WGSLsmith: Randomised Testing for the WebGPU Shading Language

Michał Andryskowski
Imperial College London
London, United Kingdom
michael.andryskowski22@imperial.ac.uk

Amber Gorzynski
Imperial College London
London, United Kingdom
amber.gorzynski22@imperial.ac.uk

Hasan Mohsin
Imperial College London
London, United Kingdom
hasan.mohsin18@imperial.ac.uk

Alastair F. Donaldson
Imperial College London
London, United Kingdom
alastair.donaldson@imperial.ac.uk

Abstract

We present WGSLsmith, a tool for randomised testing of compilers for the WebGPU Shading Language (WGSL). Under the WebGPU API—now supported by all three major browsers—GPU programs are written in the WebGPU Shading Language (WGSL), and every implementation ships a WGSL compiler that validates shaders and translates them to platform-specific code. Because shaders may come from untrusted websites, these compilers can be exercised by arbitrary web pages, making them a security-critical target for testing. WGSLsmith generates random shaders to find compiler crashes and cross-implementation mismatches. We describe its architecture and usage, and how we have broadened its language feature coverage to keep up to date with the evolving WGSL specification and support features absent from the original design. We report on a campaign that uncovered 12 bugs (all confirmed, 8 fixed) in various parts of the WebGPU stack, and evaluate the thoroughness of WGSLsmith via coverage analysis.

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

Keywords: Compiler testing, GPUs, WebGPU, fuzzing

ACM Reference Format:
Michał Andryskowski, Amber Gorzynski, Hasan Mohsin, and Alastair F. Donaldson. 2026. WGSLsmith: Randomised Testing for the WebGPU Shading Language. In *Companion Proceedings of the 2026 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '26), October 3–9, 2026, Oakland, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3837729.3840486

SPLASH Companion '26, Oakland, CA, USA

ACM ISBN 979-8-4007-2910-2/2026/10
https://doi.org/10.1145/3837729.3840486

1 Introduction

We present WGSLsmith, a tool for randomised testing of compilers for the WebGPU shading language (WGSL), intended for browser developers responsible for WebGPU implementations and for GPU vendors whose downstream compilers underpin WGSL. The WebGPU API [17] gives websites access to the GPU for highly parallel computation and advanced 3D rendering. As of June 2025 it is supported by all three major browsers—Chrome, Firefox, and Safari [21]—so this functionality is now available to a large fraction of web users. Programs that run on the GPU are called *shaders*, and under WebGPU they are written in WGSL [3]. Every WebGPU implementation ships a WGSL compiler that validates each shader and translates it to lower-level code for a platform-specific graphics API (Direct3D, Vulkan or Metal), which vendor-specific drivers then translate to GPU machine code. Because shaders may originate from untrusted websites, browsers depend on these compilers for security: a bug in a WGSL compiler is reachable from any web page, without the user downloading or running anything, making WGSL compilers a high-value target for testing.

WGSLsmith [20] generates random WGSL shaders and checks for compiler crashes and cross-implementation mismatches. It was briefly introduced in prior research [5, 11], but until this paper it has not been presented to the software testing community as a headline tool. Here we describe its architecture and usage, and our recent extensions—broadening language coverage to track changes in WGSL, supporting features absent from the original design, and improving fuzzing throughput—and report on a campaign that found 12 bugs (all confirmed, 8 fixed) across Tint and Naga (the WGSL compilers in Chromium and Firefox), DXC (the DirectX Shader Compiler) and Mesa (a widely-used open source shader compiler framework).

We overview WGSLsmith (§ 2); describe our new features and improvements (§ 3); present results on bug-finding across the WebGPU shader compiler ecosystem and evaluate thoroughness via coverage (§ 4); and finish with related work (§ 5), future directions (§ 6) and tool availability (§ 7).

