

Automated Testing of Graphics Shader Compilers

ALASTAIR F. DONALDSON, Imperial College London, UK

HUGUES EVRARD, Imperial College London, UK

ANDREI LASCU, Imperial College London, UK

PAUL THOMSON, Imperial College London, UK

We present an automated technique for finding defects in compilers for graphics shading languages. A key challenge in compiler testing is the lack of an oracle that classifies an output as correct or incorrect; this is particularly pertinent in graphics shader compilers where the output is a rendered image that is typically underspecified. Our method builds on recent successful techniques for compiler validation based on *metamorphic testing*, and leverages existing high-value graphics shaders to create sets of transformed shaders that should be semantically equivalent. Rendering mismatches are then indicative of shader compilation bugs. Deviant shaders are automatically minimized to identify, in each case, a minimal change to an original high-value shader that induces a shader compiler bug. We have implemented the approach as a tool, GLFuzz, targeting the OpenGL shading language, GLSL. Our experiments over a set of 17 GPU and driver configurations, spanning the main 7 GPU designers, have led to us finding and reporting more than 60 distinct bugs, covering all tested configurations. As well as defective rendering, these issues identify security-critical vulnerabilities that affect WebGL, including a significant remote information leak security bug where a malicious web page can capture the contents of other browser tabs, and a bug whereby visiting a malicious web page can lead to a “blue screen of death” under Windows 10. Our findings show that shader compiler defects are prevalent, and that metamorphic testing provides an effective means for detecting them automatically.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → *Rasterization*;

Additional Key Words and Phrases: GPUs, OpenGL, GLSL, testing, shaders, compilers

ACM Reference Format:

Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 93 (October 2017), 29 pages.
<https://doi.org/10.1145/3133917>

1 INTRODUCTION

Real-time 2D and 3D graphics, powered by technologies such as OpenGL [Kessenich et al. 2016c], Direct3D [Microsoft 2017a] and Vulkan [Khronos Group 2016], are at the heart of application domains including gaming and virtual reality, and are employed in rendering the graphical interfaces of operating systems, interactive web pages, and safety-related products such as automated driver

This work was supported by EPSRC Early Career Fellowship (EP/N026314/1), an EPSRC-funded studentship via the Centre for Doctoral Training in High Performance Embedded and Distributed Systems (EP/L016796/1), and Imperial College’s EPSRC Impact Acceleration Account.

Authors’ addresses: A. F. Donaldson, A. Lascu, H. Evrard, P. Thomson, Department of Computing, Imperial College London, London, SW7 2AZ, UK.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2017 Copyright held by the owner/author(s).

2475-1421/2017/10-ART93

<https://doi.org/10.1145/3133917>

assistance systems. Graphics processing units (GPUs) provide the computation required for high quality real-time graphics. Developers program GPUs by writing *shader programs*, including *vertex shaders* to specify scene transformations and *fragment shaders* to compute individual pixel colors. Graphics shaders are expressed in shading languages, such as GLSL [Kessenich et al. 2016a], HLSL [Microsoft 2017b], and the lower-level SPIR-V intermediate representation [Kessenich et al. 2016b], associated with OpenGL, Direct3D and Vulkan, respectively. These languages aim to provide an abstraction that enables portability: although GPUs from different designers vary in specifics, such as floating-point roundoff, the effect computed by a shader program should ideally be visually similar to an end user, regardless of the GPU that is actually used for rendering.

Graphics drivers are the interface between the code a developer writes and the computation that occurs on a GPU. A graphics driver includes a *shader compiler* that translates shader programs into low-level machine code specific to the target GPU. This compilation is performed at *runtime*, meaning that an application cannot rely on *a priori* knowledge of the end user's GPU. For example, graphics shaders embedded in an Android app today will execute on a GPU from ARM, Imagination, NVIDIA or Qualcomm if the app is deployed on a Samsung Galaxy S6, ASUS Nexus Player, Shield TV or HTC One M7 device, respectively (see Table 1).

Shader compilers need to be reliable. If a shader compiler crashes during runtime compilation, this may prevent operation of the application attempting to render using the shader. Worse, a defective shader compiler may silently generate incorrect machine code. This can lead to wrong images being rendered, and to security vulnerabilities, e.g. if the erroneously-generated machine code accesses out-of-bounds memory. Incorrect rendering of images is not only frustrating for graphics developers who have carefully designed and programmed visual effects, but may also be critical when shaders are used to render the displays of safety-related systems; this use case is underlined by the Safety-Critical OpenGL specification [Khronos Group 2015]. Security defects, including system crashes or freezes, and access to out-of-bounds memory, are especially concerning in the context of *web-based* graphics rendering. The WebGL standard [Khronos Group 2014] enables a web page to run shaders on a client-side GPU via JavaScript. If a WebGL shader is miscompiled in a manner that induces a security vulnerability, a user's machine might crash when they visit a malicious web page. Worse, if the vulnerability involves out-of-bounds memory accesses, it may be possible for a malicious web page to steal a user's personal data by reading from elsewhere in video memory [Context 2011; Lee et al. 2014].

These risks are compounded by runtime compilation: it is hard to work around compiler bugs by testing an application in advance, because the shaders of an application will be compiled on-the-fly for end user GPUs that may be unknown, unavailable, or that may not yet exist, at design time.

Unfortunately, shader compilers are hard to test. Shading languages are deliberately under-specified in various aspects to cater for the diversity of current and future GPUs. Examples include the level of floating-point precision that is required, and the ranges associated with integer data types. Under-specification means that there is no single reference image that should be rendered in a given scenario, to the extent that it is impossible to construct a platform-independent conformance test suite or reference implementation that rigorously checks rendering results.

These issues amplify what is already the main challenge in validating compilers: that there does not typically exist a direct oracle to determine whether a compiled program produces an acceptable result on execution.

A popular method for circumventing the oracle problem in compiler validation is *differential testing* [McKeeman 1998; Yang et al. 2011], whereby multiple compilers for the same language are cross-checked against a set of (usually randomly-generated) programs. However, differential testing is problematic in the context of shader compilers for two reasons. First, it requires not only multiple GPU devices to be available, but also a reasonable level of agreement between these devices

in interpreting the language specification. Second, it needs a rich source of diverse benchmark programs that exercise the shading language broadly and yet do not depend on language features for which behavior differs substantially across platforms. This requirement makes automatically generating such shaders a challenge.

An alternative approach to the oracle problem uses *metamorphic testing* [Chen et al. 1998], and has been applied in various ways to find defects in compilers for languages including C, OpenCL and GLSL [Donaldson and Lascu 2016; Le et al. 2014, 2015; Lidbury et al. 2015; Sun et al. 2016; Tao et al. 2010]. These approaches work by generating families of programs that should yield identical results, such that mismatches between programs indicate compiler bugs.

1.1 Our contributions

Inspired by successful applications of metamorphic testing to C compilers, via *equivalence modulo inputs* (EMI) validation [Le et al. 2014, 2015; Sun et al. 2016], and building on a preliminary study [Donaldson and Lascu 2016], we present a technique and tool, GLFuzz, for testing OpenGL shading language (GLSL) compilers based on *semantics-preserving transformations*. We have designed a set of six such program transformations for GLSL, which are detailed in Section 3.1. For a given *original* shader, GLFuzz repeatedly applies many of these transformations at random, leading to a transformed *variant* shader. The variant is typically much larger than the original, but should render a visually identical image due to the semantics-preserving nature of the transformations.

If a significantly different image is rendered, this may be indicative of a shader compiler bug. GLFuzz then performs *reduction* in a manner similar to delta debugging [Zeller and Hildebrandt 2002]: it iteratively reverses transformations, converging on a *minimal* set of transformations that still lead to a significant rendering difference, such that reversing any further transformations removes the rendering discrepancy. The result is a minimized variant that exposes the potential compiler bug via a clear and small source code change to the original shader.

Cross-checking an individual compiler across families of equivalent programs has at least three appealing properties. First, a GPU and driver version can be tested in isolation, thus divergent behavior between GPU platforms is not an issue. Second, a rich and diverse set of shader programs can be generated *automatically* from *existing* shaders of *high-value* (e.g. they ship as part of an important real-world code base, such as a game, or come from a carefully-crafted in-house test suite). Third, the defects found by our technique are presented as small changes to these high-value shaders, meaning that they are likely to have a higher priority for fixing compared with often pathological defects identified via randomly-generated programs. Prior methods for EMI testing transform programs by deleting, inserting or mutating program statements based on data gathered at runtime [Le et al. 2014, 2015; Sun et al. 2016]. This requires profiling facilities, which are typically not available in the context of graphics shader programs. In contrast, the method we present employs general-purpose static program transformations, and is thus applicable to testing the compilers of all GPU designers that support OpenGL.

We present a large experimental campaign applying GLFuzz to the 17 GPU and driver configurations summarized in Table 1. These configurations span all 7 major designers of commercial GPU drivers, henceforth referred to as *GPU designers*: AMD, Apple, ARM, Imagination Technologies, Intel, NVIDIA and Qualcomm.¹ They cover desktop, web and mobile settings, and account for multiple operating systems. As well as testing OpenGL drivers, the Windows WebGL configurations (7, 11, 12) indirectly test Direct3D drivers due to Google Chrome’s use of ANGLE (Almost Native Graphics Layer Engine) [Google 2017], which we describe in Section 2.1.

¹Configuration 9 features an Imagination PowerVR GPU, but we refer to this as “Apple PowerVR” since the iPhone GPU drivers are written by Apple (as confirmed by an Imagination employee when we reported iPhone defects).

