Imperial College
London

MEng Individual Project

Department of Computing

Imperial College of Science, Technology and Medicine

WGSLsmith: a Random Generator of WebGPU Shader Programs

Author:
Hasan Mohsin

Supervisor:
Prof. Alastair Donaldson

June 2022

Abstract

The WebGPU API is a recent addition to the web, catering to the high performance graphics needs of modern web applications. The introduction of WGSL as WebGPU's shader language enables web developers to write graphics and compute shaders to accelerate their applications, but facilitates running untrusted code from the internet on GPUs. Thus, it is crucial to ensure that WGSL compilers are free from bugs to limit the potential for exploitable vulnerabilities.

Compiler testing has been explored in the past to find bugs across various languages and compilers. WGSLsmith provides a toolkit for testing WGSL compilers through randomized testing. To date, it has been able to find 33 distinct bugs, including both crashes and miscompilations resulting in unexpected runtime behaviour. Most of these have been reported to and confirmed by compiler developers, and several have already been fixed.

WGSLsmith is able to generate test programs using a range of WGSL language features, and is the first tool for testing shader compilers with support for pointers. This uses a static pointer analysis to ensure that programs containing pointer operations behave predictably. WGSLsmith also applies the reconditioning technique [1] to avoid undefined behaviour in generated programs, enabling the use of off-the-shelf test case reduction tools to produce human-readable test cases suitable for bug reports.

Acknowledgements

Firstly, I would like to thank my supervisor, Prof. Alastair Donaldson, for his invaluable guidance, support and feedback throughout the project, as well as for helping with bug-finding on macOS. It has been a pleasure to work with you on this project.

I would also like to thank the Tint and Naga developers for providing feedback on bug reports. I am particularly grateful to Teodor Tanasoaia for their quick responses and code reviews on my pull requests.

Finally, I would like to thank my friends and family for their support throughout the past four years, without which I never would have made it this far.

Contents

Chapter 1

Introduction

With the introduction of the WebGPU API [2], the web has gained new capabilities for high performance graphics rendering and general purpose computation on GPU hardware. This enables a wide range of applications to target web browsers, from video games to graphics-intensive professional tools.

However, the nature of the web platform means that software developed for the web is a valuable target for attacks. It is crucial to ensure that web browsers are secure and free from exploitable bugs, to safeguard users' devices and personal data. The addition of WebGPU serves to expand the exploitable attack surface of web browsers, increasing the potential for misuse (see section 3.4).

Modern graphics hardware is highly programmable and can be used for general computation applications such as machine learning. WebGPU introduces WGSL [3], a new programming language for writing GPU programs, known as shaders. Implementations of WebGPU include compilers in order to transform WGSL source code into platform and driver specific shader code to be executed on the hardware.

Compilers provide critical infrastructure for a language, and are typically highly complex pieces of software. Thus, it is no surprise that existing research has shown bugs to be prevalent in compilers for mainstream programming languages such as C and C++ [4].

Most research into compiler bugs and testing approaches focuses on general purpose programming languages such as C and C++ [4–10]. This makes sense as these languages are commonly used to implement critical systems. Some recent work has been done on testing compilers for GLSL, a shader language designed for the OpenGL API [11, 12], as well as SPIR-V for Vulkan [13, 14].

1.1 Contributions

This project introduces WGSLsmith, a toolkit for performing automated testing of WGSL compilers. WGSLsmith uses randomized testing (fuzzing) to test compilers by generating random shader programs and running them to compare the behaviour across compiler implementations (section 4.1).

WGSLsmith applies the idea of reconditioning [1] to WGSL, to ensure that generated programs have predictable runtime behaviour, and to maintain this property during test case reduction (section 4.2). Additionally, WGSLsmith is the first tool for testing shader compilers that includes support for pointers, and uses a pointer analysis to ensure that pointer accesses remain valid.

WGSLsmith includes a flexible tool for executing and testing arbitrary WGSL shaders, for use in differential testing scenarios (section 4.3). This is able to execute shaders with different I/O layouts, and can perform alignment and padding sensitive buffer comparisons for use across compiler implementations[1]. It also implements a client-server model to enable remote testing workflows.

Finally, WGSLsmith also includes tools for performing automated test case reduction on shaders, to produce human-readable test cases suitable for providing in bug reports (section 4.4).

WGSLsmith has been continuously evaluated to find bugs in the current WGSL compiler implementations (Tint [15] and Naga [16]). So far, it has been able to find 33 bugs (section 5.1), most of which have been reported to compiler developers and several of which have already been fixed. Different types of bugs have been found, including both crashes and miscompilations resulting in unexpected runtime behaviour. The performance of four different reducers has also been evaluated when applied to WGSL programs using the reconditioning technique (section 5.3).

[1]Compiler implementations do not currently treat padding the same, though recent spec changes have been made to rectify this. See section 4.3.1.

Chapter 2

Background

This chapter introduces compiler testing concepts, and discusses approaches that have been used in the past by existing testing tools. As this chapter focuses on general compiler testing, an introduction to graphics programming and WebGPU is provided later in chapter 3.

2.1 Compiler Testing

Testing software is crucial to ensuring that it fulfils its requirements and is free from bugs. A common method for testing is to use manually written test cases. This approach has a number of advantages: it is very flexible as tests can be highly focused for a particular application and scenario; additionally, it is usually straightforward for the programmer to determine the expected program output for a given test case, avoiding the test oracle problem (discussed in section 2.4).

However, manual testing puts considerable burden on the programmer to write these test cases. Writing good tests can take significant time and effort. It is also limited by the programmer's imagination; the only cases that will be tested are those that the programmer has thought of. While the programmer may be good at testing the expected inputs and execution paths, identifying all possible edge cases and input variations to test is usually more difficult.