2 Overview of WGSLsmith

2.1 WebGPU and WGSL

WebGPU is an API developed by the W3C for accelerated 3D graphics and general-purpose GPU compute on the web. To run GPU workloads, developers write shaders in the WebGPU Shading Language (WGSL), a language designed with strict security and portability guarantees. A WebGPU implementation serves as an abstraction layer bridging the browser API and the host system's native graphics APIs (such as Vulkan, Metal, or Direct3D). Every implementation embeds a WGSL compiler that can target native shading languages (like SPIR-V, MSL, or HLSL). The WebGPU implementation of Chromium is *Dawn*, which includes the *Tint* WGSL compiler. In Firefox, the implementation is called *wgpu* and uses the *Naga* compiler. Once translated by Tint or Naga, the shader code is passed to downstream native toolchains and proprietary GPU drivers for final compilation and execution.

2.2 Architecture of WGSLsmith

WGSLsmith supports differential testing [14] of WGSL shader compilers, and comprises a *generator* that creates shaders randomly, a *reconditioner* that bounds their behaviour to make them deterministic, a *test harness* for running shaders on devices via a WebGPU implementation, and a *reduction driver* that reduces bug-triggering shaders to a simple form for investigation and triage.

Generator. The generator constructs an AST that is later pretty-printed to WGSL, working in two phases: it first emits supporting constructs (types, global and buffer variables) and then the entry-point function, generating auxiliary functions on demand in a top-down style similar to Csmith [23]. At each choice point it checks the preconditions of each candidate construct and samples from the valid subset, so generation never backtracks. Generated shaders are guaranteed well-formed and well-typed but may exhibit undefined behaviour or fail to terminate; the reconditioner handles these later. Literals are drawn from distributions that mix interesting edge cases (e.g. `i32::MIN`) with values biased towards small magnitudes.

Reconditioner. The reconditioner transforms a generated shader so that its execution is deterministic, both to enable differential testing and to keep test cases valid during reduction. It runs in two steps: a static analysis that detects invalid pointer aliasing and rejects offending shaders, followed by an AST transformation that guards against various forms of undefined behaviour through array-index wrappers that enforce bounds, loop limiters that bound iteration counts, and floating-point wrappers that keep values within a safe range. The reconditioner can also encode workarounds for known compiler bugs to stop them masking other bugs.

Test harness. The harness performs differential testing, executing compute shaders across multiple WebGPU configurations and comparing their outputs to detect bugs. Comparison is supported across different WebGPU implementations—e.g. Dawn and wgpu—as well as across different downstream graphics APIs—e.g. Vulkan vs. Direct3D. A client-server mode allows shaders to be dispatched to a remote process for execution, useful for testing WebGPU support on devices where executing the generation and reconditioning logic directly is non-trivial (e.g. Android). WGSLsmith also supports a crash testing mode in which numerically-unstable features are allowed (including more liberal use of floating-point operations). Here differential testing is not applicable, since numerical instability leads to legitimate result differences between implementations, but the wider feature range can be useful for triggering compiler crashes.

Reduction driver. The reduction driver minimises a bug-triggering shader by managing a pluggable external reducer (we use Perses [19] by default, as it can be armed with a WGSL grammar so as to only make syntactically correct reduction steps), together with an *interestingness test* that decides whether each candidate still reproduces the bug. Importantly, the driver reconditions every candidate before testing it, so that reduction cannot introduce undefined behaviour that would invalidate the result.

2.3 Using WGSLsmith

To launch a differential fuzzing campaign across selected targets (e.g. Dawn and wgpu on Vulkan, the trailing `0` selecting the device index), one runs `./wgslsmith fuzz -c dawn:vk:0 -c wgpu:vk:0`, which continuously generates, reconditions, and executes shaders to flag mismatches or crashes. When a bug is found, `./wgslsmith reduce` minimises the triggering shader via an external reducer driven by an automatically generated interestingness test. Components can also be used standalone: `./wgslsmith recondition`, for instance, makes an existing shader deterministic without running the full testing loop.

