2023 IEEE Conference on Software Testing, Verification and Validation (ICST) | 978-1-6654-5666-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICST57152.2023.00042

Industrial Deployment of Compiler Fuzzing Techniques for Two GPU Shading Languages

Alastair F. Donaldson*†, Ben Clayton*, Ryan Harrison*, Hasan Mohsin†, David Neto*, Vasyl Teliman‡, Hana Watson†
*Google
†Imperial College London, UK
‡Independent, Ukraine

***Abstract*—We report on our experience at Google deploying a variety of coverage-guided and black-box fuzzers to find bugs in compilers for two graphics shading languages: the WebGPU Shading Language (WGSL) and Standard Portable Intermediate Representation (SPIR-V). We discuss the deployment of coverage-guided fuzzing on ClusterFuzz and OSS-Fuzz using libFuzzer's built-in mutators, and a number of custom mutators that exploit knowledge of the syntax and semantics of WGSL and SPIR-V. We also discuss the deployment of several black box fuzzers on ClusterFuzz, including two that are based on new randomised program generators for WGSL, which we have also used in a targeted fashion for end-to-end testing of WebGPU implementations. We discuss a series of insights arising from our experience that we hope will be valuable to practitioners and researchers interested in applying fuzzing to industrial problems.**

***Index Terms*—Fuzzing, compilers, graphics shaders, WebGPU, WGSL, SPIR-V**

I. Introduction

Randomised testing, commonly known as *fuzzing*, has become a critical defence against defects and vulnerabilities in software systems, especially systems written in C and C++, due to the unsafe features that these languages offer.

WebGPU [47] is a new JavaScript API that allows graphics and compute workloads to be accelerated using client-side GPUs. To avoid the potential for denial of service or remote execution attacks arising from untrusted web pages issuing malicious workloads to end-user GPUs, it is critical that an in-browser implementation of WebGPU is robust. One of the most complex components of a WebGPU implementation is a *compiler* that translates code written in the new WebGPU Shading Language (WGSL) into a suitable GPU programming language supported by the user's system, thus techniques for thoroughly testing a WGSL compiler to quickly find and eliminate potentially exploitable bugs are important.

Separate but related, Google has developed the SwiftShader software renderer [22], which is used as a fall-back renderer for implementations of the existing WebGL API [33] (to which WebGPU has been described as a successor) as well as for WebGPU. SwiftShader allows in-browser graphics workloads to be executed by the end user's CPU, e.g. due to the GPU in the user's system being too old to meet the needs of WebGPU. SwiftShader is a conformant implementation of the Vulkan GPU programming model [32], in which graphics shaders are expressed in Standard Portable Intermediate Representation (SPIR-V) [29], thus SwiftShader incorporates functionality for processing and optimising SPIR-V programs. Due to its use in web browsers, SwiftShader may be presented with arbitrary, malicious graphics shaders.

We report on our experience at Google employing a variety of fuzzing techniques to find defects in (a) compilers for WGSL, and (b) tools for optimising and validating SPIR-V. As both are used to enable WebGL and WebGPU in popular browsers such as Chrome, Edge and Firefox, bugs in these components have major potential for negative impact.

After providing necessary background (§II), the paper is structured as follows:

Deploying coverage-guided fuzzing (§III). We explain how we have deployed a variety of coverage-guided[1] fuzzers to WGSL and SPIR-V processing tools, as well as the design of various custom mutators for WGSL.

Deploying black-box fuzzing (§IV). We have written two specialised program generators for WGSL that allow differential and validation testing of WGSL compilers, which we have also deployed on ClusterFuzz, alongside a black-box fuzzer based on an existing GPU shading language fuzzer.

Summary of bugs found (§V). Deployment of fuzzing on ClusterFuzz and OSS-Fuzz since 2018 has led to 88 security-critical issues being reported (86 fixed at time of writing), and targeted fuzzing of `tint` and `naga` has led to the discovery of 58 functional bugs (36 at time of writing).

Insights (§VI). We reflect on take-aways from our experience, including cases when our early fuzzing efforts informed the design WGSL or the implementation of the `tint` compiler, and the trade-off between the benefits of domain-aware custom mutators and the effort required to create and maintain them.

Throughout, when we talk about the current status of issues found by various fuzzers, or say "at time of writing", this refers to November 2022.

II. Background

A. Coverage-guided fuzzing with libFuzzer and ClusterFuzz

libFuzzer. The libFuzzer tool [36] is an in-process coverage-guided mutation-based fuzzer for C and C++ software com-

[1]Throughout, we use *coverage* to mean *branch coverage*.

978-1-6654-5666-1/23/$31.00 ©2023 IEEE
DOI 10.1109/ICST57152.2023.00042

ponents. This style of fuzzing can be viewed as a genetic algorithm where new tests cases are derived via mutation, and code coverage serves as the fitness function.

To use libFuzzer, one must write a *fuzz target*: a function that receives a buffer of bytes, interprets the byte buffer as an input to the software under test (SUT), and invokes the SUT using this input. This fuzz target is then called repeatedly by libFuzzer using a series of input buffers.

The tool is "mutation-based" because an input buffer is obtained by *mutating* some previous input buffer, starting from a user-provided *initial corpus*, or with a default initial buffer if no corpus is given. Inputs are mutated using a number of built-in mutation operators that work on byte buffers in a domain-agnostic manner, for example deleting, swapping or changing bytes, or combining multiple previously-used inputs. If the input format is text-based, the user can provide a dictionary of relevant tokens that libFuzzer's built-in mutators can use, e.g. allowing them to insert, delete or substitute tokens. In the genetic programming analogy, dictionaries provide building blocks analogous to base pairs / genes, while a corpus can be seen as a set of complete structures, analogous to chromosomes / DNA strands, to mutate.

An input is prioritised for further mutation if it leads to new source code coverage of the SUT; this is what makes libFuzzer "coverage-guided".

The fuzz target, SUT and the overall `main` method (which performs corpus maintenance, input selection and mutation) are all linked together into the same executable—this is what makes libFuzzer an "in-process" fuzzer. To facilitate this, using libFuzzer requires compiling the SUT using a special flag, supported by the Clang/LLVM compiler framework.

The *oracle* [3] used for detecting bugs during fuzzing is the *crash oracle*, where a bug is deemed to have been found if the SUT crashes. When using libFuzzer one typically boosts this oracle using a *sanitiser* (e.g. AddressSanitizer, for detecting buffer overflows [45]). The fuzz target must be designed so that the (sanitizer-boosted) crash oracle is precise: if the SUT crashes when invoked by the fuzz target, this must correspond to a genuine bug, otherwise the use of libFuzzer will raise false alarms. If the SUT is only supposed to deal gracefully with inputs that satisfy certain preconditions, the fuzz target must take responsibility for discarding inputs that do not satisfy these preconditions (or for transforming them so that they do). We revisit this point in §VI-C, Insight 9.

Custom mutators. The built-in libFuzzer mutators may be ineffective at yielding interesting *valid* inputs for the SUT. To overcome this, the user can provide a *custom mutator*: a function that takes a buffer of bytes and returns the byte buffer in a mutated form [19]. The custom mutator can process the byte buffer according to the known input format of the SUT and mutate it accordingly.

ClusterFuzz and OSS-Fuzz. ClusterFuzz refers to both an open source project from Google providing a software framework for continuous fuzzing [16] (using various fuzzers including libFuzzer), as well as Google's large-scale internal deployment of this framework [2]. Google also provides a fuzzing service for the open source community based on the ClusterFuzz framework, called OSS-Fuzz [21].