Another approach is to use formal verification to ensure correct program behaviour. This has the advantage of being able to verify all possible execution paths of the program, thus guaranteeing correctness. Existing work has been done to apply formal verification to compilers such as CompCert [10]. However, it is a laborious process and can take a long time for non-trivial systems, often resulting in specifications larger than the software itself. This makes it impractical for large scale software such as compilers.

Automatic test case generation can overcome the challenges associated with manual testing, by removing the burden from the programmer to design and write test cases. Fuzzing is an approach to automate the testing process by producing large numbers of randomized test cases, with the aim that some of these test cases will probabilistically trigger bugs.

However, this poses a number of new challenges.

- Firstly, it is necessary to automatically construct test programs to test the compiler with (section 2.2). For a very simple approach, the compiler could be tested with arbitrary textual input. However, it is common to impose constraints on the input; when testing middle and back-end stages of a compiler it is useful to test only syntactically valid programs to bypass uninteresting errors from the lexing and parsing stages of the compiler.
- Secondly, given a test case, the automated testing system must be able to determine the expected output of the program to establish whether the compiler correctly handles the test case. This is known as the test oracle problem (section 2.4).
- Thirdly, it must be possible to make test cases human-readable so that they can be reported to developers (section 2.5). Test programs that are difficult to read and analyse are unlikely to be well-received by developers as they significantly increase the effort required to debug an issue.

Existing approaches to solving these problems are discussed in the following sections.

2.2 Program Construction

In order to test a piece of software, a concrete test case must be generated. Compilers take source code as input, so the test case generator must be able to produce test programs.

Depending on which part of the compiler is being tested, the test program will need to be constrained. For example, to identify memory safety issues in the initial stages of lexing and parsing it may be sufficient to provide completely random input to the compiler in an attempt to trigger unsafe memory operations.

However, if it is desired to identify bugs that would result in miscompilations in the output program, the test program must be both syntactically and semantically correct to ensure it is parsed correctly. That way it can be used to test later compilation stages such as code generation and optimization. This project focuses on detecting miscompilations as well as crash-related bugs in the later stages of WGSL compilers.

There are two main approaches to automatically generating test programs. Random program generation involves randomly generating new programs from scratch. Alternatively, it is possible to apply transformations to existing programs to produce new variants of the test programs. In either case, the primary objective is usually to maximize the diversity of test cases.

2.2.1 Program Generation

A commonly employed method for constructing test programs is to generate them from scratch. Csmith [5] is a well-known tool capable of randomly generating C programs in this way.

Program generation in Csmith takes a grammar-directed approach. It first creates a random number of struct declarations, each containing a random set of members. It then begins generating a top-level entrypoint function by selecting a production based on the grammar rules for the current context. This process is continued recursively for non-terminal productions.

Csmith is able to generate additional functions and variable definitions *on demand*. During the process of generating an expression, the generator may select a variable or function call in which case it can pick a previously-defined variable or function (if possible), or decide to generate a new one. In this way, the generator uses a “top-down” approach to generate programs [5]. Compared to an approach where all variables and functions are generated upfront, this means that it is not restricted by previous decisions and can instead branch out further, enabling greater diversity in the generated programs.

To make random choices during the generation process, Csmith uses a probability table that defines probabilities for generating each of the possible constructs. Csmith also allows the user to configure certain options that influence generation, such as the maximum expression depth.

Programs generated by Csmith are deterministic and do not read from any external input. The `main` function invokes the top-level entrypoint and then computes and prints a checksum of global variables [5]. This checksum is used to compare executions of the program across compilers under test.

YARPGen [6] is another random program generator capable of producing both C and C++ programs. The generator begins by generating a set of struct declarations similarly to Csmith, as well as a set of global variables. It then randomly generates a set of functions; unlike Csmith, variables and functions are generated upfront rather than during expression generation. In the case of functions, this is necessary since YARPGen does not support function calls.

After generating the test functions, YARPGen generates a `main` function which invokes the test functions and computes a checksum from global variables, similarly to Csmith.

Global variables are also declared in the same file as `main`.

YARPGen introduces the concept of generation policies [6] with the aim of increasing program diversity. The main idea is to sample from different distributions when making decisions in the generator. For example, YARPGen can decide to limit the types of operations in a particular expression subtree to use bitwise and shift operators only. This can be used to target particular optimizations that the compiler may implement, for more focused testing.

Additionally, YARPGen uses a technique known as parameter shuffling [6] where a random distribution is used to seed the main distributions used for the generator's decisions, before beginning the generation process. This enables programs to have very different characteristics between executions of the generator.

2.2.2 Program Transformation

Recent research has been conducted into using program transformation as a technique for generating test programs. This works by taking an existing program and applying modifications to produce a different one.

Many tools using this approach are based on the idea of *equivalence modulo inputs* (EMI) [8]. At a high level, EMI relies on the idea that two programs are equivalent with respect to some input if both programs exhibit the same behaviour given the same input. Thus, the code produced by the compiler should exhibit the same behaviour for both of the input programs (when run with the same inputs).

Orion [8] is an example of a bug-finding tool that makes use of this technique. Orion applies a two-step approach to producing test programs. First, it profiles the execution of a program on a number of inputs, to extract coverage information. This enables identifying areas of code that are not executed (dead code) for later pruning. The second step is to generate a number of EMI variants. Orion does this by traversing the program's AST and randomly removing statements that have been identified as dead code by the previous step.

