WebGlitch: A Randomised Testing Tool for the WebGPU API

Matthew K. L. Wong
Department of Computing, Imperial College London, UK

Alastair F. Donaldson
Department of Computing, Imperial College London, UK

Abstract

We report on our experience designing a new technique and tool for fuzzing implementations of WebGPU, a W3C standard JavaScript API for in-browser GPU computing. We also report on our experience using our WebGlitch tool to test industrial-strength implementations of WebGPU, leading to the discovery of numerous bugs. WebGPU enables programmatic access to a device's graphics processing unit (GPU) for in-browser GPU computing, and is being implemented by Google, Mozilla and Apple for inclusion in all of the major web browsers. Guaranteeing the security and reliability of WebGPU is crucial to avoid wide-reaching browser security vulnerabilities and to facilitate portability by ensuring uniform behaviour across different platforms. To that end – inspired by randomised compiler testing techniques – our approach to fuzzing creates random, valid-by-construction programs by continuously selecting a WebGPU API function, then recursively generating all requirements necessary for that API call to be valid based on careful modelling of the API specification. This is implemented as a new open source tool, WebGlitch, which we designed in consultation with engineers at Google who work on the Chrome WebGPU implementation. WebGlitch identifies bugs through sanitiser-boosted crash oracles, differential testing, and by identifying cases where valid-by-construction API calls lead to runtime errors. We present an evaluation showing that WebGlitch can find bugs missed by an existing WebGPU fuzzer, wg-fuzz, and across the broader WebGPU ecosystem: to date, WebGlitch has found 24 previously-unknown bugs (15 fixed so far in response to our reports). Among these, 17 bugs affected WebGPU implementations from Google, Mozilla, and the Deno project. WebGlitch found an additional 4 bugs in the shader compilers used by the graphics APIs that WebGPU interfaces with. The remaining 3 bugs affect the widely-used JavaScript runtimes Node.js and Deno. Fuzzing with WebGlitch also led us to identify an ambiguity in the specification of the WebGPU shading language, for which we proposed an amendment that was accepted by W3C and which has been adopted in the latest version of the specification. Analysing the line coverage of a WebGPU implementation by WebGlitch-generated programs revealed that WebGlitch covers code missed by wg-fuzz and the official conformance test suite. Our hope is that this report on the design of WebGlitch and its deployment in practice will be useful for practitioners and researchers interested in using API fuzzing to improve the reliability of industrial codebases.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging; Software and its engineering → Object oriented languages

Keywords and phrases Fuzzing, WebGPU, WGSL, API, shaders

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.39

Category Experience Paper

Supplementary Material *Software (ECOOP 2025 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.11.2.14`

Funding This work was supported by EPSRC grant EP/R006865/1 and gift funding from Google.

Acknowledgements We are grateful to Corentin Wallez, Alan Baker and David Neto at Google for their input during the design and deployment of WebGlitch, and to Daniel Simols for assistance in using the wg-fuzz tool.

1 Introduction

In this experience report we investigate the use of randomised testing techniques for improving the reliability of implementations of the WebGPU API.

WebGPU [58] is a new W3C standardised web application programming interface (API) that allows building websites that make programmatic use of a device's graphics processing unit (GPU) for real-time rendering and general-purpose computation. WebGPU abstracts away the intricacies of platform-specific graphics APIs, providing a unified, object-oriented JavaScript interface to execute commands on the GPU and configure render and compute passes, including the running of *shader programs*, which execute across the massively parallel hardware that GPUs offer. In practice, implementations of WebGPU conform to this interface by delegating WebGPU API calls to appropriate combinations of calls in lower level, platform-specific graphics APIs such as Direct3D [37], Metal [2], and Vulkan [25].

Three major WebGPU implementations are currently under development, by Google (for Chromium-based browsers, including Chrome and Edge), Apple (for Safari), and an implementation primarily by Mozilla (for Firefox) [51].

As a web standard, WebGPU will be enabled on billions of browser instances, so defects in implementations of the API that affect WebGPU-enabled content, or worse, exploitable vulnerabilities, have the potential to be wide-reaching. Since WebGPU requires interfacing with kernel-level graphics drivers, exploits can potentially bypass browser sandboxing features [49], and these bugs have the potential to be triggered by merely visiting a web page. Indeed, many high severity CVEs have been identified for an existing web graphics API, WebGL [40, 42, 41]. WebGPU's increased complexity over WebGL presents an even wider attack surface and makes it more difficult to test. While there is an official conformance test suite (CTS) [50], it is unrealistic for handwritten tests to cover all edge cases, especially as problems may arise due to both bugs in the WebGPU implementation itself and bugs in platform-specific shader compilers and graphics APIs.

API fuzzing can help cover these corner cases by randomly generating sequences of API calls [5, 44, 22, 21]. In this work, we investigate the use of API fuzzing for finding bugs in implementations of WebGPU. In consultation with engineers at Google who work on the WebGPU implementation in Chromium, we developed a Java-based open source command line tool, WebGlitch: a black box fuzzer for the WebGPU API. We report on our experience designing and implementing WebGlitch, and deploying it in practice to find bugs in production WebGPU implementations.

WebGlitch generates JavaScript programs that make intricate use of the WebGPU API. Generated programs can then be fed to a WebGPU implementation under test, also testing the downstream graphics API and graphics drivers on which the implementation depends.

We have designed WebGlitch to ensure that the WebGPU programs it generates by default are valid-by-construction, according to the WebGPU specification. This is important because programs generated in a naive fashion would be unlikely to yield programs featuring API calls that satisfy WebGPU's complex validation rules. Invalid API calls would only exercise a limited subset of the implementation under test. Another benefit of generating valid-by-construction programs is that it allows for the automatic detection of validation bugs, where valid calls to API functions are wrongly rejected.

To ensure rigorous testing of parts of WebGPU related to compiling and executing shaders, we have integrated WebGlitch with WGSLsmith, a state-of-the-art random generator of WebGPU shading language (WGSL) programs [13, 28].

WebGlitch constructs programs incrementally, iteratively choosing random API functions to generate until a pre-defined limit is reached. Many API functions require parameters that refer to objects that must have been created previously via other API calls. At each generation step, rather than being restricted to only using previously-generated objects as the receiver and parameters for an API call, WebGlitch uses an *on-demand* approach, as follows. An *arbitrary* API function is chosen at random. A suitable receiving object and parameters for the chosen function are then obtained via a combination of (a) using in-scope objects from previous calls (if available), and (b) recursively generating randomised calls to other API functions to yield a receiving object and suitable parameter arguments when needed. On-demand generation increases the probability of rich API functions that require complex arguments being invoked: taking the simpler approach of only allowing calls to API functions for which suitable arguments have already been generated would bias generation towards favouring simple functions requiring a small number of simple arguments.

To maximise the chances of finding bugs, WebGlitch uses multiple test oracles in conjunction: sanitiser-boosted crash oracles (using sanitisers such as AddressSanitizer [47], LeakSanitizer [20], and UndefinedBehaviorSanitizer [20]), differential testing [32], and confirming that no errors are thrown for a valid WebGPU program (hereafter termed the "validity" oracle). The validity oracle hinges on the fact that WebGlitch generates programs that are valid by construction.

Over a period of 5 months, at various stages during its development, we have used WebGlitch to test the Dawn WebGPU implementation from Google [19] and the wgpu implementation from (primarily) Mozilla [45], used in the Chromium and Firefox browsers, respectively. This has led to WebGlitch identifying 24 previously-unknown bugs, of which 17 affected WebGPU implementations, 4 affected shader compilers used in downstream graphics APIs, and 3 affected the JavaScript runtimes Node.js and Deno. In addition, WebGlitch independently identified 5 bugs that turned out to have already been reported.

As well as finding bugs in implementations of WebGPU, our testing efforts identified an ambiguity in the specification of WGSL related to overflow checking for numeric types. We clarified the intended semantics with engineers at Google who are leading the design of WGSL, and based on this discussion we proposed a clarification to its specification to resolve the ambiguity, which as been accepted and incorporated into its latest version [52].

We present a set of controlled experiments on Google's WebGPU implementation, comparing the lines of code covered by running the WebGPU CTS vs. by conducting a short fuzzing campaign using WebGlitch, and a comparable fuzzing campaign using an existing WebGPU fuzzer, wg-fuzz [48]. Our results show that both WebGlitch and wg-fuzz generated programs that covered lines missed by the CTS. WebGlitch was able to cover 1,530 lines not reached by wg-fuzz, despite supporting the generation of fewer WebGPU API functions outlined in the specification than wg-fuzz. These two fuzzers complement each other, with wg-fuzz being able to cover 5,180 lines missed by WebGlitch.

We also present a summary of the insights that we gained on API fuzzing from our fuzz testing campaign of WebGPU, which may be valuable for researchers and practitioners considering deploying API fuzzing in the future. A key point here is that although our approach allowed the detection of numerous previously-unknown bugs, our fuzzing campaign required frequent manual intervention because of both the evolving and layered nature of the WebGPU API.

In summary, the contributions of this paper are:

- The development of a technique for fuzzing WebGPU implementations, implemented as the WebGlitch open source tool.

- A report on the use of WebGlitch in practice to find bugs in industrial-strength implementations of WebGPU from Google and Mozilla, to find bugs in associated infrastructure including the Node.js and Deno JavaScript runtimes, and to resolve an ambiguity in the WebGPU shading language specification.
- The findings of a controlled experiment assessing the code coverage achieved by WebGlitch in comparison to the WebGPU CTS and another WebGPU fuzzer, wg-fuzz. Our results show that WebGlitch covers lines of code that are missed by wg-fuzz, and that there are potential gaps in the CTS.

Paper Structure

We first give background on the WebGPU ecosystem, including the way in which it interfaces with downstream graphics APIs and the shader compilation process (Section 2). We then discuss the design and implementation details of WebGlitch (Section 3), before presenting an overview of the bugs found by WebGlitch (Section 4). We also conduct a controlled study to evaluate the abilities of different tools – the WebGPU CTS, wg-fuzz, and WebGlitch – to exercise and test a WebGPU implementation (Section 5). We finish by discussing key insights and lessons learned from our work (Section 6), related work on API fuzzing and compiler testing (Section 7), and directions for future work (Section 8).