3 New WGSLsmith Features

When WGSLsmith was created in 2022, WebGPU was in its early stages: the specification was a Working Draft, and no major browser supported it by default on the stable branch. The four years since have brought numerous changes, including API changes, deprecations, new features, and validation improvements. To test current implementations effectively, the fuzzer must generate valid code that complies with these updated rules. The revamp we report on here enhances the generator, reconditioner and test harness, and introduces a *concretiser*.

Expanding the scope of generation. We revamped the generator so that it now emits the full set of modern WGSL

types—matrices, atomics, and, behind a `--gen-ext` flag, optional features such as `f16` and subgroup operations that are only available when the target GPU advertises support. The original generator restricted itself to constructs it could safely recondition, silently excluding many built-in floating-point functions from generation. An `--unstable-float` flag lifts this restriction for crash-only testing, where the goal is to trigger a crash rather than compare outputs, giving substantially broader coverage of built-ins. We also stress memory-layout rules: the generator computes the minimum legal alignment and size for each struct member and, when an inflated value is chosen or with fixed probability, attaches explicit `@align` and `@size` attributes. WGSL's uniformity requirement for collective operations is handled pragmatically: subgroup functions and workgroup barriers must execute in uniform control flow, and a compiler that cannot prove uniformity rejects the shader. We emit these calls only at the top level of the entry point rather than implementing full uniformity analysis. This guarantees the precondition while still exercising the compiler's handling of these builtins. Supporting these features required corresponding extensions to the WGSLsmith parser and AST representation.

Reconditioner. The original WGSLsmith relied on reconditioning to avoid *de facto* undefined behaviour: operations the WGSL specification defines but that Tint and Naga, lagging behind the spec, did not yet implement safely e.g. integer overflow, division by zero, and similar basic arithmetic. Modern WebGPU implementations now follow the specification for these cases, so we removed the corresponding wrappers; leaving them in could have suppressed bugs the compilers would otherwise reveal.

The new language features we added in turn required new safety wrappers. For collective operations, the reconditioner masks the index arguments of subgroup and quad broadcast functions to keep their results well-defined, and we updated the pointer analysis and transformation passes to handle modern syntax such as `while` loops and `continuing` blocks.

It is tempting to disable reconditioning entirely for crash testing, since there we do not compare numerical outputs across implementations. However, some reconditioning is still necessary: loop limiters must remain to prevent non-terminating shaders, and we must avoid the WGSL *Trap* dynamic error (raised e.g. on an out-of-bounds access) which halts execution early and masks the behaviour we are trying to reach. Even in `--unstable-float` mode, we therefore continue to apply the loop and buffer-access wrappers.

Concretiser. A recent study reported a validity rate of just 0.77% for WGSLsmith [4], despite the generator originally having been designed to emit syntactically and semantically valid WGSL. On investigation, this turned out to be due to the language specification having been tightened to reject any shader containing an ill-formed compile-time *const-expression*—e.g. a compile-time division by zero such as `5i / 0i`, or even a partially evaluable variant like `x / (abs(-5i) - 5i)`. Because generated shaders contain many deeply nested expressions, and a single offending const-expression invalidates the entire module, the overwhelming majority of generated shaders caused compilation to fail with validation errors, exercising only the front-end validator and never reaching the later compilation passes that WGSLsmith is supposed to stress.

To address this, we introduced a *concretiser* pass that runs during reconditioning. The concretiser hinges on two operations: `evaluate`, which attempts to compute the compile-time value of any given sub-expression, resolving to `None` if that specific sub-expression depends on runtime state or triggers a domain error; and `concretise`, which performs a bottom-up partial evaluation of the AST. It leaves runtime expressions intact, but replaces any statically evaluable sub-expression that faults (which would otherwise trigger a validator rejection) with a safe default such as `1` or `1.0`. A subtlety is that partially non-constant expressions must also be handled: in `x / y`, if `x` is a runtime value but `y` concretises to `0`, the division is still rejected, so the pass rewrites it to its numerator. We handle this by maintaining a stack of lexical scopes so that `const` declarations can be tracked and identifiers resolved to their compile-time values during traversal.