While Orion's effectiveness at detecting bugs has been demonstrated [8], it is limited as the only transformation applied is the removal of dead code. Athena [17] introduces the ability to insert new code into regions of dead code. As there is no upper bound on the amount of new code that can be inserted, this greatly increases the number and diversity of variants that can be generated compared to Orion. Athena also introduces a technique to improve detection of bugs that require long sequences of transformations, by using Markov Chain Monte Carlo (MCMC) techniques for sampling with the aim of maximizing the difference between the original and variant programs. The authors show this to be much more effective than Orion's simple "blind mutation strategy".

Hermes [18] is another tool that takes this further by enabling additional transformations capable of modifying any part of the original program. Hermes extends the profiling stage to collect information about the concrete values of variables at each program point. It then uses this information to insert code snippets at particular program points in such a way that the values of existing variables at that point are unchanged. This guarantees that the result of the program execution is preserved.

Additional tools such as GLFuzz [11] and spirv-fuzz [14] have been developed using program transformation techniques. These are discussed further in the context of metamorphic testing, in section 2.4.2.

One important advantage of the aforementioned approaches is that they preserve correctness of the program across transformations. This is necessary to ensure that test results are meaningful, and is discussed further in the next section. However, these approaches are still limited since they can only apply a fixed set of transformations to existing programs. A *from scratch* generator can potentially achieve greater diversity in the generated programs by having more choices available during the generation process.

2.3 Ensuring Correctness

Undefined behaviour (UB) refers to behaviour that is not explicitly defined by the language specification. For example in C, dereferencing a null pointer is undefined behaviour. In the presence of UB, the compiler no longer makes any guarantees about the runtime behaviour of the program and is free to generate code with arbitrary behaviour. In practice, this can result in effects ranging from runtime crashes to silent propagation of errors through the program execution.

Along with undefined behaviour, it is possible for a language to have unspecified or implementation-defined behaviour. This is behaviour that can vary between occurrences and across compiler implementations. Implementation-defined behaviour is more common in WGSL than traditional UB (see section 3.3.5), but also leads to unpredictable runtime behaviour and can be treated similarly.

Therefore, it is necessary to ensure that the behaviour of generated test programs is predictable, so that tests can produce meaningful and reproducible results.

2.3.1 Structural Approach

An obvious technique to eliminate undefined behaviour is for the generator to simply avoid producing syntactic structures that would elicit such behaviour. For example, uninitialized memory accesses are UB in C. A generator could avoid producing accesses to uninitialized variables by always generating an unconditional initialization statement for all variables.

Quest [9] makes extensive use of this technique by only generating very simple C programs, without for example, complex arithmetic expressions or control-flow. This has the advantage of being very simple to implement and works well for Quest which specifically aims to detect bugs in C calling conventions. However, this means that test programs are much less expressive, making it unsuitable for finding bugs related to other language constructs.

In practice, most generators will use this approach to some extent, but turn to other approaches to enable more advanced testing.

2.3.2 Dynamic Checks

More complex types of UB may depend on the specific values of variables at runtime. For example, avoiding division by zero requires the generator to guarantee that the divisor is non-zero. Eliminating division operations altogether will solve this problem, but significantly limits expressiveness, so Csmith [5] solves this through dynamic checks at runtime.

Listing 2.1 contains a C program that defines a function `f`. The function divides argument `x` by `y`. On its own, this is unsafe as there is no guarantee that `y` contains a non-zero value. This operation can be made safe by including runtime checks, as shown in listing 2.2.

```c
int f(int x, int y) {
    return x / y;
}
```

Listing 2.1: A C program that may exhibit UB.

```c
int safe_div(int x, int y) {
    if (y == 0) return x;
    else return x / y;
}

int f(int x, int y) {
    return safe_div(x, y);
}
```

Listing 2.2: Using runtime checks to avoid UB.

The unsafe division is wrapped in a dynamic check to ensure that the divisor is non-zero. Note that typically, the concrete result of the division is unimportant to the test program. Thus, it is possible to return an arbitrary value in the case where the safety check fails.

Csmith uses similar dynamic checks to ensure safety in a variety of cases, such as for arithmetic checks and pointer safety. While this technique has been criticized in other works for limiting the expressiveness of generated programs [6], Csmith has been successful at detecting a wide range of bugs in C compilers [5].

Csmith's safe wrappers guarantee that operations do not invoke UB. However, they are inserted unconditionally, without regard for whether UB can actually be triggered based on the runtime values of variables. Even-Mendoza et al. show how these checks can in fact be relaxed in some cases by analysing the runtime behaviour of the test program [19]. This approach involves replacing occurrences of safe wrapper usages with instrumentation code that records whether the code fragment is executed with values that would trigger UB. Since Csmith always produces deterministic programs, executing this instrumented program identifies the areas where the safety checks are actually necessary. Thus, redundant checks can be removed from program points where the given operation would not result in UB.

2.3.3 Generation-Time Analysis

The YARPGen generator for C and C++ programs has shown that it is also possible to avoid many types of undefined behaviour through more advanced analysis during the generation process [6].

To avoid UB such as integer overflow when generating expressions, YARPGen partitions variables into three groups:

- **Input** variables can appear in expressions but are never reassigned. These will always contain their initial value.
- **Output** variables contain the results of expressions and can be written to one or more times, but never appear in an expression. These do not need to be analysed since their values are never used (except for bug checking at the end of execution).
- **Mixed** variables can appear in expressions and can be reassigned.

The values of mixed variables can change through the program, so YARPGen keeps track of their concrete values [6]. This enables the generator to statically guarantee that an expression will not result in undefined behaviour, since it knows the values of the subexpressions. If the analysis determines that an expression is not safe, the generator uses a set of rules to rewrite the expression into an alternative safe expression.

For example, the expression `-x` is unsafe if `x` is a signed integer and contains the smallest possible value for the type, as it may overflow. In this case, YARPGen will transform this into `+x` which is a safe operation.

