Putting Randomized Compiler Testing into Production

Alastair F. Donaldson
Google, UK
Imperial College London, UK
afdx@google.com

Hugues Evrard
Google, UK
hevrard@google.com

Paul Thomson
Google, UK
paulthomson@google.com

Abstract

We describe our experience over the last 18 months on a compiler testing technology transfer project: taking the GraphicsFuzz research project on randomized metamorphic testing of graphics shader compilers, and building the necessary tooling around it to provide a highly automated process for improving the Khronos Vulkan Conformance Test Suite (CTS) with test cases that expose fuzzer-found compiler bugs, or that plug gaps in test coverage. We present this tooling for test automation—gfauto—in detail, as well as our use of differential coverage and test case reduction as a method for automatically synthesizing tests that fill coverage gaps. We explain the value that GraphicsFuzz has provided in automatically testing the ecosystem of tools for transforming, optimizing and validating Vulkan shaders, and the challenges faced when testing a tool ecosystem rather than a single tool. We discuss practical issues associated with putting automated metamorphic testing into production, related to test case validity, bug de-duplication and floating-point precision, and and provide illustrative examples of bugs found during our work.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Software testing and debugging

Keywords and phrases Compilers, metamorphic testing, 3D graphics, experience report

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.22

Category Experience Report

1 Introduction

Graphics processing units (GPUs) provide hardware-accelerated graphics in many scenarios, such as 3D and 2D games, applications, web browsers, and operating system user interfaces. To utilize GPUs, developers must use a graphics programming API, such as OpenGL, Direct3D, Metal or Vulkan, and write *shader programs* that execute on the GPU in an embarrassingly-parallel manner. Shaders are written in a *shading language* such as GLSL, HLSL, MetalSL, or SPIR-V (associated with the OpenGL, Direct3D, Metal and Vulkan APIs, respectively), and are usually portable enough to run on many different GPU models. A graphics driver contains one or more *shader compilers* to translate shaders from portable shading languages to machine code specific to the system's GPU.

Functional defects in graphics shader compilers can have serious consequences. Clearly, as with any bug in any application, it is undesirable if a mis-compiled graphics shader causes unintended visual effects. Furthermore, since shaders are compiled at *runtime* (because the GPU and driver that will be present when an application executes is not known at the

application's compile time), a shader compiler crash can lead to an overall application crash. Additionally, developers cannot feasibly test for or workaround these issues, as the driver version that crashes may not have even been written at the time of application development. Worse still, because the graphics driver usually drives the whole system's display, if a shader compiler defect leads to the state of the driver being corrupted, the entire system may become unstable. This can lead to device freezes and reboots, display corruption and information leakage; see [15] for a discussion of some examples, including information leak bugs in iOS [2] (CVE-2017-2424) and Chrome [5] caused by GPU driver bugs, and an NVIDIA machine freeze [38] (CVE-2017-6259).

One way to provide a degree of graphics driver quality—and shader compiler quality in particular—is via standardized test suites. An example is the Khronos Vulkan Conformance Test Suite (Vulkan CTS, or just CTS for short) [25]. This is a large set of functional tests for implementations of the Vulkan graphics API. The Khronos Group, who define various standards and APIs including Vulkan, requires a Vulkan implementation (such as a GPU driver and its associated GPU hardware) to demonstrate that they pass the Vulkan CTS tests in order for the vendor to use the official Vulkan branding. Google's Android Compatibility Test Suite incorporates the Vulkan CTS, so that Android devices that provide Vulkan capabilities must include drivers that pass the Vulkan CTS. Improving the quality and thoroughness of the Vulkan CTS is thus an indirect method for improving the quality of Vulkan graphics drivers in general, and on Android in particular.

GraphicsFuzz (originally called GLFuzz) [16, 15] is a technique and tool chain for automatically finding crash and miscompilation bugs in shader compilers using metamorphic testing [9, 42]. Whenever GraphicsFuzz synthesizes a test that exposes a bug in a conformant Vulkan driver, this demonstrates a gap in the Vulkan CTS: the driver has passed the conformance test suite despite exhibiting this bug. If GraphicsFuzz synthesizes a test that covers a part of a conformant driver's source code, but the driver does not crash, and the code is not covered by any existing CTS tests, then this also exposes a CTS gap (albeit arguably a less severe one): it demonstrates that part of the driver's source code *can* be covered but is *not* covered by the CTS; bugs that creep into such code in the future would not be caught.

In this experience report we describe our activities at Google over the last 18 months putting the GraphicsFuzz tool chain into production, with the aim of improving implementations of the Vulkan API. We have set up a process whereby the randomized metamorphic testing capabilities of GraphicsFuzz are used to find tests that expose driver bugs or CTS coverage gaps, shrink such tests down to small examples that are simple enough for humans to read and debug, and package the resulting tests into a form whereby they are almost completely ready to be added to the Vulkan CTS. So far, this has led to 122 tests that exposed driver and tooling bugs and 113 that exposed driver coverage gaps being added to CTS. The bugs affect a range of mobile and desktop drivers, as well as tools in the SPIR-V ecosystem. Our contribution of CTS tests that expose them means that future conformant Vulkan drivers cannot exhibit them (at least not in a form that causes these tests to fail).

We start by presenting relevant background on graphics programming APIs, shader processing tools, the Vulkan CTS, and the GraphicsFuzz testing approach (§2). We then describe how we set up a pathway for incorporating tests that expose bugs found by GraphicsFuzz into the CTS, and various practical issues we had to solve to ensure valid tests (§3). With this pathway in place we were empowered to build a fuzzing framework, gfauto, for running GraphicsFuzz against a range of drivers and shader processing tools, automatically shrinking tests that find bugs and getting them into a near-CTS-ready form (§4). To aid in finding coverage gaps, we have built tooling for *differential* coverage analysis; we describe

how—by treating coverage gaps as bugs—gfauto can be used to synthesize tests that expose such gaps in a highly automatic fashion (§5). A strength of GraphicsFuzz is that it facilitates testing not only vendor graphics drivers, but also a variety of translation, optimization and validation tools that are part of the Vulkan ecosystem. We explain how this also presents a challenge: it can be difficult to determine which component of the ecosystem is responsible for a bug (§6). Throughout, we provide illustrative examples of noteworthy bugs and tests found and generated by our approach, including bugs that affect core infrastructure (such as LLVM), bugs that affect multiple tools simultaneously, and bugs for which the responsible tool is non-trivial to identify. We conclude by discussing related (§7) and future (§8) work.

Main takeaways We hope this report is simply interesting for researchers and practitioners to read as an example of successful technology transfer of research ideas to industrial practice. In addition, we believe the following aspects could provide inspiration for follow-on research:

- The pros and cons of fuzzing a low level language via a program generator for a higher level language and a suite of translation and optimization tools, including the problem of how to determine *where* in a tool chain a fault has occurred (§3.1 and §6);
- The need for image differencing algorithms that are well-suited to tolerating the degree of variation we expect from graphics shaders due to floating-point precision (§3.4);
- Threats to test validity caused by undefined behavior, long-running loops and floating-point precision, where more advanced program analyses have the potential to be applied (§3.5);
- The difficulty of correctly maintaining a test case generator and a corresponding test case reducer, especially when test case reduction needs to be semantics-preserving (also §3.5)
- The challenge of de-duplicating bugs that do not exhibit distinguished characteristics, such as wrong image bugs and message-free compile and link errors (§4.2);
- The idea of using differential coverage analysis and test case reduction to fill coverage gaps (§5), and the challenge of going beyond synthesizing tests that trivially cover new code to tests that are also equipped with meaningful oracles (§5.4 specifically).

Open sourcing Our extensions to the GraphicsFuzz tool, the new gfauto tool, and our infrastructure for differential code coverage, are open source.[1] The tests we have contributed to Vulkan CTS are also open source.[2]

2 Background

2.1 The GLSL and SPIR-V Shading Languages

GLSL The OpenGL Shading Language (GLSL) [22] is the main shading language in the OpenGL graphics API [41] (analogous to HLSL and the Direct3D API). It is used for rendering hardware-accelerated 2D and 3D graphics. OpenGL ES [32], and its associated shading language GLSL ES [43], is a subset of the OpenGL API supported by mobile devices, including Android devices.[3] We focus on the GLSL ES shading language version 3.10 and later, and henceforth drop the ES suffix and version number for brevity. Figure 1 shows an