Availability

Our WebGlitch tool is available as open source [53], and an artifact accompanying the paper is available [56]. WebGlitch is exposed as a set of command line tools: a *generator* component that generates JavaScript program that uses WebGPU; a *runner* component that runs sets of generated programs on a target platform, and a separate *fault-finder* program that mines the logs produced by the runner component to check for bugs, which are then categorised by the test oracle that found them.

2 WebGPU Background

WebGPU is a powerful new API that allows web applications to leverage the GPU for demanding tasks such as data visualisation, producing high-quality renders for 3D design tools, and on-device machine learning [57]. The WebGPU API allows a JavaScript program to encode GPU commands in buffers, which are then sent to the GPU for execution [58]. For example, there are commands for setting up render and compute passes, wherein shader program execution can be configured. The entire API is object-oriented, and all API functions are invoked on some receiving WebGPU object.

There are three major WebGPU implementations currently in development: Dawn, developed by Google and used in the Chromium project, on which the Google Chrome and Microsoft Edge browsers are based [19]; wgpu [45], primarily developed by Mozilla and used in Firefox, and an implementation from Apple [8] as part of the WebKit browser engine on which Safari is based. The wgpu implementation is noteworthy in that it is written in and made for Rust, and it is not fully compliant with the W3C WebGPU standard to make its usage more idiomatic to the Rust language. The wgpu project is used as the core implementation of WebGPU functionality in Firefox. However, an additional layer above wgpu is required to map JavaScript WebGPU API calls to wgpu ones and to fully adhere to the specification [45]. Google and Apple's WebGPU implementations are both written in C++. Although WebGPU is designed to be used in-browser, bindings are also available to

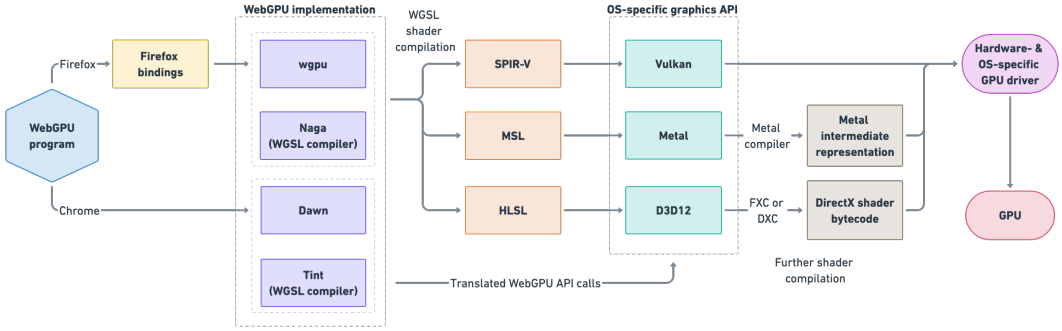

Figure 1 Simplified overview of the WebGPU architecture and WGSL compilation chain, created using information from wgpu and Chromium documentation [45, 49]. Tint is technically part of Dawn, but has been shown separately here for clarity. HLSL can be compiled by FXC [34] (deprecated) or DXC [35]. Apple's implementation in Safari, which only targets the Metal backend, has been excluded for clarity.

enable its use from JavaScript runtimes such as Node.js and Deno. As with Firefox, Deno [11] also uses wgpu as the core of its WebGPU implementation, building on top of it with its own independent implementation.

When a WebGPU API call is made in JavaScript, it is validated according to the correct usage defined in the WebGPU specification. If valid, it is then mapped on to one or more platform-specific graphics API calls (see Figure 1). At the time of writing, the APIs that are used in practice are: Direct3D 12 (D3D12) (for Windows) [37], Metal (for macOS, iOS, iPadOS) [2], and Vulkan (for Linux, Android) [25].

The downstream API calls made by the WebGPU implementation should be valid-by-construction. But bugs related to making these calls could mean that the lower-level API is used incorrectly, which can lead to undefined behaviour and crashes that could be exploited [36]. Even if there are no errors related to their usage, bugs within the implementation of these lower level APIs could be exploited through WebGPU programs. The major WebGPU implementations are also written in languages that allow for memory-unsafe code to be written (C++ and Rust[1]). As such, heap-use-after-frees and buffer overflows could be exploited for arbitrary remote code execution.

A key part of graphics programming is the issuing of shader programs for execution on GPU hardware. Because WebGPU provides an abstraction that is independent of any particular graphics API, it has its own associated shading language: WebGPU Shading Language (WGSL) [59]. Because the underlying graphics APIs expect shaders to be written in specific shading languages, such as HLSL (D3D12) [38], MSL (Metal) [3], and SPIR-V (Vulkan) [26], a WebGPU implementation must first compile a WGSL shader into the appropriate target language (Figure 1). This is accomplished using the WGSL compilers Tint (for Dawn) [19], Naga (for wgpu) [45], and wgslc (for Safari/WebKit). The resulting shader will then be further compiled by the shader compiler that ships with the driver for the user's GPU device to enable execution on the GPU hardware. Worryingly, bugs within the WGSL compilers themselves, as well as downstream compilers, have the potential to be exploited.

[1] Rust code that uses the "unsafe" feature of the language is not guaranteed to be memory-safe, and unsafe Rust is used in the wgpu project.

All these potential attack vectors combine to create a highly complex attack surface, similar to the one present in WebGPU's predecessor, WebGL [27]. This attack surface includes the WebGPU implementation and WGSL compiler used by a browser, alongside the lower-level platform-specific graphics API used by a device. However, the implementations of these APIs and their associated shader compilers are typically managed by third parties, resulting in a multi-layered attack surface that is difficult to test comprehensively.

Potential vulnerabilities have already been identified. These include an issue within the WebGPU implementation Dawn [49], an overflow bug in a downstream shader compiler [49], and several high-severity CVEs related to Tint [6].

3 Design and Implementation of WebGlitch

After an overview of how programs are generated by WebGlitch (Section 3.1), we explain how WebGPU API functions are represented (Section 3.2) and discuss the algorithm WebGlitch uses to generate valid-by-construction programs (Section 3.3). A worked example of this algorithm is then presented in Section 3.4 We then discuss how shaders are incorporated in generated programs (Section 3.5), and the test oracles that WebGlitch supports (Section 3.6). Finally, we present experimental results that give a ballpark estimate of the throughput of fuzzing using WebGlitch (Section 3.7).

3.1 WebGPU Program Generation Overview

To generate random WebGPU programs from scratch, WebGlitch incrementally adds new API calls to a skeleton JavaScript program. This skeleton contains components such as the main function and a helper function to load shaders. We refer to the program comprising all the calls that have been generated so far as a *partially-generated* program. At all times, WebGlitch keeps track of the state of the partially-generated program in order to generate valid API calls.

The following steps are repeated N times, for some fixed N that is provided as a parameter to the tool.

1. Choose a random API function, f, to call.
2. Select a receiver for f: an object on which f should be called. This may be an existing object resulting from a previous call, or may involve WebGlitch recursively generating additional API calls to bring a receiver of the right type into scope.
3. Issue any necessary function calls on the chosen receiver to bring it into a suitable state for a call to f to be valid.
4. Analyse the arguments required for the call to f. For primitive types and enumerated values, randomly choose a value from a set or range of valid ones. For other WebGPU objects, either find an in-scope object that satisfies the requirements, or generate one that does.

To illustrate step 3: suppose f needs to be called on a `GPUCommandEncoder` object with no active child passes. A `GPUCommandEncoder` is a WebGPU object that stores commands intended for execution on the GPU, such as render passes and compute passes. These "child" passes of a `GPUCommandEncoder` are considered "active" if the `end` method has not yet been invoked on it. If WebGlitch has selected a receiver representing a `GPUCommandEncoder` object with an active child pass, WebGlitch would need to invoke `end` on the child pass to bring the receiver into the necessary state.

Two programs generated using N repetitions of the above steps will have *roughly* the same size, because they will each feature N API function calls chosen via step 1. However, the sequence of functions that are called will likely be entirely different. Randomised recursive generation in steps 2–4 to fulfil the requirements of these calls will also lead to diversity.

The generation of a given call is guaranteed to terminate as there are a finite number of arguments per API function, and there are no cyclic dependencies between the requirements of functions in the API. In addition, WebGlitch uses a bound on the number of calls that will be generated, so generation is always guaranteed to terminate.

We now delve into the technical details of how this approach works, and our choice of this strategy for program generation.

3.2 Defining API Call Signatures and Requirements

Generation of valid WebGPU programs, which we describe in full in Section 3.3 below, hinges on a rigorous understanding of the signatures and requirements of API functions as defined in the prose specification.

Inspired by RESTler [5], an API-agnostic fuzzer which uses an interface definition language (IDL) to represent the structure of API functions, we designed a custom representation to specify the WebGPU API. This allowed us to add support for generating more basic API calls quickly. Each API function is specified via a JSON object, using a format that expresses characteristics shared by all API functions (e.g., number and types of arguments, and certain categories of requirements) to be expressed externally to the WebGlitch codebase (Figure 2). On execution, WebGlitch parses these definitions and uses the requirements they specify to guide its generation process. This decoupling of core API requirements from the source code of WebGlitch proved convenient during the development of the tool: it allowed additional functions to be made available to the tool, or early errors in our understanding of the API specification to be fixed without modifying or recompiling the WebGlitch source code.

```
"name": "writeBuffer",
"returnType": "none",
"properties": [{
  "buffer": {
    "type": "GPUBuffer",
    "conditions": {
      "same": ["GPUDevice"],
      "withAttributes": {
        "GPUBuffer.usage": ["GPUBufferUsage.COPY_DST"],
        "GPUBuffer.mapped": ["false"]
  }}}}
]
```

(a) Function to write data to a buffer on the GPU.

```
"name": "unmap",
"returnType": "none",
"conditions": {
  "withAttributes": {
    "GPUBuffer.mapped": ["true"]
  }
},
"setAttributes": {
  "this.mapped": ["false"]
}
```

(b) Function to return access to a buffer back to the GPU.