Id	GPU	Host CPU	OS + browser	Driver
Desktop OpenGL				
1	AMD R9 Fury	Intel E5-1630	Windows 10.0.10586	Crimson 16.9.2
2	AMD HD 7700	AMD A10-7850K	Ubuntu 16.04.1	AMDGPU Pro 16.40 348864
3	Intel HD 520	Intel i7-6500U	Windows 10.0.14393	20.19.15.4501
4	Intel HD 520	Intel i7-6500U	Windows 10.0.14393	21.20.16.4542
5	NVIDIA GTX 770	Intel E5-1630	Windows 10.0.10586	373.06
6	NVIDIA GTX Titan	Intel E5-2640	Ubuntu 14.04.5	370.28
WebGL				
7	AMD R9 Fury	Intel E5-1630	Windows 10.0.10586 + Google Chrome 55.0.2883.87	Crimson 16.9.2
8	AMD HD 7700	AMD A10-7850K	Ubuntu 16.04.1 + Google Chrome 54.0.2840.100	AMDGPU Pro 16.40 348864
9	Apple PowerVR GT7600	iPhone SE (MLM62B/A)	iOS 10.1.1 + Safari 602.1	Not available
10	ARM Mali-T628	Samsung Chromebook (XE503C12)	ChromeOS 54.0.2840.101	Not available
11	Intel HD 520	Intel i7-6500U	Windows 10.0.14393 + Google Chrome 55.0.2883.87	20.19.15.4501
12	NVIDIA GTX 770	Intel E5-1630	Windows 10.0.10586 + Google Chrome 55.0.2883.87	373.06
13	NVIDIA GTX Titan	Intel E5-2640	Ubuntu 14.04.5 + Google Chrome 55.0.2883.87	370.28
OpenGL ES under Android				
14	ARM Mali-T760 MP8	Samsung Galaxy S6 (SM-G920F)	Android 6.0.1 build MMB29K.G920FXXS4DPJ2	OpenGL ES 3.1 v1.r7p0-03rel0
15	Imagination PowerVR Rogue G6430	ASUS Nexus Player (TV000I)	Android 7.1.1 build NMF26J	OpenGL ES 3.1 build 1.6@4278818
16	NVIDIA Tegra	Shield TV (P2571)	Android 6.0 build MRA58K.324774_793.8284	OpenGL ES 3.2 NVIDIA 361.00
17	Qualcomm Adreno 320	HTC One M7 (PN071110)	Android 5.0.2 build 7.19.401.30 CL482424	OpenGL ES 3.0 V@84.0AU

Table 1. Summary of the OpenGL configurations we tested

1.2 Summary of Key Findings

Our extensive experimental campaign, using 30,000 generated shaders, has identified a large number of GPU compiler defects, including both rendering defects and *security-critical* vulnerabilities. Our key findings are as follows:

Shader compiler bugs are prevalent in graphics drivers. We identified shader compiler defects in *all* the (GPU + driver) configurations that we tested, spanning all the major GPU designers. Figure 1 and Table 2 illustrate a sample of these bugs, featuring at least one bug per designer; these are discussed in detail in Section 4. Overall, we have reported more than 60 issues to GPU designers, many of which have been confirmed and fixed (see Section 5.4 for a summary of the status of the issues at time of writing). This covers not just GLSL compilers in OpenGL drivers, but also a selection of HLSL compilers in Direct3D drivers, tested via Windows WebGL platforms that use ANGLE to implement WebGL via Direct3D. The high rate of defects, many of which are severe, indicates that more rigorous testing of commercial GLSL compilers is needed.

We maintain a repository providing details of all the bugs we have reported so far (excluding details of some security issues in the interests of responsible disclosure), tracking the status of each bug based on feedback from GPU driver developers [Multicore Programming Group 2017].

Graphics shaders present a security threat. It is well-known that graphics driver defects, as well as defective shaders running via graphics drivers, can have serious consequences [Context 2011; Lee et al. 2014; SecurityWeek 2016; van Sprundel 2014]. To guard against this, version 1 of the WebGL specification is deliberately restrictive so that e.g. it is possible to check statically whether all memory accesses made by a shader are within bounds. Nevertheless, we found a number of


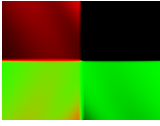
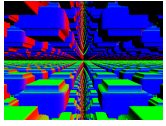
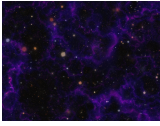
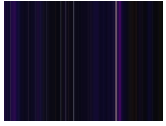
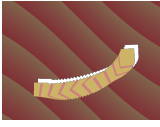

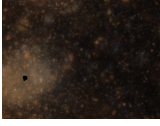


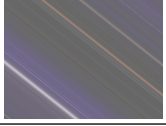
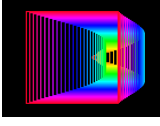

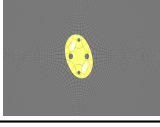
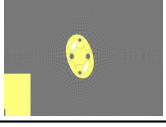
Config.	Original image	Injection	Variant image
1 (AMD)		<pre>if(injSwitch.x > injSwitch.y) { if(injSwitch.x > injSwitch.y) { return; } int f = 1; } [...]</pre>	
10 (ARM)		<pre>if(injSwitch.x > injSwitch.y) { for(int i = 0; i < 1; i++) { k = 0.0; } }</pre>	
14 (ARM)		<pre>uniform float GLF_13time; float GLF_13map() { [...] } [...] float GLF_13t = 1.0; for(int GLF_13i = 0; GLF_13i < 1; GLF_13i++) { if(GLF_13t > 1.0) { continue; } GLF_13t += GLF_13map(); }</pre>	
9 (Apple)		<pre>for(int c = 0; c < 1; c++) { if(j == 0) { return vec4(0.8, 0.5, 0.5, 1.0); } } return vec4(0.8 * injSwitch.y, 0.7, 0.4, 1.0);</pre>	
15 (ImgTech)		<pre>if(injSwitch.x > injSwitch.y) { for(int i = 0; i < 10; i++) { continue; } } [...] if(injSwitch.x > injSwitch.y) { if((p.z > 60.)) { break; } }</pre>	
3 (Intel)		<pre>[...] vec2 uvs = [...]; vec4 uvs_vec = vec4(0.0, uvs, 0.0); /* Replace uvs with uvs_vec.yz after */</pre>	
16 (NVIDIA)		<pre>vec3 hsbToRGB(float h, float s, float b) { return b * ((false ? (--s) : 1.0) - s) + (b - ((false ? (--s) : b * (1.0 - s))) * clamp(abs(abs((false ? (--s) : 6.0)) * [...]; }</pre>	
17 (Qualcomm)		<pre>/* Multiple instances of the following, where v is a literal value */ if(injSwitch.x > injSwitch.y) return v;</pre>	

Table 2. Wrong image examples from different configurations. The image produced by the original shader is on the left, the image produced by the variant shader is on the right, and the code injection (highlighted in yellow) that induces the bug is shown in the middle. Recall that injSwitch is set to (0.0, 1.0) at runtime.

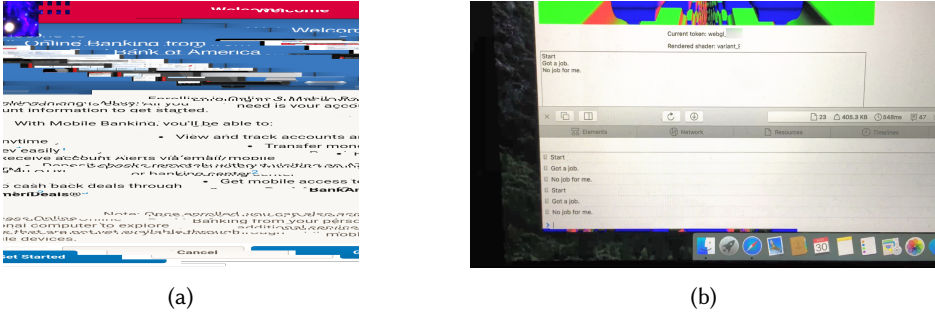


Fig. 1. Images illustrating two security bugs. Image (a) was captured by our malicious web page and includes the contents of another browser tab. The user was visiting the Bank of America page. The image has been flipped and rotated. Image (b) is a screen shot showing a web page in which graphics from the WebGL canvas are being rendered outside the web page on the Safari UI.

serious security vulnerabilities that affect WebGL. Most seriously, we found a significant remote information leak security bug in the Google Chrome web browser where a malicious WebGL page, rendering on a Samsung Galaxy S6 by an ARM Mali GPU, can capture the contents of other browser tabs, illustrated in Figure 1a. On reproducing the issue in response to our bug report, Google have reported this defect directly to ARM, requesting a fix. In response to our report, ARM fixed the issue and the fix has been deployed in recent Samsung Galaxy S6 firmware updates. We also identified a memory corruption bug that caused leakage of data between processes running on an iPhone, which Apple confirmed and have now fixed in iOS 10.3, acknowledging GLFuzz in the release notes and logging the issue as CVE-2017-2424. Another serious defect is a WebGL shader that frequently causes a “blue screen of death” (a fatal system error) under Windows 10 with an AMD GPU; this bug can be triggered simply by visiting a web page. AMD were able to reproduce the issue and have fixed it in more recent drivers. These issues highlight that there is more work to be done by driver writers in ensuring the security of WebGL. We found security-related issues affecting all other designers: on an ASUS Nexus Player (Imagination), incorrect compilation led to garbage being rendered, possibly leaking information; with an NVIDIA GPU under Linux we encountered machine freezes through visiting WebGL pages (NVIDIA have fixed the issue in recent drivers, acknowledging GLFuzz in the release notes and logging the issue as CVE-2017-6259); and in Safari on a MacBook with an Intel GPU, we found that visiting a WebGL page would lead to visual glitches in parts of the browser UI, outside the designated HTML5 canvas element (see Figure 1b).

We discuss these issues in detail in Section 4.2, and provide online a video showing the effects of the main issues [Donaldson and Thomson 2017].

Metamorphic testing is effective in identifying shader compiler bugs. All the bugs we reported were identified by GLFuzz using our metamorphic testing approach. Each of the six types of program transformation employed by GLFuzz proved necessary to trigger at least one defect.

Metamorphic testing finds bugs that GPU designers care about. A key feature of GLFuzz is that bug-inducing variant shaders can be automatically minimized via reduction. As a result we were able to report each miscompilation defect in the form of a small change to an existing, visually appealing shader, that led to defective rendering. We argue that this is a key property of our approach, because (a) the small change that induces the bug provides compiler developers with a hint as to what may be wrong, and (b) the fact that a compiler bug can be induced by only slightly changing a carefully-crafted, human-written shader suggests that this bug should have a higher

priority of fixing compared with an obscure corner-case bug. We have had positive engagement with all seven GPU designers in response to our bug reports: every company has confirmed and fixed at least one defect that we found using GLFuzz.

Metamorphic testing can *inhibit* defects. In a number of instances, we found that the *original* shader was being miscompiled, leading to an unexpected image, and that some of the variant shaders generated through our semantics-preserving transformations would evade miscompilation, rendering the expected image. This surprising result suggests a less conventional application of our approach: GLFuzz could be used by developers to automatically synthesize a temporary work-around for a shader compiler bug while a driver update is awaited.

In summary, our key contributions are:

- An approach to metamorphic testing that uses novel program transformations to automatically generate families of semantically equivalent shaders from existing high-value shader programs
- An implementation of this technique as a tool, GLFuzz, for testing GLSL fragment shader compilers
- A collection of security bugs affecting drivers from every GPU designer, demonstrating that our technique provides a rigorous method for exposing such bugs
- A large experimental evaluation over 17 OpenGL configurations from 7 GPU designers, illustrating the effectiveness of our technique by exposing more than 60 shader compiler defects, spanning every tested configuration

The paper is accompanied by a series of blog posts detailing our experiences applying GLFuzz to graphics drivers from each major GPU designer [Donaldson 2016].

2 BACKGROUND

2.1 OpenGL and GLSL

In this work, we focus on the OpenGL graphics API [Kessenich et al. 2016c] and its associated shading language, GLSL [Kessenich et al. 2016a]. This allows us to test a variety of desktop and mobile configurations (Table 1), as OpenGL is the most widely supported graphics API. We focus on fragment shaders (known as pixel shaders in Direct3D), which are programs that execute on the GPU for every pixel being rendered and usually output a color and depth value. We henceforth use *shader* and *shader compiler* to mean *fragment shader* and *fragment shader compiler* unless specified otherwise. Fragment shaders are written in a particular version of GLSL. We use GLSL version 4.40 for desktop configurations, and GLSL ES version 1.00 for Android and WebGL configurations.

