facts is maintained. The module and facts are repeatedly modified by running *fuzzer passes*, each of which sweeps through the module looking for opportunities to apply a particular combination of transformations, probabilistically deciding which of these opportunities to take. When a fuzzer pass completes, the tool probabilistically decides whether to stop or apply another pass, definitely stopping if a limit of 2000 transformations is exceeded. Randomization is controlled by a seed passed to spirv-fuzz on the command line. At present spirv-fuzz only transforms the SPIR-V module, leaving the input unchanged.

We have integrated spirv-fuzz into gfauto, a framework for graphics shader compiler testing [8]. This supports running spirv-fuzz repeatedly with a large number of seeds, identifying crashes and miscompilations, and invoking the tool's reducer in such cases. As gfauto was originally designed for running the glsl-fuzz tool, it facilitates our comparison of spirv-fuzz and glsl-fuzz discussed in §4.

We now provide some detail on the facts that transformations can manipulate and the transformations themselves.

Facts. Transformations can establish and depend on the following kinds of facts, which we illustrate below:

- DeadBlock(b): Block b will never be executed
- Synonymous($u[\vec{i}], v[\vec{j}]$): $u[\vec{i}] = v[\vec{j}]$ holds at any program point where ids u and v are both available
- Irrelevant(i): The value of id i does not affect the end result of computation
- IrrelevantPointee(p): The value of data pointed to by pointer p does not affect the end result of computation
- LiveSafe(f): Calling function f from any program point does not affect the end result of computation as long as pointer arguments satisfy IrrelevantPointee

In the case of Synonymous facts, $\vec{i}$ and $\vec{j}$ denote possibly empty vectors of literal indices. For example, if a is a scalar and m a column-major matrix, then Synonymous($a, m[0, 1]$) means that the value at column 0, row 1 of m is equal to a.

Transformations. The tool features 85 types of transformation at time of writing, some of which we discuss below. The full set of transformations is formalized in a Protocol Buffers file [11] with descriptive comments in the spirv-fuzz source tree [34]. Transformations were inspired by prior work, discussions with SPIR-V compiler developers, and reading the SPIR-V specification carefully with an eye for interesting ways to combine its features.

Several *supporting transformations* add types, constants and variables to the module. They are not interesting in isolation, but fuzzer passes frequently use them to enable more interesting transformations.

Prior work has shown that transformations that affect control flow can be effective in uncovering bugs [7, 22], leading us to devise many such transformations. For example, *AddDeadBlock* creates dynamically-unreachable blocks with associated DeadBlock facts, and *ReplaceBranchWithKill* changes a dead block's terminator to a special *OpKill* instruction, which terminates a fragment; this substantially changes the static control-flow graph with no semantic impact.

A number of transformations support moving an instruction within its block or between blocks based on a conservative dependency analysis. We discuss a bug found using the *PropagateInstructionUp* transformation in §5.

Many transformations create Synonymous facts. For example, *CompositeConstruct* creates a composite value (e.g. of vector or struct type) from appropriate constituents, creating Synonymous facts relating each index of the composite to the constituent with which it was created. Similarly, *CompositeExtract* adds a SPIR-V instruction to extract from a composite value at a particular index, adding a Synonymous fact relating the result of this extraction to the component that was extracted. Other transformations create Irrelevant facts; e.g. *AddParameter* adds a new parameter to a function and updates all call sites to provide values for the new parameter. Because the values that are provided do not matter, they are recorded as being irrelevant.

A use of an id can be replaced with a known-to-be-equal id via *ReplaceIdWithSynonym*, which exploits Synonymous facts. A use of an id for which the Irrelevant fact holds can be replaced with *any* id of the right type by *ReplaceIrrelevantId*. The fact that spirv-fuzz knows the runtime values of the module's inputs (called *uniforms* in SPIR-V) is exploited by *ReplaceConstantWithUniform*, which can e.g. obfuscate from the compiler the fact that a block is dead by making the block's dynamic reachability depend on the value of an input.

AddFunction is a large transformation that adds a new function to the module. The fuzzer pass that creates this transformation uses the *donor* modules as a source of functions. Full details of a function are encoded in an *AddFunction* instance so that the donors are not required during reduction. *AddFunction* can be configured to make its function "live-safe" (documented via a LiveSafe fact) by clamping memory accesses to be in-bounds, truncating loops via an iteration limit, and eliminating uses of *OpKill*. A live-safe function can be called from anywhere without changing the results of computation, as long as IrrelevantPointee pointers are passed for any pointer arguments. The *FunctionCall* transformation adds calls to LiveSafe functions from anywhere, and to non-LiveSafe functions from dead blocks. IrrelevantPointee pointers come from new variables that the fuzzer introduces via an *AddVariable* transformation.

Function calls can be inlined via the *InlineFunction* transformation, which duplicates the blocks (and instructions) of a callee function and inserts them in place of a particular call to that function.

MoveBlockDown swaps a block with its syntactic successor if doing so respects the SPIR-V dominance rules. A *PermuteBlocks* fuzzer pass repeatedly applies *MoveBlockDown* to shuffle the blocks of functions in interesting ways. We discuss a bug found by *MoveBlockDown* in §5.