Figure 2 Examples of the JSON format for specifying WebGPU API functions and their requirements. Not all arguments have been listed for clarity. Some arguments require WebGPU objects, and their required state can be specified through the *withAttributes* field under the *conditions* field. These values can also be dynamic. The *setAttributes* field indicates that the function modifies the state of the object on which the function is called.

We populated these function requirements through careful analysis of the WebGPU specification [58], and updated them in response to occasional changes to the specification. Gathering such requirements was manual and could not be automated, as many API functions had unique constraints that required iterative updates to the JSON representation we used.

In addition to the general classes of requirements expressed in our JSON format, some API functions have more intricate requirements. Since extending our JSON representation to express such requirements was not straightforward nor elegant, information about them is "baked in" to WebGlitch – expressed in the logic of the WebGlitch source code. For example, creating a pipeline for rendering textures (through `createRenderPipeline`) can involve specifying various information about the vertex shader that will be used, which is difficult to express in a JSON representation.

Validation of WebGPU API function signatures was principally through practical testing against two implementations of WebGPU: Dawn and wgpu. As valid-by-construction programs are generated by default, when a WebGPU implementation under test rejects such a program this indicates that there is a bug in that implementation or a bug in the way the API function has been modelled in WebGlitch, which would be fixed iteratively. In cases where this was not clear from the specification, we consulted engineers at Google who work on Dawn.

3.3 Creating Valid Programs

WebGlitch constructs programs using an internal representation akin to an abstract syntax tree, but simpler because generated programs consist of a linear sequence of API calls.[2] Once generation is complete, the internal representation is pretty-printed as a JavaScript program.

Recall from the introduction and Section 3.1 that we have designed WebGlitch generate programs that are valid-by-construction: they always call WebGPU API functions on receiving objects that are in an appropriate state, passing the right number and types of arguments, with arguments satisfying any requirements described in the API specification.

The motivation for ensuring that generated programs are valid is as follows. When an *invalid* API call is made, minimal logic of the WebGPU implementation is executed: only the logic that checks for argument validity. Importantly, no functionality of the underlying graphics API will be invoked, and the WebGPU API call will not produce a meaningful result that can be used as input for subsequent functions.

We considered two approaches to generating valid-by-construction programs: a *conservative* approach, where an API function is selected to be invoked only if the partially-generated program has already created suitable WebGPU objects to fulfil the requirements of the selected function, and an *on demand* approach, where *any* API function can be selected to be invoked, with recursive generation being used to fulfil any missing requirements.

As already illustrated by the overview in Section 3.1, we opted for on-demand generation. Although the conservative approach is simpler, it suffers from the problem that many WebGPU functions have extensive requirements that must be satisfied in order for calls to be valid. It is unlikely that all such requirements will happen to be satisfied as a result of prior API calls in the partially-generated program. As a result, the most interesting and complex API functions would rarely be invoked in generated programs.

Once an API function has been selected at random, generation of a call to this function by WebGlitch can be split into two stages:

1. choose or create a random receiver object and ensure that it is in a suitable state for the function call;

[2] From the perspective of testing WebGPU implementations, there would be no value in including loops or conditional statements in generated JavaScript programs: such constructs would exercise the JavaScript engine, but would not exercise the WebGPU API in any additional depth, since e.g. a long linear sequence of calls can be used to call particular API functions multiple times.

2. for each function argument, choose a suitable value at random (for enumerated and numeric types), or choose or create a suitable WebGPU object at random and ensure that it is in a suitable state for the function call (for object arguments).

The state of a WebGPU object o is determined by the arguments passed into the function that created o, the state of the WebGPU object on which the constructing function was called, and any API calls that have been made on o since it was constructed.

In step 1 above, if a suitable receiver object with the required state is not available, WebGPU issues calls to construct one from scratch and bring it into a suitable state, or to bring an existing object into a suitable state. This process is recursive, because the creation of a required WebGPU object may require the creation of other WebGPU objects. We illustrate this via a worked example below.

Even if suitable receiver objects are already available for a call, WebGlitch randomly decides (via a configurable probability) whether to use one of these or generate a new receiver. The motivation for this is to reduce the probability that all API calls are made on the same or a small number of objects, which would limit program diversity.

Once a suitable receiver has been found or created, WebGlitch iteratively generates a valid value for each expected argument. For arguments with enumerated types, invalid values are determined by analysing the state of the receiver and object on which the constructor was called, and are filtered out. A value is chosen at random from the remaining subset. For arguments with numeric type, a random value is chosen within a valid minimum and maximum value. Certain numeric arguments may also need to be divisible by a constant value to be considered valid, in which case the randomly chosen value is rounded to the nearest divisible number within the range.

The most complex arguments to generate are other WebGPU objects, because their validity for a given call does not only depend on their type, but also on their state. The process of selecting or creating suitable arguments for a call is analogous to that for selecting or creating the receiving object for the call. However, in some cases the chosen receiving object can constrain the choice of valid arguments for the call. We illustrate this with a worked example in Section 3.4.

To support generation of valid programs, WebGlitch uses a symbol table (akin to that of a compiler) to track the types and states of objects in the partially-generated program throughout its generation. This includes relationships between parent and child objects (because new objects are created by invoking API functions on existing objects), and history of API calls that have been invoked on each object.

Allowing Invalid Function Calls

While there are several advantages to generating WebGPU programs that are guaranteed to be valid (as argued above), we deemed it also useful to equip WebGlitch with a mode where API calls within a program have some probability of being invalid, creating programs that are *nearly valid*. This enables the detection of instances where a WebGPU implementation's validity checks deviate from the specification or differ from those of another implementation.

To cater for this, WebGlitch can be configured with a percentage chance to bypass its internal validity analyses. WebGlitch can skip verifying both the receiver's state and the validity of the arguments passed into the API call, such that invalid WebGPU objects can also be passed as arguments in API calls further down.

3.4 Worked Example: Generating a Valid WebGPU API Call

To illustrate the above ideas, we show a call to the `writeBuffer` API function (Figure 3), which writes data into GPU memory, could be generated.

```
writeBuffer(GPUBuffer buffer, GPUSize64 bufferOffset, AllowSharedBufferSource data)
```

Figure 3 Signature of the `writeBuffer` method of the `GPUDevice` class. Optional arguments not shown for simplicity. GPUBuffer: a WebGPU object of type GPUBuffer. GPUSize64: an unsigned long long. AllowSharedBufferSource: an instance or view of a JavaScript ArrayBuffer or SharedArrayBuffer object. Adapted from [58].

Figure 4 shows what the partially-generated program might look like after all the ingredients required for the call to `writeBuffer`, and the call itself, have been generated. Being able to make this call depends on various WebGPU-specific constructs, including `GPUAdapter` (from which a WebGPU object representing the GPU is requested), `GPUDevice` (programmatic representation of a GPU), `GPUQueue` (handles executing GPU commands), and `GPUBuffer` (representing GPU memory) [58]. The specific details of these constructs do not matter much for the example.

```
const adapter = await navigator.gpu.requestAdapter();
const device = await adapter.requestDevice();
const queue = device.queue;
const buffer = device.createBuffer({
  mappedAtCreation: true
  size: 12
  usage: GPUBufferUsage.COPY_DST
});
const bufferOffset = 8;
const data = new Uint8Array([1, 2, 3, 4]);
queue.writeBuffer(buffer, bufferOffset, data);
```

Figure 4 WebGPU program to write data to a buffer.

Suppose WebGlitch is at the very start of creating a WebGPU program, so that the partially-generated program is empty, and suppose that the `writeBuffer` function is randomly selected. The receiver object for a call to this function must be a `GPUQueue`, but no such object exists in the (empty) partially-generated program. Because a `GPUQueue` is accessed via a property of a `GPUDevice`, and no `GPUDevice` is available in the empty program, WebGlitch issues a call to `requestDevice` to create a `GPUDevice`. However, `requestDevice` requires a `GPUAdapter` as its receiver, thus WebGlitch issues a call to `requestAdapter` to create a `GPUAdapter`. No additional generation work is required to call `requestAdapter` because it is invoked on the `navigator.gpu` property. The `navigator` object is part of the Document Object Model and provides access to the WebGPU API.

Armed with a `GPUQueue` object, `queue`, WebGlitch can now proceed to generate a call to `writeBuffer` on `queue`. In some cases, certain state-changing functions must be invoked on an object before a particular desired function can be called, but this is not the case for `writeBuffer`: it can be safely called on a freshly-created `GPUQueue` object. WebGlitch can therefore proceed to generate suitable parameters for the call to `writeBuffer`. Instructed by the JSON files in which requirements of WebGPU API functions are encoded, WebGlitch knows that `writeBuffer` requires a `GPUBuffer` created from the same `GPUDevice` from

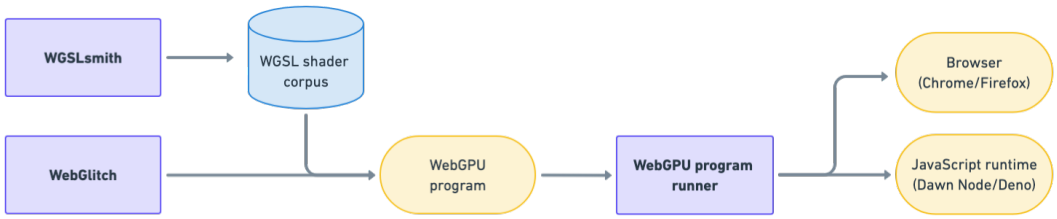

Figure 5 Overview of the WebGlitch workflow. The program runner configures the environment such that the same program generated by WebGlitch can be run within a browser and a JavaScript runtime.

which the `GPUQueue` was created, and with a size that is a multiple of 4. The buffer must also be allowed to be writeable, so it issues a call to `createBuffer` on `device` with the necessary arguments. The remaining arguments to `writeBuffer` also have restrictions (e.g., the `bufferOffset` parameter must be a multiple of 4 and the data to be written must fit within the offset and buffer size). Correspondingly, WebGlitch creates arguments that meet these requirements, employing randomisation where the requirements allow for a range of acceptable values.