WebGL [Khronos Group 2014]² is a JavaScript version of the OpenGL API that allows web pages to render GPU-accelerated graphics. WebGL disallows several GLSL features that are optional in the GLSL ES version 1.00 specification, yet these *are* typically implemented in OpenGL ES implementations, such as on Android devices. Thus, we treat the more-restrictive WebGL-compatible GLSL as a separate GLSL version. In particular, WebGL restricts loops to simple for-loops that can always be unrolled statically and array/vector/matrix indexing must use *constant-index-expressions*: a constant expression, a loop index variable, or a combination of the two. We also note that, when running on Windows, the Firefox and Chrome web browsers use Direct3D to implement WebGL. This is achieved via ANGLE (Almost Native Graphics Layer Engine) [Google 2017], which translates the GLSL shaders to HLSL as required for Direct3D. Thus, we indirectly test ANGLE and Direct3D shader compilers and drivers on our Windows WebGL configurations.

²Throughout this article, we refer to WebGL version 1.

2.2 The Oracle Problem

To test a piece of software, one generally requires an oracle: a means for determining the correct output for the software under test for a given input [Barr et al. 2015]. In many contexts, an oracle is unavailable, either because there does not exist a well defined set of correct outputs for a given input, or because determining the expected output is theoretically possible but practically infeasible [Weyuker 1982].

Compilers suffer from the latter problem in general: although for small programs it is feasible to work out by hand the results one should observe by running the program after compilation, this quickly becomes infeasible as program size grows. In addition, shader compilers suffer from the first problem: parts of GLSL are deliberately under-specified. For example, the GLSL specification states [Kessenich et al. 2016a, p. 85]:

“Any denormalized value ... can be flushed to 0. The rounding mode cannot be set and is undefined. NaNs are not required to be generated”,

and [Kessenich et al. 2016a, p. 90]:

“Without any [precision] qualifiers, implementations are permitted to perform such optimizations that effectively modify the order or number of operations used to evaluate an expression, even if those optimizations may produce slightly different results relative to unoptimized code.”

The motivation for this flexibility is to enable shaders to run efficiently on a wide range of GPUs with differing hardware support for floating-point arithmetic, and to allow aggressive compiler optimizations to be applied. The downside is that this flexibility makes it impossible to test a new OpenGL implementation via pixel-by-pixel comparison with the rendering results of a single trusted reference implementation. Similarly, writing a rigorous test suite to precisely check expected rendering results is impossible, since the flexibility afforded by the specification means that in many cases no specific result is expected.

2.3 Metamorphic Testing

One approach for solving the oracle problem is *metamorphic testing* [Chen et al. 1998; Segura et al. 2016], which we describe via an example. Suppose we are attempting to implement a program **sin** that computes the trigonometric sin function (note the different type faces used to distinguish the mathematical sin function, and the attempted implementation **sin**). Suppose our program computes **sin**(x) = y for some input x and output y . If we do not have an oracle for sin, we cannot know whether our implementation is correct, i.e. whether y does indeed correspond to $\sin(x)$. Still, we know some facts about the mathematical sin function, including that, for all x , $\sin(-x) = -\sin(x)$, and $\sin(x + 2\pi) = \sin(x)$. These input/output relations for sin provide a means for testing **sin**: if we find an input x such that **sin**($x + 2\pi$) \neq **sin**(x), or such that **sin**($-x$) \neq -**sin**(x) then we know that **sin** must have been implemented incorrectly. Here \neq tolerates a degree of floating-point roundoff error when checking for inequality between values.

More generally, let p be a mathematical function. Suppose we know from domain information that for two functions f_I and f_O , transforming the inputs and outputs of p respectively, the following holds:

$$\forall x . p(f_I(x)) = f_O(p(x)).$$

We can test an implementation **p** of p by checking, for various inputs x , whether **p**($f_I(x)$) \approx $f_O(\mathbf{p}(x))$, where \approx denotes an equality test that, if necessary, is tolerant to acceptable result differences e.g. induced by floating-point roundoff. The input/output relations are called *metamorphic relations*,

and the process of testing a function's implementation by cross-checking inputs and outputs with respect to metamorphic relations is called *metamorphic testing*.

Referring to the \sin example again, we can choose $f_I(x) = f_O(x) = -x$ as metamorphic relations for the identity $\sin(-x) = -\sin(x)$, and $f_I(x) = x + 2\pi$ and $f_O(x) = x$ for the identity $\sin(x + 2\pi) = \sin(x)$.

Notice that metamorphic testing can only be used to find bugs, and not to prove their absence. For example, a stupid \sin implementation, **sin**, that returns **sin**(x) = 0 for all x , satisfies both the identities of \sin discussed above, yet is obviously incorrect.

Metamorphic compiler testing. We now summarize how metamorphic testing can be applied to compilers [Donaldson and Lascu 2016; Le et al. 2014; Lidbury et al. 2015; Sun et al. 2016; Tao et al. 2010]. Compiling and running a program can be viewed as a function, `compileAndRun` : Bytes \times Bytes \rightarrow Bytes, taking a pair of strings corresponding to the program and its input, and producing a stream of bytes: an error message if the program fails to compile, or the output obtained by running the compiled program otherwise.³ For a shader compiler, the output will be data to be written to a display frame buffer. Suppose that f_I : Bytes \rightarrow Bytes is a semantics-preserving program transformation. That is, for a given program it returns a program that may be different syntactically, but which should yield identical output when executed.

Then, for any program P and input I for P , we should have:

$$\text{compileAndRun}(P, I) = \text{compileAndRun}(f_I(P), I)$$

Note that since f_I is semantics-preserving, f_O is simply the identity function and hence does not feature in the above equation. If we find a discrepancy between the outputs, then there must be a defect in the compiler. More accurately, there must be a defect in the implementation of the programming language, which could be a defect in the compiler, supporting runtime libraries, or even in the hardware on which compiled programs execute. A caveat here is that we assume P to be deterministic; e.g. it does not feature randomization, nondeterminism due to concurrency, or nondeterminism arising from undefined behaviors such as accessing memory out-of-bounds.

As we discuss further in Section 3, a fuzzy comparison between the results of `compileAndRun` may be required if the programming language semantics are such that small floating-point differences can arise due to the transformations that are applied (so that they are not strictly semantics preserving).

3 DETAILS OF OUR TESTING APPROACH

We now detail the techniques behind our GLFuzz tool for metamorphic testing of GLSL compilers. Our focus is on compilers for GLSL fragment shaders, though with engineering effort the approach could be applied to other shader types, as well as different shader representations such as HLSL and SPIR-V, or indeed more broadly to other, non-graphical, programming languages.

Testing with GLFuzz involves three phases:

Variant generation From an existing *original* shader, we automatically generate a number of *variant* shaders, by applying semantics-preserving transformations.

Detection of deviant variants If the image rendered by a variant is significantly different from the original image, the variant is said to be *deviant*: it might expose a shader compiler bug.

Reduction of deviant variants After identification of a deviant variant, an automatic *reduction* process is triggered: transformations are reversed as long as a different image is observed,

³In practice the program may also have external side-effects that are not captured by its output, such as writing to files or changing operating system environment variables.

until a deviant variant with a minimal difference from the original variant, and still leading to rendering discrepancies, is obtained.

3.1 Variant Generation via Program Transformations

A variant shader is obtained from an original shader by applying transformations that should not alter the shader semantics. However, if a GLSL shader manipulates floating-point data, which is always the case in practice, then technically no transformation to the shader can be *guaranteed* to be semantics-preserving. As discussed in Section 2.3, shader compilers are permitted to perform optimizations that reorder floating-point operations, “even if those optimizations may produce slightly different results relative to unoptimized code” [Kessenich et al. 2016a, p. 90]. The conditions under which compiler optimizations fire are well-known to be sensitive to the way a program is presented, so that any program change (whether explicitly floating-point related or not) might cause different optimizations to fire, which might influence floating-point computation.

We say that a transformation is *essentially semantics-preserving* to mean that the transformation will have no effect on computation, except for possibly inducing slight changes in floating-point results at the point of transformation. We cannot give a more precise definition (e.g. by bounding acceptable roundoff error) because the GLSL specification does not specify limits on the extent to which results are allowed to change due to compiler optimizations. Henceforth we shall use *semantics-preserving* to mean *essentially semantics-preserving*, for brevity.

We now detail the concept of *opaque values* and *opaque expressions* (Section 3.1.1), on which several of the GLFuzz program transformations hinge, after which we describe the six program transformations that GLFuzz currently employs (Section 3.1.2) and how they can be composed (Section 3.1.3).

3.1.1 Opaque values and expressions. Each variant shader is equipped with a new **uniform** variable,⁴ `injSwitch` (short for *injection switch*), of type **vec2**: a 2D floating-point vector. At runtime, `injSwitch` will be set to (0.0, 1.0), but the shader compiler is oblivious to this at compile-time: the values of `injSwitch.x` and `injSwitch.y` are *opaque* to the compiler. These opaque values can be used to construct expressions that are guaranteed to evaluate to *true*, *false*, 1 or 0. For example, `(injSwitch.x > injSwitch.y)` is guaranteed to be *false* at runtime, and both `(injSwitch.x + injSwitch.y)` and `injSwitch.y` are guaranteed to evaluate to 1. We use the notation **T**, **F**, **1** and **0** to refer to expressions guaranteed to evaluate at runtime to *true*, *false*, 1 and 0, respectively. In each case, GLFuzz randomly generates an appropriate expression via `injSwitch.x` and `injSwitch.y`.

3.1.2 Program transformations. Armed with these opaque values and expressions, we now describe the transformations GLFuzz employs. We selected these transformations based on (a) hypotheses that they might be effective in provoking compiler bugs (e.g. because they complicate control flow or change data layout), and (b) the ease with which they can be automatically applied during variant generation and later reversed during reduction of deviant variants. The bugs in a given driver that our method can detect is of course limited by the available transformations: there is no guarantee that a particular bug can be triggered by the transformations we have implemented so far. However, as discussed in detail in Section 5, we have found every transformation to be essential in triggering at least one bug. Furthermore, our method is parametric so that it can be augmented in the future with additional transformations.

⁴A **uniform** variable is a read-only input parameter to the shader, the value of which is independent from the pixel being rendered.

Dead code injection (dead). Code from another *donor* shader is injected into the target shader, such that the injected code is unreachable at runtime. Code fragments C and D are randomly selected from the target and donor shaders, respectively. Donor fragment D is adapted into a form D' suitable for injection before C : any variables used but not declared in D are either declared at the start of D' , or substituted in D' for the names of other appropriately-typed variables already in scope at C . A statement:

```
if( $\mathbb{F}$ ) {  $D'$  }
```

is then injected into the target shader immediately before C .

The injection is semantics-preserving because condition \mathbb{F} is guaranteed to evaluate to *false* at runtime. Because \mathbb{F} is *opaque*, the compiler cannot deduce that the injected code is unreachable and thus cannot optimize it away. Furthermore, if some of the free variables in D are substituted in D' for variables already in scope at C , it will appear to the compiler that there may be data flow between the original shader variables and D' , which might influence the optimizations applied by the compiler.