[1] https://github.com/google/graphicsfuzz
[2] https://github.com/KhronosGroup/VK-GL-CTS/tree/master/external/vulkancts/data/vulkan/amber/graphicsfuzz
[3] Strictly, OpenGL ES is not quite a subset of OpenGL: over time it has evolved with some features that have been deemed specifically important for mobile platforms.

```
precision highp float;

layout(location = 0) out vec4 _GLF_color;

void main()
{
 vec2 a = vec2(1.0);
 vec4 b = vec4(1.0);
 pow(vec4(a, vec2(1.0)), b);
 // Added manually to ensure that the shader writes red
 _GLF_color = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Figure 1 A reduced GLSL ES fragment shader that, after translation to SPIR-V, triggered a bug leading to a crash in the GPU shader compiler for a popular Android device.

example of a GLSL *fragment* shader (also known as a pixel shader in Direct3D). The code is C-like, but with some additional features useful for graphics programming. The code in `main` is conceptually executed n times on the GPU for each of the n pixels rendered into a *framebuffer* (which stores the image) using the shader. The `precision highp float;` line causes all subsequent floating-point values to be represented with 32 bits of precision by default (lower precision can be specified via the `mediump` and `lowp` qualifiers on a per-variable basis). Note that `main` has no parameters and a `void` return type; in GLSL, inputs and outputs to the shader are instead expressed using special global variables. Global variable `_GLF_color`[4] is an output variable into which the fragment shader writes the RGBA colour value that will be rendered at the pixel coordinate for which the shader is running. The `vec2` and `vec4` types are built-in float vector types (with 2 and 4 float components respectively). The vector constructor form that takes one floating-point value (e.g. `vec2(1.0)`) creates a vector with all components set to that value. A vector constructor can also take a combination of vectors and/or scalars (e.g. `vec4(a, vec2 (1.0))`) to construct a vector made up of each component in order, as long as the total number of components matches the vector type. The `pow(x,y)` function yields an approximation of x^y, and is an example of one of the many built-in math functions provided in GLSL. The example of Figure 1 was minimized with the aim of reproducing a shader compiler crash bug (discussed further as Example 1 below), and is not representative of a practically useful graphics shader: the `main` function performs some redundant computation and then writes the colour red (`vec4(1.0, 0.0, 0.0, 1.0)`) to the output colour variable. Thus, every pixel rendered by this shader will be red.

Shaders can also define *uniform* global variables (not shown in the example), using the `uniform` keyword. These are shader inputs that yield the same value for every pixel being shaded during a single shader invocation. For example, a uniform declaration `uniform float time;` could be used to pass a representation of the current time into a shader, allowing it to produce a time-varying visual effect.

SPIR-V In comparison to OpenGL, Vulkan [20] is a newer, lower-level graphics API. It is widely supported by modern desktop GPUs, as well as being available on newer Android devices. Standard, Portable Intermediate Representation - V (SPIR-V; the "V" does not stand for anything) is the Vulkan shading language [23]. Unlike GLSL, SPIR-V was designed as an intermediate representation to be stored in a binary form, and thus is not usually written

[4] The `_GLF` prefix comes from the fact that the tool was originally called GLFuzz. This prefix is used as a default for any special variable, function or macro names used by GraphicsFuzz.

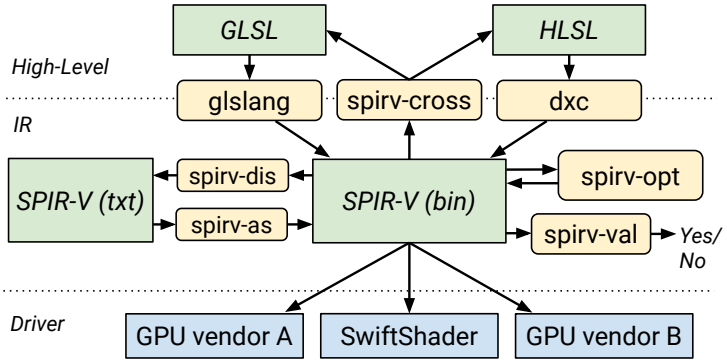

Figure 2 Diagram showing the various tools in the SPIR-V ecosystem and how they interact.

directly by programmers. Instead, programmers write their shaders in a higher-level language like GLSL or HLSL, and use a tool to compile the shaders into SPIR-V. SPIR-V modules use static single assignment (SSA) form [12], including the use of Phi instructions [12], and functions contain blocks with branches.

2.2 The SPIR-V Tooling Ecosystem

Figure 2 summarizes various open source tools for analyzing and transforming SPIR-V shaders, and translating to and from SPIR-V.

As mentioned in §2.1, most shaders are written in high level languages such as GLSL and HLSL and translated to SPIR-V. For example, glslang [24] and DXC [37] can compile GLSL and HLSL, respectively, to SPIR-V. A binary SPIR-V shader can be loaded by the Vulkan API and executed as part of a graphics pipeline on a GPU device. Google provides a software implementation of Vulkan, SwiftShader [18], which allows Vulkan applications (including their SPIR-V shaders) to be executed in the absence of Vulkan-capable hardware. This is useful to bring Vulkan support to old devices, as a fall-back renderer if a GPU driver goes into an unstable state, and as a "second opinion" for GPU driver writers.

The code generated by front-ends such as glslang and DXC is not typically optimized. In fact glslang deliberately performs as straightforward a syntax-directed translation of a GLSL shader as possible. The spirv-opt tool, part of the Khronos SPIRV-Tools framework [27], implements many target-agnostic optimizations as SPIRV-V-to-SPIR-V passes.

The philosophy of the Vulkan API is to allow drivers to assume that the Vulkan workloads with which they are presented are valid, pushing the onus of validation to the application. In support of this, the spirv-val tool (also part of the SPIR-V tools framework), checks whether a SPIR-V shader obeys the (many) rules mandated by the SPIR-V specification [23]. The spirv-dis and spirv-as disassembler and assembler (again, part of SPIRV-Tools) allow a shader to be translated into text format and back, which is useful for debugging.

Finally, the spirv-cross tool [28] allows SPIR-V to be translated into various shading languages including GLSL, HLSL and Apple's Metal shading language (MetalSL, not shown in the figure). Translation to these higher-level languages can help in understanding the intended behavior of a SPIR-V shader, and the SPIR-V-to-MetalSL pathway is used by the MoltenVK project, which provides an implementation of most of Vulkan on top of Apple's Metal graphics API [26].

2.3 The Vulkan Conformance Test Suite

The Khronos Vulkan Conformance Test Suite (Vulkan CTS) [25] is a set of tests for the Vulkan API. In theory, every part of the Vulkan specification should have one or more corresponding tests in the Vulkan CTS. Each test should invoke the relevant Vulkan API functions to check that a Vulkan implementation conforms to the Vulkan specification. Indeed, the Vulkan CTS mostly consists of a set of functional tests (there are over 550,000 Vulkan CTS tests at the time of writing) that attempt to test features in isolation. The Vulkan CTS is part of the larger Khronos Conformance Test Suite called dEQP (drawElements Quality Program[5]) that additionally contains tests for OpenGL ES and EGL.

Any implementation of Vulkan (including any Vulkan graphics driver with its associated GPUs) must pass the Vulkan CTS (and upload the results to Khronos for peer review) before the Vulkan name or logo can be used in association with the implementation. Thus, the Vulkan CTS sets a minimum quality standard for every conformant Vulkan implementation. Of course, the test suite is also extremely useful during development of a Vulkan driver; as with most test suites, it can be used to identify bugs and regressions, and to measure progress towards becoming a conformant implementation. The OpenGL ES and EGL test suite is similarly used as part of the conformance process for those APIs, and as a useful aid during driver development. The dEQP test suite is included in the Android Compatibility Test Suite (Android CTS), which is an even larger test suite for Android devices. Original equipment manufacturers (OEMs) will typically customize the Android OS for a given device, but these Android implementations must still pass the Android CTS to be deemed "compatible". Thus, the Vulkan CTS also sets a minimum quality standard for Vulkan on every compatible Android device, which can have a large impact on the Android ecosystem.

Vulkan CTS development is mostly done by Khronos members, although anybody can contribute. New tests are reviewed by GPU vendors before being accepted. Tests need to be deterministic, and clear enough to allow debugging of Vulkan implementations if a test fails.

2.4 Metamorphic Compiler Testing Using GraphicsFuzz

The GraphicsFuzz tool originated from a research project at Imperial College London, and formed the basis of a spin-out company, GraphicsFuzz Ltd., founded by the authors of this paper, which Google acquired during 2018.

Figure 3 gives an overview of the GraphicsFuzz approach to testing shader compilers. It starts with an existing reference shader which, after being compiled by the shader compiler embedded in the GPU driver and executed on the GPU hardware, leads to a given reference image. A classic way to test a GPU driver would be to compare this resulting image to what a reference implementation of the graphics API would produce. However, graphics API are purposefully relaxed to let GPU vendors reach very high performance through aggressive optimizations, such that there are various images that can be deemed acceptable. It is currently not possible, and not desirable for GPU vendors, to agree on a strict reference implementation that would serve as a test oracle.

Inspired by the *equivalence modulo inputs* method for testing C compilers [31], GraphicsFuzz works around this lack of oracle by using *metamorphic testing* [9, 42]: here the GPU input (shader) is transformed in a way that should not change its output (image). In practice, the glsl-fuzz tool applies *semantics-preserving transformations* to the reference shader source code

[5] dEQP was a commercial product developed by the drawElements company. Google acquired drawElements in 2014 and donated the dEQP test suite to Khronos, where it is now open source.

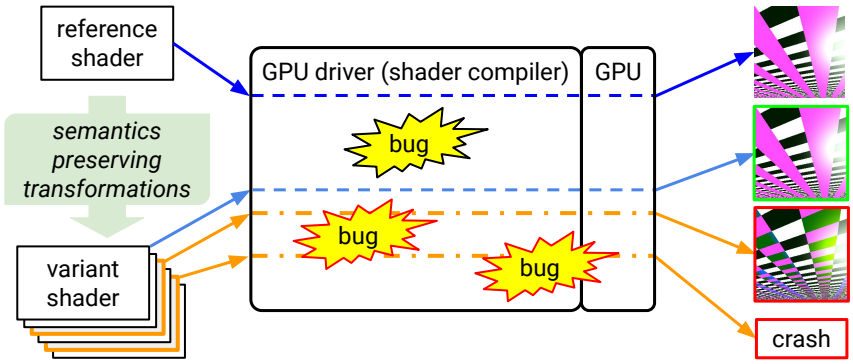

Figure 3 Illustration of the metamorphic testing approach used by GraphicsFuzz

to obtain a family of variant shaders. As a very simple example, one can add zero to an existing integer operation, `int x = y + 0`: this source code change should not impact the program behavior. The glsl-fuzz tool contains many such semantics-preserving transformations [15], including: arithmetic operations (such as adding zero, multiplying by one, etc), boolean operations (e.g. `bool b = x && true`), dead code injection (adding valid yet unreachable code, e.g. wrapped inside an `if (false) { ... }`), live code injection (adding code that will be executed while making sure to save and restore all variables affected by it, e.g. `{t = x; x = foo(x); x = t;}`), control flow wrapping (e.g. wrapping existing code in a single-iteration loop `do { ... } while(false)`), packing scalar data into composite data types (such as structs and vectors), and outlining expressions into functions. Some of the values used in these transformations, such as zero, one, true and false, are obtained via program inputs whose value is guaranteed at execution time, but unknown at compilation time. This is to make sure compilers cannot trivially remove some transformations, e.g. by statically detecting dead code injections to be unreachable.

Care is required when applying these transformations to ensure that program semantics are preserved. For instance, one cannot wrap some code in a single-iteration loop if this code contains a top-level `break` statement: the `break` would now apply to the newly-introduced loop, rather than to the original loop or switch statement in which it originally appeared.

Each variant shader is syntactically distinct from the reference, yet has the same semantics (modulo floating-point error). It may thus exercise a different path in the shader compiler but should still lead to a visually similar image being rendered, so long as the reference shader is sufficiently numerically stable. This is illustrated by the top, blue variant shader line in Figure 3. However, some variants may lead to significantly different images, or a driver crash, which are symptoms of bugs, most likely in the shader compiler but also potentially in other parts of the driver or GPU hardware. These are illustrated by the lower two orange variant shader lines in Figure 3. For a given GPU, we cannot know what to expect as a reference image, but we do expect variants to lead to an extremely similar image.

Semantics-preserving transformations are used in other contexts, e.g. compiler optimizations and code obfuscation tools modifying a program representation while keeping the same behavior. Code refactoring, when understood as improving the program structure while keeping the same functional features, can also be considered as a semantics-preserving transformation at a bigger scale than the transformations used in glsl-fuzz. For our testing purpose, we are interested in any kind of semantics-preserving transformation that may potentially have interesting effects on how the shader is processed by the GPU.

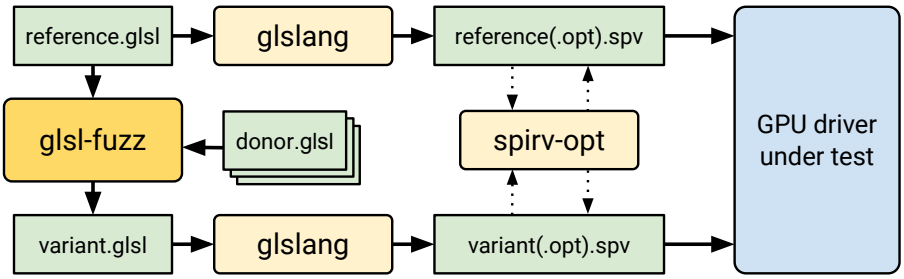

Figure 4 Targeting SPIR-V shader compilers from GLSL.

Although a bug-inducing variant can be used as a starting point for debugging, its source code is often barely understandable by a human because of the hundreds of transformations that have been applied to it. To ease debugging, the glsl-reduce tool progressively shrinks the variant source code while making sure that the bug is still triggered.

There are two reduction modes:

Semantics-preserving reduction For shader miscompilation bugs leading to wrong images, glsl-reduce performs *semantics-preserving reduction* by removing the glsl-fuzz transformations in a way that still preserves semantics. This typically leads to a variant that differs from its reference only by a handful of transformations necessary to trigger the wrong image bug. The pair of semantically identical shaders is useful as a debugging starting point.

Semantics-changing reduction For bugs leading to a driver crash, glsl-reduce performs *semantics-changing reduction* by removing source code, the only requirement being to keep it statically valid (which includes being syntactically valid and well-typed). No valid shader should not cause a driver crash, so there is no need to keep a semantic equivalence with the reference shader. Semantics-changing reductions can lead to very short crash-inducing shaders (e.g. Figure 1), which are useful for debugging and as regression test cases.

3 Integrating GraphicsFuzz Tests With Vulkan CTS

As described in §2.4, the GraphicsFuzz tool was originally designed to find bugs in OpenGL and OpenGL ES drivers by transforming shaders written in the GLSL shading language. However, our interest is in making shader compilers for the more modern Vulkan API as reliable as possible by improving the Vulkan CTS, and Vulkan uses SPIR-V as its shading language (see §2.1). We explain the process we used to allow GLSL-based fuzzing of SPIR-V shader compilers via translation (§3.1). We explain why we did *not* opt for embedding the fuzzer inside CTS, or directly contributing large numbers of fuzzer-generated tests, instead preferring to add tests that are known to expose shader compiler bugs (§3.2). We then describe how we paved the way for tests that expose crash and wrong image bugs to be added to CTS (§3.3 and §3.4, respectively).

3.1 Fuzzing SPIR-V Compilers via GLSL Shaders

In order to target SPIR-V shader compilers with a tool that operates on GLSL, we leverage the glslang translator, which takes GLSL as input and has a SPIR-V back-end. By design, glslang performs a very straightforward translation from GLSL to SPIR-V, performing no optimization beyond some basic constant folding and elimination of functions that are never invoked. As a result, the SPIR-V that glslang emits is rather basic (e.g. it rarely exhibits uses

of Phi instructions). While it is vital that SPIR-V shader compilers correctly handle this "vanilla" SPIR-V, we are also interested in testing their support for more interesting SPIR-V features. Towards this aim, we optionally invoke the spirv-opt tool on the SPIR-V that glslang generates, with its `-O` flag (optimize for speed), its `-Os` flag (optimize for size), or a random selection of its finer-grained flags (which include things like `--ssa-rewrite`, which changes variable uses to register uses, and can add Phi instructions, and `--eliminate-dead-inserts`, which avoids unnecessary insertions of data into composite structures).

This use of glslang and spirv-opt allows us to perform metamorphic fuzzing at the GLSL level to generate a variant from a reference, send both the variant and reference through glslang to turn them into SPIR-V, and then (optionally, and at random) transform the variant using a configuration of spirv-opt. The resulting SPIR-V shaders can then be compiled and executed on a Vulkan driver, and the results they compute can be compared. This process is illustrated graphically in Figure 4. This translation-based approach allows us to also find bugs in glslang and spirv-opt, which benefits the Vulkan ecosystem. However, as discussed further in §6, it can be hard to determine—in the case of wrong image bugs—which of these tools or the driver's shader compiler has miscompiled.

Making shaders "Vulkan-friendly" Unlike in GLSL, where a global variable of almost any type can be declared as `uniform` (see §2.1), SPIR-V requires that every uniform is declared as a field of a structure called a *uniform block*, with the whole structure being declared to be `uniform`. The number of uniform blocks allowed in a SPIR-V module is implementation-dependent. GLSL has been updated with "Vulkan-friendly features" to allow uniforms to be presented in this way, and glslang will only compile Vulkan-friendly shaders into SPIR-V. We thus wrote a simple pass to turn a standard GLSL shader into Vulkan-friendly form. For simplicity of implementation we approached this by placing each original uniform variable in its own (single-field) uniform block. Our pass limits the number of such blocks to 10, as we have not encountered a Vulkan implementation that supports fewer than 10 uniform blocks, and none of the reference shaders we currently use for testing feature more than 10 uniforms. When glsl-fuzz generates a variant shader with more than 10 uniforms (due to injecting code from other shaders), our Vulkan preparation pass demotes superfluous uniforms to standard global variables initialized to concrete values.

3.2 Argument for Not Running Fuzzing in CTS

We briefly considered pitching to Khronos the idea of running GraphicsFuzz as part of running CTS, so that to pass CTS a driver would have to pass all of the regular tests, and additionally survive running a certain number of GraphicsFuzz-generated tests unscathed. We quickly dismissed this idea because it is important to GPU driver makers that qualifying as Vulkan-conformant involves passing a fixed number of tests that run in a deterministic fashion. However much enthusiasm driver makers have for randomized testing as a way to discover bugs, it is understandable that there is little appetite for a conformance test suite that exhibits randomization.

Another issue with embedding GraphicsFuzz in CTS is that inevitable defects in GraphicsFuzz (such as generating a variant shader that turns out *not* to be semantically equivalent to the reference shader) would manifest as a driver failing to pass CTS.

An alternative to actually running GraphicsFuzz in CTS would be to generate a reasonably large set of shaders—e.g. 1000 shaders—and contribute them as CTS tests. We also quickly decided against this strategy for a few reasons. First, the intended behavior of a CTS test should be feasible for a Vulkan expert to understand. The generated variant shaders are large (in order to maximize the probability of finding a bug), and not feasible for humans

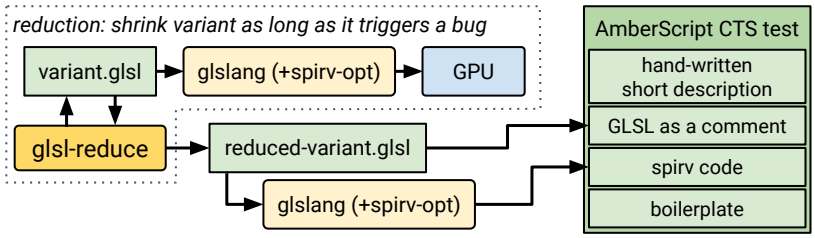

Figure 5 Overview of the reduction process and the creation of a CTS test in AmberScript.

to realistically comprehend in isolation; the reducer, glsl-reduce, is essential in shrinking a bug-inducing variant to a comprehensible form. Furthermore, 1000 large randomized shaders would be a substantial addition to CTS in terms of the test suite's runtime, but is not a large enough number of tests to run with the expectation of thoroughly testing a shader compiler.

We opted instead for setting up a continuous fuzzing process whereby we could use GraphicsFuzz to find bugs that affect current shader compilers, use glsl-reduce to shrink the associated tests down to small examples that reproduce said bugs, and contribute the resulting tests. We now explain the format we settled on for adding crash and wrong image tests to CTS. We detail our tooling for continuous fuzzing in §4.

3.3 Supporting Crash Tests

Around the same time we commenced our plan to add tests exposing shader compiler crash bugs to CTS, a new tool called Amber was launched [17]. Amber provides a simple domain-specific language, AmberScript, in which some aspects of a Vulkan graphics pipeline can be specified, including the SPIR-V shaders that should be executed, and input and output data (including uniform blocks and framebuffers) on which the shaders should operate. It also allows querying the results of running a shader, e.g. probing pixels in the output framebuffer. The motivation for Amber was to make it easy to write stand-alone shader compiler tests, hiding the (very substantial amount of) Vulkan API boilerplate required for even a simple graphics pipeline. Since early 2019, Amber has been integrated into Vulkan CTS and is now the preferred method for writing shader compiler tests.

We wrote a script that takes a reduced GLSL shader known to trigger a SPIR-V shader compiler crash (after translation to SPIR-V and possibly optimization using some specific spirv-opt flags) and produces an Amber test comprised of:

- A brief comment, supplied as an argument to the script, to describe the test and the reason why it should be expected to pass;
- A comment showing the original GLSL code for the reduced shader; this is useful because GLSL is much easier to read compared with SPIR-V;
- Assembly code for the SPIR-V fragment shader that was obtained from this GLSL by translation using glslang and (optional) optimization using spirv-opt;
- A comment listing the spirv-opt arguments that were applied (if any);
- Commands to create the target framebuffer and to populate the shader uniforms;
- A command, supplied as an argument to the script, to check some property of the image finally obtained in the framebuffer.

Figure 5 illustrates the process of Amber test creation following test case reduction.