Without taking care to generate valid programs by construction, it is very unlikely that even this relatively simple function would be called with valid arguments by chance.

3.5 Integrating Shader Programs

To exercise the portion of WebGPU related to shader compilation and execution, WebGlitch integrates with an existing WGSL shader generator, WGSLsmith [13, 28]. This combination leverages WGSLsmith's ability to construct a diverse range of shaders, which WebGlitch uses within the WebGPU API calls it generates (Figure 5). This results in WebGPU programs featuring a variety of GPU pipelines composed of randomised shader programs and configurations, which are ultimately executed in either a browser or JavaScript runtime environment. However, it is challenging to integrate shader programs and, by extension, WGSLsmith, into WebGlitch. API calls involving shaders often require specific arguments, such as for buffer sizes and bind group layouts, in order for them to be valid. It is difficult to automate this process without tooling to parse and understand WGSL shaders, which is challenging to get right. Rather than having WebGlitch invoke WGSLsmith during the program generation process, we instead used WGSLsmith to generate a corpus of shaders ahead of time, and manually determined the necessary bindings for these, encoding information about bindings in a JSON format that WebGlitch processes. The corpus of shaders used has varied over time: as discussed in Section 4.2 after finding that some shaders triggered interesting shader compiler bugs (which we reported), we removed them to avoid such bugs persistently occurring. We also removed some shaders due to changes in the WGSL specification which meant that certain features of WGSLsmith-generated shader programs were no longer valid.

3.6 Test Oracles

WebGlitch uses three test oracles to detect bugs: sanitiser-boosted crash oracles, the validity oracle, and differential testing. The validity oracle is enabled by default as long as the option to generate invalid programs is turned off. Straightforward crashes are also detected by default. The sanitiser-boosted crash oracle amounts to compiling the WebGPU

implementation under test with AddressSanitizer [47], which automatically enables LeakSanitizer [20], or UndefinedBehaviorSanitizer [20]. AddressSanitizer helps detect memory errors like out-of-bounds accesses and use-after-frees by making them more likely to cause a crash with a clear error message when they may otherwise go unnoticed. In a similar manner, UndefinedBehaviorSanitizer detects undefined behaviour like integer overflows and dereferencing a null pointer.

The differential testing oracle requires the user of WebGlitch to first execute the same set of generated WebGPU programs on different platforms and save the console output. We then use a script to parse and compare the results, flagging any discrepancies and notable error messages. WebGlitch then highlights programs which have unexpected differing outputs. The sanitiser-boosted crash oracle and validity oracle are straightforward to detect bugs with. Although differential testing is conceptually straightforward (run the same program on different implementations and check for differences), there are various nuances that must be considered when applying differential testing to WebGPU, which we now discuss.

Comparing Shader Outputs

One challenge is with accessing compute shader outputs, which are stored in GPU memory rather than main memory. Copying back and printing the results of every shader executed requires many additional API calls. But forcing every WebGPU program to always include these API calls would limit program diversity, and potentially prevent some bugs from being discovered. For these reasons, when invoking a compute shader WebGlitch makes a random choice as to whether each of the buffers associated with the compute shader should be copied back and printed.

Another issue applies to both graphics and compute shaders. While WebGPU contains very few implementation-dependent behaviours, the result of executing shaders can be different across different GPUs and drivers due to floating-point rounding [28]. Round-off error can lead to legitimate differences between results computed by shaders on different platforms, which could confound differential testing. As such, comparing outputs would require distance metrics and threshold values to identify differences [14], which is non-trivial. However, for compute shaders, this issue can be addressed using WGSLsmith [13, 28]. WGSLsmith only generates compute shaders and uses reconditioning [28] to avoid floating-point rounding. During reconditioning, all floating point operations are substituted with calls to functions that always return numeric values that can be represented exactly using the IEEE-754 standard. Implementation-dependent behaviours can also arise from "dynamic errors" in WGSL, which can be caused by out-of-bounds array accesses and unbounded loop iterations. WGSLsmith also avoid these behaviours are also by inserting checks on array bounds and loop iterations [28]. If WebGlitch were to be integrated with other shader generators, this would need special attention.

Comparing Error Messages

Recall that WebGlitch has a mode wherein validity checking has a configurable chance of being skipped for any receiver chosen and argument generated. In this case, we *do* expect to receive errors from the implementations under test. The type of error thrown for an invalid WebGPU API call is standardised in the WebGPU specification, so we can use differential testing to check for equivalent error output for a WebGPU program executed using different WebGPU implementations. A challenge here, however, is WebGPU does not standardise error message contents [58], so we parse and convert to an implementation-

independent representation for comparison. Different WebGPU implementations are also at varying degrees of completion [51], requiring manual analysis to determine the subset of API functions and arguments supported by all implementations for differential testing.

Differential Testing at WebGPU Program Runtime

A practical challenge is that it is difficult to directly differentially test WebGPU programs across multiple platforms as this would require platform-specific graphics APIs and drivers to be available simultaneously. To work around this, we run generated WebGPU programs separately on different platforms and log their outputs. The outputs are then parsed and compared post-execution by a separate program *FaultFinder*, which looks for discrepancies in outputs, error messages in valid programs, and crashes.

3.7 Throughput of WebGlitch

To provide an estimate of the throughput of fuzzing using WebGlitch, we used commit 29a4e87 to generate 1000 programs. We then executed each program against the Dawn implementation of WebGPU, targeting the Vulkan downstream graphics API via Node.js. Throughput was not measured on wgpu/Deno due to a bug that resulted in a Rust panic when executing almost every WebGPU program generated (Table 1, row 26). Experiments were performed on a machine with a 5.4 GHz Intel i7-13700K CPU, 64 GB DDR5 RAM, and an NVIDIA RTX 4080. They were run on Ubuntu 24.04. With default settings, WebGlitch took an average of 0.675 seconds to generate a WebGPU program. The majority of the time was spent on execution, which averaged 2.932 seconds.

4 Practical Impact on the WebGPU Ecosystem

To date, our use of WebGlitch has identified 29 bugs across the WebGPU ecosystem, consisting of 24 previously-unknown bugs and the independent discovery of 5 bugs that turned out to have already been reported.

Of the previously-unknown bugs, all have been reported, with 15 fixed following our submitted reports and 5 others confirmed.

We focused our testing efforts on two of the three major WebGPU implementations, Dawn (from Google and used by Chrome) and wgpu (primarily from Mozilla, and used in Firefox), on the major graphics APIs supported by Windows (D3D12), macOS (Metal) and Linux/Android (Vulkan).

We began by testing Dawn and wgpu through their JavaScript runtime bindings for Node.js and Deno, respectively. We then moved on to direct testing of the Chrome and Firefox browsers for completeness, because WebGPU is ultimately exposed to end users via browsers. Testing was conducted in-browser and within JavaScript runtimes on macOS,[3] Windows,[4] and Linux.[3]

Table 1 provides a summary of the 29 bugs identified by WebGlitch. The "Issue" column provides the ID of the bug in the issue tracker for the relevant project, with a hyperlink. The table is separated into specification issues, issues that might affect WebGPU support in Chromium and Firefox, shader compilation issues, issues impacting WebGPU bindings in Node.js and Deno, and JavaScript runtime bugs. WebGlitch was able to find bugs across

[3] MacBook Pro, Apple M2 Pro, 16 GB RAM
[4] Intel i7-13700K, 32 GB DDR5 RAM, NVIDIA RTX 4080

Table 1 Summary of all issues found by WebGlitch and the test oracles that found them. All "Duplicate" bugs marked with a * were first found by wg-fuzz. All other bugs classed as "Duplicate" were user-reported. FXC [34] is an HLSL shader compiler used in D3D12. The LLVMpipe driver from the Mesa 3D graphics library [33] allows for commands intended for execution on the GPU to be run on the CPU instead. Graphics libraries have been specified in parentheses for cases where bugs were specific to them. Otherwise, bugs affected all graphics libraries tested for that operating system (Linux: Vulkan, Windows: D3D11 with FXC & D3D12 with DXC, macOS: Metal).

	Issue	Type	Backend	Oracle	Description	Status
Issue affecting WGSL specification						
1	#4795	Ambiguity	N/A	Differential	Description of language behaviour when inferring types of numeric literals unclear	Fixed
Issues affecting Chromium (via Dawn)						
2	#350775867	Logic	All	Crash	Error message thrown for valid WebGPU program (exhibited only via Chrome)	Duplicate
3	#355605693	Logic	Windows	Validity	Validation error thrown for correct program	Confirmed
Issues affecting Firefox (via wgpu, or its Firefox integration)						
4	#1923237	Logic	All	Validity	Incorrectly throws error for valid program (exhibited only via Firefox)	Confirmed
5	#24798	Crash/hang	All	Crash	Crashes/hangs when specific buffer is created, followed by the destruction of a GPUDevice	Fixed
6	#24813	Logic	All	Differential	Fails to check that all WebGPU objects within a pass are from the same GPUDevice	Fixed
7	#24832	Logic	All	Differential	Missing check for GPUTextureView validity within a render pass	Fixed
8	#6040	Implementation lag	All	Validity	wgpu unknowingly lagging behind current WebGPU spec	Confirmed
9	#6041	Logic	All	Differential	Incorrect error message thrown	Fixed
Issues affecting shader compilation						
10	#6023	Logic	All	Differential	Naga: Implementation of constant evaluator deviates from WGSL specification	Fixed
11	#352496179	BSOD	Windows (FXC)	Crash	FXC: Poor performance of FXC resulted in 100% CPU and memory usage	Fixed
12	#42250583	Crash	Windows (FXC)	Crash	FXC: Valid HLSL code causes FXC to crash when paths with no side effects are encountered	Duplicate
13	#355603413	Crash	macOS	Crash	Metal shader compiler: Valid MSL involving division by 0 crashes Metal on Apple GPUs only	Confirmed
14	#372512081	ASan	Linux (LLVMpipe)	Sanitiser	LLVMpipe: Floating point exception	Unconfirmed
Issues affecting Node.js bindings for Dawn						
15	#346264226	Crash	All	Crash	Unreachable statement reached	Duplicate*
16	#346356636	Crash	All	Crash	Unreachable statement reached	Duplicate*
17	#346197748	Crash	All	Crash	Fatal error when making illegal API call	Duplicate*
18	#364871809	Crash	All	Crash	Fatal error when retrieving error message	Unconfirmed
19	#351644575	ASan	All	Sanitiser	Heap-use-after-free (object accessed after destruction)	Fixed
20	#355006329	LeakSan	All	Sanitiser	Not all resources freed on program exit	Fixed
Issues affecting wgpu integration in Deno						
21	#24806	Crash	All	Crash	Missing argument in error message constructor	Fixed
22	#24672	Logic	All	Differential	Fails to throw error message for invalid render/compute pass	Fixed
23	#24821	Logic	All	Differential	Arguments used in an unrelated render pass affects compute pass output	Confirmed
24	#25874	Logic	All	Validity	GPUAdapter incorrectly invalidated after specific API call	Fixed
25	#25870	Crash	All	Crash	Panics when destroyed texture used in render pass	Fixed
26	#28966	Crash	All	Crash	Panics when command encoder is finished	Unconfirmed
Issues affecting JavaScript runtimes (unrelated to WebGPU)						
27	#24575	LeakSan	All	Sanitiser	Deno leaks memory when JavaScript VM is exited	Fixed
28	#849	LeakSan	All	Sanitiser	Deno leaks memory when JavaScript VM is exited	Fixed
29	#54178	LeakSan	Linux	Sanitiser	Node.js leaks memory when "requiring" a non-existent module	Unconfirmed