Dead jump injection (dead-jmp). It has been hypothesized that metamorphic testing of C compilers is effective because it complicates the control-flow graph of the program being compiled [Le et al. 2014]. Inspired by this hypothesis, we equipped GLFuzz with a transformation that randomly adds a statement to the target shader of the form:

```
if( $\mathbb{F}$ ) { jump; }
```

Here *jump* is one of **break**, **continue**, **return** or **discard**; the former two are only allowed when injecting into the body of a loop. Again, this is semantics-preserving because \mathbb{F} will evaluate to *false* at runtime. This transformation can be seen as a special case of dead designed to provoke compiler bugs that manifest in the presence of complex control flow.

Live code injection (live). While the dead transformation injects unreachable code, live injects donated code that is actually executed. A code fragment D is chosen from the donor and turned into a code fragment D' by renaming all variables and functions referred to in D so that their names do not clash with names declared in the target shader. All free variables appearing in D are declared at the start of D' . Code fragment D' is then inserted at a random position in the target shader. All functions of the donor shader that D' might call (directly or indirectly) are added to the target shader, and are also subject to renaming to avoid name clashes.

Intuitively, this transformation is semantics-preserving because the injected code accesses disjoint data from the original shader code. Some extra care is required to guarantee preservation of semantics in relation to jump statements. For example, if the donor fragment D includes a **return** statement, this must be removed in D' as it would cause the original shader to perform a function return when usually this would not be the case; similar care is required with **break**, **continue** and **discard** statements.

Expression mutation (mutate). Mathematical identities are used to rewrite a numeric or boolean expression e into an equivalent form, *identity*(e), which is one of:

- $(\mathbb{T} ? e : d)$ or $(\mathbb{F} ? d : e)$, where d is a randomly generated expression with the same type as e
- $(e + 0)$, $(0 + e)$, $(e * 1)$ or $(1 * e)$, if e is numeric
- $(e \ \&\& \ \mathbb{T})$, $(\mathbb{T} \ \&\& \ e)$, $(e \ || \ \mathbb{F})$ or $(\mathbb{F} \ || \ e)$, if e is boolean

Analogous transformations are applied to vector expressions.

Vectorization (vec). Multiple variables are packed into a vector, and occurrences of the original variables replaced with vector accesses. For example, suppose the target shader declares variables

a , b and c , where a and c have type **float** and b has type **vec2**. The target shader is equipped with a new *merged* variable, a_b_c say, of type **vec4**. All references to a , b and c are replaced with $a_b_c.x$, $a_b_c.yz$ and $a_b_c.w$, respectively (where “.yz” returns the middle two components of a **vec4** as a **vec2**). For instance, a statement $b.x = a + b.y + c$; becomes:

```
a_b_c.yz.x = a_b_c.x + a_b_c.yz.y + a_b_c.w;
```

This is semantics-preserving because it merely changes where values are stored. GLFuzz implements a generalized version of this transformation in which floating-point and integer variables are randomly selected for packing into vectors.

Control flow wrapping (wrap). Like dead-jmp, wrap aims to provoke compiler bugs by complicating control flow. A randomly-selected target shader code fragment C is replaced with one of:

- **if**(\mathbb{T}) { C }
- **if**(\mathbb{F}) { /* empty */ } **else** { C }
- **for**(**int** temp = 0; temp < 1; temp++) { C }
- **do** { C } **while**(\mathbb{F});

This preserves semantics because in each case C is executed exactly the number of times it was executed originally. Due to the use of opaque conditions \mathbb{T} , \mathbb{F} , and opaque loop bounds 0 and 1, the compiler under test cannot statically determine that the added control flow is redundant.

3.1.3 Composing transformations. GLFuzz generates a variant by repeatedly applying the above transformations to an original shader, leading to composition of transformations. For example, wrap might wrap a piece of code previously donated by live or dead, vec might subsequently apply vector packing to this part of this code, mutate might apply identity functions to expressions that now refer to vector components, and further applications of mutate might be applied to sub-expressions of already mutated expressions. Combined and applied repeatedly, these six transformations offer a rich space of variant programs that can be generated from a given original shader. Also notice that each transformation is easy to *reverse* during reduction (see Section 3.3). In general, the GLFuzz framework can be extended with any additional program transformation that is semantics-preserving and reversible.

3.2 Identification of deviant variants

Deviant variants are identified by checking whether the variant image and the original image are visually distinguishable, in which case the variant is said to be deviant as it might expose a shader compiler bug. A pixel-per-pixel comparison of images is not suitable as the program transformations are only *essentially semantics-preserving*, as discussed in 3.1.

In particular, if a shader is numerically unstable, it is possible that transformations inducing slight floating-point result changes propagate and accumulate, ultimately leading to large differences in what is rendered. When this occurs it is arguable that the associated shader is unlikely to be suitable for portable rendering. Indeed, identifying rendering differences via semantics-preserving transformations might serve as a mechanism for identifying such instability.

As a proxy for “visually distinguishable”, we compare images using the chi-squared distance between image histograms, and regard images as sufficiently different if this distance is above a given threshold. However, GLFuzz is parameterized so that any other image comparison metric could be used. Details of how this threshold was set for our experiments are given in Section 5.2 and we assess the false alarm rate associated with the chi-squared distance metric in Section 5.3.

3.3 Reduction of deviant variants

A deviant variant is typically large, featuring lots of transformations. As such, it provides very little information about which transformations have led to discrepancies in image rendering. It is therefore desirable to reduce a deviant variant into a more manageable form that still triggers the rendering discrepancy. GLFuzz accomplishes reduction via the following process.

In a *reduction step*, a random set of transformations that were applied to the variant are reversed, and the associated image rendered. If the new image is still sufficiently different from the original, GLFuzz performs further reduction steps to try to reverse further transformations. Otherwise, the reduction step is retried using another randomly selected set of transformations. This process iterates until a minimal set of transformations is reached, such that reversing any further transformation causes the distance between image histograms to fall below the given threshold. Due to the random choice of transformations to reverse at each reduction step, this approach actually converges to a local minimum. In practice, local minima seem satisfactory: in our experiments, reductions nearly always converged to small shaders featuring only a small number of transformations.

The reduction process is similar to *delta debugging* [Zeller and Hildebrandt 2002], and, like state-of-the-art delta debugging algorithms, uses heuristics to accelerate the process. A reduction step first reverses a large set of transformations, favoring larger transformations, such as those that inject large blocks of code. If the attempt fails, an exponentially smaller set of transformations is chosen on each retry, and this set is more likely to contain smaller transformations from *within* large injected blocks of code.

This automated process leads to a variant that is only slightly different from the original shader, that *should* render a visually identical image (because the transformations are semantics-preserving), but that renders an image sufficiently *different* to have been flagged by the histogram-based comparison. If, on manual inspection, the images are indeed visually distinguishable this means either (a) the original shader is numerically unstable, so that small compiler-induced floating-point differences have a significant impact on what is rendered, or (b) the mismatch is due to a compiler bug. If the original shader is of high-value then both outcomes are noteworthy.

4 EXAMPLE BUGS

Before describing our full testing campaign (Section 5) we highlight the effectiveness of our approach by detailing several wrong image bugs (Section 4.1) and security bugs (Section 4.2). In both categories, the bugs cover all GPU designers, and the wrong image bugs exercise all six of the semantics-preserving transformations described in Section 3.1. We also found and reported many front-end issues, such as incorrect compiler errors and compiler crashes, but we do not focus on these here.

We have reported each issue to the associated GPU designer and comment on cases where they have responded. Our repository of shader compiler bugs provides further details on the bugs we discuss [Multicore Programming Group 2017], and we associate with each bug the number of a corresponding issue in our GitHub issue tracker.

Because our testing method is black-box, in general we do not have access to low-level details associated with the issues that GLFuzz triggers. However, in Section 4.3 we present insights into two bugs affecting recent AMD and NVIDIA drivers, for which it is possible to inspect generated assembly code.

4.1 Wrong Image Bugs

We present details of eight bugs, alphabetically ordered by the designer of the GPU associated with the configuration that exhibited the bug. For each bug we note the GPU designer, the configuration

id (referring to Table 1), the GLFuzz transformations required to trigger the bug (see Section 3.1), and the issue number associated with the bug in our issue tracker. We chose this selection of bugs in order to cover all GPU designers and all transformations.

Table 2 summarizes details of the bugs in the same order, with one bug per row showing (left-to-right) the configuration id and GPU designer, the image produced by the original shader, the small semantics-preserving transformation that is present in the reduced variant shader (with injected code highlighted in yellow), and the significantly different image produced by the variant shader. In all cases the injections are *minimal*: simplifying the injected code further leads to identical rendered images.

AMD (1) – dead-jmp, dead, issue 7. A combination of dead and dead-jmp injections causes a space scene to become a completely white image. We have observed similar injections causing wrong images to be rendered on our AMD Windows and Linux configurations. What these injections have in common is a variable declaration (*f* in the presented example) and a second dead-code injection. Other generated variant images include a solid background color with artifacts or odd missing shapes. AMD have confirmed they can reproduce this issue and that a fix will be available in an upcoming driver. In Section 4.3 we provide details of the assembly code generated by AMD’s shader compiler for a simple shader that is affected by what appears to be the same issue.

ARM (10) – dead, wrap, issue 26. An unreachable block containing the assignment $k = 0.0$; is introduced by dead, and then wrapped in a single-iteration loop by wrap. The injection changes the “flat” image produced by the original shader to the three-dimensional scene in the variant image. Interestingly, the *variant* image is similar to what we see when rendering the original shader on other platforms. This suggests that this configuration exhibits a compiler bug that is triggered by the original shader and that the injection actually *fixes* (i.e. works around) this issue, by preventing a compiler optimization. ARM confirmed that they could reproduce this in driver versions up to “r11p0” (October 2015), and that it has been fixed independently in driver versions since “r12p0” (the next driver version); it is possible that this defect adversely affected an end-user. We have observed similar instances where GLFuzz injections can work around shader compiler bugs in various configurations, particularly the Android devices. This suggests a possible additional application of semantics-preserving transformations.

ARM (14) – live, issue 31. The injection adds, via donation from another shader, a new uniform variable, GLF_live13time, and a new function, GLF_live13map, that uses the variable. The GLF_liven prefix is used to avoid name clashes as described in Section 3.1. The function is then called from a live, side-effect free context. The injections turn the space scene image into stripes. Interestingly, we were not able to remove or simplify the remaining statements in GLF_live13map (not shown) without making the issue disappear. We reported the issue to ARM, who were able to reproduce it on an old version of their GPU drivers (as used by our Samsung Galaxy S6, see Table 1). The bug was reportedly caused by a register allocation bug that has been independently fixed in newer drivers available to OEMs, but not yet used by the Samsung Galaxy S6. Again, independent fixing indicates that this bug may have affected end users directly.

Apple (9) – wrap, mutate, dead, issue 36. This injection contains wrap and mutate transformations, as well as a dead transformation (not shown in Table 2) that injects an *empty if* statement: `if(injSwitch.x < injSwitch.y) { }`. These transformations cause the foreground “slug” object in the original image to disappear. Removing any of the mutations (even the empty *if* statement) causes the slug to reappear. We reported the issue to Apple who confirmed the issue and have fixed the problem in iOS 10.3.