Since YARPGen programs do not read from any external input, it is possible for concrete values to be statically tracked through the program. Using generation-time analysis means that YARPGen can avoid the runtime checks used by other generators such as Csmith [5]. This may enable detecting bugs that are otherwise obscured by the additional code for performing dynamic checks.

However, YARPGen has a number of limitations. It experimentally implements limited support for loops, requiring that all iterations of a loop operate on the same values [6]. This constrains the types of behaviour that can be tested within loops. Additionally, YARPGen does not support certain key features such as function calls and pointer arithmetic.

2.4 Test Oracles

When employing manual testing approaches, the programmer is typically able to encode the expected behaviour of each test case. With automatic test case generation comes the challenge of automatically determining what the correct outcome of a test case should be. This information is provided by a *test oracle* [20].

In the context of compilers, this is a challenging problem as compilers are typically highly complex pieces of software. This makes it difficult to statically determine the expected behaviour of a test program. In practice, two main techniques for deriving a test oracle have been applied to compiler testing: *differential testing* and *metamorphic testing*. The latter is closely related to the program transformation approaches outlined in section 2.2.2.

2.4.1 Differential Testing

Differential testing, introduced by McKeeman [7], relies on the availability of multiple compiler implementations for a given programming language. The implementations can then be tested by comparing the results of executing the same program compiled by each compiler implementation under test. Assuming that the test program is guaranteed to have predictable behaviour (i.e. it is free of undefined, unspecified or implementation-defined behaviour), differences in the execution results will indicate the existence of a compiler bug.

There are a number of differential testing strategies: *cross-compiler*, *cross-optimization* and *cross-version* [21]. Cross-optimization and cross-version strategies enable testing a single compiler implementation across different versions and optimization levels. Testing different versions is useful to avoid regressions in compiler behaviour between releases. Comparing different optimizations can also be good at identifying issues with how optimizations affect each other when used in interesting combinations. Comparing entirely different implementations is the most general strategy, but requires the language to actually have multiple implementations available.

2.4.2 Metamorphic Testing

Metamorphic testing [22] is an alternative approach to providing a test oracle that does not require multiple compilers to test against. This approach is built on the program transformation techniques discussed in section 2.2.2. Using the idea of EMI, transforming a program into an equivalent program by e.g. inserting dead code should not change the result produced when executing the program. Thus, if a difference in the output is observed between the transformed program and the original, it indicates a compiler bug.

Metamorphic testing has been used in both Orion and Hermes [8, 18] by generating variants of programs using the EMI technique (section 2.2.2). The main difference between the two is the types of transformations they are capable of producing, as previously discussed.

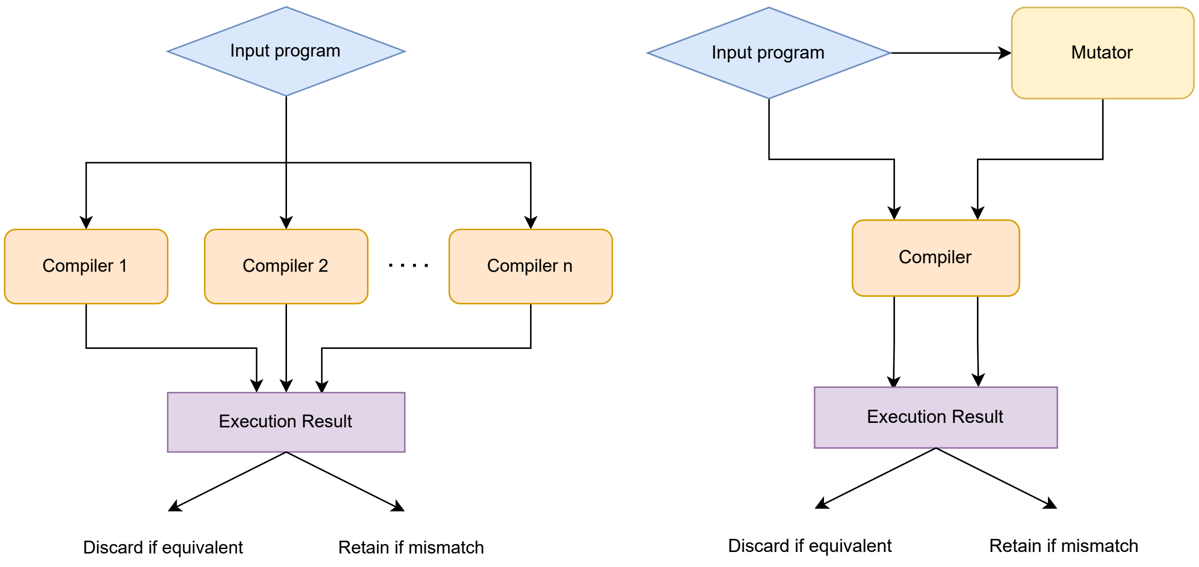
Metamorphic testing is often used in conjunction with a random program generator to produce the seed programs. Orion and Hermes make use of Csmith to generate initial C programs, as well as other sources of existing programs such as compiler test suites and open-source projects.

Metamorphic testing has also been applied to shader compilers by Donaldson et al. in GLFuzz for testing GLSL compilers [11], and spirv-fuzz for testing SPIR-V compilers [14]. They introduce the idea of *essentially semantics-preserving* transformations which preserve the behaviour of the program similarly to EMI. However, the precise semantics of floating-point operations are typically underspecified in languages such as GLSL; thus, a level of error is allowed when comparing the results of these operations.

Transformation-based testing also presents an alternative strategy for test case reduction (section 2.5) [14]. Delta debugging is used by spirv-fuzz to find the minimal subset of transformations applied to the original program to reproduce a bug. This can be easier to implement compared to other approaches, and guarantees reduced programs are free from UB if the original program is also UB-free.