the WebGPU ecosystem, including WebGPU implementations, WGSL compilers, the WGSL specification itself, downstream shader compilers, and JavaScript runtimes. Four new bugs (Table 1, rows 6, 7, 24, 25) were found by executing WebGPU programs containing invalid API calls, highlighting the importance of designing WebGlitch with a configuration that allows for invalid API calls in addition to its default mode where generated programs are valid-by-construction (see Section 3.3). The remaining bugs were all found through programs that were valid-by-construction. The range of sources within which bugs were detected demonstrates the importance of fuzz testing WebGPU in as many environments as possible.

We now elaborate on some of the bugs found by WebGlitch and their impact on the WebGPU ecosystem.

4.1 Specification Issue

WGSL is statically-typed and supports integer types of various bit-widths [59]. It also allows the bit-widths of certain numeric expressions to be left unspecified and inferred based on usage. This concept is referred to as *abstract numerics* [59]. The WGSL specification requires that expressions involving abstract numerics are checked at compile-time for overflow, so that a program is statically rejected if the value of an abstract numeric does not fall within the allowed range of values supported by a type of a specific bit-width.

To illustrate this, consider the first declaration in Figure 6. The abstract numeric expression `-1 * -2147483648` evaluates to `2147483648`, which is too large to be represented by the `i32` type (a 32-bit signed integer). The declaration should thus be rejected.

```
// Correctly rejected by Tint & Naga
const num1 : i32 = -1 * -2147483648

// Incorrectly rejected by Naga only
const num2 : i32 = -1 * i32(-2147483648)
```

Figure 6 We fixed a problem in the WGSL language specification where rules for implicit type casting were ambiguous. This was brought to light by the Naga compiler incorrectly rejecting the second declaration in this example.

In contrast, the second declaration should *not* be rejected. This is because the expression `i32(-2147483648)` specifies that `-2147483648` should be stored as a 32-bit signed integer, which is allowed because the value is in the representable range of the `i32` type. Overload resolution means that the abstract numeric argument `-1` is implicitly converted to an `i32` value (as an abstract numeric value is used with an `i32` value). The WGSL specification mandates two's complement semantics for `i32` operations, so the overall expression evaluates to `-2147483648`.

Differential testing between Dawn and wgpu revealed that the Tint shader compiler (used by Dawn) was behaving correctly, while the Naga shader compiler (used by wgpu) was incorrectly rejecting the second declaration [39].

However, when we investigated this discrepancy, it was not clear to us from the wording of rules about abstract numerics in the WGSL specification which of Tint or Naga were behaving correctly. Our understanding of the type-checking rules described above only came about after discussions with engineers at Google leading the development of the WGSL specification: they were not clear from the specification itself. We submitted a proposal to clarify this behaviour in the specification, which has since been accepted and incorporated [52]. After investigation, the bug in Naga stemmed from its constant evaluator rather than the way in which it handles type casting. A fix to then align the behaviour of Naga with the WGSL specification was submitted and accepted [55].

4.2 Bugs Affecting In-browser Execution

We found 13 previously-unknown bugs that affected either the core logic of a WebGPU implementation or downstream shader compiler behaviour. Although we found most of these bugs by fuzzing using JavaScript runtimes, rather than through in-browser fuzzing, they would affect any WebGPU workload that exercised these parts of the relevant WebGPU implementations, or that used shaders that would trigger the shader compiler bugs. Discussed below, we also found some bugs via in-browser testing that do not trigger via out-of-browser JavaScript runtimes, demonstrating the value of also conducting in-browser testing.

Bugs in the Implementation of API Functions

We found 8 bugs arising due to the incorrect implementation of API functions (7 of them previously-unknown); these are under the "Issues affecting Chromium (via Dawn)" and "Issues affecting Firefox (via wgpu, or its Firefox integration)" headings in Table 1. We now discuss a selection of these.

Two bugs were due to wgpu failing to reject invalid API calls related to configuring render passes (Table 1, rows 6, 7). One WebGlitch program generated by WebGlitch (shown in reduced form in Figure 7) caused wgpu to crash on macOS, and to hang on Linux and Windows.

```
const adapter = await navigator.gpu.requestAdapter();
const device = await adapter.requestDevice();
const buffer = device.createBuffer({
  mappedAtCreation: true,
  size: 12,
  usage: GPUBufferUsage.COPY_SRC
});
device.destroy();
```

Figure 7 Minimal test case causing wgpu to hang on Linux and Windows but crash on macOS.

Identifying the component responsible for this bug was difficult due to the layered architecture in use: it was initially detected while testing Deno, which builds on top of the Rust-native wgpu. As such, we had to manually translate the WebGPU program to a corresponding sequence of direct Rust API calls to confirm the bug was in wgpu.

Two bugs could *only* be triggered by in-browser fuzz testing with Firefox and Chrome – i.e. they could not be found by fuzzing against the Node.js and Deno runtimes.

For Firefox, this is expected: like Deno, Firefox uses wgpu, but builds on top of wgpu in a separate and non-trivial way to enable its use through the Firefox internal JavaScript engine. As such, bugs in this layer are only reachable by fuzz testing in Firefox; this is the case for the bug that we found (Table 1, row 4).

In the case of the bug found in Chrome, which led to an error being thrown for a valid WebGPU program (Table 1, row 2), it is unclear why it was not reachable through Dawn's Node.js bindings, and the cause of the bug is still under investigation.

Shader Compiler Bugs

In addition to the Naga bug discussed in Section 4.1 and Figure 6, we found 4 bugs affecting downstream shader compilers of specific graphics APIs, demonstrating the need to thoroughly test on a range of platforms.

One such bug (Table 1, row 13) affected the Metal compiler for Apple's Metal shading language. Recall from Section 2 that the WGSL compiler in a WebGPU implementation emits code for a downstream shading language. On macOS, we found that Tint, the WGSL compiler that ships with Dawn, generated Metal shading language code that caused the Metal compiler to fail. Determining that the problem was in the Metal compiler was non-trivial due to the layered nature of WebGPU: it is often unclear whether some invalid program was compiled by the WGSL compiler, or whether the source of the bug is further downstream. Moreover, this bug only affected shader compilation when targeting Apple GPUs – we found the crash did not happen when targeting Intel and AMD GPUs on Apple platforms, showing the importance of testing on different hardware configurations. To avoid repeatedly triggering this bug (which would impede the discovery of further bugs), we had to remove the offending shader from the corpus of shaders used by WebGlitch, something that we had to do for various shader programs that would reliably trigger compiler bugs.

4.3 Bugs Related to Runtime Bindings

A number of the previously-unknown bugs we found – 6 in total – turned out to relate to problems with the bindings between the Node.js and Deno JavaScript engines and the Dawn and wgpu implementations of WebGPU, and could not be triggered via in-browser execution (which does not use these bindings).

```
const adapter = await navigator.gpu.requestAdapter();
const device = adapter.requestDevice();
console.log(adapter.info);
```

Figure 8 Minimal test case where valid WebGPU program rejected by Deno.

While less serious in terms of impact than browser-related bugs, our reports of these problems improved the stability of these runtime bindings, and early fixes for some of these issues unblocked our work, allowing us to perform more intensive testing of Dawn and wgpu via these JavaScript runtimes without the binding-related problems masking the presence of other bugs.

Figure 8 shows a reduced test case that triggered a bug in Deno (Table 1, row 24). Our report led to this being raised in the wgpu project's weekly WebGPU meeting [54]. Within this test case, the `adapter.info` call was incorrectly rejected by Deno, throwing an error message that the `GPUAdapter` had been invalidated because a `GPUDevice` had been created from it.