```
# A test for a bug found by GraphicsFuzz.

# Short description: A fragment shader that uses pow

# We check that all pixels are red. The test passes because main does
# some computation and then writes red to _GLF_color.

SHADER vertex variant_vertex_shader PASSTHROUGH

# variant_fragment_shader is derived from the following GLSL:
# #version 310 es
#
# precision highp float;
# precision highp int;
#
# layout(location = 0) out vec4 _GLF_color;
#
# void main()
# {
#  vec2 a = vec2(1.0);
#  vec4 b = vec4(1.0);
#  pow(vec4(a, vec2(1.0)), b);
#  _GLF_color = vec4(1.0, 0.0, 0.0, 1.0);
# }
SHADER fragment variant_fragment_shader SPIRV-ASM
; SPIR-V
; Version: 1.0
; Generator: Khronos Glslang Reference Front End; 7
; Bound: 28
; Schema: 0
                OpCapability Shader
          %1 = OpExtInstImport "GLSL.std.450"
                OpMemoryModel Logical GLSL450
                OpEntryPoint Fragment %main "main" %_GLF_color
                OpExecutionMode %main OriginUpperLeft
                OpSource ESSL 310
                OpName %main "main"
                OpName %a "a"
                OpName %b "b"
                OpName %_GLF_color "_GLF_color"
                OpDecorate %_GLF_color Location 0
       %void = OpTypeVoid
          %3 = OpTypeFunction %void
      %float = OpTypeFloat 32
    %v2float = OpTypeVector %float 2
                ...
         %21 = OpCompositeConstruct %v4float %17 %18 %19 %20
         %22 = OpLoad %v4float %b
         %23 = OpExtInst %v4float %1 Pow %21 %22
                OpStore %_GLF_color %27
                OpReturn
                OpFunctionEnd
END

BUFFER variant_framebuffer FORMAT B8G8R8A8_UNORM

PIPELINE graphics variant_pipeline
  ATTACH variant_vertex_shader
  ATTACH variant_fragment_shader
  FRAMEBUFFER_SIZE 256 256
  BIND BUFFER variant_framebuffer AS color LOCATION 0
END
CLEAR_COLOR variant_pipeline 0 0 0 255

CLEAR variant_pipeline
RUN variant_pipeline DRAW_RECT POS 0 0 SIZE 256 256

EXPECT variant_framebuffer IDX 0 0 SIZE 256 256 EQ_RGBA 255 0 0 255
```

Figure 6 CTS test that exposed a shader compiler crash bug, in AmberScript form. Some of the SPIR-V assembly has been omitted.