Using recommendations to drive fuzzing. Many of our transformations have the potential to interact, e.g. adding new functions creates opportunities for adding function calls, which in turn creates opportunities for inlining. To increase the likelihood of such interactions we implemented a *recommendations* strategy. For every fuzzer pass, we used our knowledge of the tool to manually select a (possibly empty) set of *follow-on* passes that may be worth running soon after the pass. When recommendations are enabled, spirv-fuzz maintains an initially empty *recommendation queue* of fuzzer passes. When deciding which fuzzer pass to run next, the tool chooses with uniform probability to select a pass at random or pop a pass from the queue (if non-empty). After running a pass p, spirv-fuzz pushes a random subset of follow-on passes for p to the queue. We evaluate the effectiveness of recommendations with respect to bug finding ability in §4.

3.3 Transformation Design

We give some examples of how we have followed the principles of §2.3 when designing transformations; similar approaches were followed for many other transformations.

Maximizing independence. Recall that the *InlineFunction* transformation duplicates the blocks and instructions of a called function. Fresh ids are required for this duplication. The simplest way to implement this would be to obtain fresh ids on-the-fly while applying the transformation, from the set of ids not yet in use. However, suppose we have a transformation sequence T_1 that adds instructions to a function f, T_2 that inlines f, and T_3 that manipulates some instruction in the inlined version of f. If T_3 depends on the existence of an instruction with a particular id then T_3's applicability is highly sensitive to the specific fresh ids used by T_2 when inlining f. Suppose that during reduction T_1 is eliminated. Applying T_2 now involves inlining a *smaller* function, requiring *fewer* fresh ids. The inline-expanded version of f will now feature different ids, and so T_3 may no longer apply. To avoid this problem, an *InlineFunction* instance is equipped with an explicit mapping from ids of the to-be-inlined function to fresh ids. During fuzzing it is easy to construct an appropriate mapping, and the mapping can then be used unchanged during reduction.

Favoring simple transformations. Instead of having a transformation that directly permutes function blocks, the *PermuteBlocks* fuzzer pass achieves a permutation by applying many instances of *MoveBlockDown*. If a permutation triggers a bug the reducer eliminates unnecessary *MoveBlockDown* instances to converge on a simpler permutation.

The *FunctionCall* transformation must provide parameters when adding a function call. Instead of choosing interesting ids for parameters, trivial constants are used (e.g. 0 for an integer parameter), and the IsIrrelevant fact is added for each of them. Subsequent fuzzer passes can then use *ReplaceIrrelevantId* to replace these simple parameters with more interesting expressions. This allows the reducer to simplify parameters back to 0 if doing so still triggers the bug.

Common types for related transformations. The tool features a *WrapRegionInSelection* transformation that can be applied in one of two forms: a region of blocks can be wrapped in the 'then' branch of a true conditional construct

or in the 'else' branch of a false conditional construct. Rather than having a separate type of transformation for each form, both are captured by the same transformation type. Test-case deduplication will thus regard reduced test cases that feature this transformation as similar, even if they use different forms of the transformation.

3.4 Test-case Reduction

On detecting a bug, gfauto generates an *interestingness test* for the bug: a script that takes a SPIR-V binary with its input values, runs it on the compiler under test, and returns true if and only if the bug appeared to be triggered. For crashes and internal errors, the interestingness test looks for a *crash signature* associated with the bug. This is automatically derived from the error message emitted by the compiler, the text associated with an assertion failure, or the stack trace that triggered the bug. We have fine-tuned a script for extracting useful crash signatures for a range of SPIR-V compilers [14].

For miscompilations, which manifest as an unexpected image being rendered when a SPIR-V binary is executed on the GPU, the interestingness test compares the pair of images rendered via the SPIR-V binary (passed to the script) and the original SPIR-V binary.

The spirv-fuzz *reducer* then uses delta debugging [40] to find a 1-minimal subsequence of transformations that still passes the interestingness test. The reduction algorithm maintains a *chunk size* c, initialized to $\lfloor n/2 \rfloor$ where n is the size of the initial transformation sequence. It divides the sequence into chunks of size c, starting from the last transformation and working backwards (the chunk of transformations at the start will be smaller than c if c does not divide n). It considers each chunk in turn, checking whether the interestingness test still passes when a chunk is removed, and eliminating the chunk if so. When no chunk of size c can be removed, c is halved. Reduction terminates when no chunk of size 1 can be removed.

Recall from §3.2 that instances of *AddFunction* encode entire functions. Sometimes these functions turn out to be larger than is necessary to trigger a bug, and *AddFunction* is the one transformation type that we found difficult to split into a smaller sequence of transformations. After delta debugging, the reducer applies spirv-reduce, a generic test case reducer for SPIR-V [20], to any remaining *AddFunction* transformations in an attempt to simplify their associated functions such that the interestingness test still passes. This use of spirv-reduce is merely an optimization, and without spirv-reduce, the delta between a minimally-reduced program and the same minimally-reduced program with one fewer transformation applied (as explained in §2.1) is still small even if functions are added, since adding a function is usually an enabler for subsequent transformations that actually trigger a bug. We also note that even though spirv-reduce is available, it does not provide the kind of features necessary for reduction in transformation-based testing or of miscompilations in general—it cannot revert transformations, and it does not preserve semantics.