In the Dawn bindings for Node.js we discovered a use-after-free (Table 1, row 19) and memory leak (Table 1, row 20). These were problematic for our fuzz testing campaign: even a trivial program Figure 9, which consists of just the entry point into the WebGPU API, triggered ASan and LeakSan errors Figure 9. Since these errors were thrown upon using the WebGPU API, we had to disable AddressSanitizer and LeakSanitizer while testing Dawn until engineers at Google fixed the problems. Because these bugs were limited to Dawn's Node.js bindings, which, understandably, are a lower priority compared than issues reproducible in Chrome, they took longer to address. As such, it temporarily blocked the discovery of a few other bugs in Dawn.

```
const adapter = await navigator.gpu.requestAdapter();
const device = await adapter.requestDevice();
```

(a) Minimal reproducible WebGPU program that raised repeated Sanitiser errors.

```
==87490==ERROR: AddressSanitizer: heap-use-after-free on address 0x00010d317d38 at pc 0x000127ec6a08 bp 0x00016b21e190 sp 0x00016b21e188
  WRITE of size 1 at 0x00010d317d38 thread T0
SUMMARY: AddressSanitizer: heap-use-after-free AsyncRunner.cpp:63 in wgpu::binding::AsyncRunner::QueueTick(Napi::Env)
  ::$_0::operator()(Napi::CallbackInfo const&) const

==17903==ERROR: LeakSanitizer: detected memory leaks
Direct leak of 108 byte(s) in 1 object(s) allocated from:
  #0 0x7f660d2c1357 in operator new[](unsigned long) ../../../../src/libsanitizer/asan/asan_new_delete.cpp:102
SUMMARY: AddressSanitizer: 60285 byte(s) leaked in 55 allocation(s).
```

(b) Erroneous output detected by WebGlitch. The full error message has been excluded for brevity.

Figure 9 Minimal reproducible test case and console output for a bug raising ASan and LeakSan errors on Dawn node.

4.4 Bugs Affecting JavaScript Runtimes

Our testing led to the discovery of three previously-unknown bugs – all memory leaks – that affected the Node.js and Deno JavaScript runtimes and which turned out to be unrelated to WebGPU: they could be triggered by JavaScript programs that do not issue any WebGPU

API calls. We found such errors by running generated WebGPU programs against WebGPU implementations within JavaScript runtimes. When we reduced these programs to their minimal reproducible example, in each case we found that we could remove all WebGPU API calls while preserving the memory leak. This led to very simple JavaScript programs, e.g. a program that simply imports a module that cannot be found. The lack of any use of WebGPU indicated that such bugs lay within the underlying runtimes. One of the memory leak bug reports we submitted to Deno (Table 1, row 27) helped identify the root cause of an earlier, elusive memory leak that had been challenging to track down. Although this was not a WebGPU-related issue, and we did not set out to discover such types of issues, the impact of this bug was arguably more substantial than that of WebGPU-specific bugs as it affects all developers using Deno, not just those working with WebGPU.

5 Code Coverage Achieved by WebGlitch

To evaluate how thoroughly fuzzing with WebGlitch exercises a WebGPU implementation, we report on statement coverage on the Dawn codebase achieved via a short fuzzing campaign. We compare this to coverage achieved through fuzzing with wg-fuzz. This is the only other open source fuzzer we are aware of for the WebGPU API [48], and the coverage achieved by running the WebGPU conformance test suite (CTS) [50]. The WebGPU CTS is a fixed test suite to check if a WebGPU implementation conforms to the specification, ensuring coverage results gathered from it are comprehensive. We decided to focus on Dawn because at time of writing it is more mature than other open source WebGPU implementations.

Dawn already ships with tooling to measure coverage achieved by the CTS using llvm-cov [30]. To maintain a consistent environment for collecting coverage data, we generated 2000 WebGPU programs using each of WebGlitch and wg-fuzz, but in a format expected by the CTS, so that the Dawn coverage infrastructure could work with these programs as if they were CTS tests. We attempted using more WebGPU programs but found this led to an out-of-memory error as it exceeded the available RAM (32 GB) in our test system. However, analysis of coverage from one of these aborted experiments suggested that coverage had already saturated after 2000 programs.

Among the 2000 programs generated by each fuzzer, 1000 programs were valid by construction, and the remaining programs had a 10% chance for any given call to be invalid.

We then collected coverage data on the Dawn codebase separately by executing (a) the 2000 programs generated by WebGlitch, (b) the 2000 programs generated by wg-fuzz, and (c) the WebGPU CTS. Due to the randomised nature of WebGlitch and wg-fuzz, we repeated coverage collection for these tools three times, using three different sets of 2000 programs, and averaged the absolute percentage coverage values that we obtained. We then analysed the results to identify coverage gaps in the CTS: lines of code in Dawn covered by the WebGPU fuzzers but not by the CTS.

WebGlitch, wg-fuzz, and the CTS were able to cover 13.52%, 15.71%, and 28.00% of Dawn, respectively [56]. These values are all low because many parts of a WebGPU implementation are graphics API-specific, and it is not possible to execute the entirety of any WebGPU implementation on one machine; e.g. the macOS platform that we used for this experiment would only exercise the Metal back-end of Dawn. Even though the coverage of Dawn achieved by both WebGlitch and wg-fuzz is lower than that of the CTS, both fuzz testing tools successfully uncovered new bugs. Absolute coverage alone does not capture the distinction between deep, exploratory execution paths and broad functional coverage. As such, these fuzzing tools serve as a valuable complement to the CTS.

WebGlitch and wg-fuzz both achieve similar coverage, but the coverage achieved by wg-fuzz is slightly higher. This is likely because wg-fuzz covers all API functions, unlike WebGlitch which currently supports 75% of all WebGPU API functions. However, WebGlitch can generate many optional arguments that are not considered by wg-fuzz. Supporting these optional arguments required additional engineering to ensure that generated programs are still valid by construction. The use of these optional arguments will lead to different code paths being taken in the underlying WebGPU implementation under test. Despite WebGlitch supporting fewer WebGPU API functions, it managed to cover 1,530 lines that wg-fuzz missed and uncovered many bugs that wg-fuzz could not. Conversely, wg-fuzz covered 5,180 lines of Dawn that WebGlitch missed. Together, these tools complement each other.

Both WebGlitch and wg-fuzz were also able to cover lines of Dawn missed by the CTS, although most were related to formatting outputs. But among these lines of code was a Dawn function entirely missed by the CTS, and this function was linked to a correctness bug within Dawn that was identified by WebGlitch (Table 1, row 3). This underlines the value that fuzzing can play in identifying and filling coverage gaps in a CTS, as prior work on shader compiler fuzzing has demonstrated [15].

6 Insights and Lessons Learned

We now summarise a number of key lessons learned from our experience designing and deploying WebGlitch, which we hope will be insightful to other researchers and practitioners interested in API fuzzing.

On-demand Generation of API Calls Is Good for Bug Finding

Recall from Section 3 that WebGlitch takes an *on-demand* approach to API call sequence generation, whereby the conditions for a particular API function to be called are satisfied by issuing previous API function calls to create necessary ingredients. In contrast, the wg-fuzz fuzzer for WebGPU takes a more cautious approach where an API function is only selected for invocation if suitable prerequisite objects are already in scope.

WebGlitch uncovered many bugs (24) that wg-fuzz could *theoretically* find, but missed in practice due to its conservative approach: the probability of specific calls being generated is low. For example, the minimal reproducible test case for one identified bug (Table 1, row 23) consisted of a valid WebGPU program with 24 WebGPU API calls and 15 different WebGPU objects. It is already unlikely that a generated program will contain these 24 API calls in the required sequence, and the probability of generating such a program would be even lower when using a conservative approach that only invokes API functions using objects already available in the partially generated program. WebGlitch is capable of exercising API functions that require complex dependencies by creating these on-demand, and the success of WebGlitch at finding bugs in practice provides evidence that the tool is effective at exercising complex WebGPU implementations in depth.

Based on our experience, we recommend considering an on-demand approach when designing a generator of programs that make calls into an API whose functions have non-trivial prerequisites.

Fuzz Blockers Limit the Automated Nature of Fuzzing

Although much of the fuzz testing process can be automated (test case generation, execution, and analysis), our fuzz testing campaign required frequent manual intervention to make fuzzing productive. While using WebGlitch to find bugs is fully automatic, since WebGlitch

uses a black box approach, determining the root causes of bugs is still manual. This was mainly due to *fuzz blockers*, where "a fuzzer encounters the same bug very often, which blocks further fuzzing of downstream code" [12]. In our setting this corresponds to bugs in a WebGPU implementation (or bugs affecting a JavaScript runtime) that are extremely easy to trigger, manifesting so often that they make it hard to find deeper bugs. For example, we had to identify and remove certain shaders that had a high probability of triggering shader compiler bugs, or disable sanitisers until certain bugs were addressed by maintainers.

The issue of fuzz blockers is well known to fuzzing experts; e.g. Google's ClusterFuzz framework assigns a "Fuzz-blocker" label to a bug that is deemed to trigger so frequently that it hurts fuzzer performance [4], and indeed fuzz blockers were a real impediment to our work deploying WebGlitch in practice.

Finding the Root Cause of a Bug Within a Layered API Is Unclear

Generally, the standardization of WebGPU as JavaScript API makes the task of testing WebGPU implementations easier than it would otherwise be: a single generated WebGPU program can be executed across a range of operating systems and graphics drivers, potentially triggering bugs in any of the components shown in the layered diagram of Figure 1.

However, standardization also presents a challenge: when a test fails it can be hard to identify the source of the bug due to the layered nature of the API. Root-causing bugs requires a degree of detective work, e.g. ascertaining that a bug is likely to be Vulkan-specific because it triggers only when a Vulkan back-end is used, regardless of operating system or graphics driver, or ascertaining that a bug lies in an NVIDIA graphics driver by differentially testing across drivers from multiple vendors. This insight is likely to be relevant to fuzzing efforts for other APIs with diverse, layered implementations.

Testing Implementations of an Evolving Standard: A Challenge and an Opportunity

We also faced challenges due to the WebGPU API standard being incomplete during our fuzzing efforts. A benefit of fuzzing at this early stage was that we were able to find some issues with the specification of the API itself, prompting discussion and refinement. However, it meant that we had to actively work on keeping the fuzzer up-to-date with the evolving specification. We also had to be mindful of whether implementations under test were up-to-speed with the latest specification changes, otherwise updating the fuzzer to reflect the latest specification changes would waste developer time by identifying known issues where developers have not yet caught up with such changes. Similar observations were made in recent work on the deployment of compiler fuzzing techniques for WGSL, where early fuzzing of WGSL compilers uncovered an edge case unaccounted for by the specification, showcasing a benefit of applying fuzzing at an early stage, but where the evolving WGSL syntax introduced additional engineering overhead and posed similar challenges with differential testing [13].