ClusterFuzz takes care of managing fuzzing jobs that run indefinitely, with functionality for automatically reducing bug-triggering test cases, determining whether bugs are reproducible, de-duplicating bug reports, auto-reporting non-duplicate bugs that reliably reproduce, bisecting revision histories to try to pinpoint the change that introduced a bug, and re-running bug-triggering test cases as changes are pushed to the SUT, auto-closing issues that appear to have been fixed.

TABLE I
SHADING LANGUAGES FOR VARIOUS GRAPHICS APIS

Graphics API	Shading language	Acronym
WebGPU	WebGPU Shading Language	WGSL
Vulkan	Standard Portable Intermediate Representation	SPIR-V
Direct3D	High Level Shading Language	HLSL
Metal	Metal Shading Language	MSL
OpenGL ES	OpenGL Shading Language	GLSL

B. The WGSL and SPIR-V languages and tool ecosystems

WebGPU, WGSL, and WGSL compilers. WebGPU is an emerging standard for graphics programming on the web [47], exposed as a JavaScript API. A web page using WebGPU makes a number of JavaScript calls to create buffers for communication between host and GPU, copy memory to and from these buffers, and run either a graphics or compute pipeline—comprising one or more *shader* programs—on the GPU. A shader is a massively parallel program that reads from an input buffer and either renders to a framebuffer, or writes the computed result to an output buffer.[2] At the heart of WebGPU is a new graphics shading language, the WebGPU Shading Language (WGSL) [46]. WGSL shares many similarities with other high level shading languages such as the OpenGL Shading Language (GLSL) from the Khronos Group [28], and Microsoft's High Level Shading Language (HLSL) [39], but has been tailored specifically to meet the needs of the WebGPU API.

An implementation of WebGPU in a web browser must leverage a graphics API native to the host system. These native APIs have no knowledge of WGSL. Instead, each expects shaders to be presented in a particular shading language: SPIR-V [29] for Vulkan (used on Android/Linux), HLSL [39] for Direct3D (used on Windows), the Metal Shading Language (MSL) [1] for Metal (used on iOS/OSX), and the OpenGL Shading Language (GLSL) [28] for OpenGL ES (used on some older Android devices). Table I provides a summary of these shading language acronyms. A portable WebGPU implementation must thus include a *shader compiler* with a WGSL front end, and back-ends for each of SPIR-V, HLSL and MSL. (If older devices are to be supported, a back-end

[2]This covers two important kinds of shaders: *fragment* and *compute* shaders; a graphics pipeline features various other shader stages such as *vertex* shaders.

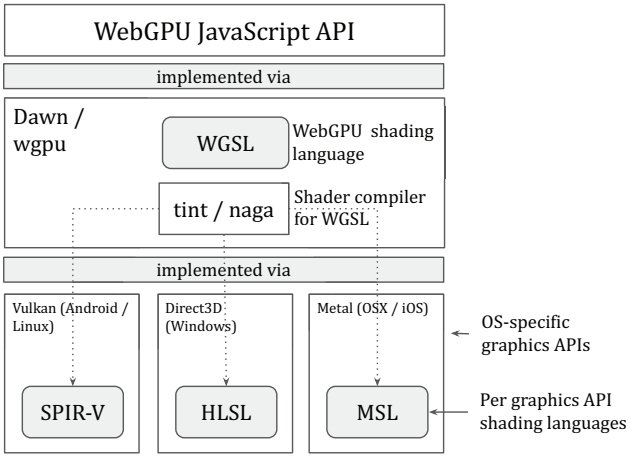

Fig. 1. Illustration of a WebGPU implementation built on top of a native graphics API

TABLE II
SUMMARY OF VARIOUS SHADER TRANSLATION TOOLS

Translator	Relevant front-ends	Relevant back-ends
tint	WGSL, SPIR-V	HLSL, MSL, SPIR-V, WGSL
naga	WGSL, SPIR-V, GLSL	HLSL, MSL, SPIR-V, WGSL
glslang	GLSL	SPIR-V
spirv-opt	SPIR-V	SPIR-V

for GLSL may also be needed, but for brevity we henceforth do not discuss support for OpenGL ES.)

Figure 1 gives an overview of how a WebGPU implementation is structured, with details of current open source implementations. The Dawn project from Google [18], written in C++, provides a WebGPU implementation for Chromium-based browsers (including Chrome and Edge), and features the tint shader compiler. The wgpu project [43], written in Rust, provides a WebGPU implementation for Firefox, and features the naga shader compiler.

The tint and naga translators also provide pathways for ingesting code in other shading languages and emitting WGSL: tint has a SPIR-V front-end and naga has front-ends for SPIR-V and GLSL. This is to make it easier for graphics developers to port their existing shader assets to WGSL. The tint and naga entries of Table II summarise the front-end and back-end languages for these tools that are relevant for the current paper (we discuss glslang and spirv-opt below).

The non-WGSL front-ends for these tools do not form part of a WebGPU implementation and thus will not be invoked from inside a web browser. While it is important to test them to improve their reliability, they are lower-priority for fuzzing compared with the WGSL front-ends.

SwiftShader and SPIRV-Tools. As discussed in §I, SwiftShader is a CPU implementation of Vulkan that serves as a fall-back renderer in Chromium-based web browsers when the client-side GPU is too old to meet the needs of WebGL or WebGPU. SwiftShader directly supports execution of shaders written in Vulkan's SPIR-V shading language, and can be used to execute GLSL shaders by first translating them to SPIR-V using the glslang tool from the Khronos Group [30].

Before executing a SPIR-V shader, SwiftShader pushes the shader through spirv-opt, the SPIR-V *optimiser*, which performs a number of target-agnostic optimisations that make the SPIR-V code simpler to process. For example, SwiftShader uses spirv-opt to fully inline all function calls so that the SwiftShader execution engine can assume all shaders are call-free (this works because the Vulkan dialect of SPIR-V does not allow recursion [32, VUID-StandaloneSpirv-None-04634]).

The spirv-opt tool is part of SPIRV-Tools, a collection of tools for working with SPIR-V [31]. Another tool in this collection is spirv-val, which is currently used as part of the tint compiler to check whether code emitted by its SPIR-V back-end is valid. Due to their security-related importance as part of the front-end of SwiftShader and the validation stage of tint, we have focused significant attention of using fuzzing to find bugs in spirv-opt and spirv-val. For thoroughness, we have also applied fuzzing to spirv-as and spirv-dis, an assembler and disassembler for SPIR-V that form part of SPIRV-Tools.

The final two rows of Table II complete the picture of shader translation tools discussed in this paper, with entries for glslang and spirv-opt.

III. COVERAGE-GUIDED FUZZING

We discuss our experience writing libFuzzer targets and various sorts of custom mutators to target the tint WGSL compiler, and the SPIRV-Tools suite of tools.

A. Fuzz targets and standard mutators

Fuzz targets for SPIRV-Tools. For SPIRV-Tools, a set of libFuzzer targets were deployed starting from August 2018. These comprised fuzz targets for the optimiser (spirv-opt), validator (spirv-val), and assembler/disassembler tools (spirv-as/spirv-dis). These fuzz targets treat their input buffer as SPIR-V binary, with the exception of the spirv-as fuzz target, which treats its input buffer as a textual representation of SPIR-V assembly. These fuzz targets were deployed on ClusterFuzz until the end of August 2021, but were moved to OSS-Fuzz in September 2021 for reasons discussed in §VI-A.