Thanks to the concretiser, WGSLsmith now achieves a validity rate of almost 100%: to test this, we generated 70,000 test cases and validated them using Tint, which reported just 4 to be invalid (due rare edge case behaviours).

4 Experiments with WGSLsmith

We report on recent bugs found using WGSLsmith (§ 4.1) and on the coverage achieved through WGSLsmith-based fuzzing (§ 4.2).

4.1 Bug Hunting Using WGSLsmith

Table 1 details 12 bugs uncovered by WGSLsmith through continuous fuzzing runs spanning several months at various stages of development, half of which were found as a result of new features we added to the tool during our work, the remainder being bugs that the original version of WGSLsmith could have found but for the validity-related problems solved by the concretiser.

The bugs occur in the Tint and Naga WebGPU shader compilers, and in the DirectX Shader Compiler (DXC) [16], which is used on Windows to process translated shader code before it is pushed to downstream drivers, and in Mesa [15], an open-source project that provides graphics drivers, including driver-level shader compilers to generate final machine code.

ID	SUT	Description	Type	Status
8900	Naga	Dot overflow should have wrapping semantics	Validation	Fixed
8942	Naga	(PR) Constant evaluation of sign(0.0f)	Miscompilation	Fixed
9540	Naga	Dot4I8Packed compilation produces invalid MSL	Crash	Open
476936378	Tint	extractBits, insertBits constant folding	Miscompilation	Fixed
478792488	Tint	DXC short circuits select() if cond is a constant	Miscompilation	Fixed
498646773	Tint	DXC error: Assignment of undefined values to UAV	Crash	Fixed
505317119	Tint	Internal Compiler Error in MSL writer	Crash	Fixed
506180954	Tint	insertBits: e=0 causes GPU Hang	Crash	Fixed
519471056	Tint	Nested firstLeadingBit polyfill explosion	Crash	Fixed
5328	DXC	Null-dereference. Found earlier by static analysis, but contributed a test case	Crash	Open
8094	DXC	Bad optimisation of shift-div	Miscompilation	Open
15343	Mesa	Null-dereference when compiling shader	Crash	Open

Table 1. Summary of bugs discovered by WGSLsmith during our fuzzing campaign. Highlighted IDs refer to bugs that could only be found thanks to improvements we made to WGSLsmith.

Issue 505317119

```
var<workgroup> v:
    array<vec3<bool>, 1>;
@compute @workgroup_size(1)
fn main() {
    v = array<vec3<bool>, 1>();
}
```

Issue 478792488

```
@compute @workgroup_size(1)
fn main() {
    // foo() has side effects
    let x = select(foo(), false, true);
}
```

Figure 1. Minimised examples of test programs triggering a compiler crash and a miscompilation, found by WGSLsmith in stable versions of Tint.

As representative examples, Figure 1 shows two minimised test cases. The left snippet triggers an internal compiler crash in Tint on macOS. The right snippet demonstrates a miscompilation uncovered via differential testing. In WGSL, function arguments evaluate eagerly. However, the `select` expression was incorrectly constant folded to `false`, which caused an output mismatch as `foo` had side effects.

4.2 Coverage Analysis for Tint and radv

To assess how thoroughly WGSLsmith exercises WebGPU compilers, we ran a six-hour campaign (three repetitions) measuring the branch coverage of our generated shaders on the Tint front-end and the Mesa `radv` driver back-end. As a baseline we collected the coverage of the official test suites of Tint, Naga, and WebKit's `wgslc` compiler, with 12,675, 319 and 63 tests respectively (we include an archive of these as part of our artifact). On Tint, WGSLsmith's standalone generator reached on average 26,754 branches, just short of the 32,172 from the official suites, and reached 1,180 branches not reached by the official suites; on `radv` it reached on average 77,439 branches, slightly exceeding the 77,318 of the official suites and including 19,422 branches not reached by the official suites.