GLFuzz and spirv-fuzz are able to compare images in order to test graphics shaders. By default, GLFuzz uses chi-squared distances to compare images, determining whether they are "visually indistinguishable" [11]. This is necessary due to the potential for accumulation of errors in floating-point operations, making a direct comparison of images impossible.

A comparison of differential testing and metamorphic testing is presented in figure 2.1.



(a) Differential testing **(b)** Metamorphic testing

Figure 2.1: Differential testing vs metamorphic testing.

2.5 Test Case Reduction

Program generators typically produce programs that are very large and highly obfuscated, due to the nature of automatic code generation and the desire to test language features in atypical scenarios. Listing 2.3 contains a small extract from a program generated by the WGSLsmith tool, the design and implementation of which is presented in chapter 4. The program consists of many complex expressions as well as opaque variable and function identifiers, and the full program contains many functions with large bodies.

```
fn main() {
    if (var_0.x) {
        let var_1 = 711513123u;
    }
    let var_1 = ~((countOneBits(max(~(2610881137u), ~(577783483u)))) * (~((~(4128073036u))
    var_0 = vec2<bool>(any(!(!(func_1(vec3<u32>(var_1, var_1, var_1))))), !(false));
    if (!(((clamp(reverseBits(var_1), 3961180244u, dot(vec2<u32>(var_1, 1912858368u), vec2<
        var var_3 = vec4<u32>(dot(vec4<u32>(dot(max(vec4<u32>(var_1, 2847045454u, var_1, 36
        var var_4 = (dot((-(-(vec4<i32>(1364496193, 1843907678, -828384492, 893683990)))) >
    }
    let var_3 = !((all(var_2)) | ((countOneBits((472260210) - (-1613635233))) > ((-11199131
    ...
}
```

Listing 2.3: Extract from a program generated by WGSLsmith. The code is largely unreadable, and unsuitable for directly reporting to compiler developers.

Submitting bug reports to a compiler development team containing such unreadable programs is unlikely to be helpful or well-received. Therefore, it is useful to find a minimal reproduction of the bug in a reduced version of the program if possible. Test case reduction techniques attempt to automatically reduce programs into smaller programs that remain *interesting*, where the definition of *interesting* is supplied the user.

Delta debugging [23, 24] is a simple technique to find the minimal set of changes applied to a program that result in a test failure. Delta debugging typically works by splitting up the input text into lines, and removing lines to find the smallest interesting input. This can work well in general for programs that accept arbitrary text input, but is not always ideal when applied to compiler testing as code fragments often have intricate dependencies that can be broken by removing entire lines at once. However, this technique has been applied in practice for program reduction, through tools such as Picire [25].

Regehr et al. show that existing delta debugging algorithms are specific solutions to a more general test reduction framework [26] and present three new test case reducers that are able to produce smaller programs.

The first two reducers, Seq-Reduce and Fast-Reduce, only work with programs generated by Csmith. Seq-Reduce uses a special mode in Csmith to bypass its internal random number generator and take control of decisions. Starting with a Csmith-generated program, it randomly modifies the specification of the Csmith generator's decisions and uses Csmith to regenerate a new program. The advantage of this is that Csmith is able to guarantee that the resulting program is valid. The program is further reduced in this way if it still exhibits the compiler bug and is smaller than the original.

Fast-Reduce instead applies rule-based transformations to the program such as dead code elimination to remove parts of the code. This is done by analysing both the static structure of the program and its runtime behaviour through instrumentation code added to the Csmith output.

The third reducer they present is C-Reduce, which is capable of reducing arbitrary C programs. C-Reduce is very flexible as it uses pluggable transformations to reduce a program until a fixpoint is reached. Several of the transformations implemented in C-Reduce are not specific to C, making it potentially useful for reducing programs written in other languages as well.

Sun et al. present Perses as an alternative reducer that uses a more "syntax-directed" approach to reduce programs, with improved speed and a smaller size compared to delta debugging and C-Reduce [27]. Perses takes a context-free grammar as input, which describes the language of the program being reduced. This allows it to work for a variety of languages, provided a grammar is available. A key advantage of this is that it ensures that the reduced program remains syntactically valid, thus decreasing the search space.

Reducers are usually parameterized by an interestingness test, in the form of a shell script. This test supplies the definition of *interesting*, and is invoked for each intermediate reduction candidate. It will often perform some initial validation of the candidate program before compiling it and checking if the bug is still manifested.

2.5.1 Reconditioning

Ensuring that the reduced program remains valid is a key challenge when performing automated reduction. Syntactic validity can be ensured by approaches such as Perses', while additional validation such as type checking can be performed in the interestingness test. However, it is possible for the program to be modified during reduction such that the changes introduce undefined behaviour, despite the unreduced program being correct. In this case, the reduced test case is useless as its behaviour is no longer guaranteed by the compiler. Even if it exhibits the same bug as the original, this can now be considered to be *expected* behaviour.

For widely used languages such as C and C++, existing sanitizer tools can solve this problem by detecting UB in reduced programs. However, for other languages that are not as commonly used, these tools may not be available, making it impossible to check reduced programs for UB. Program reconditioning provides a solution to this, by extracting the function of ensuring validity from a program generator into a separate process [1, 12]. Thus, a reconditioner is able to take an input program and transform it such that the resulting program is guaranteed to be valid.

Reconditioning integrates with the reduction loop by processing the output of the reducer to ensure it is valid before testing for the presence of the compiler bug. The reconditioner is also used to ensure validity of the original test programs produced by a program generator, before performing differential testing. This has been shown in the past to work effectively for GLSL, which suggests that it is likely to be effective when applied to WGSL, given their similarities.