Fuzz targets for tint. Fuzzing with libFuzzer was deployed for tint from the start of the project in March 2020. Recall from §II-B that tint has two front-ends, for WGSL and SPIR-V, and four back-ends, for SPIR-V, MSL, HLSL and WGSL. Initially 10 main fuzz targets were deployed:

- X-reader, for $X \in \{\text{WGSL}, \text{SPIR-V}\}$, and
- X-reader-Y-writer, for $X \in \{\text{WGSL}, \text{SPIR-V}\}$ and $Y \in \{\text{SPIR-V}, \text{HLSL}, \text{MSL}, \text{WGSL}\}$.

Each of the WGSL-reader[-Y-writer] fuzz targets treats its input as a WGSL string and attempts to parse the string and (if successful) type-check the resulting AST. The reader-only fuzz targets stop at this point, while the reader-writer

fuzz targets go further by applying a number of backend-sanitising transformations to the AST and emitting target code in the associated output language. The SPIR-V-reader[-Y-writer] fuzz targets work similarly, accepting SPIR-V binary instead of WGSL text.

For a given input language X and output language Y, the X-reader-Y-writer fuzz target exercises *at least* the functionality of the X-reader fuzz target. The purpose of the X-reader fuzz targets is to ease bug triage. If ClusterFuzz reports a bug triggered by, say, the WGSL-reader fuzz target, this bug *must* pertain to WGSL parsing or type-checking, rather than code generation. For similar reasons, we deployed separate fuzz targets per `tint` back-end, so that a bug found by, say, the WGSL-reader-MSL-writer fuzz target cannot pertain to the generation of output in a language other than MSL. Furthermore, a bug found by one fuzz target can be checked against other fuzz targets during triage. For example, a crash found by the WGSL-reader-HLSL-writer fuzz target can be checked against the WGSL-reader fuzz target. If the crash still occurs, the issue is likely related to WGSL parsing or type-checking (exercised by both fuzz targets). Otherwise, the other WGSL-reader-Y-writer fuzz targets can be checked, to see whether the crash is specific to HLSL code generation, or related to back-end-agnostic transformations.

We also developed a number of more fine-grained fuzz targets. For example, `tint` provides functionality to clone an abstract syntax tree (AST). Knowing from experience that cloning code can be error-prone, we deployed a special fuzz target that parses its input into an AST and—if the AST is confirmed to be well-typed—clones it.

B. Custom mutators

The built-in, domain-agnostic mutators that libFuzzer uses by default tend to lead to highly malformed inputs. These can be effective in triggering crashes and vulnerabilities in error handling code, but are less likely to exercise type-checking, AST transformation and code generation functionality. We thus designed a number of custom mutators geared towards generating well-formed or almost-well-formed programs.

Regex-based mutator. We designed a lightweight custom mutator based on matching and replacing patterns of text in a piece of WGSL-like code. Our goals for this custom mutator were threefold. First, it should be fast to develop and easy to maintain. Second, it should have a much higher chance of preserving the well-formedness of a WGSL shader compared to libFuzzer's default domain-agnostic mutators. Third, it should have a good chance of generating interesting "almost-valid" test cases: WGSL shaders that are invalid either due to being ill-typed or syntactically correct, but subtly so. In these regards, our regex-based mutator is similar to the Universal Mutator tool [23], [24], which we discuss further in §VII, though it was conceived of and developed independently.

Given a (possibly invalid) piece of WGSL, presented as a text buffer, the regex mutator applies a mutation operator chosen uniformly at random from the set of operators summarised in Table III. Various heuristics are used to decide whether a given index into the input buffer is *probably* part of a function or loop body. For example, to look for a program point that is probably part of a loop, the "Insert break/continue" mutation operator uses the following regex:

```
[^a-zA-Z_0-9](for|while|loop)[^\{]*\{
```

The `'[^a-zA-Z_0-9](for|while|loop)'` part matches a region of text starting with one of the WGSL loop keywords. The `'[^\{]*\{'` part matches any text up to and including the first occurrence of the `'{'` character following this; this is *likely* to be the opening brace corresponding to the body of a loop associated with the previous keyword. The regex mutator then examines the remainder of the input looking for a corresponding `'}'` character, ignoring all balanced pairs of `'{'` and `'}'` characters that occur in-between. If found, this is *likely* to signify the end of the loop body.

Matching schemes such as this are easy to implement. While having obvious disadvantages, such being confounded by text that occurs in comments, they have the advantage that they can be applied to *invalid* WGSL-like inputs. This means that if an *almost*-valid WGSL-like input achieves extra coverage, it might be possible to mutate this input into a form that achieves even more coverage, something that would not be possible if the mutator took a more principled approach that assumed syntactically correct inputs.

AST-based mutator. Nevertheless, we did also design a more principled mutator that produces well-formed programs by construction, by carefully mutating the abstract syntax tree (AST) of a well-formed WGSL program. This mutator rejects invalid inputs. For an input that is successfully parsed to a well-typed AST, the mutator randomly applies one of the operators summarised in Table IV to the AST.

The "Change binary operator" and "Change unary operator" mutations of Table IV can be viewed as principled versions of the "Replace operator" mutation of Table III. To illustrate the differences, the regex mutator might identify an arbitrary occurrence of '−' and change it to '|' (bitwise-or). This would: have no semantic effect if the occurrence of '−' occurs in a code comment; yield an invalid program if '−' occurs in a context where one floating-point expression is being subtracted from another (because '|' cannot be applied to floating-point expressions); and also yield an invalid program if '−' occurs as a unary operator (because '|' cannot be used as a unary operator). In contrast, the AST-based mutator only considers mutating genuine operator occurrences appearing in the AST, and only considers well-typed replacements.

Similarly, while "Delete statement" in Table IV may seem related to "Delete interval" in Table III, the difference is that the AST-based mutator's "Delete statement" operator carefully looks for a statement that can be deleted without making the program become invalid (for example, it will not delete a variable declaration if this would invalidate statements that refer to the declared variable), while "Delete interval" is much more aggressive: it has a reasonably high probability

TABLE III
TEXT-LEVEL MUTATION OPERATORS SUPPORTED BY OUR REGEX-BASED MUTATOR

Mutation name	Example before	Example after	Notes
Swap intervals	`A; B; C;`	`A; C; B;`	An interval is the portion of text between a pair of (not-necessarily successive) '`;`' tokens; `A` and `B` denote arbitrary fragments of code
Delete interval	`A; B; C;`	`A; C;`	Similar
Duplicate interval	`A; B; C;`	`A; B; A; C;`	Similar
Replace identifier	`a = b + c;`	`a = b + angle;`	The replacement identifier (`angle` here) is chosen based on identifiers mined from the file
Replace literal	`x = 42;`	`x = -1;`	The replacement literal (`-1` here) is chosen based on literals mined from the file
Insert return	`A; B;`	`A; return ...; B;`	If it looks like the enclosing function is non-void, a literal is suggested
Replace operator	`x + y`	`x * y`	The mutator does not attempt to respect the argument types of operators
Insert break/continue	`A; B;`	`A; break; B;`	Only applied if it appears that the code fragment is in a loop
Replace function call with built-in	`foo(x)`	`sin(x)`	The parentheses after `foo` make it likely to be a function invocation
Add swizzle	`v = w.x;`	`v = w.xxy.x;`	The application of '`.x`' to '`w`' suggests that it is vector