Qualitatively, the generator misses some coverage (e.g. related to override substitution and certain memory-layout adjustments) but explores other regions: in Tint it heavily exercises compile-time constant evaluation, integer range analysis, and loop analysis passes, while in `radv` its deeply nested expressions trigger additional optimization patterns and register-spilling routines.

5 Related Work

WGSLsmith follows the seminal Csmith project [23]; other "smith" tools include CLsmith [13], RustSmith [18], VeriSmith [9] and CUDAsmith [10]. There has also been significant work on testing other GPU shading-language compilers, e.g. GLFuzz/GraphicsFuzz for OpenGL [6, 7], the GLSLsmith tool, and spirv-fuzz [8], which targets the SPIR-V IR used by Vulkan. WGSLsmith enables differential testing in the same spirit, but is the only such tool to cater for WGSL.

An earlier version of WGSLsmith was presented as a vehicle for the idea of program reconditioning [11], and an industry experience report on randomised graphics shader compiler testing makes use of it [5]. The version reported on here is brought up to date with the latest WGSL specification and adds significant new features.

DarthShader [4] also fuzzes WGSL shader compilers and has uncovered numerous security vulnerabilities. It differs from WGSLsmith in its oracle and pipeline coverage: it is a coverage-guided crash fuzzer that instruments the Tint and Naga front-ends and DXC, using sanitisers to detect memory-safety violations, but stops at compilation—so it

neither executes shaders nor supports differential testing, and its reliance on coverage instrumentation prevents it targeting downstream proprietary GPU driver compilers.

WebGlitch [22] randomly tests WebGPU API implementations and can consume a corpus of WGSLsmith-generated shaders, so our improvements to WGSLsmith improve WebGlitch in turn. Levine et al. [12] used WGSLsmith in WGSLMemSmith, a metamorphic fuzzer generating parallel WGSL programs with controlled data races, which uncovered a memory isolation vulnerability in a pre-release Firefox.

6 Conclusions and Future Work

We have presented WGSLsmith, a randomised testing tool for WGSL compilers, and described extensions that bring it into line with the modern WGSL specification. A fuzzing campaign with the extended tool uncovered 12 bugs (8 since fixed) across Tint, Naga, DXC, and Mesa, and our coverage analysis shows it exercises regions of Tint and the radv driver only partially reached by the official test suites. These results indicate there remains scope for hardening this security-critical compiler ecosystem, and that fuzzing with WGSLsmith is a practical means of doing so.

7 Tool Availability

WGSLsmith is open source (Apache 2.0) on GitHub [20], with usage instructions in the README. An artifact capturing the version of WGSLsmith described in this paper, together with a README walking through how to use the tool, is available [2], as is a video showing WGSLsmith in action [1].

Acknowledgements

Thanks to the Tint and Naga developers for their work triaging bug reports and developing patches, and in particular to James Price and Dan Sinclair at Google who always responded promptly to resolve any issues within Tint. This work was supported by gift funding from Google.

References

[1] Michał Andryskowski. 2026. WGSLsmith (YouTube Video). Accessed: 2026-06-25. https://youtu.be/nGXbtYX4c5c

[2] Michał Andryskowski, Amber Gorzynski, Hasan Mohsin, and Alastair F. Donaldson. 2026. Artifact for "WGSLsmith: Randomised Testing for the WebGPU Shading Language". doi:10.5281/zenodo.21910004

[3] Alan Baker, Mehmet Oguz Derin, and David Neto. 2026. *WebGPU Shading Language*. W3C Candidate Recommendation Draft. World Wide Web Consortium (W3C). https://www.w3.org/TR/WGSL/