Table 2. The SPIR-V targets we test, where **Version** denotes either a driver version, git revision, or device factory image

Target	Version	GPU type
AMD-LLPC	git-4781635	Discrete
Mesa	20.2.1	Integrated
Mesa-Old	19.1.0	Integrated
NVIDIA	440.100	Discrete
Pixel-5	RD1A.201105.003.C1	Mobile
Pixel-4	QD1A.190821.014.C2	Mobile
spirv-opt	git-02195a0	N/A
spirv-opt-old	git-2276e59	N/A
SwiftShader	git-b5bf826	Software

3.5 Test-case Deduplication

We have implemented the algorithm of Figure 6 in a Python script accompanying spirv-fuzz. We have refined the script to totally ignore a fixed list of transformation types: supporting transformations for adding types and constants, as well as *SplitBlock* and *AddFunction* (which are enablers for other transformations rather than being interesting in isolation) and *ReplaceIdWithSynonym* (which reaps the benefits of prior transformations but is not interesting in isolation). We fixed this list *before* commencing controlled experiments to evaluate test-case deduplication (see §4.3).

4 Controlled Experiments

We study the effectiveness of spirv-fuzz in practice, comparing it with glsl-fuzz [7] where appropriate; recall that glsl-fuzz operates on the OpenGL shading language but can be used to test SPIR-V compilers via cross compilation [8]. Our evaluation focuses on the following research questions:

RQ1 How effective is spirv-fuzz compared with glsl-fuzz in its ability to find defects in SPIR-V implementations?

RQ2 How effective is the "free" test-case reduction afforded by spirv-fuzz compared with the hand-crafted reducer used by glsl-fuzz?

RQ3 Is our heuristic for "free" deduplication of test cases that trigger bugs effective in practice?

Our experiments were performed using git revision 02195a0 of spirv-fuzz and 751148b of glsl-fuzz. When running the test case experiments of §4.2 we found a minor bug in the glsl-fuzz reducer, so switched to glsl-fuzz git revision a0f2f9a which contains a fix for this bug (the fix did not affect the fuzzing component of glsl-fuzz).

Targets. We use "latest" to mean "latest when we commenced our experiments". We conducted experiments on the SPIR-V targets summarized in Table 2. AMD-LLPC is the latest version of AMD's LLVM-based Pipeline Compiler (LLPC) [12]. Mesa is the most recent release of the Mesa 3D

drivers [9], targeting an Intel HD Graphics 620 GPU; Mesa-Old is an approx. 1-year older version targeting the same GPU. NVIDIA is the latest driver for the Quadro P1000 GPU rolled out to Linux workstations at the company where some of the authors work. Pixel-5 is a Pixel 5 Android phone with a Qualcomm Adreno 620 GPU, with the latest factory image. Pixel-4 is a Pixel 4 phone with an Adreno 640 GPU and 1-year old factory image. spirv-opt is the latest version of the SPIR-V optimizer [20]; spirv-opt-old is a version from October 2019. SwiftShader is the latest version of the SwiftShader software renderer [10]. We did not have access to an AMD GPU, and spirv-opt is not a full Vulkan implementation, so we could not use these targets to render images, but could still find crashes and internal errors. The targets of Table 2 cover a range of GPU types plus supporting tooling. Using both the latest and 1-year old versions of Mesa and spirv-opt, as well as new and older Pixel device images, provides access to a diverse range of compiler bugs.

References, donors and test execution. As references we used a set of 21 OpenGL ES shaders from the GraphicsFuzz project [13] that are known to produce numerically-stable images and are thus suitable for detecting miscompilations. As donors we used the full set of 43 shaders from GraphicsFuzz. To feed these shaders to spirv-fuzz we converted them to SPIR-V using the glslang front-end [18]. We also provided spirv-fuzz with an optimized version of each shader by running the spirv-opt tool with the `-O` argument (standard optimizations). We could not provide optimized shaders to glsl-fuzz because it does not accept SPIR-V code. Tests are executed by the gfauto framework (see §3.2). With glsl-fuzz, gfauto applies glslang to turn a generated OpenGL shader into SPIR-V. For both spirv-fuzz and glsl-fuzz, gfauto runs the test on the target of interest. If no bug is detected, gfauto applies spirv-opt with the `-O` argument, then runs the optimized test, again checking to see whether a bug is triggered.

4.1 Bug-Finding Ability (RQ1)

Table 3 presents data comparing the effectiveness of spirv-fuzz and glsl-fuzz. We also evaluated the effectiveness of the *recommendations* strategy used by spirv-fuzz and discussed in §3; we use spirv-fuzz-simple to refer to a configuration of spirv-fuzz where the recommendations strategy is disabled.