```
#version 310 es
precision highp float;

layout(location = 0) out vec4 _GLF_color;

vec3 GLF_live6mand()
{
    return mix(uintBitsToFloat(uvec3(38730u, 63193u, 63173u)),
      floor(vec3(463.499, 4.7, 0.7)), vec3(1.0) + vec3(1.0));
}
void main()
{
    GLF_live6mand();
    _GLF_color = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Figure 7 Reduced shader that triggered a floating-point constant folding bug in LLVM

▶ **Example 1.** The GLSL shader of Figure 1, which we used to illustrate the GLSL language in §2.1, triggered a SPIR-V shader compiler crash in the GPU driver of a popular Android device, after translation to SPIR-V (and without requiring any use of spirv-opt). This shader was reduced from a much larger variant shader generated by GraphicsFuzz, which we edited by making the variable names simpler, and by adding the final line of executable code, which ensures that the colour red is written to the framebuffer. We believe the shader compiler crash was due to an assertion failing in the lowering of the `pow` intrinsic to LLVM bytecode. This is somewhat surprising given that the result of `pow` is not used, but was presumably due to dead code elimination being executed after lowering.

An abbreviated version of the Amber test corresponding to this shader is shown in Figure 6 (we omit some of the SPIR-V assembly). The test and its intent are described on lines 1–6; line 8 indicates that a standard trivial *vertex* shader (not otherwise relevant in this experience report) should be used in the test pipeline; lines 10-24 show the GLSL code for the fragment shader, and match Figure 1; the corresponding SPIR-V shader (emitted by glslang) is shown on lines 26–52 as SPIR-V assembly (notice the invocation of the `Pow` intrinsic on line 49); line 55 declares a framebuffer, and lines 57–62 define a graphics pipeline based on the vertex and fragment shaders, with the framebuffer attached; line 63 sets the back buffer to black (so that any pixels not rendered to would remain black); lines 65-66 run the pipeline, and line 68 asserts that the framebuffer ends up red at every pixel.

The purpose of adding this test to CTS was to expose the driver bug that it triggered, so that future drivers cannot be Vulkan conformant unless the underlying bug is (at least partially) fixed. We write red to the framebuffer and assert that the framebuffer indeed ends up being red so that the test has at least some runtime oracle; it does a little more than just checking that shader compilation succeeds. The test would be better if the shader stored values into one or more components of `_GLF_color` using the result of the call to `pow`, and then asserted a suitable framebuffer colour; as it stands the test would pass even if `pow` were compiled incorrectly but without a compiler crash. We occasionally work to contribute higher quality test oracles, but do not agonize over this since the main motivation for adding these tests is to force the elimination of compiler crash bugs from conformant drivers.

▶ **Example 2.** Figure 7 shows a reduced shader that triggered a bug in AMD's LLVM-Based Pipeline Compiler (LLPC) [19]: an assertion failed during constant folding:

```
amdllpc: external/llvm/lib/Support/APFloat.cpp:1521: llvm::lostFraction llvm::detail::IEEEFloat::
addOrSubtractSignificand(const llvm::detail::IEEEFloat &, bool): Assertion ‘!carry’ failed.
```

We reported this bug,[6] and the LLPC compiler developers traced its root cause to a bug in LLVM's floating-point emulation code related to handling of subnormal numbers, which was promptly fixed.[7] This demonstrates that shader compiler fuzzing can have positive impact on common infrastructure (LLVM in this case) that is used by many compilers for C-family languages. We contributed a Vulkan CTS test based on this bug, with a structure similar to the example of Figure 6.[8]

3.4 Supporting Wrong Image Tests

Recall from §2.4 and Figure 3 that GraphicsFuzz finds miscompilation bugs via a variant shader that renders a significantly different image compared to the image rendered by the associated reference shader. In this case glsl-reduce reverses as many of the transformations that were applied to the variant shader as possible while the difference persists. To create Vulkan CTS tests suitable for exposing such bugs we worked with the Amber developers to add AmberScript features related to comparing the outputs of multiple graphics pipelines. In particular, we added the ability to compare framebuffers in a *fuzzy* manner. This allows us to turn a GraphicsFuzz reference and reduced-variant shader pair into a single AmberScript file that (a) creates and runs a separate pipeline for each shader, rendering to distinct framebuffers, and (b) asserts fuzzy equality between these framebuffers.

A challenge associated with this is the selection of a suitable fuzzy comparison metric for our purpose. We collected a corpus of image pairs that—based on our shader compiler fuzzing experience—we would like to be deemed similar, and a set of pairs that we would like to be deemed different. The corpus includes image pairs produced by graphics drivers during our testing efforts, plus a few manually crafted image pairs that we believe could occur in theory and that we thought may prove challenging for certain comparison algorithms. We experimented with various image comparison algorithms provided by the `scikit-image` [47] Python library, including MSE, NRMSE, SSIM, and PSNR. We also tried several custom image comparison algorithms based on obtaining and comparing image histograms. We found that image histogram comparison was very effective at correctly classifying image pairs in our corpus, except for some manually crafted image pairs where one image was a rotation or mirror of the other. Indeed, the key weakness of image histogram comparison is that all spatial information is lost. A key advantage is it is very resilient to minor differences that other algorithms flag as important, but which we would typically like to be ignored. We chose to initially proceed with an image histogram comparison algorithm for the following reasons: it correctly classifies image pairs in our corpus as well as or better than most other algorithms; it is very simple to understand and implement (which is important because we don't want GPU vendors to struggle to understand why a Vulkan CTS test has failed and have to debug the image comparison algorithm itself); it has fairly low performance requirements;[9] with a high tolerance value, it is fairly forgiving of minor differences, and—to achieve a low false alarm rate—we would prefer to incorrectly classify an image pair as similar than incorrectly classify the pair as different (most image differences we encounter in practice are easily detected with a forgiving algorithm/tolerance).

[6] `https://github.com/GPUOpen-Drivers/llpc/issues/211`
[7] `https://reviews.llvm.org/D69772`
[8] `https://github.com/KhronosGroup/VK-GL-CTS/blob/master/external/vulkancts/data/vulkan/amber/graphicsfuzz/mix-floor-add.amber`
[9] When running the Vulkan CTS on Android, the image comparison is done on the Android device using the CPU, which has some overhead, especially when using a simulated (software) CPU, as is commonly done when testing next-generation hardware.

We implemented our image comparison algorithm, *Histogram EMD* (where EMD stands for Earth Mover's Distance [29]), in the Amber code base, and added a command to AmberScript of the form:

```
EXPECT buffer_1 EQ_HISTOGRAM_EMD_BUFFER buffer_2 TOLERANCE value
```

where `buffer_1` and `buffer_2` are framebuffers containing the images we wish to compare and `value` is the tolerance value. The test fails if the difference value returned by the algorithm exceeds the tolerance value.

These extensions to Amber provide a pathway for landing tests that expose wrong image bugs in CTS, and we have implemented the necessary scripts to directly generate such tests. We recently put several such tests up for Vulkan CTS code review, and a reviewer quickly found that the validity of one of the tests was questionable due to floating-point precision issues. We discuss this as Example 3 in §3.5. To err on the side of caution, we retracted the other wrong image tests we had put forward and manually simplified each one to double-check that it really did correspond to a driver bug rather than a floating-point precision issue. After sufficient manual simplification, we were able to add an Amber test for each of these bugs, consisting of a *single* shader (and a single pipeline) with a straightforward assertion to check that the single output image is red.

In order to be able to add wrong image tests with a pair of shaders to CTS with confidence, we are working on a corpus of reference shaders that are highly numerically stable.

3.5 Avoiding Invalid Tests

We are anxious not to waste Vulkan CTS reviewer time by proposing tests that turn out to be invalid and get rejected, or—worse—that get accepted (due to the invalidity being subtle, and not leading to failures on current drivers) and subsequently found to be invalid (necessitating their removal from every CTS release they have made it into). We discuss our main concerns related to possible invalid tests.

Preserving semantics during generation and reduction As explained in §2.4, GraphicsFuzz produces a variant shader by having glsl-fuzz repeatedly apply semantics-preserving transformations to a reference, and upon finding a potential wrong image bug, invokes glsl-reduce to reduce the test case by repeatedly attempting to reverse or simplify transformations. For wrong image bugs, it is critical that all transformations preserve semantics both when applied and reversed/simplified. The way GraphicsFuzz has been designed, all information about the transformations that have been applied is recorded by glsl-fuzz via syntactic markers in the generated shaders. Examples of syntactic markers include using special preprocessor macros, and giving variables and functions special names or name prefixes. The glsl-reduce tool then needs to understand these markers and use them to reverse and simplify certain transformations without spoiling the syntactic markers that represent other transformations. In practice we have encountered several hard-to-diagnose bugs where glsl-reduce has erroneously changed the semantics of a shader, usually due to reversal of one transformation having messed up the syntactic markers associated with another transformation, which as a result gets incorrectly reversed.[10]

We guard against this in practice via a degree of manual inspection of the final reduced shader emitted by glsl-reduce, and as glsl-fuzz and glsl-reduce continue to become more stable this issue becomes less relevant. However, based on our experience, we regard having a

[10] See `https://github.com/google/graphicsfuzz/pull/599` as an example pull request that fixes such an issue.

separate generator and reducer that must understand one another in an intricate manner to be a serious pitfall of the GraphicsFuzz approach. Recent research on *internal* test case reduction has the potential to avoid the need for a separate generator and reducer [34], and could thus be useful in our domain.

Loop limiters Recall that the *live code injection* transformation performed by glsl-fuzz (see §2.4) injects code from a donor shader into the shader under transformation in a manner such that the injected code really gets executed at runtime. A problem here is that the injected code may contain loops, and these loops may run for potentially large numbers of iterations. In particular, if the declarations of variables that control loop execution are not themselves injected, glsl-fuzz creates declarations for such variables and initializes them to randomized expressions, which can lead to infinite loops. Programs that risk containing infinite loops are used for compiler testing by tools such as Csmith [50], with the philosophy that it is better to accept that some programs will not terminate, and to use a timeout to bound the runtime of any individual test, than to put in place draconian measures to ensure that all loops terminate. Unfortunately, in the world of GPU shader compilers, long-running shaders cause display freezes, leading to the operating system's GPU watchdog killing the executing shader. This can lead to the shader rendering what appears to be an incorrect image when in fact the image was simply incomplete.