Imagination Technologies (15) — dead, issue 49. We add two unreachable code segments, each containing **break** or **continue**, embedded in a loop or conditional. Though similar to the transformations arising from dead-jmp, these transformations arise from dead as they do more than merely inject jump statements. The injections cause the image to become darker and seemingly zoomed in. We have reported this bug to Imagination Technologies, who have managed to reproduce it on their latest internal drivers and filed it to the driver team for fixing.

Intel (3) — vec, issue 44. The original shader contains a **vec2** variable, **uvs**. The vec transformation encloses **uvs** in a **vec4**, initialized to $(0.0, \text{uvs}, 0.0)$. The value of **uvs** is then accessed by referring to the contents of the new **vec4**. The original transformation packed additional variables into the x and w components of the vector, which were pulled out again automatically during reduction. The injections turn the space scene image into stripes. We reported this to Intel, who asked us to try a more recent driver version, with which we could not reproduce the issue. This suggests that it may have been fixed independently, perhaps having been reported by end users.

NVIDIA (16) — mutate, issue 12. In this example, there are multiple applications of mutate. Several close-proximity expressions are mutated to the same form: an expression e is transformed to $(\text{false} ? (--s):e)$, where s is a local variable. The injections cause the colored squares to disappear. NVIDIA confirmed they could reproduce this issue, and have deployed a fix in a recent SHIELD Android TV device update.

Qualcomm (17) — dead-jmp, issue 9. Our final example contains a variety of one-line dead jump injections. All injections are guarded by the usual `injSwitch` guard. The contents of the injections are all **return** statements, sometimes returning a vector value. The variant image has a yellow rectangle corruption in the bottom left. This rectangle flickers when we view it via our client, whereas all images produced by our shaders are meant to be static. Qualcomm informed us that they are no longer maintaining drivers for our relatively old HTC device, and did not investigate this issue further to our knowledge.

4.2 Security-related

Our approach is ideally suited to finding wrong image bugs, which are arguably the most challenging bugs to detect due to the lack of an oracle. However, we encountered several interesting security bugs. We now discuss one such bug per GPU designer, in what we judge as arguably decreasing order of severity. Due to responsible disclosure, we do not describe any of the shaders in detail. However, we provide an online video that visually illustrates each issue [Donaldson and Thomson 2017]. For each issue we indicate the affected GPU designer, the issue number for the bug as logged in our issue tracker [Multicore Programming Group 2017], and the associated start time for the issue in our video, in the form MM:SS.

Stealing data across browser tabs (ARM), issue 80, 00:05. On multiple configurations, we encountered cases where variants produce garbage when rendered. That is, the image (which is supposed to be static) is animated with somewhat random patterns, suggesting that uninitialized (and possibly previously used) memory is being rendered as part of the image. We found a particularly severe case when testing the Samsung Galaxy S6 (configuration 14): when rendering a particular variant in Chrome (or the stock browser) using WebGL, the contents of other tabs is frequently visible in the image. An example is shown in Figure 1a and in our video. The HTML5 canvas element in which the image is shown can be captured and uploaded to a server via JavaScript. Thus, we found a significant remote information leak security bug where a malicious web page can capture the contents of other browser tabs. We created a proof of concept that is under review for a security bug bounty. Our issue has been reproduced by the Chrome team who reported the issue directly to

ARM. As a result of this report, the issue was fixed by ARM and the fix was deployed in recent Samsung Galaxy S6 firmware updates.

“Blue screen of death” by visiting a WebGL page (AMD), issue 28, 00:24. We encountered a variant that frequently causes a “blue screen of death” (a fatal system error) on our Windows 10 AMD machine (configuration 7). Crucially, the variant is WebGL compatible and so the bug can be triggered simply by visiting a web page. AMD were able to reproduce the issue and issued a fix in driver version 17.1.1. The shader that triggered this bug was large: 109K. We used our reducer to remove code fragments while a blue screen could still be triggered; this involved running our reducer on a separate machine and repeatedly crashing and restarting the AMD machine. The result was a shader of size 491B that reproduces the crash on our system. However, AMD could not reproduce the issue internally using this small shader, suggesting that the bug might be somewhat system-dependent.

HTC phone restarts on visiting a WebGL page (Qualcomm), issue 82, 00:44. We found a variant that causes the HTC One M7 device to freeze and then restart (configuration 17). The issue is severe because the shader is WebGL-compatible, so the restart can be triggered via a malicious web page. Qualcomm confirmed that they could reproduce this issue, but stated that they will not fix it due to the age of this device model, and because the issue does not affect more recent devices that they still support.

Ubuntu freeze by visiting WebGL page (NVIDIA), issue 46, 00:58. On our NVIDIA Linux system (configuration 13), we found several WebGL-compatible shaders that cause a display freeze from which the OS does not recover. We reported this issue to NVIDIA who were able to reproduce the problem, and stated that it is due to a deadlock in their driver. NVIDIA have confirmed this as a security issue, logging it as CVE-2017-6259 with the description: “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 denial of service.” They have issued fixes in recent driver versions (which driver fixes the issue is device-dependent), acknowledging GLFuzz in a security bulletin.⁵

Glitches in the Safari UI (Intel/Apple), issue 81, 01:23. Although we did not include Mac OS in our experimental campaign, we performed some exploratory testing on a 13” Retina MacBook Pro (Mid 2014). We found some variants that, when rendered in Safari via WebGL, cause the display to freeze and graphical glitches to occur outside of the web page, i.e. in the Safari UI, as shown in Figure 1b. We have reported these issues to Intel because the device uses an Intel GPU and drivers.

“Blue screen of death” by launching fragment shader (Intel), issue 45, 01:51. Similar to the AMD blue screen issue, we found a variant that causes a blue screen of death on our Windows Intel machine (configuration 4). The shader is not WebGL compatible, thus we regard this issue as less severe. We used our reducer to shrink the bug-inducing shader from 975K to 15K.

iPhone renders garbage via WebGL (Apple), issue 37, 02:14. We encountered a variant that causes garbage to be rendered on the iPhone (configuration 9). The shader is WebGL compatible and so the garbage can be rendered by visiting a web page in Safari. Unlike with the Samsung Galaxy S6 example, we could not see any information in the garbage. Nevertheless, the garbage appears to “react to” (i.e. the colors and patterns change) on-screen changes like showing the notification center and switching between apps, and the garbage can still be captured by a malicious web page as before, and so could potentially leak information. Apple have confirmed this as a security issue,

⁵http://nvidia.custhelp.com/app/answers/detail/a_id/4525/

logging it as CVE-2017-2424 with the description: “Processing maliciously crafted web content may result in the disclosure of process memory”. They have issued a fix in iOS 10.3, acknowledging GLFuzz in the release notes.⁶

Nexus player renders garbage via WebGL (Imagination), issue 48, 02:38. We found a variant that causes garbage to be rendered on the Nexus Player (configuration 15). The shader is WebGL compatible. As with the iPhone example above, we could not see any information in the garbage, but the garbage can be captured by a malicious web page and so could potentially leak information. Imagination Technologies told us that they could not reproduce the issue on their newer internal drivers. Thus, the bug may have been independently fixed or a slightly different shader may be required to trigger the bug on the newer drivers.

4.3 Further Insights into Shader Compiler Bugs

In order to shed more light on the low-level causes for the bugs that GLFuzz can detect, we undertook a deeper investigation into two bugs affecting NVIDIA and AMD drivers, which provide support for dumping vendor-specific generated assembly code. We used GLFuzz to find and reduce an issue affecting the most recently available Windows drivers from NVIDIA and AMD. In each case, this led to an original shader, P , and a minimally different variant shader, $P + \delta$. We then undertook a further manual process of simplifying the code common to both P and $P + \delta$, i.e. P , leading to a pair of minimally different and very small shaders, Q and $Q + \delta$ (where shader Q is significantly smaller than P) that should be equivalent and yet render different images. Having undertaken this manual reduction it was feasible to compare the generated assembly code for each pair, to understand the manner in which the compiler has generated incorrect code.

We report on these examples, noting feedback from developers at NVIDIA and AMD related to the issues. Again, we provide the relevant issue numbers associated with the bugs in our issue tracker.

A control-flow bug affecting NVIDIA drivers, issue 76. Figure 2a shows a fragment shader that should write RGBA value (1.0, 0.0, 0.0, 1.0) to `gl_FragColor`, leading to an opaque red pixel. To see this, observe that `bar()` must return 1: the statement **return** -1 is unreachable as it directly follows a **continue** statement. This shader corresponds to Q in the above discussion, and was derived from a much larger original shader.

The variant shader of Figure 2b, corresponding to $Q + \delta$, is identical except that the dead-jmp transformation has injected a **continue** statement and a **return** statement, each guarded by **if** (false). Note that the false condition is in fact the literal false rather than an opaque boolean.

With driver version 384.76 under Windows 10, and also with driver version 381.22 under Ubuntu, NVIDIA’s shader compiler generates the single assembly instruction shown in Figure 2c when applied to the original shader of Figure 2a. This produces the desired result color of (1.0, 0.0, 0.0, 1.0) by using the swizzle mask `xyyx` to select, in order, the x , y , y and x components of the literal vector $\{1, 0, 0, 0\}$ (vectors are indexed as $\{x, y, z, w\}$). The compiler has used control-flow simplification and constant propagation to produce the most efficient code possible.

However, when applied to the equivalent shader of Figure 2b the compiler produces the instruction shown in Figure 2d. This produces a green result color, (0.0, 1.0, 0.0, 1.0), which is obviously incorrect. It is as if the compiler wrongly concluded that `bar()` should return -1; if this would be the case then green would be the correct output color.

⁶<https://support.apple.com/en-gb/HT207617>

```

int bar() {
    for(int i = 1; i <= 1; ++i) {

        continue;

        return -1;
    }
    return 1;
}

```

```

vec2 foo() {
    if (bar() < 0)
        return vec2(0, 1);
    return vec2(1, 0);
}

```

```

void main(void) {
    gl_FragColor = vec4(foo(), 0, 1);
}

```

(a) This fragment shader should yield a red pixel, because `bar()` must return 1

```
MOV.F result_color0, {1, 0, 0, 0}.xyyx;
```



(c) NVIDIA's shader compiler correctly generates a single instruction to produce a red image

```

int bar() {
    for(int i = 1; i <= 1; ++i) {
        if(false)
            continue;
        continue;
        if(false)
            return 1;
        return -1;
    }
    return 1;
}

```

```

vec2 foo() {
    if (bar() < 0)
        return vec2(0, 1);
    return vec2(1, 0);
}

```

```

void main(void) {
    gl_FragColor = vec4(foo(), 0, 1);
}

```

(b) The injections into this fragment shader are guarded by `if(false)`, so should have no effect

```
MOV.F result_color0, {0, 1, 0, 0}.xyxy;
```



(d) NVIDIA's shader compiler incorrectly generates a single instruction to produce a green image

Fig. 2. Two equivalent fragment shaders, and associated correct and incorrect assembly code generated by NVIDIA driver version 384.76

Our contacts at NVIDIA have confirmed that they could reproduce the issue internally, and that it was due to “a latent bug in the shader compiler’s control-flow simplification pass”. They report that a fix will be available in an upcoming driver release.