2.5.2 Bug Slippage

Bug slippage can be another challenge during reduction [28]. When reducing a test case exhibiting bug a, it is possible that some transformations may cause the program to instead exhibit bug b. A naive reducer may continue reducing the test case further, not knowing that the wrong bug is now being reduced. This is especially undesirable if b is a known, uninteresting bug, as it now masks a which may be much more interesting. Slippage is commonly avoided through heuristics such as checking error codes in the interestingness test. Holmes et al. [28] propose ways that slippage can be harnessed to find more bugs, by collecting sets of reduced test cases rather than just one.

Chapter 3

Graphics Programming

This chapter provides an overview of graphics programming concepts that are relevant to this project. Section 3.1 describes general graphics concepts, while section 3.2 introduces WebGPU and section 3.3 provides an overview of the WGSL shader language. A brief discussion of WebGPU's security considerations is also contained in section 3.4.

3.1 Overview

In the past, the capabilities of graphics hardware were typically restricted to a set of fixed and specialized functionality built into the hardware, which could be invoked by a programmer to accelerate aspects of graphics rendering. In contrast, modern graphics hardware is highly programmable, allowing users to execute arbitrary programs on a GPU. This can be used for a wide range of graphics software, as well as other applications that may benefit from the highly parallel nature of GPUs, such as machine learning algorithms.

In order to control a GPU, commands must be sent from the CPU. While the low-level details of this communication may be proprietary, standardized APIs are available which can be implemented by a vendor in the graphics driver, to allow programmers to interact with different devices in a uniform manner.

There are several standard APIs available, some of which are operating-system specific. Notably, DirectX is available from Microsoft for Windows [29], while Apple provides Metal for macOS and iOS [30]. Additionally, several platform-agnostic standards have been developed by the Khronos group such as OpenGL [31] and Vulkan [32], which are supported on Linux (including Android) and Windows, as well as a few other platforms.

Programs written to be executed on a GPU are called shaders. These are typically written using a specialized language for shader programming. The aforementioned graphics APIs each provide their own languages for this purpose, as shown in table 3.1.

Graphics API	Shader Language	Supported Platforms
DirectX	HLSL [33]	Windows
Metal	MSL [34]	macOS/iOS
Vulkan	SPIR-V [35]	Windows, Linux
OpenGL	GLSL [36]	Windows, Linux

Table 3.1: Common graphics APIs

Broadly, there are two categories of shaders – *graphics* and *compute* shaders. Graphics shaders are used to perform specific tasks within the graphics rendering process, such as determining the colour to assign each pixel. In contrast, compute shaders are much more general and can be used to perform arbitrary computation.

Writing a program to control the GPU using these APIs involves setting up a *pipeline*. The pipeline supplies a shader program to execute, and describes features of the execution such as the inputs and outputs of the shader(s). A graphics pipeline consists of multiple shader stages which can pass data between them, with the final result being a rendered image. A compute pipeline consists of a single compute shader which can write data to one or more storage buffers. These can later be read by the CPU.

Comparing images to detect bugs can be challenging as many aspects of rendering are not well-specified to allow for flexibility in implementations. This can result in small differences in the outputs, which need to be accounted for. While prior work has been done in this area [11], this project will focus on testing compute shaders to simplify this process. Most capabilities available in graphics shaders are also available to compute shaders, excluding certain graphics specific functionality such as texture manipulation functions.

3.2 WebGPU

Web browsers have supported graphics rendering for many years through the canvas API, which allows for high-level drawing of 2D shapes and text. More recently, WebGL has provided a more advanced API based on OpenGL ES, a simplified variant of OpenGL used on mobile and embedded devices.

WebGPU aims to provide a lower-level API allowing more control compared to the existing offerings, to cater to high-performance graphics applications not well-supported by existing web APIs, as well as to enable the use of compute shaders on the web.

Web browsers implement WebGPU by exposing a JavaScript API for use by programmers. In the case of Chrome and Firefox, the underlying implementation of the API is delegated to an external library, which builds a platform-agnostic abstraction over the existing APIs available on each platform, such as DirectX (Windows), Metal (macOS, iOS) and Vulkan (Linux, Android). This job is performed by the Dawn [37] and wgpu [38] libraries for Chrome and Firefox respectively. The WebGPU architecture is illustrated in figure 3.1.

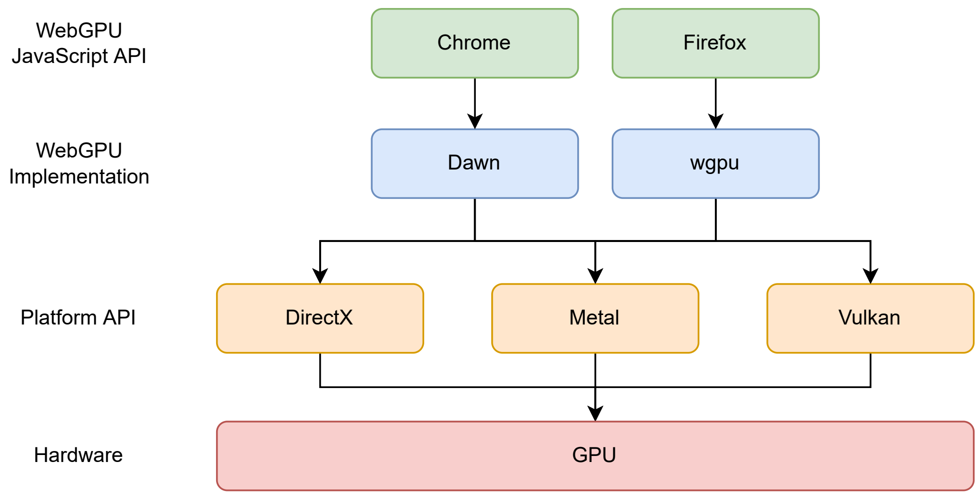


Figure 3.1: Overview of the WebGPU architecture. Additional layers between platform APIs and the physical hardware have been omitted for simplicity.