We found that this problem confounded our test results, requiring significant manual inspection of final shaders to check for long-running loops. To overcome this we decided to go ahead and put a relatively draconian measure in place: every loop in every live-injected shader is truncated via a *loop limiter*. This is an additional counter variable specific to a loop. It is initialized to zero immediately before the loop. A conditional statement at the start of the loop body breaks from the loop if the counter exceeds a small positive value (randomly chosen at generation time), and increments the counter otherwise.

With reference to our discussion above about keeping the generator and reducer in sync: loop limiters are given special names when inserted by glsl-fuzz, and when simplifying live-injected code glsl-reduce checks for these names and takes care not to remove loop limiters unless removing the entire associated loop. Again, this coupling between generator and reducer is fragile and can be hard to maintain.

When reducing a compiler crash bug glsl-reduce aggressively shrinks a shader. In this case we allow it to remove loop limiters, which can mean that finally-reduced shaders may contain infinite loops. While the resulting shaders are good enough to reproduce a compiler crash, they are not suitable for addition to CTS, as all CTS tests should be runnable. We therefore inspect shaders manually and edit them to avoid any infinite loops—while preserving the compiler crash—before submitting them for CTS review.

Array bounds clamping Live-injected code may also contain access into arrays and vector/matrix types, which have the potential to be out-of-bounds if their indexing expressions depend on variables that glsl-fuzz initializes to randomized expressions. SPIR-V for Vulkan requires that all accesses are in-bounds. Fortunately, array and vector/matrix sizes are always known statically in GLSL and there are no pointers in the language. We therefore rewrite every array index expression e that appears in live-injected code as $\texttt{clamp}(e, 0, N-1)$, where N is the size of the array or vector/matrix being accessed, and $\texttt{clamp}(a, b, c)$ is the GLSL built-in that clamps a into the range $[b, c]$. An exception to this is when e is a literal that is already in-bounds. As with loop limiters, glsl-reduce is responsible for preserving these in-bounds clamping expressions during test case reduction.

Floating-point stability We use an example to illustrate the risk of submitting invalid CTS tests posed by floating-point instability.

▶ **Example 3.** A transformation that GraphicsFuzz may try to apply is to replace a floating-point expression e with an expression e/ONE, where ONE is an expression guaranteed to evaluate to 1.0 at runtime. GraphicsFuzz has many possible ways of synthesizing an expression that is expected to evaluate to 1.0, one method being to generate an expression of the form `length(normalize(`v`))`, where v is some non-zero vector. The `normalize` GLSL built-in yields a unit vector (when applied to a non-zero vector), and `length` yields the length of a vector, so the expression intuitively should evaluate to 1.0. However, it turns out that the floating-point precision requirements on SPIR-V instructions mean that the result might not *quite* evaluate to 1.0; some round-off error is allowed [20, pp. 1754–1759].

We thought we had found a wrong image bug in SwiftShader upon finding a major image difference to be caused by transforming the following code snippet:

```
// 'ref' and 's' are 'float' variables; 'ref' has value 32.0 at runtime
for (int i = 1; i < 800; i++) {
  // 'mod' is the floating-point modulus operation
  if (mod(float(i), ref) <= 0.01) {
    s += 0.2;
  }
  ...
}
```

This code snippet causes `s` to increase by 0.2 every time `i` is a multiple of 32, since this is the only scenario where `mod(float(i), ref)` will be sufficiently small for the `if` condition to evaluate to true. GraphicsFuzz replaced `ref` with `ref / length(normalize(vec3(...)))`, where the `...` is a placeholder for a non-trivial but sensible expression that evaluates to 1.0 (so that the resulting vector is (1.0, 1.0, 1.0)).

What we assumed was a bug in SwiftShader turned out to be a false alarm. After some manual analysis we found that the divisor `length(normalize(vec3(...)))` evaluated to a value *slightly larger* than 1.0, so that the second argument to the floating-point `mod` built-in was *slightly smaller* than 32.0 (due to `ref` being exactly 32.0). As a result, the statement `s += 0.2` became *unreachable*, even for loop iterations where `i` is a multiple of 32 since the modulus of a multiple of 32 with a value v slightly smaller than 32.0 leads to the value v.

Floating-point precision issues like this hammer home the importance of using numerically stable shaders when searching for wrong image bugs using GraphicsFuzz—the code snippet in Example 3 demonstrates that the shader in question was *not* numerically stable. It is also important to maximize the extent to which the transformations that GraphicsFuzz applies actually preserve floating-point semantics. The deliberately ambiguous approach that graphics shading languages take to floating-point (in order to accommodate many disparate GPUs) means that we can never be certain that a program transformation will completely preserve semantics (since it can affect the optimizations the shader compiler performs, and those optimizations are permitted to have small effects on floating-point results). However, where possible we try to take measures to avoid floating-point error; for instance we have changed the representation of 1.0 discussed in Example 3 from `length(normalize(`v`))` to `round(length(normalize(`v`)))`, where the `round` GLSL built-in rounds its floating-point argument to the nearest integer value; this ensures that the result will indeed be 1.0.

4 gfauto

gfauto (short for GraphicsFuzz auto) is a set of tools for using GraphicsFuzz in a "push-button" fashion with minimal interaction, geared towards generation of tests that can be added to CTS using the pathways described in §3. Pre-gfauto, performing a fuzzing run required

manually generating a set of variant shaders offline from a set of reference shaders, followed by a number of manual steps to run the reference and variant shaders on target devices, waiting for the shaders to finish, and then manually triggering reductions of interesting variant shaders. This approach is unnecessarily inefficient when the main objective is to find as many interesting variants (i.e. those that expose bugs) for a given device as possible within a fuzzing run. In contrast, the high-level, *automatic* workflow of gfauto is: generate a variant shader from a reference shader; run the shaders on the target device; reduce the variant if it is interesting, otherwise discard it; repeat. This process can run continuously for long periods of time, without interaction, which maximizes the number of interesting variant shaders, and thus the potential number of new CTS tests. Using gfauto greatly decreases the length of time needed to perform a fuzzing run and submit a number of CTS tests from that run; we estimate the time period has gone from about 1-2 days (pre-gfauto) down to 1-2 hours (when using gfauto).

We detail three key features of gfauto: creation and replay of self-contained tests (§4.1), bug de-duplication and prioritization (§4.2), and automatic Vulkan CTS test export (§4.3).

4.1 Creation and replay of self-contained tests

Pre-gfauto, the output of a fuzzing run was a directory of images and log files from running reference and variant shaders; the reference and variant shaders themselves were stored in a different directory. Collecting the shaders and output files needed to reproduce and investigate a bug required copying files from different directories, and these files were stored in an ad-hoc format. Furthermore, the versions of the tools required in order to run the test (such as glslang and spirv-opt) were not captured. Details about the target device were available but were again outputted in yet another directory and were typically archived in an ad-hoc format, if at all. Thus, reproducing and investigating a bug was difficult and time-consuming, and useful information was often lost.

gfauto generates a self-contained test from the start. The generated test directory contains a `test.json` metadata file and the reference and variant GLSL shaders. The metadata file contains all information needed to run the test, including a list of required tools and their versions (which are downloaded on-the-fly), an error signature for the test (described below, and initially empty until a crash or wrong image is observed), details of the device on which the test should be run (including the driver version), and the steps needed to run the test (e.g. running spirv-opt with a given series of optimization passes). In particular, gfauto runs the test for the first time using the test metadata file, and is restricted to the tools specified in the metadata; this ensures that no tool dependency can be missed. A test directory can thus be replayed with a single command. In the case of Android, the `test.json` file even captures the Android device serial number so that the test can be automatically replayed on the target device, with no interaction, as long as it is connected to the host machine.

4.2 Bug de-duplication and prioritization

A fuzzing run pre-gfauto would often find a large number of crash-inducing variant shaders, but upon inspection of crash stack traces it would become apparent that many variants were exposing the same bug. Clearly, we would like to prioritize the *unique* bugs found. We wrote several ad-hoc scripts to classify variants that caused crashes into unique "buckets", so that each bucket represents a unique bug (based on the top function name in the stack trace). However, this process was still tedious (as it involved several manual steps) and unreliable (as the scripts were typically hand-tuned for a given fuzzing run). Furthermore,

this classification was never made permanent, so the information would typically be absent in future fuzzing runs. Thus, we would often re-find bugs that had already been found in previous runs and we would have to manually avoid investigating these.

In gfauto, generated tests that expose bugs are stored in buckets in the file system, where a bucket is a directory named using the "signature" (usually the top function name in the stack trace). This makes it trivial to identify tests that expose unique bugs (pick one test from each bucket). The signature is also stored in the test metadata, ensuring the information is never lost, even if the test is moved. A Python function `get_signature` takes the log contents as its only input and outputs the signature string; we update this function as needed to get an accurate bug signature in a number of scenarios. For example, if a stack trace is present (in one of several different formats), the top function name of the stack trace is used, if available, falling back to the hex offset of the function otherwise. Alternatively, if a recognized error message or assertion failure pattern is seen, the error message itself can be used as the signature. This approach ensures we reliably classify tests in most cases. A configurable threshold ensures only a small number of tests are stored in each bucket; subsequent tests are discarded and, crucially, do not need to be reduced, which is expensive. Additionally, gfauto supports downloading and running our Vulkan CTS tests on the target device, capturing the signatures (if an error occurs), and ignoring these signatures during the next fuzzing run. This allows gfauto to ignore bugs that can already be found by existing tests, even if the signatures change between fuzzing runs; this might happen due to a graphics driver update on the target device or due to changes in gfauto's `get_signature` function. In particular this allows unfixed bugs found in previous fuzzing runs to be ignored, assuming appropriate CTS tests were created.

Bug de-duplication challenges The above approach works well most of the time, but some issues remain. Some bugs are nondeterministic in nature. In particular, some of our tests appear to trigger memory leaks in certain shader compilers, which can cause an abort to occur at arbitrary places. Our glsl-reduce tool runs the test up to five times initially (before commencing reduction) in order to validate that the the originally-observed crash signature can be reproduced. Highly nondeterministic tests will often fail this validation step, as the signature will be different every time.

Another issue is when a driver returns a "shader compile error" or "shader link error" message, even though the provided shaders are valid. The driver often provides no additional information, and so there is no way to further distinguish the shader compiler bug. Thus, if we find hundreds of "shader compile error" bugs, we may have found hundreds of distinct compiler bugs, or just one, or any number in between. The same issue applies for tests that expose wrong image bugs, which are simply given a signature of "`wrong_image`". In future work, we hope to identify tests that likely expose distinct wrong image bugs by comparing the semantics-preserving transformations that remain in the fully-reduced variant shaders. Tests that contain very distinct transformations are perhaps more likely to be triggering different shader compiler bugs than tests that contain similar transformations. For compile/link errors (where reduction need not be semantics-preserving) we may be able to use a similarity measure on fully-reduced shaders for de-duplication purposes, drawing on ideas for "taming" compiler fuzzers [10].