TABLE IV
AST-LEVEL MUTATION OPERATORS SUPPORTED BY OUR AST-BASED MUTATOR

Mutation name	Example before	Example after	Notes
Change binary operator	$\mathsf{expr}_1 \circ \mathsf{expr}_2$	$\mathsf{expr}_1 \bowtie \mathsf{expr}_2$	Binary operators $\circ$ and $\bowtie$ are type-compatible
Change unary operator	$\circ\mathsf{expr}$	$\bowtie \mathsf{expr}$	Unary operators $\circ$ and $\bowtie$ are type-compatible
Insert unary operator	expr	$\circ\mathsf{expr}$	Unary operator $\circ$ can be applied to expressions of the type of expr
Delete statement	$\mathsf{stmt}_1; \mathsf{stmt}_2;$	stmt_2	stmt_1 does not declare variables that are later used, and removal of stmt_1 does not lead to a syntactically infinite loop or a function that no longer returns a value
Replace identifier	expr	$\mathsf{expr}[x/y]$	Identifier x occurs in expr, and identifier y is also in scope and has the same type as x

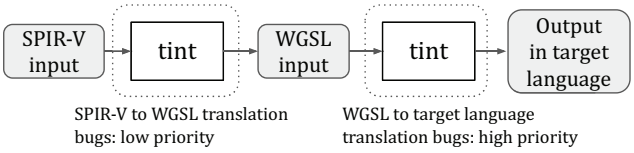

Fig. 2. Fuzz target for testing WGSL compilation via a SPIR-V shader

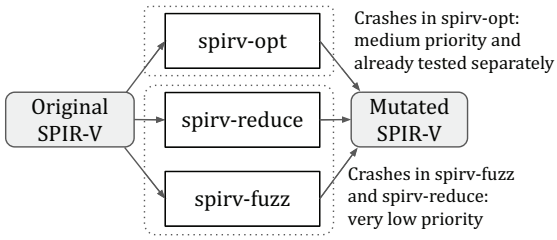

Fig. 3. Custom mutator that leverages SPIRV-Tools

of yielding an invalid program, but may also chance upon an interesting interval deletion that happens to preserve validity (such as deleting all code from mid-way through one loop body to mid-way through the next, in a manner that happens to lead to all uses of variables referring to suitable declarations).

A SPIR-V-based mutator for WGSL. Recall from §II-B and Table II that tint features a front-end for SPIR-V in addition to WGSL. Because SPIR-V is a more mature language than WGSL, there exist a number of off-the-shelf tools for transforming SPIR-V programs. These tools—spirv-opt, spirv-fuzz and spirv-reduce—ship as part of the SPIRV-Tools project [31]. In addition to its default modes, the spirv-opt optimiser (see §II-B) exposes its optimisation passes as a library such that any pass can be applied in isolation. For instance, spirv-opt has an "eliminate-local-single-store" pass that will (according to the help option of the tool) "Replace stores and loads of function scope variables that are only stored once". A pass such as this can be seen as a custom SPIR-V to SPIR-V mutator. Likewise, two other tools, spirv-fuzz and spirv-reduce, feature transformations that can be exposed at a fine level of granularity: spirv-fuzz implements a number of semantics-preserving transformations on SPIR-V programs (as part of a metamorphic approach to compiler testing [14]), while spirv-reduce, a test case reducer for SPIR-V, implements various simplifying transformations.

We built a special fuzz target and custom mutator for testing tint's WGSL support *via SPIR-V*, as illustrated in Figures 2 and 3. The fuzz target, illustrated in Figure 2, treats its input as a SPIR-V binary. It uses tint to transform this into WGSL textual form, and then invokes tint *again* to turn the WGSL text into the desired output format. The SPIRV-Tools-based custom mutator is illustrated in Figure 3. Because the fuzz target accepts SPIR-V binary, the mutator accepts and returns SPIR-V binary. It randomly selects between applying a transformation offered by spirv-opt, spirv-fuzz or spirv-reduce and, for the chosen tool, applies one of its available transformations at random.

The *intention* of this fuzz target and custom mutator is to test tint's support for translating WGSL into various target languages. As a by-product, the fuzz target also tests tint's support for translating SPIR-V into WGSL, and the custom mutator indirectly tests spirv-opt, spirv-fuzz and spirv-reduce (as it may crash due to bugs in these tools). In the long term this is positive, because thorough testing of all these components is ultimately desirable. However, in the short term it presents a problem, as discussed further in §VI-A, Insight 6 and §VI-B, Insight 8, because bugs in these components have much lower priority for fixing compared with bugs in tint's WGSL-handling code.

C. Expanding input corpora

From its deployment in August 2018 until August 2021, the SPIRV-Tools fuzz targets described in §III-A were deployed with an initial corpus comprising a *single*, relatively simple SPIR-V program. Even with this trivial corpus, fuzzing did lead to the discovery of some defects. Revisiting the fuzzers to enhance this corpus fell by the wayside for a number of years. In August 2021, this initial corpus was augmented with 87 SPIR-V examples derived from shaders from the GraphicsFuzz project (OpenGL shaders compiled to SPIR-V using the glslang tool; see §II-B). We discuss the impact of this in §VI-A, Insight 6.

From the start of the tint project in March 2020 until July 2021, the tint fuzz targets were deployed with *no initial corpus*. One reason for this is that the WGSL language was in complete flux when the tint project started: a concrete syntax for the language was being developed, and the tint developers were getting started on designing an intermediate representation and back-end code generators based on a provisional input format. Writing a corpus for this input format that was expected to be subject to major change was not a priority. We revisit this point in §VI-A, Insight 3.

In July 2021, by which time the WGSL language had taken substantial shape, we improved the tint build system so that every WGSL shader included as a regression test in the tint repository is included in the initial corpus of every fuzz target that accepts WGSL input, and similarly for SPIR-V regression tests and SPIR-V-consuming fuzz targets. At time of writing these corpora are populated from 3,095 WGSL test shaders and 1,750 SPIR-V test shaders. It is possible that these corpora are now larger than is desirable for a coverage-guided fuzzing effort, and may benefit from corpus distillation [26].

The WGSL fuzz targets based on libFuzzer's built-in mutators also utilise a dictionary that includes all WGSL keywords, operators and symbols.

IV. Black-box fuzzing

We discuss two novel program generation tools for WGSL that we have used to test tint and naga in a targeted fashion, as well as the deployment of these and other black-box fuzzers on ClusterFuzz for additional testing of tint.

A. Targeted black-box fuzzing via program generation tools

Coverage-guided fuzzing using libFuzzer can be excellent for finding crash bugs, but is less effective at detecting deeper functional errors due to the limited nature of the crash oracle.

We thus decided to investigate the use of *differential testing* [38] to allow finding miscompilation bugs in WGSL compilers—cases where the compiler generates semantically invalid code without crashing in the process. Differential testing of compilers was popularised by the Csmith project [49], which generates random C programs that are, by construction, free from undefined behaviour. A generated program can then be compiled using multiple compilers that agree on implementation-defined behaviour, and the output obtained by running the associated executables can be compared. A mismatch indicates that (at least) one compiler has miscompiled the program, which can be investigated.

Two undergraduate students at Imperial College London worked on program generators for WGSL, leading to the wgslsmith [40], [41] and wgslgenerator [48] tools.

The focus of wgslsmith is specifically on differential testing to find miscompilation bugs. In order for the results of differential testing to be meaningful, the tool puts in place mechanisms to defend against undefined behaviour. We distinguish between *fundamental undefined behaviour*, where a shader may legitimately exhibit nondeterminism according to the WGSL specification (such as due to accessing arrays out-of-bounds), and *current undefined behaviour*, where a program should compute an unambiguous result according to the WGSL language specification, but may deviate from this due to current known limitations of WGSL compilers. To illustrate *current* undefined behaviour, WGSL mandates that the expression e_1/e_2 should yield e_1 in the case where e_2 evaluates to 0. However, at time of writing, WGSL compilers do not emit target code to enforce this behaviour, compiling a division expression to an associated division instruction in the target shading language, all of which regard division by zero as an undefined behaviour or error.