3.3 WGSL

WGSL is the language used to write shaders for WebGPU applications. As WebGPU implementations are built on top of platform graphics APIs, an implementation includes a compiler that translates WGSL source code into the appropriate language for the targeted graphics API, as described in table 3.1. Dawn uses the Tint compiler [15], while wgpu uses Naga [16]. Both of these are capable of translating WGSL into HLSL, MSL and SPIR-V (they also provide some support for translating to GLSL for OpenGL). WGSL shares similarities with existing shader languages such as GLSL, as well as general purpose languages including Rust and C.

WGSL is currently still in active development and sees regular changes to the specification, affecting both syntax and behaviour. This has been a challenge throughout the development and testing of WGSLsmith, and is discussed multiple times in later sections of this report.

3.3.1 Overview

The primitive types supported by WGSL include booleans (`bool`), 32-bit signed and unsigned integers (`i32` and `u32`), and 32-bit floats (`f32`). WGSL supports vectors of between 2 and 4 components, containing any of the numeric types. These are denoted by `vecN<T>`, where `N` is the number of components and `T` is the element type.

WGSL supports fixed-size array types, denoted by `array<T, N>` where `T` is the element type and `N` is the size of the array. Runtime-sized arrays are also supported; however, a runtime-sized array cannot actually be constructed in a WGSL program – they may only be used in types for host-shareable buffers (section 3.3.2). Users are also able to define additional composite data types using structs.

Vector components can be accessed in multiple ways. The dot operator (`.`) may be used to access a single component by name, or multiple names can be used together to construct a new vector using the specified components (known as swizzling). Finally, the indexing operator (`[]`) may be used to access a component by index, similarly to an array.

Most standard arithmetic, logical and bitwise operators are supported. Operations are component-wise when one or more operands is a vector.

WGSL does not support any implicit conversions between different types (except for converting from an abstract numeric literal to a type with a concrete size). Conversions between built-in scalar and vector types are supported through explicit type constructors and bitcast operations.

WGSL has three ways to assign names to expressions: `var` statements, `let` statements and function parameters. `var` statements declare *variables*, which are names for memory locations. The type of a variable must be *storable* – this includes most primitive types as well as arrays and structs (notably, pointers are not storable). In contrast, `let` statements create a new *name* for a value. These are read-only and must be initialized at declaration time. Function parameters have similar semantics to `let` statements; notably, both can only be used in functions and cannot be reassigned.

The difference between variables compared to `lets` and parameters is largely semantic, influencing how they behave in certain cases, and means that they do not always have to refer to storable types. Variable identifier expressions are considered to be reference types, while `lets` and parameters are not. Since `lets` and parameters cannot be reassigned and cannot escape from their lexical scope, they are safe to store pointers in without risk of dangling pointers (pointers that do not point to valid memory locations). This is not the case for variables, which can be assigned to from nested scopes where the referenced data may not live as long as the lifetime of the variable. Pointers are discussed further in section 3.3.3.

3.3.2 Shader I/O

In a compute shader, passing data between a shader and the host is performed through host-shareable buffers. There are two buffer types available to compute shaders: uniform and storage buffers. Uniform buffers are read-only and can be used for passing inputs to the shader. Storage buffers can also be written to by the shader, so are useful for passing results back to the host.

The host application is able to create and initialize one or more buffers when setting up the pipeline, through the WebGPU API. Each buffer is bound to a particular slot in the pipeline, referred to by a combined group index and binding index. The shader is able to reference these buffers by declaring a special buffer variable at the global scope. For each buffer variable, the shader must specify the group and binding index using the `@group` and `@binding` attributes, as shown in listing 3.1.

```
struct MyUniformBuffer {
    a: u32,
    b: f32,
    c: vec3<i32>,
}

// The @group and @binding attributes specify the buffer that this variable is
// bound to in the pipeline.
@group(0) @binding(0)
var<uniform> input_buffer: MyUniformBuffer;

// Storage buffers are read-only by default but can be made writable using
// the read_write qualifier.
@group(0) @binding(1)
var<storage, read_write> output_buffer: array<u32, 16>;
```

Listing 3.1: Host-shareable buffer variables

From the perspective of the host, buffers are simply byte arrays, without any structure. While it is possible to treat buffers similarly as simple arrays in the shader, a more structured representation can also be used via structs, specifying different types for fields within the buffer.

Buffer variables have size and alignment requirements which are defined by the WGSL specification [3]. This means that determining the minimum size required for a buffer variable is not a trivial calculation, as there may be additional padding bytes inserted between fields of a struct or at the end to satisfy alignment requirements, which increase its size. This must be considered when calculating the sizes of buffers to be allocated on the host.

All variables in WGSL are associated with an address space. Module-scoped variables (those defined at the top-level of the program) must specify the address space explicitly, as shown in listing 3.1. The `uniform` and `storage` address spaces are used for uniform and storage buffers respectively. Additionally, the `private` address space can be used for global variables that are not shared between invocations (multiple invocations of a shader can run in parallel). Global variables can also be shared between invocations using the `workgroup` address space. Local variables in functions implicitly use the `function` address space.