4.3 Vulkan CTS test export

Creating a Vulkan CTS test from a bug found by GraphicsFuzz (using gfauto or otherwise) requires some manual iteration on the test. As explained in §4.1, reproducing and investigating a bug pre-gfauto was time-consuming, and information was liable to be lost. The self-contained

nature of a gfauto test greatly improves the experience. Iterating on a CTS test typically requires tweaking the original GLSL shaders and re-generating the SPIR-V (using the correct versions of glslang and spirv-opt) again and again. As already stated above, we believe this has greatly decreased the time needed to get from a fuzzing run to a number of submitted CTS tests, from about 1-2 days (pre-gfauto) down to 1-2 hours (when using gfauto).

We took the push-button nature of gfauto further by automating the end-to-end process of adding a Vulkan CTS test (after manually tweaking the GLSL shaders). Alongside each gfauto test, we store a Python script that generates the final `.amber` file for the Vulkan CTS test. The `.amber` file is similar to the one generated when running the test, but with a copyright header and, as illustrated in Figure 6, a short description and a comment explaining *why* the test passes; note that the short description and comment are manually written by us. The Python script includes the name of the output `.amber` file, the contents of these comments, and some optional tweaks, such as additional AmberScript commands that we might want to add to provide an oracle for the test. Another utility tool then takes this `.amber` file and inserts it into the Vulkan CTS source tree, taking care of updating various index files based on the `.amber` file name and the short description comment. This yields a patch that can be directly put up for Vulkan CTS code review.

5 Finding Test Coverage Gaps Using GraphicsFuzz and gfauto

5.1 Absolute Code Coverage and its Limitations

Line coverage is a widely-used metric for assessing the adequacy of a test suite at a basic level. While many more thorough notions of coverage have been proposed [1], line coverage is appealing because it is both simple to compute and *actionable* [4]—a lack of line coverage can typically be addressed by crafting appropriate tests. A simple idea for growing Vulkan CTS is therefore to run CTS on an open source Vulkan driver and then attempt to write tests to cover parts of the driver that are not reached.

This simple idea suffers from two key problems:

1. It might be *inappropriate* for a Vulkan CTS test to reach certain driver code;
2. For code that could be covered in principle, it is likely very labour-intensive to manually write tests that cover it in practice.

To illustrate problem 1, a recent run of Vulkan CTS on the open source Mesa driver with an AMD back-end [11] identified much uncovered code, but a lot of this code turned out to be (a) debug code (such as routines for dumping data structures in text format), (b) code specific to APIs other than Vulkan (such as OpenCL), and (c) code specific to non-AMD GPUs. It is perfectly legitimate for this code to remain uncovered.

5.2 Differential Code Coverage

To partially solve problem 1 of §5.1 we appeal to *differential* code coverage. Suppose we know which lines of an open source Vulkan driver are covered during a CTS run; call these lines A. Suppose further that we know which lines of the driver are covered by running some other valid Vulkan workload, such as a Vulkan-based game, or a 24-hour run of gfauto; call these lines B. For any line $l \in B \setminus A$, we know that l *can* be exercised by valid use of the Vulkan API, so the fact that a CTS run does not exercise l demonstrates a coverage gap in

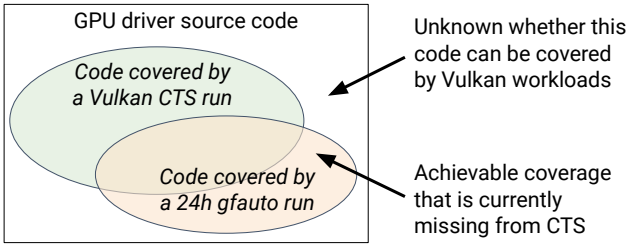

Figure 8 Illustration of differential code coverage. Driver code covered during a gfauto run but not during a CTS run is code that *can* be exercised but for which CTS test coverage is lacking. The tests that gfauto generated to achieve such coverage provide a basis for new CTS tests.

CTS that can certainly be plugged in principle.[11]

The idea of using gfauto and differential coverage to identify code that CTS could in principle cover is illustrated in Figure 8. One might also imagine that differential coverage analysis could be used to drive improvements in GraphicsFuzz: code that CTS can cover but that gfauto cannot might indicate that gfauto should be seeded with a richer set of reference shaders, or that GraphicsFuzz should implement more adventurous transformations. However, the scope of GraphicsFuzz is limited to *shader compilers*, while CTS tests the whole of the Vulkan API, so some knowledge of which parts of the driver relate to shader compilation specifically would be required.

Although the idea of differential coverage analysis is not new (e.g., continuous integration systems often provide facilities for visualizing the coverage trajectory of a project), we could not find a suitable open source project that provides it, so we implemented our own tooling for differential coverage, which we describe in §5.5.

5.3 Using Test Case Reduction to Synthesize Small Tests

While differential coverage helps with problem 1 of §5.1, it does not help directly with problem 2: just knowing that a line is coverable in principle does not yield a suitable CTS test that covers the line. If workload B (see §5.2) were an interactive game, it might be very difficult to reverse-engineer a stand-alone test that provides coverage of a particular line.

However, if workload B is a gfauto run, we can at least obtain a number of GraphicsFuzz-generated variant shaders that provide new coverage. Adding these tests to CTS would serve to fill the coverage gap, but recall from §3.2 that generated tests are very large, and virtually impossible for humans to understand in practice, thus unsuitable for direct addition to CTS.

To overcome this problem, we appeal to *test case reduction* in the following manner. Having performed a gfauto run for, say, 24 hours, and identified a set of driver source code lines $B \setminus A$ that were reached by gfauto but not by CTS, we manually choose one such line and prefix it with `assert(false)`—i.e., we pretend that it is *erroneous* to reach the line. We recompile the driver without coverage instrumentation and run gfauto again using the same parameters (random seed and corpus of shaders) as in the original run. gfauto will, once again, reach the line, this time leading to an assertion failure. gfauto will treat the assertion failure as a shader compiler crash and invoke glsl-reduce to shrink the shader to a

[11] It is theoretically possible that, e.g. due to concurrency, reachability of line l might be nondeterministic, but we have not encountered this in practice.

minimal form that still covers the line. The minimized shader is an excellent candidate for being added to CTS since it is small enough to be human-readable. The process is repeated by choosing another line from $B \setminus A$, avoiding lines that we believe are likely to already be covered by the candidate CTS tests found so far. We periodically re-run CTS after adding the new tests to update workload A, thus ensuring we don't miss any coverage gaps.

At present we have been adopting this approach by collecting differential line coverage of SwiftShader, which incorporates spirv-opt and large parts of LLVM internally. The fact that SwiftShader is open source simplifies the process considerably, although it should be possible to apply a similar process to closed-source binary drivers using instruction coverage instead of line coverage, and by overwriting an instruction with an interrupt instruction (or some invalid instruction) instead of prefixing a line with `assert(false)`.

Custom interestingness test Like many reducers, glsl-reduce supports a custom "interestingness test" script that signals to the reducer whether the shader that is being reduced is still "interesting" (e.g. still crashes the driver). Thus, instead of modifying the driver source code, we could simply provide an interestingness test that runs the shader using the driver with coverage instrumentation, processes the coverage data, and checks if the line of interest was covered. Unfortunately, processing the coverage data is slow, and thus usually done offline. glsl-reduce will typically run the interestingness test hundreds or thousands of times before finding a minimal shader. Thus, making the driver crash via an assertion failure is a much faster approach and, conveniently, already triggers a reduction in gfauto without requiring a customized interestingness test.

Less coverage after reduction A potential downside of our approach is that, after reduction, a shader may cover fewer lines of interest than before. For example, an unreduced shader might cover three seemingly unrelated functions, `f`, `g`, and `h`, that are all not covered by CTS, while the reduced shader might cover just one of the functions, `f`, because we only added an assertion in `f` and thus the reducer did not try to preserve coverage of `g` and `h`. Although this may seem undesirable, focusing on just one function at a time typically allows the reducer to go further (potentially *much* further) in minimizing the shader. We would much rather have three *simple* and *small* CTS tests, each covering a different function, than one *complex* and *large* CTS test that covers all three functions.

5.4 Manually Tweaking Tests to Improve Oracles

Although the reduced tests could be added to the CTS directly, this is almost never appropriate. As with crash bugs, we need to add an oracle to the test, else a driver that does nothing could pass the test. Again as with crash bugs, we typically add code to the shader to make it render the colour red and add a check to the test to ensure all rendered pixels are indeed red. However, the test can be made much more useful if the newly covered lines affect the output colour value so that if a bug was introduced in the newly covered lines, the test would fail. Also note that a coverage gap test will fill a coverage gap for the Vulkan driver that we were running (e.g. SwiftShader), but the hope is that it may also be a meaningful test for other drivers, especially if it relates to a feature for which test coverage is generally lacking. The test should be written with this in mind, as we will see below.

▶ **Example 4.** The following is a generated, reduced fragment shader that covers constant folding code in SwiftShader that replaces a dot product call with zero if one of the operands is a vector of zeros:

```
1 void main() {
2   if(1.0 >= dot(vec2(1.0, 0.0), vec2(0.0, 0.0))) {
```

```
    _GLF_color = vec4(1.0);
  }
}
```

To transform the shader into a form suitable for the Vulkan CTS, we could simply add code to the end of `main` that assigns the colour red to `_GLF_color`, but this has two key disadvantages: (1) any driver that *incorrectly* constant folds the dot function call will still always pass the test, and; (2) a driver might eliminate everything above our final write to `_GLF_color`, and thus, on this hypothetical driver, the test would not even cover code related to the dot product operation. A simple fix is to use the output of the dot function call in the output colour value, as follows:

```
void main() {
  float zero = dot(vec2(1.0, 0.0), vec2(0.0));
  _GLF_color = vec4(1.0, zero, 0.0, 1.0);  // we expect red
}
```

However, even this is not ideal; if a driver incorrectly replaced the dot call with a negative float value, the output colour would still be red, as the output colour components are clamped by the driver to be between 0 and 1, as required by the Vulkan specification. In our final version of the test,[12] we check that the result of the dot call is exactly 0, and we only output red in this case:

```
void main() {
  if(dot(vec2(1.0, 0.0), vec2(0.0)) == 0.0) // precise check
      _GLF_color = vec4(1.0, 0.0, 0.0, 1.0);  // we expect red
  else
      _GLF_color = vec4(0.0);
}
```

The process of manually changing the test to be useful is non-trivial and probably cannot be automated as it requires some creativity, but the reduced test synthesized by gfauto is an excellent starting point and is usually not that different to the final version of the test.

5.5 Implementing Differential Code Coverage

The GCC compiler supports compiling an application with *coverage instrumentation*, causing coverage data to be output when the application runs, which can then be processed with the gcov tool. For example, to capture line coverage of SwiftShader when running the Vulkan CTS, we could perform the following steps:

- **Compile SwiftShader with GCC, adding the `--coverage` flag.** This builds the SwiftShader Vulkan library with coverage instrumentation. For each .o file that was written, the compiler also writes a `.gcno` file at the same location. The `.gcno` (gcov notes) files describe the control flow graph of the corresponding `.o` files, and include mappings to source code file paths and line numbers.
- **Run the Vulkan CTS using SwiftShader.** Due to the coverage instrumentation in the SwiftShader library, `.gcda` files are output alongside the corresponding `.o` and `.gcno` files. The `.gcda` (gcov data) files contain the control flow graph block and edge execution counts (i.e. the number of times each block and edge was executed).