Taking inspiration from Csmith, wgslsmith works around undefined behaviour by using "safe math" wrapper functions to avoid issuing operations that would lead to *fundamental* or *current* undefined behaviour. As WGSL compilers mature, so that *current* undefined behaviour becomes less of a problem, the use of these wrapper functions can be reduced.

Our wgslgenerator tool currently assumes that WGSL compilers are not subject to *current* undefined behaviour; i.e. it generates programs that assume the semantics of WGSL is respected and does not take measures to avoid current implementation limitations in tint and naga. For this reason we have mainly used wgslgenerator to detect cases where tint and naga produce invalid code, because the results obtained by full-blown differential testing are confounded by these current implementation limitations.

Both wgslsmith and wgslgenerator share common infrastructure for (a) validating generated shaders using appropriate per-target-language validation tools (e.g. spirv-val for SPIR-V), and (b) executing valid shaders on top of the Dawn and wgpu APIs (see Figure 1). This allows differential testing between tint and naga, by comparing results obtained via Dawn vs. wgpu, as well as differential testing between different back-ends, e.g. by comparing the results obtained via Dawn's Direct3D back-end vs. Dawn's Vulkan back-end.

B. Black-box fuzzers on ClusterFuzz

The ClusterFuzz infrastructure is mainly concerned with the deployment of coverage-guided fuzzing (based on libFuzzer and related approaches). However, it also provides a *black-box* fuzzing mode [10]. To use this mode, one simply uploads an input generator as a pre-built binary. The entry point to this input generator must be a specially-named Python script that ClusterFuzz can invoke with command line arguments to request the generation of a specified number of inputs. ClusterFuzz then repeatedly invokes the input generator to obtain sets of inputs, and invokes the SUT on these inputs. Just like when fuzzing using libFuzzer, bugs are found via the (sanitizer-boosted) crash oracle (see §II-A).

We deployed ClusterFuzz black-box fuzzers that use wgslsmith and wgslgenerator to generate WGSL shaders (see §IV-A). We also deployed a fuzzer that leverages the spirv-fuzz tool [31] to generate a set of SPIR-V shaders (as described in [14]), which are then converted to WGSL using tint. In all cases, ClusterFuzz feeds a generated WGSL shader into an entry point that invokes each WGSL-reader-Y-writer fuzz target (see §III-A) in turn (where $Y \in \{\text{SPIR-V}, \text{HLSL}, \text{MSL}, \text{WGSL}\}$), aiming to expose crashes in the tint functionality that these fuzz targets exercise.

V. Summary of bugs found

A. Issues filed by fuzzers deployed on ClusterFuzz / OSS-Fuzz

Table V provides a summary of the status of all issues filed by the fuzzers that we have deployed on ClusterFuzz and OSS-Fuzz, as of 27 October 2022. This includes all issues filed via the use of coverage-guided fuzzers (§III) and black-box fuzzers deployed on ClusterFuzz (§IV-B), but *excludes* bugs found via targeted use of black-box fuzzers (§IV-A).

Under the **Fuzzer(s)** column, 'WGSL-reader[-Y-writer]' and 'SPIR-V-reader[-Y-writer]' refer to the standard fuzz targets for testing tint's support for WGSL and SPIR-V, respectively, discussed in §III-A; 'WGSL-misc' refers to the collection of fine-grained fuzz targets for WGSL mentioned briefly in §III-A; 'Regex-based fuzzer', 'AST-based fuzzer' and 'SPIRV-Tools-based fuzzer' refer to the fuzz targets that use the custom mutators discussed in §III-B; 'wgslsmith', 'wgslgenerator' and 'spirv-fuzz' refer to the black-box fuzzers deployed on ClusterFuzz discussed in §IV-B; and 'spirv-opt-fuzzer', 'spirv-val-fuzzer' and 'spirv-as/spirv-dis-fuzzer' refer to the fuzz targets for SPIRV-Tools discussed in §III-A.

For each group of fuzzers the table notes the component(s) that are tested, whether the fuzzer is coverage-guided or black-box, and in the former case whether it uses default or custom mutators. The **# Issues** column records the total number of issues ClusterFuzz/OSS-Fuzz has auto-filed for these fuzzers, with **(resolved)** recording how many have been resolved for any reason—this is largely due to related bugs being fixed, but in some cases is due to the issue having been flagged as a duplicate or marked as "won't fix" e.g. because it relates to a bug in a custom mutator that is low priority for fixing in its own right. The **# Repro.** column records how many of the total number of issues are found by ClusterFuzz/OSS-Fuzz to be reliably reproducible; we mark in bold the cases where some issues are not reliably reproducible. The **# Security** column records how many of the total issues are reproducible security issues; that is, ClusterFuzz/OSS-Fuzz has marked flagged them as being potentially security-critical (e.g. because they trigger buffer overflows), and the **(fixed)** column reports how many of these security issues have been fixed.

The data under **# Issues** and **(resolved)** should only be seen as giving ball-park figures for the bug-finding ability of the fuzzers we have deployed and the priority of filed issues for fixing. First, the data are subject to problems of duplication (e.g. resource-exhaustion bugs are hard to de-duplicate and thus may be reported by multiple fuzzers) and false alarms (e.g. due to configuration problems in fuzz targets, and bugs in custom mutators). Second, the fuzzers were deployed at different points in time, so that fuzzers deployed early had more chance to detect "low hanging fruit" issues compared with fuzzers deployed later.

Bugs found using default mutators vs. custom mutators. The first four rows of Table V show that the straightforward fuzz targets for WGSL support in tint, and the specialised fuzz targets for checking particular aspects of WGSL support (such as testing of cloning functionality, discussed in §III-A), led to a higher issue rate compared with fuzz targets using the regex- and AST-based mutators. We have not delved into the reasons for this in depth, but our hypotheses include: these fuzz targets were deployed on ClusterFuzz before the fuzz targets that use custom mutators; the overhead of custom mutators means that fuzz targets that use default mutation have higher throughput; although the custom mutators have the benefit of focusing testing on mid- and back-end functionality by generating valid or almost-valid inputs, they miss out, in return, on finding front-end bugs via extremely malformed inputs; and overlap between the bug-finding ability of various fuzz targets, combined with the higher throughput of fuzz targets that use default mutators, may mean that the latter get

TABLE V
SUMMARY OF THE NUMBER OF ISSUES OPENED BY EACH GROUP OF FUZZERS THAT WE DEPLOYED ON CLUSTERFUZZ AND OSS-FUZZ

Fuzzer(s)	Component(s) tested	Nature of fuzzer	Mutator	# Issues	(resolved)	# Repro.	# Security	(fixed)
WGSL-reader[-Y-writer]	tint (WGSL)	Coverage-guided	Default	64	64	64	2	2
WGSL-misc	tint (WGSL)	Coverage-guided	Default	105	105	105	17	17
Regex-based fuzzer	tint (WGSL)	Coverage-guided	Custom	32	32	**28**	6	6
AST-based fuzzer	tint (WGSL)	Coverage-guided	Custom	19	17	**18**	5	5
wgslsmith	tint (WGSL)	Black-box	N/A	1	1	1	1	1
wgslgenerator	tint (WGSL)	Black-box	N/A	2	2	2	0	0
spirv-fuzz	tint (WGSL)	Black-box	N/A	2	2	2	1	1
SPIRV-Tools-based fuzzer	tint (SPIR-V + WGSL)	Coverage-guided	Custom	58	41	**43**	3	2
SPIR-V-reader[-Y-writer]	tint (SPIR-V)	Coverage-guided	Default	119	107	119	21	21
spirv-opt-fuzzer	spirv-opt	Coverage-guided	Default	259	246	**254**	26	25
spirv-val-fuzzer	spirv-val	Coverage-guided	Default	24	23	24	2	2
spirv-as/spirv-dis-fuzzer	spirv-as and spirv-dis	Coverage-guided	Default	19	18	**17**	4	4
All	All components			704	658	677	88	86

credit for quickly discovering easy-to-find bugs.