We used each of spirv-fuzz, spirv-fuzz-simple and glsl-fuzz to generate a set of 10,000 transformed shaders, with each shader obtained by running the respective tool configuration with a distinct random seed. For each tool configuration, Table 3 shows the total number of distinct *bug signatures* observed for each target when running all 10,000 tests. A bug signature is either a crash signature (see §3.4), or a special signature to indicate that the bug is a miscompilation. Because all miscompilations contribute the same bug signature, the results do not provide insight into how many different miscompilations the tools can detect.

To allow statistical analysis we separated each set of 10,000 tests into 10 disjoint sets of 1,000 tests. This allowed us to compute the number of distinct bug signatures observed for each target when running a particular set of 1,000 tests. For each target, Table 3 reports the median number of distinct bug signatures observed across the 10 disjoint subsets of size 1,000. The last two columns show the results of running the Mann-Whitney U (MWU) test [1] to compare the effectiveness of spirv-fuzz vs. each of spirv-fuzz-simple and glsl-fuzz, using the number of distinct bug signatures in each 1,000-sized test group as populations. Each percentage indicates the certainty with which spirv-fuzz is (or is not) more effective according to MWU.

In answer to **RQ1**, the results show that spirv-fuzz is more effective than glsl-fuzz at bug-finding: according to the MWU test this is the case with 99.98% confidence overall, and with $\geq$ 95% confidence for 6 out of 9 targets. The results comparing spirv-fuzz with spirv-fuzz-simple are less clear-cut: spirv-fuzz performs better with high confidence for 3 configurations, but results for other targets are not statistically significant, and MWU regards spirv-fuzz as superior overall with only 85% confidence.

Each segment in each Venn diagram of Figure 7 shows the number of bug signatures that were found by all the tool configurations associated with that segment at least once across the full set of 10,000 tests. For every configuration, spirv-fuzz is able to find at least one bug signature not found by glsl-fuzz or spirv-fuzz-simple. Still, there is complementarity between spirv-fuzz and glsl-fuzz, and room for improving the strategies used by spirv-fuzz: ideally the *recommendations* strategy would lead to strictly more bugs being found.

4.2 Quality of Test-Case Reduction (RQ2)

To investigate **RQ2**, we ran reductions for the AMD-LLPC, spirv-opt, spirv-opt-old and SwiftShader targets, capping the number of reductions per bug signature at 100. This was feasible because these targets do not require running tests on GPUs so we could run a very large number of reduction instances in parallel. This led to 932 reduced tests from spirv-fuzz and 245 reduced tests from glsl-fuzz—there are more reductions for spirv-fuzz because spirv-fuzz was able to find more bugs than glsl-fuzz, as discussed in §4.1.

Recall that a reduction results in a *reduced variant*, which is derived from an original program by applying a minimized sequence of transformations (see Figure 2). As the aim of this process is to yield a delta between the original and reduced variant that is easy for a developer to understand, a good measure of the success of reduction is the size of this delta. For each reduction, we computed the size of this delta as the difference between the number of instructions in the original SPIR-V module and the reduced variant SPIR-V module.

The median delta size across all reductions was 8 instructions for spirv-fuzz and 29 instructions for glsl-fuzz. This is not an entirely like-for-like comparison because the tools

Table 3. Comparing the bug-finding ability of spirv-fuzz, spirv-fuzz-simple and glsl-fuzz

	Distinct bug signatures						spirv-fuzz beats...	
Target	**spirv-fuzz**		**spirv-fuzz-simple**		**glsl-fuzz**		**spirv-fuzz-simple?**	**glsl-fuzz?**
	Total	Median	Total	Median	Total	Median	(% confidence)	(% confidence)
AMD-LLPC	6	3.0	4	2.5	6	2.5	Yes (83.80%)	Yes (57.26%)
Mesa	10	5.5	7	4.5	11	5.5	Yes (97.43%)	No (14.99%)
Mesa-Old	15	12.0	13	11.5	16	11.5	Yes (95.06%)	Yes (93.04%)
NVIDIA	29	13.0	30	14.0	9	5.0	No (77.35%)	Yes (99.98%)
Pixel-5	11	7.0	10	7.5	6	5.0	No (34.99%)	Yes (99.91%)
Pixel-4	10	7.5	10	8.0	7	4.5	No (45.47%)	Yes (99.97%)
spirv-opt	6	1.0	6	1.5	0	0.0	No (63.57%)	Yes (99.75%)
spirv-opt-old	15	7.0	9	5.5	1	0.0	Yes (95.87%)	Yes (99.98%)
SwiftShader	16	9.0	13	8.0	2	1.0	Yes (77.35%)	Yes (99.98%)
All	118	66.0	102	63.0	58	35.5	Yes (84.91%)	Yes (99.98%)