This is a trade-off that should be borne in mind when deciding how early to commence a fuzzing effort for implementations of a new API.

Diversity of Testing Environments Is Important

Our work has benefited from the application of WebGlitch to WebGPU implementations in the context of a variety of browsers and operating systems.

However, it is likely that some bugs that WebGlitch would be capable of finding have gone undetected because the range of environments in which we were able to test is nevertheless limited. For example, some bugs affecting WebGPU are known to be specific to operating

system versions [1], or hardware-specific [18]. Any configuration that is left untested is a potential source of undiscovered bugs. This insight is particularly applicable to performance-critical APIs where it is likely that implementations of the API will leverage low level features of operating systems and hardware.

Supporting a new environment in the testing process is not always straightforward. For example, enabling sanitisers on macOS requires a particular environment variable to be set, but this variable is cleared prior to command execution for security purposes. Although there are workarounds, some can be fairly involved [46].

We are working with the WebGPU team at Google to deploy WebGlitch as part of ClusterFuzz, Google's continuous fuzzing infrastructure, which will ensure that fuzzing of WebGPU support in Chromium is conducted on a wide range of target platforms.

7 Related Work

WebGlitch generates valid-by-construction WebGPU programs from scratch, inspired the random C program generator Csmith [60]. Both these fuzzers are specific to a system under test. Many API fuzzers are more general [21, 23, 31, 5]. Among these, RESTler [5] is notable for generating syntactically correct calls by parsing an IDL and attempts to learn the sequences of calls required for validity by gathering and analysing the API call responses. FuzzGen [22], a fuzzer for C++ libraries, attempts to determine which sequences of API calls are valid by constructing a dependency graph of API calls based on analysis of a corpus of programs that use a given API. This is useful when a library's API has not been defined using an IDL. Although applicable to many different APIs, these fuzzers are unlikely to capture all nuances of any specific API, especially related to call validity.

Like Fuzzgen, the Randoop fuzzer for Java classes [43] also determines valid sequences of function calls at runtime. The design of WebGlitch is also similar to Randoop [43]: they both randomly choose a method to call and populate it with arguments created from previously generated calls. But rather than having prior knowledge of each API function's requirements, dependencies, and exactly how to construct each object like WebGlitch, Randoop executes the partially generated program as soon as it is created [43]. It then checks the result of executing the program against a set of pre-defined rules (e.g., a function should not throw an exception under certain conditions). If executing the partially-generated program did not throw any errors or violate any rules, then it is added to a corpus of partially-generated programs, otherwise it is discarded, unless the program threw an error while not violating any conditions, suggesting a potential bug. Randoop can then take objects created within these partially-generated programs and use them in another method call by extending the partially-generated program [43].

One drawback of this approach is that the search space for programs is large as Randoop does not have exact knowledge of which arguments are valid. Moreover, program generation can be inefficient as each time a method call is added, it must be executed to check if it is still valid. Most importantly, such an approach would not work well with the WebGPU API: the API contains many functions to configure the state of objects such as render/compute pipelines, buffers, and textures. However, errors are not always thrown immediately: it is possible to pass invalid configurations to an object, but if that object is never passed to the GPU for manipulation, no error is thrown. Given how unlikely it is that a partially-generated program within the corpus could be extended to include a valid call that passes objects to the GPU, most programs would fail to meaningfully exercise the underlying graphics

APIs. While this could potentially be addressed by adding extensive custom validation when executing programs, it would likely be more efficient to adopt a fuzzing approach that is tailored towards WebGPU instead.

One such example is wg-fuzz [48], which generates valid WebGPU programs from the ground up like WebGlitch, but takes a different approach to ensuring validity. It iteratively chooses random API calls to construct based on the state of the partially-generated program rather than first constructing all the API calls needed for a chosen call to be valid like WebGlitch. WebGlitch nevertheless found 25 bugs that wg-fuzz could not and rediscovered 3 that wg-fuzz had already reported (Section 4). That said, the two tools remain complementary, as wg-fuzz can cover some parts of Dawn that WebGlitch does not and vice versa (Section 5).

Fuzzing the WebGL and WebGPU ecosystems has not been limited to program generation from scratch. GLeeFuzz [44], which targets WebGL, uses an error message-guided mutation approach. GLeeFuzz is not aware of the validity of the programs it mutates, but uses the error messages thrown by mutant programs to guide its mutation. It uses error message diversity as a proxy for coverage, intensively exercising the call validation aspect of WebGL implementations. A GLeeFuzz-like technique would be interesting to investigate for WebGPU, but because its mutation-based approach primarily generates invalid programs, it would likely serve as a complement to WebGlitch, which focuses on generating valid programs.

GLFuzz [14] is a testing technique targeting compilers for the OpenGL shading language, a language similar to WGSL and used by the OpenGL, OpenGL ES and WebGL APIs [24]. GLFuzz uses metamorphic testing [7], wherein semantics-preserving mutations are applied to a seed program and the outputs of the original and mutant are compared for differences. Unlike WebGlitch, GLFuzz focuses solely on shader compiler testing and not on API testing.

There has also been work on fuzzing WebGPU's embedded shading language, WGSL, rather than the WebGPU API itself [6, 13, 28]. The WGSLsmith tool [13, 28] exclusively tests WebGPU shader compilers by generating random WGSL shader programs and detecting errors during compilation or execution of the shaders post-compilation. However, WGSLsmith exercises the WebGPU API in a limited way, as WGSL shaders are executed using a fixed WebGPU test harness. By contrast, WebGlitch makes extensive use of the broader WebGPU API in general and incorporates a fixed corpus of shaders into the API calls it generates, complementing WGSLsmith's shader-focussed approach. A recent paper [13] reports on the use of coverage-guided mutation-based fuzzing (via libFuzzer [29]) and black box fuzzing (via WGSLsmith and spirv-fuzz [17]) to directly test the Tint shader compiler in Google's Dawn WebGPU implementation, and to test tooling associated with SPIR-V, the Vulkan shading language. DarthShader [6] combines both program generation and mutation, fuzzing WGSL compilers in a coverage-guided manner. It has successfully found numerous bugs including several high severity CVEs affecting Chrome [6]. Part of its success can be attributed to its ability to mutate both the abstract syntax tree representation of a shader and its intermediate representation [6], enabling it to more effectively find bugs across different compilation stages. However, this approach is not directly applicable to the WebGPU API layer.

As discussed in Section 6, the WebGlitch and WGSLsmith projects both required manual effort to keeping the respective tools up-to-date with changes to the WebGPU specification. Furthermore, both tools tailored specifically to the WebGPU setting. Recent work has investigated Large Language Model (LLM)-based approaches that could potentially address these limitations [9, 10]. Both TitanFuzz [9] and FuzzGPT [10] use program generation techniques that can be applied to any API or programming language. TitanFuzz first gives a LLM step-by-step instructions to generate seeds, which are then mutated using an evolutionary algorithm to mask portions of the seed. This is then fed back to an LLM for

infilling, and mutants that execute without errors can then act as new seeds [9]. FuzzGPT extends TitanFuzz by either prompting or fine-tuning the LLM with code snippets from historical bugs to generate more unusual programs [10]. Both have been evaluated on the deep learning libraries PyTorch and TensorFlow, finding 21 and 27 new bugs, respectively via crash oracles and differential testing.

We speculate that the effectiveness of these LLM-based approaches on fuzzing for WebGPU may be more limited, due to the much smaller amount of WebGPU data available for training and/or fine-tuning compared with that for PyTorch and TensorFlow, and the fact that some existing training data may already be obsolete due to recent changes to the WebGPU standard [58]. These issues would also apply to generating WGSL shaders for use in such WebGPU programs. Given that WebGPU also requires complex setups of object states and buffers in specific sequences for calls to be valid, it would be more challenging for LLMs to produce valid programs that meaningfully exercise downstream graphics APIs (roughly a third of the programs generated by TitanFuzz were valid [9]).

As discussed in Sections 4.1 and 6, the deployment of fuzzing can identify problems not only in software but also in language specifications. WebGlitch was able to detect an area of ambiguity in the WGSL specification, and similar issues have been found before by WGSLsmith [13] and other fuzzers for the programming language Dafny [16].

8 Conclusions and Future Work

We have introduced a new method for fuzzing the WebGPU API that generates sequences of API calls that are valid-by-construction, based on careful modelling of the requirements of API functions, and an on-demand generation approach. We have implemented this in a new open source tool, WebGlitch, which has helped to find bugs across the WebGPU ecosystem – in the WGSL specification, the Dawn and wgpu implementations of WebGPU, the Node.js and Deno JavaScript runtimes, and various downstream shader compilers – in a manner that would be difficult to achieve with manually written tests alone.

Avenues for future work include testing of Apple's WebGPU implementation in WebKit/Safari once it is in a more complete state, as well testing on a broader range of hardware configurations (the importance of this is evidenced by the fact that one of our bugs was specific to Apple GPUs). Currently, WebGlitch supports around 75% of all functions in the WebGPU API; adding support for the full API would improve the thoroughness of testing and might increase bug-finding ability. In the meantime, WebGlitch is ready for browser development teams in industry to integrate into their testing processes to help identify bugs as early as possible.

References

1 Anon. Automated tests fail on macOS 10.13, 2024. Accessed 23rd April 2025. URL: `https://github.com/gfx-rs/wgpu/issues/4632`.

2 Apple. Metal, 2024. Accessed 23rd April 2025. URL: `https://developer.apple.com/documentation/metal`.

3 Apple. Metal shading language specification, 2024. Accessed 23rd April 2025. URL: `https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf`.

4 Abhishek Arya and Oliver Chang. ClusterFuzz: Fuzzing at Google scale. In *Black Hat Europe 2019*, 2019. Accessed 23rd April 2025. URL: `https://i.blackhat.com/eu-19/Wednesday/eu-19-Arya-ClusterFuzz-Fuzzing-At-Google-Scale.pdf`.

5 Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. RESTler: stateful REST API fuzzing. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 748–758. IEEE / ACM, 2019. `doi:10.1109/ICSE.2019.00083`.