A control-flow bug affecting AMD drivers, issue 79. The fragment shaders of Figures 3a and 3b should each yield red pixels, owing to their common first line of code; the remaining statements of the shaders are enclosed in dead code blocks, added by the dead and dead-jmp transformations of GLFuzz. Notice that the shaders are identical except that the positions of the unreachable **return** and **discard** statements have been swapped. Each shader can be thought of as having the form $Q + \delta$, where Q is a shader consisting of the single statement `gl_FragColor = vec4(1., 0., 0., 1.);`.

In practice, with driver version 17.6.2 under Windows 10, we find that the shader of Figure 3a yields red pixels as expected, while the shader of Figure 3b does not render anything, yielding a transparent image.

```

void main(void) {
    gl_FragColor = vec4(1., 0., 0., 1.);
    if (injSwitch.x > injSwitch.y) {
        if (injSwitch.x > injSwitch.y)
            discard;
        int a = 0;
    }
    if (injSwitch.x > injSwitch.y)
        return;
}

```

(a) This fragment shader should yield a red pixel, because `injSwitch` is set to (0.0, 1.0)

```

...
s_waitcnt lgkmcnt(0)
v_mov_b32 v0, s1
v_cmp_gt_f32 vcc, s0, v0
s_andn2_b64 s[8:9], s[8:9], vcc
s_cbranch_scc0 label_0011
s_and_b64 exec, exec, s[8:9]
v_mov_b32 v0, 1.0
v_mov_b32 v1, 0
label_0011:
s_mov_b64 exec, s[8:9]
v_cvt_pkrtz_f16_f32 v2, v0, v1
v_cvt_pkrtz_f16_f32 v3, v1, v0
...

```



(c) For the above shader, AMD's shader compiler generates correct code, a fragment of which is shown here

```

void main(void) {
    gl_FragColor = vec4(1., 0., 0., 1.);
    if (injSwitch.x > injSwitch.y) {
        if (injSwitch.x > injSwitch.y)
            return;
        int a = 0;
    }
    if (injSwitch.x > injSwitch.y)
        discard;
}

```

(b) Similarly, this fragment shader should yield a red pixel

```

...
s_waitcnt lgkmcnt(0)
v_mov_b32 v0, s1
v_cmp_ngt_f32 vcc, s0, v0
s_andn2_b64 s[8:9], s[8:9], vcc
s_cbranch_scc0 label_0011
s_and_b64 exec, exec, s[8:9]
v_mov_b32 v0, 1.0
v_mov_b32 v1, 0
label_0011:
s_mov_b64 exec, s[8:9]
v_cvt_pkrtz_f16_f32 v2, v0, v1
v_cvt_pkrtz_f16_f32 v3, v1, v0
...

```



(d) For the above shader, AMD's shader compiler generates code identical to that of Figure 3c, except that a boolean test is erroneously negated, leading to pixels being discarded

Fig. 3. Two equivalent fragment shaders, and snippets of associated correct and incorrect assembly code generated by AMD driver version 17.6.2

The full assembly code generated for each shader, in the Graphics Core Next 3 instruction sets architecture [AMD 2016], is provided via our issue tracker. The code generated is identical with the exception of one instruction. Figures 3c and 3d show snippets of the generated code, with the instruction that differs emphasized in bold. Our understanding of the code of Figure 3c is as follows: the **`v_cmp_gt_f32`** instruction performs the comparison corresponding to `injSwitch.x > injSwitch.y`; the `s_andn2_b64` sets a *predicate mask* in a manner such that threads for which the **`v_cmp_gt_f32`** result was false will be disabled, so that they will not render any pixels; the remainder of the instructions cause enabled threads to render red pixels. This is expected behavior: if `injSwitch.x > injSwitch.y` would hold, the **`discard`** statement would be reachable.

In contrast, the generated code of Figure 3d changes the condition to `v_cmp_ngt_f32`, so that pixels are discarded if `injSwitch.x > injSwitch.y` does not hold. Looking at the source code of Figure 3b this is evidently wrong.

AMD have confirmed that they can reproduce this issue and that it will be fixed in a forthcoming driver release. They reported that: “The issue was resolved by improving the optimization that replaces conditional returns.”

Notice that the structure of the incorrectly-compiled code of Figure 3b is similar to that of the issue shown at the top of Table 2; we expect that the issues have a common root cause.

5 OUR TESTING CAMPAIGN

We now describe the large-scale testing campaign that led to the discovery of the bugs reported in Section 4, among many others. We explain how we generated variant shaders (Section 5.1) and the process for testing configurations against these shaders (Section 5.2), and we examine quantitatively the effectiveness of the various GLFuzz transformations (Section 5.3). We also summarize the status, at time of writing, of the issues we have discovered during our testing efforts so far (Section 5.4).

5.1 Generating Variants

We used GLSLsandbox.com as a rich source of original fragment shaders. These shaders are suitable because they carry high associated value for the expert enthusiasts who contribute them, are intended to work cross-platform (because GLSLsandbox.com is not tied to any particular GPU designer), and exercise the GLSL language in order to produce interesting visual effects. We retrieved all shaders from the first 20 pages of GLSLsandbox.com on 13 July 2016 (1000 shaders), and removed shaders that depended on a particular GLSL version, or failed one of various sanity checks such as being regarded as invalid by the Khronos Reference Compiler. Of the remaining shaders, we chose the largest 100, measured by file size, as original shaders. These ranged between 1,431 and 9,980 bytes (median: 3,302, mean: 3,705). We also used these shaders as donors for live and dead code injection.

From each of these original shaders, we generated three *shader sets*, each consisting of 100 variants, one targeting each of the classes of configurations listed in Table 1. Generated variants for the desktop OpenGL platforms (configurations 1–6) feature transformations that exercise the GLSL 4.40 language; variants for the WebGL platforms (configurations 7–13) are restricted so that transformations exercise only those aspects of GLSL 1.00 compatible with WebGL; while variants for the Android platforms (configurations 14–17) are also restricted to GLSL 1.00, but include features such as non-constant array indexing that we found were supported by our Android devices. With 100 variants per shader set, we generated 30,000 variants in total across the three classes of configuration.

We use *swarm testing* [Groce et al. 2012] to ensure diversity in the combinations of transformations that are applied during variant generation. On generating a variant, each available transformation is selected to be enabled or disabled, uniformly at random. For each enabled transformation, we also randomly choose an associated probability of applying the transformation to each relevant program point. We have not rigorously studied the effectiveness of using swarm testing, but anecdotally we have found it helpful in avoiding repeatedly finding a known bug that is easy to trigger via a particular transformation, thanks to each transformation being often disabled, or sometimes applied with low probability.

5.2 Testing Configurations

Infrastructure. We adopted a client-server architecture to efficiently test a large number of devices. Each configuration runs a “dumb” client, which pulls the next shader from the server and responds

with the rendered image or an error message. The client is unaware of whether the shader is an original, a variant, or a variant with some transformations removed (i.e. a reduction step). Thus, the “smart” server offers original and variant shaders, compares images, and performs reductions. This approach allowed us to cope with configurations that crash due to driver bugs, and made the client, which had to be written several times for different platforms, relatively simple. To test the configurations of Table 1, we have developed clients that run under Windows, Linux, Android, and in a browser via JavaScript and WebGL.

Running shader sets and reducing variants. We aimed to render all 100 original shaders and the corresponding set of 10,000 variant shaders using each of the configurations in Table 1. To identify a candidate wrong image issue we used OpenCV version 3.2, invoking the `compareHist` routine with the `COLOR_BGR2HSV` color space and the chi-squared distance (`HISTCMP_CHISQR`) algorithm, to compare image histograms. This routine returns a floating-point histogram distance, and based on early experimentation, we used 100 as a default threshold value above which an image difference would be deemed interesting. For each shader set, we proceed to the automatic reduction of up to 5 wrong image issues. Where there were more than 5 candidate wrong images, we prioritized reducing variants with a high associated histogram distance.

Incomplete results. While we designed our client-server infrastructure to be robust to configuration crashes and restarts where possible, some configurations would crash in a manner that required a device to be rebooted, an Android app restarted, or a WebGL page refreshed, in each case manually. For several configurations this occurred so frequently that it was infeasible for us to baby-sit the testing process to obtain a complete set of results. We obtained complete or near-complete data sets for configurations 1, 3, 4, 5, 7, 11, 12 and 15; in some cases the data is not quite complete due to e.g. a reduction being abandoned as a result of a sporadic machine failure. The partial data sets for other configurations were sufficiently large for us to identify interesting bugs.

We discovered a late bug in our implementation of the vec transformation (see Section 3.1), which meant that this transformation would sometimes be semantics-changing. While we found reduced variants in which the transformation had been correctly applied and induced a bug (see the **Intel** bug in Section 4.1), we also found cases where a reduced variant was semantically different from the original shader. We thus ignore reduced variants that exhibit the vec transformation in our quantitative analysis in Section 5.3.

5.3 Properties of Reduced Variants

We now study the effectiveness of the GLFuzz transformations in inducing visually identifiable differences in rendering. These differences may be due to shader compiler bugs, or a shader being numerically unstable, prone to significant rendering differences arising from small differences in floating-point computation. We investigate this by considering the reduced shaders for the 8 configurations for which we have near-complete experimental data. Ignoring reduced shaders that still feature the vec transformation, due to the vec bug discussed in Section 5.2, this led to 975 reductions.

Identifying false positives. Recall that we used the chi-squared distance between image histograms, with a fixed threshold determined via pilot experiments, to decide whether rendering differences might be worth investigating (see Section 5.2). To better classify those cases where the difference was visually distinguishable, we used the following procedure. We asked three fellow researchers, not involved in the project, to indicate via a key-press whether they thought each of the 975 pairs

of (original, reduced variant) images were identical.⁷ The participants undertook the experiment at the same machine, were presented with the image pairs in a random order, and were instructed to respond to each pair within a matter of seconds. No participant was color-blind.

We treat a reduction as a *false positive* if at least two participants indicated that the associated image pair looked identical, and as a *true positive* otherwise. A false positive suggests that the pixel-level differences leading to a non-trivial histogram distance between images may be merely due to acceptable floating-point roundoff error; a true positive suggests that the difference may be interesting; indicative of a compiler bug or of numerical instability. We say that a false positive (resp. negative) is *unanimous* if all participants agreed that the image pair was identical (resp. different).

Among the 975 reductions, this experiment led to 568 being classified as true positives (513 unanimous), and 407 false positives (339 unanimous). The false positive rate suggests that our histogram distance threshold is a reasonable proxy for “visually distinguishable”, but perhaps could be tightened. The fact that a non-trivial number of ratings were not unanimous emphasizes the difficulty associated with deciding whether rendering differences are acceptable or not. Our experiment does not account for false *negatives*: cases where a visually distinguishable difference in images leads to a histogram distance below our threshold; assessing this would require having users rate a much larger collection of image pairs.

Assessing the effectiveness of transformations. Table 3 summarizes the ratio of performed reductions, and the extent to which the GLFuzz transformations (excluding vec) are present in the reductions classified as true positives and false positives. In both cases, the bottom table shows, for each listed configuration, and for each transformation, the number of reduced variants in which the transformation still appears. If a reduced variant features multiple transformation types, the count for each transformation type is incremented.