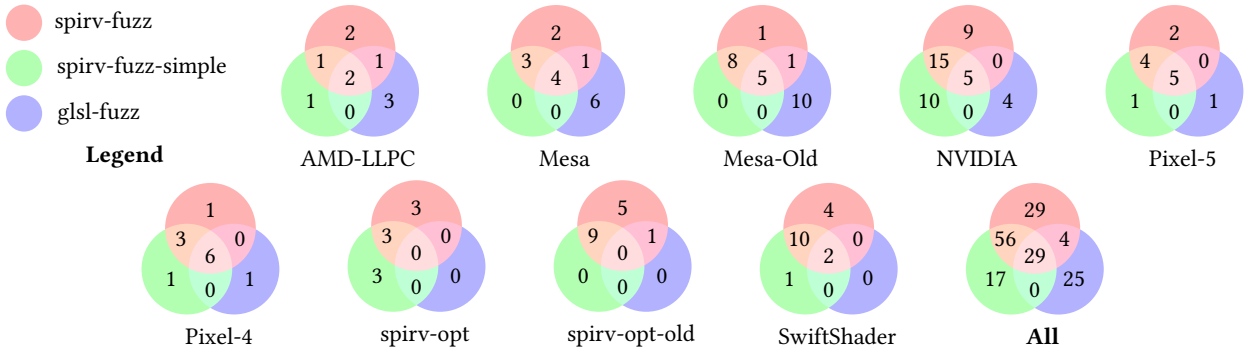


Figure 7. Complementarity of spirv-fuzz, spirv-fuzz-simple and glsl-fuzz with respect to bug-finding. Each number denotes the number of distinct bug signatures found by all tools represented by the associated part of the Venn diagram.

found different sets of bugs and the number of reductions available for analysis with spirv-fuzz is much larger than for glsl-fuzz. However, with respect to **RQ2**, the median sizes indicate that (a) both tools are effective at test-case reduction, since the size differences between original and *unreduced* variant modules are in the order of thousands or tens of thousands of instructions, and (b) the quality of "free" reduction that spirv-fuzz offers automatically due to its design appears to be somewhat superior to the quality of reduction offered by the hand-crafted glsl-fuzz reducer.

We found that spirv-fuzz reductions were a lot faster than glsl-fuzz reductions, but we do not dwell on this since the tools are implemented in different programming languages.

4.3 Effectiveness of Test-Case Deduplication (RQ3)

Investigating **RQ3** requires a baseline against which to compare the set of test cases that the deduplication algorithm predicts as triggering distinct bugs, i.e. a set of bugs that are *known* to be distinct, and a large corpus of reduced test cases that trigger these bugs, such that we know which bug each test case triggers. To obtain a suitable baseline we took the large set of reduced test cases gathered to assess reduction quality (§4.2), and also gathered a set of reduced test cases for all remaining targets, capping the maximum number of reductions per bug signature at 20 to allow experiments to complete in reasonable time. We were unable to gather this data for NVIDIA due to a driver bug that caused frequent machine freezes. We restricted attention solely to crash bugs (i.e., ignoring miscompilation bugs), because for crash bugs we can obtain reliable crash signatures as discussed in §3.4.

Table 4. The effectiveness of test-case deduplication

Target	Tests	Sigs	Reports	Distinct	Dups
AMD-LLPC	131	6	6	4	2
Mesa	60	8	6	6	0
Mesa-Old	202	13	9	7	2
Pixel-5	94	9	6	6	0
Pixel-4	96	8	9	6	3
spirv-opt	16	5	3	3	0
spirv-opt-old	511	15	6	5	1
SwiftShader	357	14	4	4	0
Total	1467	78	49	41	8

For each target we then ran the transformation-based deduplication algorithm on the set of associated reduced test cases. The results are summarised in Table 4. For each target, **Tests** shows the total number of reduced test cases on which

the deduplication algorithm was invoked and **Sigs** shows the number of distinct crash signatures we know that these test cases collectively exhibit. For example, for AMD-LLPC we ran deduplication on 131 reduced test cases, each of which triggers one of 6 distinct crashes. **Reports** shows the number of test cases that the deduplication algorithm suggests investigating and **Distinct** shows how many distinct crashes are actually triggered by the suggested test cases. In the ideal case the numbers for **Sigs**, **Reports** and **Distinct** would all be equal: this would mean that exactly one test case per bug signature was suggested, with no duplicates and no missed signatures. If **Distinct** is less than **Reports** then some of the suggested test cases trigger duplicate bugs; the number of duplicates is shown under **Dups**. Exemplifying again using AMD-LLPC, the algorithm suggests investigating 6 test cases that cover 4 of the 6 distinct AMD-LLPC bugs that the full set of test cases trigger, meaning that there are 2 duplicates.

The results show that the recommended tests tend to cover a reasonable number of distinct crash signatures with a low duplicate rate. Overall, 41 out of 78 distinct crash signatures are covered by at least one suggested test case (53% of crash signatures are covered), with 8 out of 49 suggested test cases turning out to be duplicates (16%). Per target, the crash signature coverage rate ranges from as low as 29% (SwiftShader) to as high as 67% (AMD-LLPC and Mesa), while the duplicate rate is zero for several targets (Mesa, Pixel-5, spirv-opt-old and SwiftShader) and as high as 33% (AMD-LLPC and Pixel-4). While the deduplication heuristic is far from perfect, these results show that it covers more than half of the distinct crash signatures with a reasonably low rate of duplicates.