6 Lukas Bernhard, Nico Schiller, Moritz Schloegel, Nils Bars, and Thorsten Holz. DarthShader: Fuzzing WebGPU shader translators & compilers. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, pages 690–704. ACM, 2024. doi:10.1145/3658644.3690209.

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

8 Jon Davis. Release notes for Safari Technology Preview 185, 2024. Accessed 23rd April 2025. URL: https://www.webkit.org/blog/14885/release-notes-for-safari-technology-preview-185/.

9 Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 423–435. ACM, 2023. doi:10.1145/3597926.3598067.

10 Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 70:1–70:13. ACM, 2024. doi:10.1145/3597503.3623343.

11 Deno. Deno, 2024. Accessed 23rd April 2025. URL: https://github.com/denoland/deno.

12 Zhen Yu Ding and Claire Le Goues. An empirical study of OSS-fuzz bugs. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*, pages 131–142. IEEE, 2021. doi:10.1109/MSR52588.2021.00026.

13 Alastair F. Donaldson, Ben Clayton, Ryan Harrison, Hasan Mohsin, David Neto, Vasyl Teliman, and Hana Watson. Industrial deployment of compiler fuzzing techniques for two GPU shading languages. In *IEEE Conference on Software Testing, Verification and Validation, ICST 2023, Dublin, Ireland, April 16-20, 2023*, pages 374–385. IEEE, 2023. doi:10.1109/ICST57152.2023.00042.

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

15 Alastair F. Donaldson, Hugues Evrard, and Paul Thomson. Putting randomized compiler testing into production (experience report). In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, volume 166 of *LIPIcs*, pages 22:1–22:29. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.ECOOP.2020.22.

16 Alastair F. Donaldson, Dilan Sheth, Jean-Baptiste Tristan, and Alex Usher. Randomised testing of the compiler for a verification-aware programming language. In *IEEE Conference on Software Testing, Verification and Validation, ICST 2024, Toronto, ON, Canada, May 27-31, 2024*, pages 407–418. IEEE, 2024. doi:10.1109/ICST60714.2024.00044.

17 Alastair F. Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpinski. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, pages 1017–1032. ACM, 2021. doi:10.1145/3453483.3454092.

18 Benjamin Fyre. Test failures on Dx12/AMD, 2024. Accessed 23rd April 2025. URL: https://github.com/gfx-rs/wgpu/issues/6830.

19 Google. Dawn, 2024. Accessed 23rd April 2025. URL: https://github.com/google/dawn.

20 Google. Sanitizers, 2024. Accessed 23rd April 2025. URL: https://github.com/google/sanitizers.

21 Harrison Green and Thanassis Avgerinos. Graphfuzz: Library API fuzzing with lifetime-aware dataflow graphs. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1070–1081. ACM, 2022. doi:10.1145/3510003.3510228.

22 Kyriakos K. Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. FuzzGen: Automatic fuzzer generation. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2271–2287. USENIX Association, 2020. Accessed 23rd April 2025. URL: https://www.usenix.org/system/files/sec20-ispoglou.pdf.

23 Jianfeng Jiang, Hui Xu, and Yangfan Zhou. RULF: Rust library fuzzing via API dependency graph traversal. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pages 581–592. IEEE, 2021. doi:10.1109/ASE51524.2021.9678813.

24 Khronos Group. The OpenGL shading language, version 4.60.8, 2023. Accessed 23rd April 2025. URL: https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.60.pdf.

25 Khronos Group. Khronos Vulkan Registry, 2024. Accessed 23rd April 2025. URL: https://registry.khronos.org/vulkan/.

26 Khronos Group. SPIR-V specification, 2024. Accessed 23rd April 2025. URL: https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html.

27 Khronos Group. WebGL specification, 2024. Accessed 23rd April 2025. URL: https://registry.khronos.org/webgl/specs/latest/1.0/.

28 Bastien Lecoeur, Hasan Mohsin, and Alastair F. Donaldson. Program reconditioning: Avoiding undefined behaviour when finding and reducing compiler bugs. *Proc. ACM Program. Lang.*, 7(PLDI):1801–1825, 2023. doi:10.1145/3591294.

29 LLVM Compiler Infrastructure. libFuzzer – a library for coverage-guided fuzz testing, 2024. Accessed 23rd April 2025. URL: http://llvm.org/docs/LibFuzzer.html.

30 LLVM Project. llvm-cov - emit coverage information, 2024. Accessed 23rd April 2025. URL: https://llvm.org/docs/CommandGuide/llvm-cov.html.

31 Chenyang Lyu, Jiacheng Xu, Shouling Ji, Xuhong Zhang, Qinying Wang, Binbin Zhao, Gaoning Pan, Wei Cao, Peng Cheng, and Raheem Beyah. MINER: A hybrid data-driven approach for REST API fuzzing. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 4517–4534. USENIX Association, 2023. Accessed: 23rd April 2025. URL: https://www.usenix.org/system/files/usenixsecurity23-lyu.pdf.

32 William M. McKeeman. Differential testing for software. *Digit. Tech. J.*, 10(1):100–107, 1998. Accessed: 23rd April 2025. URL: http://www.dtjcd.vmsresource.org.uk/pdfs/dtj_v10-01_1998.pdf.

33 Mesa 3D project. LLVMpipe, 2024. Accessed 23rd April 2025. URL: https://docs.mesa3d.org/drivers/llvmpipe.html.

34 Microsoft. Effect-Compiler Tool, 2020. Accessed 23rd April 2025. URL: https://learn.microsoft.com/en-us/windows/win32/direct3dtools/fxc.

35 Microsoft. DirectX Shader Compiler, 2024. Accessed 23rd April 2025. URL: https://github.com/microsoft/DirectXShaderCompiler.

36 Microsoft Corporation. GPU-based validation and the Direct3D 12 debug layer, 2021. Accessed 23rd April 2025. URL: https://learn.microsoft.com/en-us/windows/win32/direct3d12/using-d3d12-debug-layer-gpu-based-validation.

37 Microsoft Corporation. DirectX graphics and gaming, 2022. Accessed 23rd April 2025. URL: https://learn.microsoft.com/en-us/windows/win32/directx.

38 Microsoft Corporation. HLSL specifications, 2024. Accessed 23rd April 2025. URL: https://github.com/microsoft/hlsl-specs.

39 David Neto and Alan Baker. Personal correspondence, July 2024.

40 NIST National Vulnerability Database. CVE-2016-2824, 2018. Accessed 23rd April 2025. URL: https://nvd.nist.gov/vuln/detail/CVE-2016-2824.

41 NIST National Vulnerability Database. CVE-2012-3968, 2020. Accessed 23rd April 2025. URL: https://nvd.nist.gov/vuln/detail/CVE-2012-3968.

42 NIST National Vulnerability Database. CVE-2017-5112, 2023. Accessed 23rd April 2025. URL: https://nvd.nist.gov/vuln/detail/CVE-2017-5112.

43 Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 75–84. IEEE Computer Society, 2007. doi:10.1109/ICSE.2007.37.

44 Hui Peng, Zhihao Yao, Ardalan Amiri Sani, Dave Tian, and Mathias Payer. GLeeFuzz: Fuzzing WebGL through error message guided mutation. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 1883–1899. USENIX Association, 2023. Accessed 23rd April 2025. URL: https://www.usenix.org/system/files/usenixsecurity23-peng.pdf.

45 Rust Graphics Mages. wgpu, 2024. Accessed 23rd April 2025. URL: https://github.com/gfx-rs/wgpu?tab=readme-ov-file.

46 Hynek Schlawack. Surprising consequences of macOS's environment variable sanitization, 2023. Accessed 23rd April 2025. URL: https://hynek.me/articles/macos-dyld-env/.

47 Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Address-Sanitizer: A fast address sanity checker. In Gernot Heiser and Wilson C. Hsieh, editors, *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX ATC 2012, Boston, MA, USA, June 13-15, 2012*, pages 309–318. USENIX Association, 2012. Accessed 23rd April 2025. URL: https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf.

48 Daniel Simols. wg-fuzz, 2024. Accessed 23rd April 2025. URL: https://github.com/wg-fuzz/wg-fuzz.

49 Brendon Tiszka, Chris Bookholt, and Matthew Riley. WebGPU technical report. Technical report, Google, 2023. Accessed 23rd April 2025. URL: https://chromium.googlesource.com/chromium/src/+/main/docs/security/research/graphics/webgpu_technical_report.md?pli=1#.

50 W3C group for GPU web standards. Conformance Test Suite, 2024. Accessed 23rd April 2025. URL: https://github.com/gpuweb/cts.

51 W3C group for GPU web standards. Implementation status, 2024. Accessed 23rd April 2025. URL: https://github.com/gpuweb/gpuweb/wiki/Implementation-Status.

52 Matthew Wong. Add extra example for operations involving abstract numerics, 2024. Accessed 23rd April 2025. URL: https://github.com/gpuweb/gpuweb/pull/4795.

53 Matthew Wong. WebGlitch, 2024. Accessed 23rd April 2025. URL: https://github.com/matthew-wong1/WebGlitch.

54 Matthew Wong. WebGPU adapter.info call fails after device creation, 2024. Accessed 23rd April 2025. URL: https://github.com/denoland/deno/issues/25874.

55 Matthew Wong. naga change i32 arithmetic operations to use wrapping instead of checked, 2025. Accessed 23rd April 2025. URL: https://github.com/gfx-rs/wgpu/pull/6835.

56 Matthew Wong. WebGlitch artifact, 2025. Accessed 25th April 2025. doi:10.5281/zenodo.15281221.

57 World Wide Web Consortium. WebGPU explainer draft community group report, 2024. Accessed 23rd April 2025. URL: https://gpuweb.github.io/gpuweb/explainer/.

58 World Wide Web Consortium. WebGPU W3C working draft, 2024. Accessed 23rd April 2025. URL: https://www.w3.org/TR/webgpu/.

59 World Wide Web Consortium. WebGPU shading language W3C working draft, 4 September 2024, 2024. Accessed 23rd April 2025. URL: https://www.w3.org/TR/2024/WD-WGSL-20240904/.

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