[4] Lukas Bernhard, Nico Schiller, Moritz Schloegel, Nils Bars, and Thorsten Holz. 2024. DarthShader: Fuzzing WebGPU Shader Translators & Compilers. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie (Eds.). ACM, 690–704. doi:10.1145/3658644.3690209

[5] Alastair F. Donaldson, Ben Clayton, Ryan Harrison, Hasan Mohsin, David Neto, Vasyl Teliman, and Hana Watson. 2023. 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*. IEEE, 374–385. doi:10.1109/ICST57152.2023.00042

[6] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 93:1–93:29. doi:10.1145/3133917

[7] Alastair F. Donaldson, Hugues Evrard, and Paul Thomson. 2020. Putting Randomized Compiler Testing into Production (Experience Report). In *34th European Conference on Object-Oriented Programming, ECOOP 2020, Berlin, Germany (Virtual Conference), November 15-17, 2020 (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:29. doi:10.4230/LIPICS.ECOOP.2020.22

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

[9] Yann Herklotz and John Wickerson. 2020. Finding and Understanding Bugs in FPGA Synthesis Tools. In *FPGA '20: The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, February 23-25, 2020*, Stephen Neuendorffer and Lesley Shannon (Eds.). ACM, 277–287. doi:10.1145/3373087.3375310

[10] Bo Jiang, Xiaoyan Wang, Wing Kwong Chan, T. H. Tse, Na Li, Yongfeng Yin, and Zhenyu Zhang. 2020. CUDAsmith: A Fuzzer for CUDA Compilers. In *44th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2020, Madrid, Spain, July 13-17, 2020*. IEEE, 861–871. doi:10.1109/COMPSAC48688.2020.0-156

[11] Bastien Lecoeur, Hasan Mohsin, and Alastair F. Donaldson. 2023. Program Reconditioning: Avoiding Undefined Behaviour When Finding and Reducing Compiler Bugs. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1801–1825. doi:10.1145/3591294

[12] Reese Levine, Ashley Lee, Neha Abbas, Kyle Little, and Tyler Sorensen. 2025. SafeRace: Assessing and Addressing WebGPU Memory Safety in the Presence of Data Races. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025), 697–725. doi:10.1145/3763075

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

[14] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10 (1998), 100–107. https://api.semanticscholar.org/CorpusID:14018070

[15] Mesa3D Contributors. 2026. *The Mesa 3D Graphics Library*. Accessed: 2026-05-21. https://www.mesa3d.org/

[16] Microsoft Corporation. 2026. *DirectXShaderCompiler*. Accessed: 2026-06-26. https://github.com/microsoft/DirectXShaderCompiler

[17] Kai Ninomiya, Brandon Jones, and Jim Blandy. 2025. *WebGPU*. W3C Candidate Recommendation Draft. World Wide Web Consortium (W3C). https://www.w3.org/TR/webgpu/

[18] Mayank Sharma, Pingshi Yu, and Alastair F. Donaldson. 2023. RustSmith: Random Differential Compiler Testing for Rust. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 1483–1486. doi:10.1145/3597926.3604919

[19] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 361–371. doi:10.1145/3180155.3180236

[20] The WGSLsmith Project Authors. 2026. Dredd. Accessed: 2026-06-25. https://github.com/wgslsmith/wgslsmith

[21] WebGPU Community Group. 2026. *WebGPU Implementation Status*. GitHub Wiki. Accessed: 2026-06-26. https://github.com/gpuweb/gpuweb/wiki/Implementation-Status

[22] Matthew K. L. Wong and Alastair F. Donaldson. 2025. WebGlitch: A Randomised Testing Tool for the WebGPU API (Experience Paper). In *39th European Conference on Object-Oriented Programming, ECOOP 2025, Bergen, Norway, June 30 - July 2, 2025 (LIPIcs, Vol. 333)*, Jonathan Aldrich and Alexandra Silva (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 39:1–39:26. doi:10.4230/LIPICS.ECOOP.2025.39

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

Received 2026-06-27; accepted 2026-07-25