The results show that every transformation contributes both true and false positives for some configurations, but that various transformations are ineffective in yielding true positives for some configurations, e.g. `dead-jmp` for Intel desktop configurations (3 and 4), and `wrap` for the NVIDIA desktop (5) and AMD WebGL (7) configurations. Overall, `wrap` is least effective in terms of number of true positives, but also has a low false positive rate. The `mutate` transformation has a noticeably higher rate of false positives and negatives compared with other transformations. We hypothesize that this transformation has the most impact on floating-point compiler optimizations because, for example, it can change whether an expression is a compile-time constant or not, which might trigger compiler bugs or provoke numerical instability.

The number of reductions varies greatly between platforms, with the Nexus player (15) exhibiting a particularly large number of true positives. Configurations 3 and 4 correspond to our Intel Windows machine with older and newer drivers. With the old drivers, `live` was very effective in provoking true positives, becoming much less so with the newer drivers. The reverse is true for `mutate`, which becomes more effective with respect to the newer drivers, though with an increased false positive rate as well. For AMD, Intel and NVIDIA, configuration pairs (1, 7), (3, 11) and (5, 12) correspond to Windows desktop and WebGL configurations, respectively, with the same drivers. There is, however, little correlation between the effectiveness of the various transformations between configurations in each pair. We attribute this to the use of ANGLE to implement WebGL in Chrome under Windows, so that the Direct3D rather than OpenGL drivers are being tested in the case of WebGL. This is encouraging as it shows that GLFuzz is able to find diverse defects between the OpenGL and ANGLE+Direct3D stacks.

⁷One of the colleagues subsequently became involved in the project and is an author of this paper, but knew little about GLFuzz when participating in the experiment.

Performed reductions					
Config.	True positives			False positives	
	#affected shader sets	mean #red.	#affected shader sets	mean #red.	
1	55	1.33	9	2.44	
3	28	1.46	6	1.00	
4	13	2.08	14	3.21	
5	32	1.69	20	2.40	
7	5	2.00	10	1.80	
11	17	1.94	42	3.52	
12	15	2.00	6	3.50	
15	89	3.37	37	2.68	

Transformations remaining after reduction												
Config.	#red.	True positives					#red.	False positives				
		#dead	#dead-jmp	#live	#mutate	#wrap		#dead	#dead-jmp	#live	#mutate	#wrap
1	73	56	65	3	72	6	22	8	3	1	22	1
3	41	0	0	33	5	0	6	1	0	2	4	1
4	27	1	0	3	22	1	45	10	0	1	45	1
5	54	2	2	1	53	0	48	1	2	5	44	1
7	10	1	4	1	10	0	18	2	3	2	17	1
11	33	5	7	4	31	2	148	33	5	1	148	1
12	30	2	9	4	29	1	21	1	1	0	20	1
15	300	40	142	51	284	22	99	10	34	8	92	0

Table 3. For a subset of configurations (Config.) and for both **true** and **false** positives, the top table lists the number of shader sets in which at least one reduction was performed (#affected shader sets), together with the mean number of variants that were reduced (mean #red.) in these shader sets; the bottom table lists the number of reduced variants (#red.) and details the number of transformations remaining (of each type) after reduction.

5.4 Status of Discovered Issues

We discovered and logged 71 issues affecting shader compilers during our study. We classify these issues into eight categories according to whether we reported them to associated developers and the response we received if so. The categories are:

- *Fix deployed*: the issue was reproduced and fixed, and we have checked that the issue does not manifest in newer drivers.
- *Fix pending*: the issue was reproduced, and a fix has been promised but is not yet available.
- *Confirmed*: the issue was reproduced, but not yet fixed to our knowledge.
- *Reported*: we have not yet received feedback on our issue report.
- *Unreported*: we have logged the issue but not yet reported it, either because we did not want to bombard a particular company with bug reports, or due to us awaiting feedback on earlier reports.

Status of issue	# issues
Fix deployed	20
Fix pending	1
Confirmed	9
Reported	10
Unreported	8
Bug vanished	13
Won't fix	3
False positive	7
Total:	71

Table 4. Summary of the number of issues in shader compilers that we reported so far, and their status at time of writing

- *Bug vanished*: we reported the issue with respect to a particular driver version, but it could not be reproduced with a later driver version (either the company's latest internal drivers, or a newer public release).
- *Won't fix*: the company did not investigate the bug report because they no longer support the device on which the issue presented.
- *False positive*: the issue turned out not to be a bug.

Table 4 uses these categories to provide a breakdown of the 71 issues that we found. The issues cover our testing of the platforms summarized in Table 1, plus a small amount of additional testing using open source Mesa drivers,⁸ the SwiftShader software renderer,⁹ and some variations of the platforms of Table 1 equipped with more recent drivers.

The false positives were mainly due to using certain original shaders that turned out to be numerically unstable, so that significant changes in rendering arose due to differences in floating-point roundoff error. We also reported two issues where the live transformation induced very long-running loops, which caused a black image to be rendered on some platforms. However, it is permissible (in fact, desirable) for an operating system to kill a long-running shader to free the GPU for other rendering tasks, and this turned out to be the reason for the change in rendering.

The false positives arose from our early issue reports. Given our better understanding of the field since then, we are reasonably confident that the 21 issues in the *reported* and *unreported* categories are genuine.

Issues in the *bug vanished* category may correspond to cases where bugs were independently fixed by compiler developers in newer drivers. However, they may also correspond to cases where the underlying bug has *not* been fixed, but rather that the optimizations applied by the shader compiler have changed. In this case, it may simply be that the reduced variant that triggered the bug in an older driver version does not trigger the bug in newer drivers, but that a differently-structured shader would. Except in the few cases where developers gave us direct feedback on bug root causes we have no way to tell the difference.

Table 5 breaks down the 64 issues that we believe are not false positives according to their nature. As expected, most bugs lead to bad image rendering, but we also found 20 shaders that triggered distinct compile-time errors, one shader that caused the SwiftShader renderer to go into a seemingly-infinite loop, and 10 security-related issues, most of which we discuss in Section 4.2, split

⁸<https://www.mesa3d.org/>

⁹<https://github.com/google/swiftshader>

Nature of issue	# issues
Bad image	33
Compile error	20
Non-termination	1
System instability	6
Potential information leak	4
Total:	64

Table 5. Among the non-false positive issues we reported, most are cases of bad image rendering

into system instability issues (e.g. machine crashes and device restarts) and potential information leaks (e.g. the issue discussed in Section 4.2 where data can leak between browser tabs).

6 RELATED WORK

Metamorphic testing. Metamorphic testing [Chen et al. 1998] has been used to test software in application domains including web services, embedded systems and machine learning; see [Segura et al. 2016] for a recent survey. In graphics, metamorphic testing has been applied to test implementations of image processing algorithms [Guderlei and Mayer 2007; Jameel et al. 2016].

An initial approach to metamorphic testing of compilers by generating families of equivalent random programs [Tao et al. 2010] does not generate minimal bug-inducing programs, making it hard to identify the root cause of a behavioral mismatch. Metamorphic compiler testing via program transformation was first proposed in the form of *equivalence modulo inputs* (EMI) testing [Le et al. 2014], where profiling is used to identify program statements that are unreachable for a given input I . A family of programs is then generated by randomly pruning the identified statements; such programs should still behave identically to the original when executed on input I . The technique has been extended to incorporate profile-based dead code insertion and live code mutation [Le et al. 2015; Sun et al. 2016]. A practical difficulty associated with applying profiling-based EMI testing to shader compilers is the lack of detailed runtime analysis facilities for GPU platforms. In the context of OpenCL this has been circumvented by injecting *dead-by-construction* code [Lidbury et al. 2015].

In an initial endeavor applying metamorphic testing to GLSL, we developed the dead and mutate transformations, and showed that metamorphic testing for GLSL is feasible by identifying two bugs in a pair of desktop GPU configurations [Donaldson and Lascu 2016]. Our early work only supported one desktop version of GLSL. In our current work, we introduce a set of new transformations, dead-jmp, live, vec and wrap, and have implemented all transformations in our tool, GLFuzz, adding support for two additional versions of GLSL so that Android and web platforms can be tested. We evaluate the techniques on a set of 17 GPU configurations, covering desktop, Android and web platforms, incorporating devices from every GPU designer, and finding multiple defects per configuration, including exploitable security vulnerabilities. Our results show that *every* transformation employed by GLFuzz is necessary for identifying at least one defect. To our knowledge, ours is the largest reported cross-vendor study of graphics driver reliability.

Other approaches to compiler testing. Compiler testing, through validation suites and program generation, has a long history, and a number of surveys are available discussing early works in the field [Boujarwah and Saleh 1997; Burgess and Saidi 1996; Scowen and Ciechanowicz 1983].

A particularly successful method for testing compilers is *differential testing* using randomly-generated programs [McKeeman 1998], which has been especially popularized by the widely used Csmith tool [Yang et al. 2011]. As well as being applied to numerous C compiler, Csmith has been used to test the Frama-C static analyzer [Cuoq et al. 2012], and has been extended to provide testing

of OpenCL compilers [Lidbury et al. 2015]. A recent study compared the effectiveness of differential testing, both using multiple compilers and the same compiler at different optimization levels, with EMI testing [Chen et al. 2016].

We hypothesize that differential testing will not work well for testing graphics compilers: the significant freedom afforded by languages such as GLSL with respect to floating-point means that, especially in randomly-generated programs, significant differences in results are to be expected. In contrast, metamorphic testing via program transformations is not subject to significant, inter-device rounding differences.

Just as GLFuzz is equipped with test case reduction facilities, methods for testing compilers via randomly-generated programs require test case reduction facilities. The task of reducing a randomly-generated bug-inducing program to a small example that triggers the bug of interest is hard, due to the challenge of ensuring freedom from undefined behavior during the reduction process. The C-Reduce tool provide a best-effort solution to this problem, drawing on a number of static and dynamic analyses [Regehr et al. 2012], and has been extended to provide test case reduction for OpenCL kernels [Pflanzner et al. 2016]. In contrast, reducing GLFuzz-generated variants is much simpler as it involves reversing known transformations; avoiding undefined behavior during this process is straightforward.

Effective fuzzing involves devising strategies that not only achieve a high rate of detected defects, but also achieve *diversity* in the defects that are found, to avoid repeatedly finding and reducing test cases that trigger a single known defect. As discussed in Section 5.1, GLFuzz uses *swarm testing* [Groce et al. 2012] during variant generation, so that different variants generated from the same original shader may be generated with different combinations of program transformations enabled, and with varying probabilities. Other work aims to help in prioritizing compiler test cases, either prioritizing the order in which tests are executed to accelerate testing [Chen et al. 2017], or the order in which a human engineer should approach investigating their root causes, to save developer time [Chen et al. 2013]. So far we have not required these sophisticated strategies to find defects in shader compilers, but they may prove useful in future as shader compilers become hardened to our approach.

Graphics driver vulnerabilities. A major contribution of our study is the identification of *security* bugs affecting every GPU designer, the most critical being those that affect WebGL configurations. It is well known that graphics drivers and shaders pose a security threat (see e.g. [Lee et al. 2014; SecurityWeek 2016; van Sprundel 2014]), and the specific issue of stealing data via WebGL has been studied previously [Context 2011]. We contribute a rigorous testing method to help guard against these threats.