[12] `https://github.com/KhronosGroup/VK-GL-CTS/blob/master/external/vulkancts/data/vulkan/amber/graphicsfuzz/cov-const-folding-dot-determinant.amber`

- **Run gcov to process the `.gcno` and `.gcda` files to get line coverage information.** Manually executing gcov on every `.gcno` file from the required directory (or directories) while avoiding output filename clashes is a tedious process. Additionally, the line coverage output from gcov is fairly primitive. Thus, there are third-party tools, such as lcov and gcovr, that invoke gcov automatically, yielding information in an intermediate data format, and then further process this data to generate, say, an HTML report that shows every source file annotated with the execution count of each line.

Unfortunately, we could not find any tools capable of obtaining *differential* line coverage as described in §5.2. Thus, we created our own set of tools[13] that are similar in spirit to lcov and gcovr, but support obtaining differential line coverage.

cov_from_gcov processes `.gcno` and `.gcda` files into a single `.cov` output file that contains the set of source file lines that were executed. The `.gcno` and `.gcda` files are processed by invoking `gcov` in each required directory to output intermediate data files that are then processed further to produce the `.cov` output file.

cov_new takes `A.cov` and `B.cov` as inputs, and outputs the differential coverage to `new.cov`. The `new.cov` file is simply `A.cov` minus `B.cov` (i.e. $A \setminus B$ from §5.2).

cov_to_source takes `new.cov` as its input, and outputs two parallel directory structures `zero/` and `new/`. Both directories contain copies of the original source files with a prefix added to every line: the `zero/` directory has a "0" prefix added to every line; the `new/` directory has a "1" prefix added to every line present in `new.cov`, and a "0" prefix for every other line. The two directory structures can be compared using a diff tool; lines that are different are the lines of interest (i.e. the lines in $A \setminus B$ from §5.2), and will be highlighted. The approach of generating parallel directory structures means we avoid generating large HTML reports, which can be slow to open and navigate, and instead allows the use of any existing diff tool that is already optimized for this type of task.

6 Fuzzing the SPIR-V Tooling Ecosystem

Recall the rich ecosystem of SPIR-V-related tools shown in Figure 2. Because gfauto uses many of these tools during a fuzzing run, we have the potential to find bugs in them as well as finding bugs in vendor shader compilers. Furthermore we have also conducted some fuzzing runs that include the spirv-cross tool (not part of the default gfauto workflow).

We illustrate, via examples, both the strength of being able to find bugs in multiple tools and the challenge associated with determining which tool is to blame when a problem arises.

▶ **Example 5.** Running spirv-opt on a shader generated by gfauto led to a non-zero exit code. We reported this bug to the SPIRV-Tools project,[14] assuming it to be a bug in spirv-opt. The spirv-opt authors investigated and determined that in fact the shader contained *invalid* SPIR-V that spirv-val (which gfauto runs at every transformation stage) had missed. This identified a validator bug (in the form of a validator omission) but raised the question of what had created the invalid SPIR-V. It turned out that gfauto had run spirv-opt as part of generating the shader, and its block merging pass had been too aggressive: loops in SPIR-V assembly have designated *merge* and *continue* blocks, the *merge* block denoting the loop's exit, and the *continue* block denoting the start of a region of code that must be traversed in order to return to the loop head. The block merging pass was allowing the merge block of

[13] `https://github.com/google/graphicsfuzz/blob/master/gfauto/docs/coverage.md`
[14] `https://github.com/KhronosGroup/SPIRV-Tools/issues/3031`

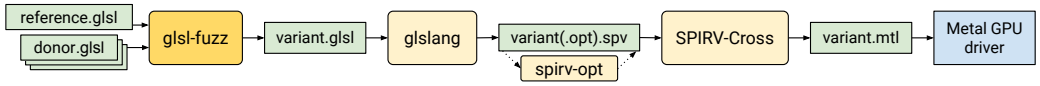

Figure 9 Fuzzing Metal drivers from GLSL.

one loop and the continue block of another loop to be merged. This turned out to be illegal according to the SPIR-V specification, though the wording of the specification did not spell the rule out very clearly. The SPIRV-Tools team enhanced spirv-val to detect this kind of invalidity, and fixed the bug in spirv-opt's block merging pass.[15]

This resolved the combined validator and optimizer bug that we had found with gfauto. Unfortunately, it turned out that 55 existing Vulkan CTS tests contained SPIR-V that was invalid for the same reason—in many cases the SPIR-V in question had been processed by spirv-opt's previously buggy block merging pass. This necessitated fixing these tests in the master branch of CTS, as well as in multiple release branches.

The SPIR-V working group are discussing how to clarify the specification with respect to its rules about the structure of loops and other control flow constructs, in part due to this (and other) reports from our fuzzing efforts.

▶ **Example 6.** An early assertion failure that we triggered in spirv-opt,[16] by fuzzing using a random combination of optimizer flags, turned out to be due to a "merge return" optimization pass being applied to a SPIR-V control flow graph that it was known not to be able to handle. The SPIRV-Tools team hardened spirv-opt by having the "merge return" pass explicitly check for unsupported control flow graph idioms and, on encountering an unsupported idiom, gracefully exit with an error informing the user that they should run the "eliminate dead code" pass first.

Subsequently, we ensured that gfauto only generates lists of spirv-opt optimization passes in which "merge return" (if present) runs after "eliminate dead code".

▶ **Example 7.** We ran some experiments testing MoltenVK [26], an implementation of most of Vulkan on top of Apple's Metal graphics API, on a MacBook Pro. MoltenVK uses spirv-cross to translate SPIR-V to the Metal shading language (MetalSL) so it can be sent to the Metal shader compiler within the Metal driver on a Mac or iOS device. The full translation pipeline is illustrated in Figure 9. When a wrong image is produced in this setup, the bug could be in the Metal driver or in any of the tools that come before (shown as rounded-rectangles in Figure 9). Differential testing can come to the rescue here: if the bad image is also produced by the variant shader in a "vanilla" setup, e.g. by using glslang and rendering the resulting SPIR-V using some other Vulkan driver, the bug is very likely in glsl-fuzz or glslang. Otherwise, if the bug manifests only when adding the same spirv-opt passes (before running on this other Vulkan driver), the problem is likely in spirv-opt. Otherwise, the bug is likely in spirv-cross or the Metal driver.

We found such a wrong image bug in our testing, and used differential testing to conclude that the bug was likely in spirv-cross or the Metal driver. After inspecting the MetalSL code, we found it to be incorrect and so ascertained that the bug was in spirv-cross. We submitted a bug report[17] and the bug was promptly fixed.

[15] https://github.com/KhronosGroup/SPIRV-Tools/pull/3068
[16] https://github.com/KhronosGroup/SPIRV-Tools/issues/1962
[17] https://github.com/KhronosGroup/SPIRV-Cross/issues/1091

7 Related Work

Randomized and metamorphic compiler testing techniques Randomized testing of compilers has a long history; see e.g. [21] for a very early example, and multiple surveys [8, 3, 30]. Random differential testing (RDT) of C compilers was investigated to some extent by McKeeman [36], and the Csmith project from the University of Utah [50] has triggered a lot of interest in the topic over the last decade. An early approach to metamorphic compiler testing involved generating equivalent programs from scratch [46]. More recent work on *equivalence modulo inputs* testing (EMI) [31, 44], a form of metamorphic testing, showed that approaches based on transforming programs in a manner that preserves semantics at least for certain inputs can be an effective way of triggering wrong code bugs. The RDT and EMI approaches have been extended to allow testing of OpenCL compilers [33], and the EMI approach was the inspiration for the approach to metamorphic testing employed by GraphicsFuzz [16, 15], on which the work described in this experience report was built. Randomized differential testing has also been applied to other program processing tools, such as refactoring engines [14] and static analyzers [13], and there is scope for applying techniques from compiler testing to program analyzers more generally [6].

Experience reports related to compiler testing A short report on work at the UK's National Physical Laboratory describes some experiences testing compilers for Pascal, Ada and Haskell using random program generators, mainly focusing on the relative difficulty of constructing program generators for each of these languages [48]. McKeeman's seminal paper on differential testing includes a section on randomized compiler testing that is written in the style of an experience report [36]. In comparison to our paper, these reports do not discuss the challenges of setting up a pathway from randomly-generated test cases to test cases suitable for incorporation into a standard compiler test suite. An edited volume on validation of Pascal compilers provides a number of experience reports related to the testing and validation process [49]. These reports discuss issues related to constructing compiler conformance tests, but do not mention randomized testing. Furthermore, none of the aforementioned experience reports discuss the challenges of testing *graphics* compilers.

Empirical studies related to compiler bugs and compiler testing There have been three recent empirical studies related to compiler bugs and randomized compiler testing: a study on the relative effectiveness of compiler testing based on RDT vs. EMI testing [7], a study on the characteristics of bugs in the GCC and LLVM compilers (not specifically focusing on bugs found via randomized testing) [45], and a study that aims to assess the relative impact on end-user software of fuzzer-found compiler bugs compared with compiler bugs encountered and reported "in the wild" by users [35]. Unlike our work these studies all focus on C/C++ compilers, not compilers for graphics shading languages. A main finding from [35]—that bugs found by fuzzers appear to have at least as much practical impact as bugs reported by users—supports our belief that adding fuzzer-found compiler bugs to compiler regression test suites is a worthwhile endeavour.

Compiler test case reduction and bug de-duplication The *semantics-changing* reduction mode of glsl-reduce is similar in nature to the approach taken by the C-Reduce tool [40], following the well-known delta debugging method [51]. A difference between glsl-reduce and C-Reduce is that glsl-reduce exclusively uses valid abstract syntax tree transformations to reduce a shader, whereas C-Reduce uses a combination of methods, including language-aware transformations built on top of the Clang framework, language-agnostic transformations based on line and token deletion, and methods in-between that assume only basic language properties, such as that the language is block-structured, with blocks delimited by braces.

As a result, C-Reduce can be applied to programs from a variety of languages (e.g. it has been successfully applied to OpenCL C [39]), while glsl-reduce is specific to GLSL. It would be interesting to investigate how well C-Reduce works for the reduction of GLSL programs that induce shader compiler crashes.

The semantics-preserving mode of glsl-reduce is intimately tied to the semantics-preserving transformations applied during metamorphic testing. As discussed in §3.5, the tight coupling between glsl-fuzz and glsl-reduce related to this mode has made it hard to maintain the pair of tools. Recent work proposes leveraging a test-case generator to provide test case reduction "for free", by repeatedly re-generation to search for smaller tests that still trigger a bug [34]. An approach along these lines may be effective in avoiding the need for a tightly coupled generator and reducer in our domain.