With respect to reproducible security issues, custom mutators show more promise. Fuzzers based on standard fuzz targets led to the reporting of 2 security-related issues out of 64 reports, while 6 out of 32 and 5 out of 19 issues reported by fuzz targets using the regex-based mutator and AST-based mutator, respectively, were security-related.

B. Bugs found via targeted black-box fuzzing

While the results for deployment of black-box fuzzing on ClusterFuzz shown in Table V are rather disappointing, we had significant success using wgslsmith and wgslgenerator in a targeted manner to find bugs in tint and naga. Via wgslsmith we were able to find and report 11 bugs in tint, of which 9 have been fixed so far, and 18 bugs in naga, of which 9 have been fixed so far. Using wgslgenerator we were able to find and report 6 bugs in tint, of which 3 have been fixed so far, and 23 bugs in naga (or wgpu more generally), of which 15 have been fixed so far. The major advantage of our targeted use of black-box fuzzing compared with deployment of black-box fuzzing on ClusterFuzz is that the targeted approach uses a stronger oracle. We discuss this further in §VI-C, Insight 11.

VI. INSIGHTS

We discuss insights arising from our experience, related to the timeliness of applying fuzzing (§VI-A), use of custom mutators (§VI-B), and oracle-related issues (§VI-C). Some of these insights are already well known to engineers and researchers who have worked with fuzzing in practice, but we believe there is value in bringing them to the attention of readers who have less experience working with fuzzing, and discussing them explicitly in our context for those who do.

A. Timeliness of applying fuzzing

Insight 1: Early fuzzing can inform language design

An innovative feature of WGSL is the notion of *abstract* numeric values. Similar to constants in Go [15], abstract numeric types allow compile-time constant expressions to be evaluated at 64-bit precision and then concretised to various lower-precision numeric types when they are used in dynamic contexts. The WGSL specification lists the scenarios where an abstract numeric value needs to be concretised. The WGSL-reader-MSL-writer fuzz target triggered a code generation error due to an abstract vector being indexed using statically-unknown value. The WGSL specification did not mandate that the abstract vector should be concretised in such a scenario. In response to the fuzzer-reported bug [5] a member of the tint development team opened a WGSL specification issue [9] which led to the language specification being fixed to account for this missing case [42].

Insight 2: Fuzzing can drive implementation choices

WGSL features constant array expressions, which can either enumerate the values of an array, or specify that array elements should default to 0. An early implementation of constant propagation in tint handled array constants on an element-by-element basis. Our fuzzers quickly identified a problem with this implementation by synthesising a shader that declared a huge, default-initialised array, causing a memory-out in tint's constant propagation routine. In response, the tint developers provided a more nuanced implementation that handles default-initialised structures efficiently [8].

Insight 3: Fuzzing can be useful even before an input format has been finalised

As discussed in §III-C, our tint fuzzers were initially not provided with a corpus of input examples. This was in part because, at that time, the WGSL language was too unstable for reliable examples to be available. Although the bug-finding ability of our fuzzers exhibited a step-change when we introduced an input corpus, our fuzzers nevertheless found a number of important bugs by starting from a default buffer. Thus "our input format is not finalised yet" should not be seen as an excuse for delaying the deployment of fuzzers.

Insight 4: A changing input format creates a fuzzer maintenance burden

As a counterpoint to Insight 3, wgslsmith and wgslgenerator (§IV-A) had to cope with the syntax of WGSL undergoing relatively frequent changes. The tint and naga compilers

sometimes took a while to catch up with changes to the language specification, leading to scenarios where these tools did not quite agree on certain syntactic points. To allow differential testing we had to implement workarounds to transform a generated WGSL shader into separate tint- and naga-friendly forms before compiling and executing it. If we had been less eager and designed these tools after the language had fully stabilised, we would have saved on engineering effort (at the cost of missing out on early bug discovery).

Related to this: while coverage-guided fuzz targets are re-built by ClusterFuzz and OSS-Fuzz each time the SUT changes, black-box fuzzers are manually uploaded as binary archives (see §IV-B). If the input format for the SUT changes, a black-box fuzzer may now yield inputs that are deemed invalid and hence no longer deeply exercise the SUT. There is no easy way to know that this has happened, meaning that black-box fuzzers must be manually re-built and re-deployed from time to time. It may thus be more fruitful to deploy black-box fuzzers on ClusterFuzz once the input format of a project has reached maturity.

Insight 5: Fuzzing too early may waste time by flagging already-known issues

While our targeted use of black-box fuzzing to find bugs in tint and naga via differential and validation testing led to the discovery of some subtle issues, there are several cases where it merely highlighted known issues that were already on the road-maps of developers of these projects. For example, our fuzzers highlighted several already-known cases where built-in functions of WGSL had not yet been implemented, and highlighted the already-known fact that tint and naga were not yet taking measures to enforce WGSL semantics for edge case behaviours, such as division by zero.

Insight 6: Bugs found by fuzzers in low-priority components can be a distraction

Prior to our introduction of strong corpora for our fuzzers (see §III-C), all fuzz targets were running on ClusterFuzz as part of the Chromium testing process. From time to time this would lead to discovery of security issues in non-critical components of SPIRV-Tools, and in the tint SPIR-V front-end, which is also non-critical. However, this happened sufficiently rarely that there was sufficient engineering bandwidth for them to be promptly fixed. The introduction of strong corpora led to a flood of issues being discovered in non-critical components of SPIRV-Tools, and in tint's support for SPIR-V. Among these were a number of security issues that were automatically tagged as Chromium release blockers. While these issues have all now been fixed, it was important for the development team to be able to focus on issues that directly affected WGSL support in tint, without being distracted by SPIR-V issues being flagged as release-blocking. For this reason, in August 2021 we moved all SPIR-V-related fuzzers to OSS-Fuzz so that they still run on Google's ClusterFuzz infrastructure, but do not lead to Chromium release-blocking bug reports.

B. Experience with custom mutators

Insight 7: There is a trade-off between the effectiveness of a custom mutator and the effort required for its creation and maintenance

The three custom mutators we have designed (see §III-B) discovered several bugs, including a number of security issues (see Table V). Of these mutators, we argue that the regex-based fuzzer has been the most worthwhile. It was simple to implement (not requiring any compiler construction expertise), and has provided a good return on investment. Adding additional mutation operators is a relatively straightforward process as these operators are based on string manipulation. In contrast, the implementation of each mutation operator in the AST fuzzer was akin to writing a simple compiler pass, which is a non-trivial task.

Insight 8: Bugs in custom mutator code can be a source of false alarms

The SPIRV-Tools-based mutator and associated fuzz target has the potential to provide the most bang-for-buck, since (a) it leverages existing technology—spirv-opt, spirv-fuzz and spirv-reduce—making it relatively cheap to build, and (b) it simultaneously tests tint's support for SPIR-V and WGSL, as well as exercising various components of SPIRV-Tools during the mutation process. However, as discussed in Insight 6, bugs in tint's SPIR-V front-end have lower priority for fixing compared with WGSL-related bugs (see Figure 2). Worse still, crashes in the custom mutator itself arising from bugs in spirv-fuzz, spirv-reduce and stand-alone spirv-opt optimisation passes may leaded to ClusterFuzz auto-reporting bugs that have even lower priority for fixing (see Figure 3).