5 Using spirv-fuzz in the Wild

Over the last 6 months we have used spirv-fuzz, at various stages of its development, to test a number of SPIR-V targets: the SPIR-V optimizer, spirv-opt, and validator, spirv-val, part of the SPIRV-Tools project that supports the Vulkan ecosystem [20]; the SwiftShader software renderer, which is widely-used as a reference implementation of Vulkan [10]; the Mesa [9] and LLPC [12] open source driver projects; and various Android devices including Pixel 4 and Pixel 5 phones.

So far we have reported 74 issues: 14 miscompilations, 49 crashes/internal errors, 7 cases where spirv-opt emits illegal SPIR-V (that spirv-val rejects), 3 cases where spirv-val rejects valid SPIR-V, and one SPIR-V specification issue. We have also submitted 34 new test cases to the Vulkan Conformance Test Suite (CTS) in response to a variety of these issues, which have been accepted [19]. This ensures that fixes to the associated bugs will be rolled out to devices (since all GPU vendors must pass CTS), and defends against them being re-introduced.

We provide details of two interesting miscompilation bugs.

Example Mesa bug. Figure 8a illustrates a miscompilation bug we reported to Mesa. The left control flow graph (CFG) represents a loop within a SPIR-V module; block c, the loop body, also contained load and store operations that we omit from the figure. The *PropagateInstructionUp* transformation was used to duplicate instruction $x_1 = i_1 \leq 100$ in block b into each of its predecessors, a and c, selecting between their results using a ϕ instruction, leading to the middle CFG. *PropagateInstructionUp* was then applied twice more to move the increment and comparison from block c into b, leading to the right CFG. The images rendered by the full SPIR-V modules containing the (unsimplified versions of these) CFGs are shown beneath each graph. The final transformation leads to a significantly different image. The optimization bug, fixed in response to our report, caused the last loop iteration to be skipped.

Example Pixel 5 bug. Figure 8b illustrates a bug that *MoveBlockDown* triggered in the Vulkan driver of a Pixel 5 phone (with a November 2020 factory image). The two CFGs shown in the figure are identical. Both top-to-bottom block orders shown in the figure are also valid, because in both cases each block appears before the blocks it dominates. However, the second ordering leads to holes in the image rendered by the Pixel 5 driver. The actual CFGs associated with the bug were larger than the ones we show, but the transformation still involved swapping a single pair of blocks.

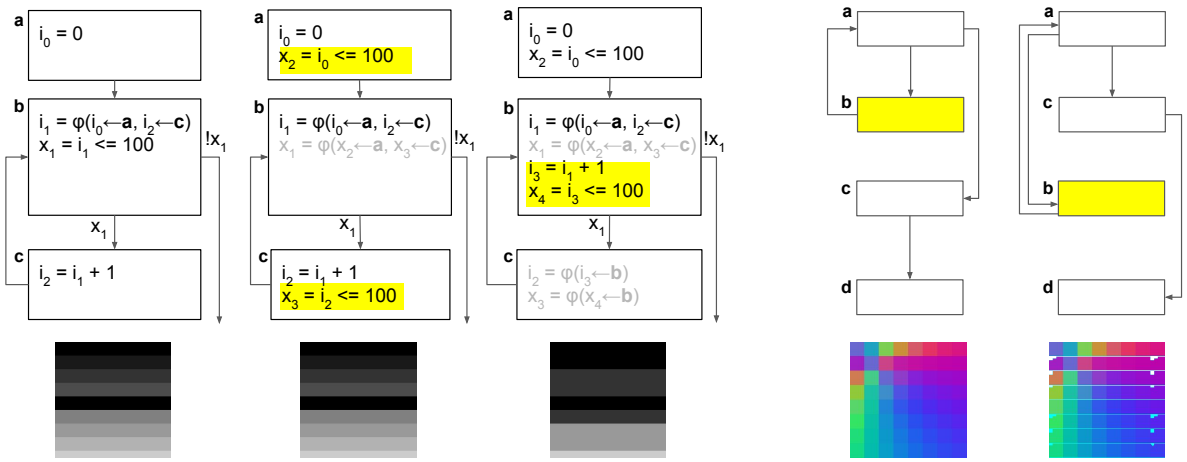
6 Related Work

Transformation-based testing is employed by Orion and the EMI family of tools [22, 23, 33] for testing C compilers, the CLsmith tool for testing OpenCL compilers [25], and the glsl-fuzz tool [7]. We believe the transformations employed by these tools could be readily expressed using the precondition and effect formalization (§2.2), providing test-case reduction benefits. The EMI tools rely on external reducers such as C-Reduce [32] to find a small well-defined program that exhibits a result mismatch between the compiler under test and an oracle compiler (usually CompCert [24]), and C-Reduce relies on the availability of undefined behavior sanitizers, which may be incomplete such that the final results of reduction are not always valid.