Validating graphics drivers. Graphics drivers are typically validated using conformance tests: the Khronos Group and Microsoft provide conformance tests suites for OpenGL/Vulkan and Direct3D, and open source test suites are available, including *piglit* (which incorporates *gleam*) [Hähnle 2017], and *drawElements Quality Program* (dEQP) [Android Community 2017] for OpenGL. The dEQP suite is now maintained by Google as part of the Android Compatibility Test Suite. Microsoft provide a reference renderer and Khronos a reference compiler front-end. These test suites and reference implementations assess the extent to which API functions and shading language features are supported, but do little to check what is actually rendered. In contrast, our metamorphic approach is successful in identifying rendering defects by cross-checking equivalent shaders with respect to a single device and driver. We have contributed to dEQP by adding a new class of test case

that compares rendering results between equivalent shaders, together with several examples of equivalent shaders found using GLFuzz that triggered bugs in various Android device drivers.¹⁰

7 CONCLUDING REMARKS AND FUTURE WORK

We have presented a large-scale study using metamorphic testing to find defects in graphics shader compilers via semantics-preserving program transformations. Our results show that defects are prevalent, including exploitable security vulnerabilities that affect WebGL, and that our testing method is effective at finding them: we exposed bugs in all configurations that we tested.

A natural question to ask is whether our approach is fundamentally effective in finding shader compiler bugs, or whether shader compilers are simply under-tested, so that our approach has detected “low-hanging” bugs. The shader compilers we have considered are all commercial products that are required to pass conformance tests from the Khronos Group, and during our discussions with GPU driver developers we heard from several companies that they test their compilers on shaders captured from complex video games. Some companies also told us that they use fuzzers for in-house testing, but only to check whether their compiler crashes, rather than to check whether rendering is accurate. Our impression is that, due to the oracle problem, GPU driver developers do not have good methods to test at scale whether their compilers generate correct code. Our techniques make steps towards solving the oracle problem in this domain, and the bugs we have found indicate that commercial GPU compilers could benefit from the better testing we offer.

One means for assessing whether our method is fundamentally effective is to re-validate GLFuzz on successive driver versions that incorporate fixes for previous bugs detected by the tool, to see whether new bugs are discovered. We predict that, relatively quickly, a given shader compiler may become “immune” to the current program transformations that GLFuzz employs (especially if vendors incorporate our bug reports into their regression suites), but that equipping the tool with more refined or additional program transformations will be effective in finding new bugs. On the other hand, a large number of the issues we found are likely caused by buggy compiler optimizations; the compiler developers are likely to continue adding and updating optimizations throughout the lifetime of the compiler, leading to new bugs or regressions that may not manifest using the variants we tested, particularly in the case of reduced variants. Thus, continued testing with even just the current program transformations may provide lasting benefit.

Related to the program transformations we have investigated so far: of the endless program transformations one could imagine, we chose these based on their ease of implementation and our hypothesis from prior work that complicating control flow may be effective in triggering compiler bugs [Le et al. 2014]. As with any testing method, our approach is incomplete. There will exist shader compiler bugs that our current transformations could find *in principle*, but with very low probability, e.g. if the bug requires multiple transformations to be combined in a specific way, and applied to existing code with a very particular form, in order to trigger. There will also exist shader compiler bugs that *cannot* be triggered by our current set of transformations, because they require a shader to exhibit structural features that our transformations cannot induce. Knowledge of the root causes of typical shader compiler bugs, mined e.g. from the bug tracker of the open source Mesa 3D project,¹¹ could provide inspiration for additional program transformations.

Other near-term future work includes applying our method to additional real-time shader languages, in particular HLSL and SPIR-V, and testing open source graphics drivers (e.g. the Mesa drivers, for which we have commenced testing activities). A longer-term avenue is to apply our method to offline rendering scenarios, generating equivalent families of shaders or input scenes

¹⁰<https://android-review.googlesource.com/#/c/platform/external/deqp/+375574/>

¹¹<https://www.mesa3d.org/bugs.html>

for software such as RenderMan, and even to other domains, generating inputs e.g. to hardware-accelerated video decoders.

ACKNOWLEDGMENTS

Our involvement in the ICURe Programme was instrumental in building relationships in the GPU industry that allowed us to report issues to relevant engineers, and we thank the many engineers and users who provided feedback on our bug reports, via email, forum posts, and in response to our blog series [Donaldson 2016]. We thank Tillmann Karras for help understanding the GCN3 assembly code of Figure 3. We are grateful to Daniel Liew and Tyler Sorensen for participating in the user study described in Section 5.3, and again to Tyler Sorensen, as well as the anonymous reviewers, for feedback on an earlier draft of our paper.

REFERENCES

- AMD. 2016. Graphics Core Next Architecture, Generation 3. (2016). <http://gpuopen.com/compute-product/amd-gcn3-isa-architecture-manual/>.
- Android Community. 2017. OpenGL ES Testing. (2017). <http://source.android.com/devices/graphics/testing.html>.
- Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Software Eng.* 41, 5 (2015), 507–525.
- A.S. Boujarwah and K. Saleh. 1997. Compiler test case generation methods: a survey and assessment. *Information and Software Technology* 39, 9 (1997), 617 – 625.
- C.J. Burgess and M. Saidi. 1996. The automatic generation of test cases for optimizing Fortran compilers. *Information and Software Technology* 38, 2 (1996), 111 – 119.
- Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to prioritize test programs for compiler testing. In *Proc. International Conference on Software Engineering*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 700–711. <http://dl.acm.org/citation.cfm?id=3097451>
- Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An empirical comparison of compiler testing techniques. In *Proc. International Conference on Software Engineering*. ACM, 180–190.
- T.Y. Chen, S.C. Cheung, and S.M. Yiu. 1998. *Metamorphic testing: a new approach for generating next test cases*. Technical Report HKUST-CS98-01. Hong Kong University of Science and Technology.
- Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 197–208.
- Context. 2011. WebGL: More WebGL Security Flaws. (2011). <https://www.contextis.com/resources/blog/webgl-more-webgl-security-flaws/>.
- Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. 2012. Testing Static Analyzers with Randomly Generated Programs. In *Proc. NASA Formal Methods Symposium*. Springer, 120–125.
- Alastair F. Donaldson. 2016. Crashes, Hangs and Crazy Images by Adding Zero: Fuzzing OpenGL Shader Compilers. (2016). https://medium.com/@afd_icl/crashes-hangs-and-crazy-images-by-adding-zero-689d15ce922b.
- Alastair F. Donaldson and Andrei Lascu. 2016. Metamorphic Testing for (Graphics) Compilers [Short Paper]. In *Proc. International Workshop on Metamorphic Testing*. ACM, 44–47.
- Alastair F. Donaldson and Paul Thomson. 2017. Automated Testing of Graphics Shader Compiler: Video Illustration of Security Bugs. (2017). <https://youtu.be/d3CNfMoP2t8>.
- Google. 2017. ANGLE: Almost Native Graphics Layer Engine. (2017). <https://chromium.googlesource.com/angle/angle>.
- Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. 2012. Swarm testing. In *Proc. International Symposium on Software Testing and Analysis*. ACM, 78–88.
- Ralph Guderlei and Johannes Mayer. 2007. Towards Automatic Testing of Imaging Software by Means of Random and Metamorphic Testing. *International Journal of Software Engineering and Knowledge Engineering* 17, 6 (2007), 757–781.
- Nicolai Hähnle. 2017. Piglit - OpenGL driver testing framework. (2017). <https://people.freedesktop.org/~nh/piglit/>.
- Tahir Jameel, Mengxiang Lin, and Liu Chao. 2016. Metamorphic Relations Based Test Oracles for Image Processing Applications. *International Journal of Software Innovation* 4, 1 (2016), 16–30.
- John Kessenich, Dave Baldwin, and Randi Rost. 2016a. The OpenGL Shading Language, Language Version 4.50. (2016). <https://www.opengl.org/registry/doc/GLSLangSpec.4.50.pdf>.
- John Kessenich, Boaz Ouriel, and Raun Krisch. 2016b. SPIR-V Specification (Provisional). (2016). <https://www.khronos.org/registry/spir-v/specs/1.1/SPIRV.pdf>.

- John Kessenich, Graham Sellers, and Dave Shreiner. 2016c. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V* (9 ed.). Addison-Wesley.
- Khronos Group. 2014. WebGL Specification, Version 1.0.3. (2014). <https://www.khronos.org/registry/webgl/specs/1.0/>.
- Khronos Group. 2015. Khronos Invites Industry Participation to Create Safety Critical Graphics and Compute Standards. <https://www.khronos.org/news/press/>. (August 2015).
- Khronos Group. 2016. Vulkan 1.0.38 – A Specification. (2016). <https://www.khronos.org/registry/vulkan/specs/1.0/pdf/vkspec.pdf>.
- Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 216–226.
- Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proc. ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 386–399.
- Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. 2014. Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities. In *Proc. IEEE Symposium on Security and Privacy*. IEEE, 19–33.
- Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core compiler fuzzing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 65–76.
- William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.
- Microsoft. 2017a. Direct3D 12 Programming Guide. (2017). [https://msdn.microsoft.com/en-us/library/windows/desktop/dn899121\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dn899121(v=vs.85).aspx).
- Microsoft. 2017b. HLSL. (2017). [https://msdn.microsoft.com/en-us/library/windows/desktop/bb509561\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/bb509561(v=vs.85).aspx).
- Multicore Programming Group. 2017. A collection of shader compiler bugs. (2017). <http://github.com/mc-imperial/shader-compiler-bugs>.
- Moritz Pflanzner, Alastair F. Donaldson, and Andrei Lascu. 2016. Automatic Test Case Reduction for OpenCL. In *Proc. International Workshop on OpenCL*. ACM, 1:1–1:12.
- John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 335–346.
- R.S. Scowen and Z.J. Ciechanowicz. 1983. Compiler validation—a survey. In *PASCAL Compiler Validation*, B.A. Wichmann and Z.J. Ciechanowicz (Eds.). Wiley-Blackwell, Chapter 13, 90–144.
- SecurityWeek. 2016. Code Execution Flaw Plagues Intel Graphics Driver. (2016). <http://www.securityweek.com/code-execution-flaw-plagues-intel-graphics-driver>.
- Sergio Segura, Gordon Fraser, Ana B. Sánchez, and Antonio Ruiz Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Trans. Software Eng.* 42, 9 (2016), 805–824.
- Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proc. ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 849–863.
- Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. 2010. An Automatic Testing Approach for Compiler Based on Metamorphic Testing Technique. In *Proc. Asia Pacific Software Engineering Conference*. IEEE, 270–279.
- Ilja van Sprundel. 2014. Windows Kernel Graphics Driver Attack Surface. (2014). <https://www.blackhat.com/docs/us-14/materials/us-14-vanSprundel-Windows-Kernel-Graphics-Driver-Attack-Surface.pdf>.
- Elaine J. Weyuker. 1982. On Testing Non-Testable Programs. *Comput. J.* 25, 4 (1982), 465–470.
- Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 283–294.
- Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200.