More generally, a risk of deploying custom mutators in an in-process fuzzer such as libFuzzer is that one cannot immediately distinguish between crashes stemming from the custom mutators vs. crashes stemming from the fuzz target itself. We have suggested to the ClusterFuzz team that it might be worth detecting when crashes originate from a custom mutator and handling them differently so that they are not considered to be release blockers [17].

In the **# Repro.** column of Table V we have highlighted cases where the number of reproducible issues is lower than the total number of issues in the **# Issues** column. All three fuzzers that use custom mutators suffer from unreproducible issue reports, due to custom mutator crashes.

C. Test oracles

Insight 9: Fuzz targets must be carefully engineered to avoid false alarms

Recall from §II-A that to work effectively, a libFuzzer fuzz target must be designed so that a crash in the fuzz target really does correspond to a bug in the SUT. The tint compiler features various sanitising transformations that can be applied to an AST. In production, the Dawn engine (see §II-B and Figure 1) will apply these transformations in a specific manner, depending on the underlying native graphics API that is

being targeted. We had several false alarms where our fuzz targets would apply sanitising transformations in unexpected orders that will never be used in productions, leading to false alarm bug reports. As an example, one issue report features a comment from a possibly exasperated developer: "Looks like yet another case of the fuzzers not running all necessary transforms before invoking the backend." [6]. This highlights the point that fuzz targets must be designed with care in order for the crash oracle to yield meaningful bug reports.

Insight 10: It may be necessary to make modifications to the SUT to accommodate fuzzing, or to accept some number of false alarms

SPIR-V allows loops to be annotated with a hint requesting that they be fully unrolled, if a compiler can deduce a static bound on the number of times the loop will iterate. By design, spirv-opt honours such requests. The spirv-opt fuzz target merrily produced a program featuring a loop with a static bound of more than 1 million iterations, annotated with this unrolling hint. This led to OSS-Fuzz auto-reporting an out of memory error [7].

As this does not actually correspond to a bug in spirv-opt, nor a real-world usage scenario, the spirv-opt developers were reluctant to change the tool's production behaviour for the sake of pacifying a fuzzer. As a compromise, we modified spirv-opt so that when it is compiled as part of a fuzz target, loops with more than 100 iterations will not be unrolled [11]. This pacified the fuzzer for a while, but eventually it synthesised an example featuring a nest of loops with each loop exhibiting a double-digit iteration count, leading to a similar issue report.

This highlights the fact that when applying fuzzing to complex real-world software it is hard to avoid bug reports that, in a domain-specific sense, are false alarms.

Insight 11: Advanced oracles for fully automated fuzzing present interesting future challenges

Recall from §V that while our targeted application of wgslsmith and wgslgenerator led to a large number of bugs being discovered, their deployment on ClusterFuzz as black-box fuzzers led to very few bug reports. This is due to a difference in test oracles: ClusterFuzz merely checks whether tint crashes, while in our targeted use of these program generators we (a) used various back-end-specific tools to validate the output generated by tint, and (b) used differential testing to cross-check tint-generated code against naga-generated code, as well as to cross-check the behaviour of code generated by multiple tint back-ends.

Integrating these more advanced oracles into coverage-guided fuzzing is hindered by practical and conceptual challenges. Practically, using target-specific validation tools requires these to be available on the machines used for fuzzing. In the case of tint's MSL back-end, for example, validation requires the use of a Metal shader validator that would not be easy to deploy on ClusterFuzz's standard Docker image. Conceptually, differential testing is good for identifying that one of the systems being compared is wrong, but cannot automatically pinpoint which is the culprit. It is not clear how automatic bug triage—a big selling point of ClusterFuzz—could be adapted to effectively shed light on the root cause of issues found using differential testing.

VII. Related Work

Compiler fuzzing in the domain of graphics has been explored previously via the GraphicsFuzz (formerly GL-Fuzz) [12] and spirv-fuzz [14] tools, including an experience report on the use of GraphicsFuzz [13]. These approaches aim to find miscompilation bugs via metamorphic testing [4], [44]. These tools build on ideas from *equivalence modulo inputs* testing [34], generating families of equivalent programs via semantics-preserving transformations. We have used spirv-fuzz as the basis of a ClusterFuzz black-box fuzzer (see §IV-B) and as part of a custom mutator (see §III-B), but these uses of the tool do not exploit the fact that it applies semantics-preserving mutations; we merely exploit the fact that it applies *some* form of mutation in an attempt to generate programs that may uncover crash bugs in tint.

The wgslsmith and wgslgenerator tools were inspired by the Csmith generator of C programs [49]. In the domain of GPU computing, program generation has also been used to test OpenCL and CUDA compilers [27], [35].

Our regex-based mutator is similar to the Universal Mutator tool [23], [24], which as been applied in the context of compiler fuzzing [25]. The interval deletion mutations of our regex-based fuzzer were inspired by the structureshrink test case reducer [37], which considers deleting regions of an input that are delimited by particular character sequences.

VIII. Conclusions

We have described our experience applying a variety of fuzzing techniques, included coverage-guided fuzzing with and without custom mutators, and black-box fuzzing, to test compilers and processing tools for the WGSL and SPIR-V graphics shading languages. These include security-critical components that ship as part of WebGPU and WebGL implementations in modern browsers. Our deployment of fuzzing has led to the discovery and fixing of a large number of potentially security-critical issues.

An important direction for future work is to bridge the gap between targeted black-box fuzzing—where differential testing provides a strong oracle—and scalable fuzzing on ClusterFuzz and OSS-Fuzz. The former can lead to the discovery of more intricate functional bugs than the latter, but currently requires a human in the loop for bug triage and reporting. Another direction is to use Google's new FuzzTest project [20] to reduce the amount of custom code that needs to be written to interpret input byte buffers in a structured manner.

Acknowledgements

Thanks to Paul Thomson for his valuable feedback on an earlier draft of this work. This work was partly supported by the EPSRC IRIS Programme Grant (EP/R006865/1).

REFERENCES

[1] Apple, "Metal Shading Language Specification," 2022, https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf, last accessed 2022-10-23.

[2] A. Arya and O. Chang, "ClusterFuzz: Fuzzing at Google scale," in *Black Hat Europe 2019*, 2019, https://i.blackhat.com/eu-19/Wednesday/eu-19-Arya-ClusterFuzz-Fuzzing-At-Google-Scale.pdf, last accessed 2022-10-23.

[3] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Trans. Software Eng.*, vol. 41, no. 5, pp. 507–525, 2015. [Online]. Available: https://doi.org/10.1109/TSE.2014.2372785

[4] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: A new approach for generating next test cases," The Hong Kong University of Science and Technology, Tech. Rep. HKUST-CS98-01, 1998.

[5] Chromium, "Issue auto-reported by ClusterFuzz, revealing problem related to abstract vectors," 2022, https://bugs.chromium.org/p/chromium/issues/detail?id=1345468, last accessed 2022-11-03.

[6] ——, "Issue auto-reported by ClusterFuzz that turned out to be due to a misconfigured fuzz target," 2022, https://bugs.chromium.org/p/chromium/issues/detail?id=1314938, last accessed 2022-11-03.

[7] ——, "Issue auto-reported by OSS-Fuzz that turned out to relate to a loop with many iterations being fully unrolled," 2022, https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=39010, last accessed 2022-11-03.