The glsl-fuzz reducer is able to revert transformations because transformations leave a trail of syntactic markers in the transformed program [7]. This requires keeping the fuzzer and reducer in sync, which has been a source of bugs [6, 8].

Our approach avoids the need for external reduction tools, oracle compilers, or sanitizers, and its correctness does not require maintaining a relationship between fuzzer and reducer. A downside is that if the original program is very large, the difference between it and a minimally transformed version may not provide an actionable bug report.

The Hypothesis property-based testing tool [30] uses *internal reduction* to avoid the need for an explicit external test case reducer [29]. Internal reduction takes the sequence of *choices* that the test case generator made, and searches for a



(a) Mesa compiler bug

(b) Pixel 5 compiler bug

Figure 8. Illustrations of miscompilations that affected SPIR-V compilers in Mesa and Pixel 5 drivers

shorter choice sequence that still leads to a bug-triggering input when fed to the generator. If applied in our domain, internal reduction might have the advantage of being able to find a small sequence of *different* transformations that trigger a bug, but will be incapable of finding a particular subsequence of original transformations if no path through the fuzzer would directly generate this sequence.

Chen et al. have shown that data sources such as compiler code coverage and metrics gathered from the compiler output, can be used to rank tests to prioritize a diverse set of bugs [5]. Looking at a prefix of the resulting bugs in priority order then provides a means of deduplication. The recent YARPGen tool [26] attempts to distinguish between bugs by determining the sequence of compiler optimizations that are required to trigger them, an instance of a long-standing idea [37]. These could be used to complement our transformation-based deduplication approach, which works in a black box manner, but only if coverage information and compiler metrics are available or if the compiler supports fine-grained optimization control. In the context of SPIR-V, usually neither are available from compilers embedded in deployed graphics drivers.

A metric for deduplicating historic compiler bugs is based on the notion of *correcting commits* [3]. If two test cases are successfully compiled by the latest version of a compiler, and incorrectly compiled by some older revision of the compiler, they can be deemed to trigger distinct underlying bugs if the compiler revision at which they start passing differs—i.e., the underlying bugs were corrected by two different commits. This measure is not useful for deduplicating test cases that trigger newly-found, unfixed bugs. However, it is an effective deduplication technique for historic bugs when a compiler's source code and revision history are available, and thus could be used to provide an alternative baseline against which to compare deduplication heuristics such as the one proposed in this paper.

A forthcoming paper on sources of inspiration for metamorphic relations uses spirv-fuzz as one of three case studies [21].

7 Conclusions and Future Work

We have described an approach to transformation-based compiler testing that provides test-case reduction and a heuristic for test-case deduplication out-of-the-box, and demonstrated its effectiveness via spirv-fuzz, the first compiler-testing tool targeting the SPIR-V language. Our experiments show that spirv-fuzz out-performs the glsl-fuzz tool with respect to bug-finding, the "almost free" reduction of spirv-fuzz is competitive with the hand-crafted reducer of glsl-fuzz, and the heuristic for test-case deduplication has merit.

It would be interesting to extend spirv-fuzz with transformations that modify both a SPIR-V module and its input in sync, for example changing struct padding and array sizes to find layout-related bugs, and to improve the *recommendations* strategy of the tool. There is scope for applying our approach to other programming languages, or to compiler intermediate representations such as LLVM IR and MLIR. We also believe that transformation-based testing approaches, with corresponding reduction and deduplication advantages, may be of interest beyond the domain of compilers.

Acknowledgments

We are grateful to Hugues Evrard, Vasileios Klimis, David Neto, John Wickerson, Matthew Windsor, the PLDI 2021 reviewers and the PLDI 2021 Artifact Evaluation Committee for insightful feedback on earlier drafts of this work. This work was supported by EPSRC projects EP/R011605/1 and EP/R006865/1, and two Google Summer of Code projects.

References

[1] On a Test of Whether One of Two Random Variables is Stochastically Larger Than The Other. 1947. H.B. Mann and D.R. Whitney. *Annals of Mathematical Statistics* 18 (1947), 50–60. https://doi.org/10.1214/aoms/1177730491

[2] Android Authority. 2019. Qualcomm will let you update GPU drivers via the Play Store. https://www.androidauthority.com/qualcomm-gpu-driver-updates-1063096/, last accessed 2021-04-02.

[3] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 180–190. https://doi.org/10.1145/2884781.2884878

[4] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1 (2020), 4:1–4:36. https://doi.org/10.1145/3363562

[5] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Z. Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 197–208. https://doi.org/10.1145/2491956.2462173

[6] Alastair Donaldson. 2019. GraphicsFuzz pull request: Fix issue where the reducer was replacing a dead code injection with its body. https://github.com/google/graphicsfuzz/pull/599, last accessed 2021-04-02.

[7] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *PACMPL* 1, OOPSLA (2017), 93:1–93:29. https://doi.org/10.1145/3133917

[8] Alastair F. Donaldson, Hugues Evrard, and Paul Thomson. 2020. Putting Randomized Compiler Testing into Production (Experience Report), See [16], 22:1–22:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.22

[9] Freedesktop.org. 2021. The Mesa 3D Graphics Library. https://www.mesa3d.org/, last accessed 2021-04-02.