Work on automated ranking of compiler bug reports proposes several metrics that can be used to order bug-inducing tests, with the aim of presenting a diverse selection of test cases exposing distinct bugs first [10]. Our de-duplication of crash bugs based on crash signatures (see §4.2) has not yet required this level of sophistication, but we believe such techniques could be brought to bear for de-duplication of wrong image bugs for which there is no analogue to a crash signature.

8 Conclusions and Future Work

We have described our experience rolling out graphics shader compiler fuzzing, based on the GraphicsFuzz tool chain, in a production environment with the goal of improving the Vulkan Conformance Test Suite via new tests that expose shader compiler bugs or provide additional coverage of shader processing tools. We hope the various insights in this report will be useful to researchers interested in testing programming language implementations.

We identify several directions for future practical work in this area.

Direct fuzzing for SPIR-V We have gotten significant mileage from testing SPIR-V shader compilers via GLSL shaders, but the SPIR-V features this flow will exercise are inevitably limited, motivating the need for a fuzzer that works at the SPIR-V level.

Stability tests We discuss the avoidance of invalid tests in §3.5. However, unintentionally invalid tests (due to bugs in the GraphicsFuzz tooling) have sometimes led to the discovery of serious driver stability issues, e.g. Android devices rebooting after accessing invalid memory, or failing to gracefully recover from long-running shaders [15]. It would be valuable to put in place a suite of tests, distinct from Vulkan CTS, to check that invalid shaders cannot derail an operating system.

Higher confidence in wrong image bugs As discussed in §3.4 we are presently exercising caution regarding adding wrong image tests to CTS. A corpus of highly numerically stable shaders would allow us to proceed with greater confidence here, as would a more detailed analysis of the possible floating-point effects of the transformations that glsl-fuzz employs.

Acknowledgements

We are grateful to David Neto and to the anonymous ECOOP 2020 reviewers for their feedback on an earlier draft of this work.

References

1 Paul Ammann and Jeff Offutt. *Introduction to Software Testing.* Cambridge University Press, 2 edition, 2017.

2 Apple. About the security content of ios 10.3, 2017. https://support.apple.com/en-gb/HT207617, see "Processing maliciously crafted web content may result in the disclosure of process memory".

3 Abdulazeez S. Boujarwah and Kassem Saleh. Compiler test case generation methods: a survey and assessment. *Information & Software Technology*, 39(9):617–625, 1997. URL: https://doi.org/10.1016/S0950-5849(97)00017-7, doi:10.1016/S0950-5849(97)00017-7.

4 Arkady Bron, Eitan Farchi, Yonit Magid, Yarden Nir, and Shmuel Ur. Applications of synchronization coverage. In Keshav Pingali, Katherine A. Yelick, and Andrew S. Grimshaw, editors, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2005, June 15-17, 2005, Chicago, IL, USA*, pages 206–212. ACM, 2005. URL: https://doi.org/10.1145/1065944.1065972, doi:10.1145/1065944.1065972.

5 bugs.chromium.org. Issue 675658: Security: Malicious WebGL page can capture and upload contents of other tabs, 2016. https://bugs.chromium.org/p/chromium/issues/detail?id=675658.

6 Cristian Cadar and Alastair F. Donaldson. Analysing the program analyser. In Laura K. Dillon, Willem Visser, and Laurie Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, pages 765–768. ACM, 2016. URL: https://doi.org/10.1145/2889160.2889206, doi:10.1145/2889160.2889206.

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

8 Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing techniques. *ACM Computing Surveys*, 2020. To appear.

9 T.Y. Chen, S.C. Cheung, and S.M. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, The Hong Kong University of Science and Technology, 1998.

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

11 Igalia / codecov.io. Coverage report for vulkan cts on open source mesa driver with amd bck-end, 2020. https://codecov.io/gh/Igalia/mesa/.

12 Keith Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2002.

13 Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. Testing static analyzers with randomly generated programs. In Alwyn Goodloe and Suzette Person, editors, *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*, volume 7226 of *Lecture Notes in Computer Science*, pages 120–125. Springer, 2012. URL: https://doi.org/10.1007/978-3-642-28891-3_12, doi:10.1007/978-3-642-28891-3_12.

14 Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In Ivica Crnkovic and Antonia Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 185–194. ACM, 2007. URL: https://doi.org/10.1145/1287624.1287651, doi:10.1145/1287624.1287651.

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

16 Alastair F. Donaldson and Andrei Lascu. Metamorphic testing for (graphics) compilers. In *Proceedings of the 1st International Workshop on Metamorphic Testing, MET@ICSE 2016, Austin, Texas, USA, May 16, 2016*, pages 44–47. ACM, 2016. URL: https://doi.org/10.1145/2896971.2896978, doi:10.1145/2896971.2896978.

17 Google. Amber GitHub repository, 2020. https://github.com/google/amber.

18 Google. SwiftShader GitHub repository, 2020. https://github.com/google/SwiftShader.

19 GPUOpen Drivers. LLVM-based pipeline compiler GitHub repository, 2020. https://github.com/GPUOpen-Drivers/llpc.

20 The Khronos Vulkan Working Group. *Vulkan 1.1.141 - A Specification (with all registered Vulkan extensions)*. The Khronos Group, 2019. https://www.khronos.org/registry/vulkan/specs/1.1-extensions/pdf/vkspec.pdf.

21 K. V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9:242–257, 1970.

22 John Kessenich, editor. *The OpenGL Shading Language Version 4.60.7*. The Khronos Group, 2019. https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf.

23 John Kessenich, Boaz Ouriel, and Raun Krisch, editors. *SPIR-V Specification, Version 1.5, Revision 2, Unified*. The Khronos Group, 2019. https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.pdf.

24 Khronos Group. glslang GitHub repository, 2020. https://github.com/KhronosGroup/glslang.

25 Khronos Group. Khronos Vulkan, OpenGL, and OpenGL ES conformance tests GitHub repository, 2020. https://github.com/KhronosGroup/VK-GL-CTS.

26 Khronos Group. MoltenVk GitHub repository, 2020. https://github.com/KhronosGroup/MoltenVK.

27 Khronos Group. SPIR-V Tools GitHub repository, 2020. https://github.com/KhronosGroup/SPIRV-Tools.

28 Khronos Group. SPIRV-Cross GitHub repository, 2020. https://github.com/KhronosGroup/SPIRV-Cross.

29 Jeffery Kline. Properties of the d-dimensional earth mover's problem. *Discrete Applied Mathematics*, 265:128 – 141, 2019. URL: http://www.sciencedirect.com/science/article/pii/S0166218X19301441, doi:https://doi.org/10.1016/j.dam.2019.02.042.

30 Alexander S. Kossatchev and Mikhail Posypkin. Survey of compiler testing methods. *Programming and Computer Software*, 31(1):10–19, 2005. URL: https://doi.org/10.1007/s11086-005-0008-6, doi:10.1007/s11086-005-0008-6.

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

32 Jon Leech, editor. *OpenGL ES Version 3.2*. The Khronos Group, 2019. https://www.khronos.org/registry/OpenGL/specs/es/3.2/es_spec_3.2.pdf.

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

34 David R. MacIver and Alastair F. Donaldson. Test-case reduction via test-case generation: Insights from the hypothesis reducer. In *34th European Conference on Object-Oriented Programming, ECOOP 2020*, volume 166 of *LIPIcs*, pages 13:1–13:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

35 Michaël Marcozzi, Qiyi Tang, Alastair F. Donaldson, and Cristian Cadar. Compiler fuzzing: how much does it matter? *PACMPL*, 3(OOPSLA):155:1–155:29, 2019. URL: https://doi.org/10.1145/3360581, doi:10.1145/3360581.

36 William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998. URL: http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf.

37 Microsoft. DirectX shader compiler GitHub repository, 2020. https://github.com/microsoft/DirectXShaderCompiler.

38 NVIDIA. Security bulletin: Nvidia gpu display driver contains multiple vulnerabilities in the kernel mode layer handler, 2018. https://nvidia.custhelp.com/app/answers/detail/a_id/4525/, see "NVIDIA GPU Display Driver contains a vulnerability in the kernel mode layer handler where an incorrect detection and recovery from an invalid state produced by specific user actions may lead to a denial of service".

39 Moritz Pflanzer, Alastair F. Donaldson, and Andrei Lascu. Automatic test case reduction for opencl. In *Proceedings of the 4th International Workshop on OpenCL, IWOCL 2016, Vienna, Austria, April 19-21, 2016*, pages 1:1–1:12. ACM, 2016. URL: https://doi.org/10.1145/2909437.2909439, doi:10.1145/2909437.2909439.

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

41 Mark Segal and Kurt Akeley, editors. *The OpenGL Graphics System: A Specification Version 4.6 (Core Profile)*. The Khronos Group, 2019. https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf.

42 Sergio Segura, Gordon Fraser, Ana B. Sánchez, and Antonio Ruiz Cortés. A survey on metamorphic testing. *IEEE Trans. Software Eng.*, 42(9):805–824, 2016. URL: https://doi.org/10.1109/TSE.2016.2532875, doi:10.1109/TSE.2016.2532875.

43 Robert J. Simpson and John Kessenich, editors. *The OpenGL ES Shading Language Version 3.20.6*. The Khronos Group, 2019. https://www.khronos.org/registry/OpenGL/specs/es/3.2/GLSL_ES_Specification_3.20.pdf.

44 Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In Eelco Visser and Yannis Smaragdakis, editors, *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*, pages 849–863. ACM, 2016. URL: https://doi.org/10.1145/2983990.2984038, doi:10.1145/2983990.2984038.

45 Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in GCC and LLVM. In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 294–305. ACM, 2016. URL: https://doi.org/10.1145/2931037.2931074, doi:10.1145/2931037.2931074.

46 Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. An automatic testing approach for compiler based on metamorphic testing technique. In Jun Han and Tran Dan Thu, editors, *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010*, pages 270–279. IEEE Computer Society, 2010. URL: https://doi.org/10.1109/APSEC.2010.39, doi:10.1109/APSEC.2010.39.

47 Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu, and the scikit-image contributors. scikit-image: image processing in Python. *PeerJ*, 2:e453, 6 2014. URL: https://doi.org/10.7717/peerj.453, doi:10.7717/peerj.453.

1175 **48** Brian A. Wichmann. Some remarks about random testing, 1998. Available online at `https://www.semanticscholar.org/paper/Some-Remarks-about-Random-Testing-Wichmann/2ad3c4c2e1b0b5867a1aa3e7c2de4a17d9facead`.

1178 **49** Brian A. Wichmann and Z. J. Ciechanowicz, editors. *Pascal Compiler Validation*. John Wiley & Sons, Inc., 1983.

1180 **50** Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011. URL: `https://doi.org/10.1145/1993498.1993532`, `doi:10.1145/1993498.1993532`.

1185 **51** Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002. URL: `https://doi.org/10.1109/32.988498`, `doi:10.1109/32.988498`.