[8] B. Clayton, "Change list to Dawn project," 2022, https://dawn-review.googlesource.com/c/dawn/+/94942, last accessed 2022-11-03.

[9] ——, "Github issue: specification needs to clarify behavior of indexing of abstract composite type with runtime value," 2022, https://github.com/gpuweb/gpuweb/issues/3210, last accessed 2022-11-03.

[10] ClusterFuzz, "Blackbox fuzzing," 2023, https://google.github.io/clusterfuzz/setting-up-fuzzing/blackbox-fuzzing/, last accessed 2023-01-28.

[11] A. F. Donaldson, "SPIRV-Tools pull request: Avoid unrolling large loops while fuzzing," 2022, https://github.com/KhronosGroup/SPIRV-Tools/pull/4835, last accessed 2022-11-03.

[12] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, "Automated testing of graphics shader compilers," *PACMPL*, vol. 1, no. OOPSLA, pp. 93:1–93:29, 2017. [Online]. Available: https://doi.org/10.1145/3133917

[13] A. F. Donaldson, H. Evrard, and P. Thomson, "Putting randomized compiler testing into production (experience report)," in *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, ser. LIPIcs, R. Hirschfeld and T. Pape, Eds., vol. 166. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 22:1–22:29. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2020.22

[14] A. F. Donaldson, P. Thomson, V. Teliman, S. Milizia, A. P. Maselco, and A. Karpinski, "Test-case reduction and deduplication almost for free with transformation-based compiler testing," in *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, S. N. Freund and E. Yahav, Eds. ACM, 2021, pp. 1017–1032. [Online]. Available: https://doi.org/10.1145/3453483.3454092

[15] Go, "The Go programming language specification: Constants," 2022, https://go.dev/ref/spec#Constants, last accessed 2022-11-03.

[16] Google, "ClusterFuzz," 2022, https://google.github.io/clusterfuzz/, last accessed 2022-10-23.

[17] ——, "ClusterFuzz GitHub issue about bugs in custom mutators," 2022, https://github.com/google/clusterfuzz/issues/2827, last accessed 2022-11-03.

[18] ——, "Dawn, a WebGPU implementation," 2022, https://dawn.googlesource.com/dawn/+/refs/heads/main/README.md, last accessed 2022-10-24.

[19] ——, "Fuzzing forum: Structure-aware fuzzing with libFuzzer," 2022, https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md, last accessed 2022-10-23.

[20] ——, "FuzzTest," 2022, https://github.com/google/fuzztest, last accessed 2022-11-03.

[21] ——, "OSS-Fuzz," 2022, https://google.github.io/oss-fuzz/, last accessed 2022-10-23.

[22] ——, "Swiftshader, CPU-based vulkan implementation," 2022, https://swiftshader.googlesource.com/SwiftShader, last accessed 2022-10-23.

[23] A. Groce, "Univeral mutator," 2022, https://github.com/agroce/universalmutator, last accessed 2022-10-27.

[24] A. Groce, J. Holmes, D. Marinov, A. Shi, and L. Zhang, "An extensible, regular-expression-based tool for multi-language mutant generation," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 25–28. [Online]. Available: https://doi.org/10.1145/3183440.3183485

[25] A. Groce, R. van Tonder, G. T. Kalburgi, and C. L. Goues, "Making no-fuss compiler fuzzing effective," in *CC '22: 31st ACM SIGPLAN International Conference on Compiler Construction, Seoul, South Korea, April 2 - 3, 2022*, B. Egger and A. Smith, Eds. ACM, 2022, pp. 194–204. [Online]. Available: https://doi.org/10.1145/3497776.3517765

[26] A. Herrera, H. Gunadi, L. Hayes, S. Magrath, F. Friedlander, M. Sebastian, M. Norrish, and A. L. Hosking, "Corpus distillation for effective fuzzing: A comparative evaluation," 2019. [Online]. Available: https://arxiv.org/abs/1905.13055

[27] B. Jiang, X. Wang, W. K. Chan, T. H. Tse, N. Li, Y. Yin, and Z. Zhang, "Cudasmith: A fuzzer for CUDA compilers," in *44th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2020, Madrid, Spain, July 13-17, 2020*. IEEE, 2020, pp. 861–871. [Online]. Available: https://doi.org/10.1109/COMPSAC48688.2020.0-156

[28] Khronos Group, "The OpenGL ES shading language, version 3.20.6," 2019, https://registry.khronos.org/OpenGL/specs/es/3.2/GLSL_ES_Specification_3.20.pdf, last accessed 2022-10-24.

[29] ——, "A complete registry of all official SPIR-V specifications," 2022, https://www.khronos.org/registry/SPIR-V/, last accessed 2022-10-23.

[30] ——, "glslang GitHub repository," 2022, https://github.com/KhronosGroup/glslang, last accessed 2022-06-30.

[31] ——, "SPIRV-Tools repository, including spirv-opt and spirv-val," 2022, https://github.com/KhronosGroup/SPIRV-Tools, last accessed 2022-06-30.

[32] ——, "Vulkan 1.3 - a specification (with all registered Vulkan extensions)," 2022, https://www.khronos.org/registry/vulkan/specs/1.3-extensions/html/vkspec.html, last accessed 2022-10-23.

[33] ——, "WebGL 2.0 specification," 2022, https://registry.khronos.org/webgl/specs/latest/2.0/, last accessed 2022-10-23.

[34] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, M. F. P. O'Boyle and K. Pingali, Eds. ACM, 2014, pp. 216–226. [Online]. Available: https://doi.org/10.1145/2594291.2594334

[35] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, D. Grove and S. Blackburn, Eds. ACM, 2015, pp. 65–76. [Online]. Available: https://doi.org/10.1145/2737924.2737986

[36] LLVM Compiler Infrastructure, "libFuzzer – a library for coverage-guided fuzz testing," 2022, http://llvm.org/docs/LibFuzzer.html, last accessed 2022-10-23.

[37] D. MacIver, "Structureshrink," 2022, https://github.com/DRMacIver/structureshrink, last accessed 2022-11-03.

[38] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998. [Online]. Available: http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

[39] Microsoft, "Reference for HLSL," 2019, https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-reference, last accessed 2022-10-23.

[40] H. Mohsin, "wgslsmith," 2022, https://github.com/wgslsmith/wgslsmith, last accessed 2022-10-25.

[41] ——, "WGSLsmith: a random generator of WebGPU shader programs," Master's thesis, Imperial College London, 2022, https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/2122-ug-projects/2122-individual-projects/WGSLsmith---a-Random-Generator-of-WebGPU-shader-programs.pdf, last accessed 2022-10-25.

[42] D. Neto, "Github pull request: wgsl: indexing X by non-const-expr index requires X to be concrete," 2022, https://github.com/gpuweb/gpuweb/pull/3497, last accessed 2022-11-03.

[43] Rust Graphics Mages, "The wgpu project," 2022, https://github.com/gfx-rs/wgpu, last accessed 2022-10-24.

[44] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 805–824, 2016.
[45] The Clang Team, "Clang documentation," 2022, https://clang.llvm.org/docs/, last accessed 2022-10-23.
[46] W3C, "WebGPU Shading Language W3C working draft," 2022, https://www.w3.org/TR/WGSL/, last accessed 2022-10-23.
[47] ——, "WebGPU, W3C Working Draft," 2022, https://www.w3.org/TR/webgpu/, last accessed 2022-10-23.
[48] H. Watson, "wgslgenerator," 2022, https://github.com/hanawatson/wgslgenerator, last accessed 2022-10-25.
[49] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, M. W. Hall and D. A. Padua, Eds. ACM, 2011, pp. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532