[10] Google. 2020. SwiftShader GitHub repository. https://github.com/google/SwiftShader, last accessed 2021-04-02.

[11] Google. 2021. Protocol Buffers. https://developers.google.com/protocol-buffers, last accessed 2021-04-02.

[12] GPUOpen Drivers. 2020. LLVM-Based Pipeline Compiler GitHub repository. https://github.com/GPUOpen-Drivers/llpc, last accessed 2021-04-02.

[13] GraphicsFuzz project authors. 2021. GraphicsFuzz. https://github.com/google/graphicsfuzz, last accessed 2021-04-02.

[14] GraphicsFuzz project authors. 2021. Script for extracting crash signatures. https://github.com/google/graphicsfuzz/blob/master/gfauto/gfauto/signature_util.py, last accessed 2021-04-02.

[15] The Khronos Vulkan Working Group. 2019. *Vulkan 1.1.141 - A Specification (with all registered Vulkan extensions).* The Khronos Group. https://www.khronos.org/registry/vulkan/specs/1.1-extensions/pdf/vkspec.pdf, last accessed 2021-04-02.

[16] Robert Hirschfeld and Tobias Pape (Eds.). 2020. *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference).* LIPIcs, Vol. 166. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[17] John Kessenich, Boaz Ouriel, and Raun Krisch (Eds.). 2019. *SPIR-V Specification, Version 1.5, Revision 2, Unified.* The Khronos Group. https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.pdf, last accessed 2021-04-02.

[18] Khronos Group. 2020. glslang GitHub repository. https://github.com/KhronosGroup/glslang, last accessed 2021-04-02.

[19] Khronos Group. 2020. Khronos Vulkan, OpenGL, and OpenGL ES Conformance Tests GitHub repository. https://github.com/KhronosGroup/VK-GL-CTS, last accessed 2021-04-02.

[20] Khronos Group. 2020. SPIR-V Tools GitHub repository. https://github.com/KhronosGroup/SPIRV-Tools, last accessed 2021-04-02.

[21] Andrei Lascu, Matt Windsor, Alastair F. Donaldson, Tobias Grosser, and John Wickerson. 2021. Dreaming up Metamorphic Relations: Experiences from Three Fuzzer Tools. In *Proceedings of the 1st International Workshop on Metamorphic Testing, MET@ICSE 2021, Online, June 2, 2021.* To appear.

[22] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 216–226. https://doi.org/10.1145/2594291.2594334

[23] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 386–399. https://doi.org/10.1145/2814270.2814319

[24] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43, 4 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

[25] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 65–76. https://doi.org/10.1145/2737924.2737986

[26] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 196:1–196:25. https://doi.org/10.1145/3428264

[27] Martin Liška. 2021. C-Vise. https://github.com/marxin/cvise, last accessed 2021-04-02.

[28] LLVM Project. 2021. libFuzzer – a library for coverage-guided fuzz testing. http://llvm.org/docs/LibFuzzer.html, last accessed 2021-04-02.

[29] David Maciver and Alastair F. Donaldson. 2020. Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer (Tool Insights Paper), See [16], 13:1–13:27. https://doi.org/10.4230/LIPIcs.ECOOP.2020.13

[30] David R. MacIver, Zac Hatfield-Dodds, and Many Other Contributors. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019), 1891. https://doi.org/10.21105/joss.01891

[31] John Regehr. 2020. Responsible and Effective Bugfinding. https://blog.regehr.org/archives/2037, last accessed 2021-04-02.

[32] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 335–346. https://doi.org/10.1145/2254064.2254104

[33] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 849–863. https://doi.org/10.1145/2983990.2984038

[34] The Khronos Group. 2021. spirv-fuzz source code. https://github.com/KhronosGroup/SPIRV-Tools#fuzzer, last accessed 2021-04-02.

[35] The LLVM compiler infrastructure. 2021. Clang Static Analyzer. https://clang-analyzer.llvm.org/, last accessed 2021-04-02.

[36] The LLVM compiler infrastructure. 2021. How to submit an LLVM bug report. https://llvm.org/docs/HowToSubmitABug.html, last accessed 2021-04-02.

[37] David B. Whalley. 1994. Automatic Isolation of Compiler Errors. *ACM Trans. Program. Lang. Syst.* 16, 5 (1994), 1648–1659. https://doi.org/10.1145/186025.186103

[38] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. https://doi.org/10.1145/1993498.1993532

[39] Michal Zalewski. 2014. Technical "whitepaper" for afl-fuzz. https://lcamtuf.coredump.cx/afl/technical_details.txt, last accessed 2021-04-02.

[40] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200. https://doi.org/10.1109/32.988498

[41] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schröter, and Cathrin Weiss. 2010. What Makes a Good Bug Report? *IEEE Trans. Software Eng.* 36, 5 (2010), 618–643. https://doi.org/10.1109/TSE.2010.63