3.3.3 Pointers

WGSL includes limited support for pointers compared to other languages such as C. For example, WGSL does not support pointer arithmetic nor conversions between integers and pointers, and it is impossible to obtain a null pointer. It is also forbidden to return a pointer from a function. As described in section 3.3.1, pointers are not storable so cannot be stored in variables (though they can be named by function parameters and `let` statements). These rules make it impossible to produce a dangling pointer. WGSL also does not provide a facility for dynamic memory allocation; thus, all memory locations are known statically.

Another key rule is that the address-of operator (`&`) can only be applied to expressions of reference type. Since `lets` and function parameters are not considered references, it is not possible to obtain a pointer to a `let` or a parameter.

These restrictions mean that many dynamic pointer-related errors that are possible to trigger at runtime in C are eliminated in WGSL. However, WGSL does have one rule that must be enforced by the programmer, involving aliasing (where multiple pointers/references are used to access the same memory location). This is discussed in detail in section 4.2.5, along with a technique for ensuring that a WGSL program does not contain invalid aliasing.

3.3.4 Floating-Point

WGSL supports floating-point values and operations according to the IEEE-754 standard [39]. However, certain requirements of the standard are relaxed to enable greater implementation flexibility. This can lead to differences in behaviour between implementations.

In particular, the rounding mode in WGSL is unspecified, meaning that an implementation is allowed to select any scheme to use when rounding values (up or down). Additionally, WGSL programs do not generate floating-point exceptions or signalling NaNs, and implementations may ignore the sign of zero values (floating-point zero may be positive or negative).

All floating-point operations have a specified level of accuracy. However, the precise results of operations may differ across implementations. Certain operations are required to produce *correct results*, but most will have a range of possible values that they are allowed to produce. However, it is possible to restrict values and operations in such a way that they can have more predictable semantics. This is discussed in section 4.2.2.

3.3.5 Predictability and Undefined Behaviour

In contrast to other shader languages such as GLSL and HLSL, WGSL contains very little undefined behaviour, in the traditional sense. The WGSL specification aims to explicitly define the semantics of WGSL programs to as large an extent as possible, which is important given the security critical context in which WGSL programs will be executed.

Nevertheless, it is difficult to precisely define the behaviour of all operations, without compromising on implementation complexity and runtime performance. Thus, there are certain operations that may have multiple allowable behaviours, making their precise behaviour implementation specific. In other cases, there are operations that can result in dynamic errors (at runtime), such as the invalid pointer aliasing mentioned previously. From the perspective of a program generator, these are as important to avoid as undefined behaviour, since it is necessary to ensure that test cases have well-defined and predictable behaviour.

Type	**Implementation-defined behaviours and dynamic errors**
Arithmetic	Floating-point accuracy Floating-point rounding modes
Data layout	Mismatch between host and shader memory layout
Memory access	Invalid pointer aliasing Out-of-bounds array access
Control-flow	Infinite loops
Concurrency	Race conditions Atomic operations on invalid memory locations

Table 3.2: Summary of implementation-defined behaviours and dynamic errors in WGSL.

Table 3.2 summarizes the implementation-defined behaviours and dynamic errors that are present in WGSL. However, there is an additional challenge for a WGSL program generator due to the current status of WGSL compiler development. While there are several behaviours such as arithmetic overflow that are specified by WGSL, it is possible that compiler developers have not yet implemented the necessary checks to enforce them. Given the nature of WGSL compilers as translators from WGSL to an API-specific (typically high-level) backend language, these behaviours may be undefined in the backend language resulting in them being effectively undefined in WGSL for the purpose of program generation.

Type	**Undefined behaviours**
Arithmetic	Integer overflow
	Division by zero
	Invalid shift values
	Modulo of negative values
Built-in functions	Order of arguments in `clamp`
	Bit operations on signed integers

Table 3.3: Undefined behaviours in other shader languages targeted by WGSL compilers.

It is difficult to provide a comprehensive list of such behaviours as it is dependent on the set of functionality used by WGSL compilers for each targeted language. However, table 3.3 lists some of the issues that have been identified and corrected in WGSLsmith, across the targeted backends. These are in addition to the specific WGSL issues in table 3.2. Eventually, these behaviours will be implemented and enforced correctly in WGSL compilers. However, in the short-term, it is necessary for a program generator to handle these as they affect basic operations, and are likely to produce significant noise in results that will hinder finding actual bugs.

3.4 Security Concerns

The addition of WebGPU and WGSL to the web adds new ways for web applications to access GPU hardware. WGSL shaders enable running code on GPUs and add support for compute shaders, which were previously not possible to use on the web. However, this means that any website will be able to download and execute untrusted WGSL code from the internet. While the web already enables untrusted code execution through JavaScript, WebAssembly and WebGL, this creates an additional attack surface which must be made secure.

There are several potential security issues associated with WebGPU and WGSL, such as accessing memory belonging to other programs, CPU and GPU-based undefined behaviour, driver bugs and DoS attacks [2]. In addition to these security concerns, the web also exposes privacy issues that can result in leaking sensitive information that identifies users. The very existence of implementation bugs can provide information about the specific hardware and software that a user is running, as an additional data-point to identify a user. Testing WGSL compilers and WebGPU implementations is therefore crucial to ensuring that web browsers are free from these security issues and can adequately address privacy concerns.

Chapter 4

WGSLsmith

WGSLsmith is a toolkit for automated WGSL compiler testing. There are two main workflows that WGSLsmith aims to facilitate: the fuzzing process, which involves generating and testing shaders to find test cases exhibiting potential bugs, and the reduction process which involves finding a minimized version of a buggy test case. Thus, WGSLsmith provides a fuzzing driver that continuously generates and tests shaders and a reduction driver that oversees reduction using a pluggable reducer.

From an implementation perspective, WGSLsmith consists of several components:

1. A **generator**, which produces random WGSL shader programs to be tested.
2. A **reconditioner**, which processes a shader to remove undefined behaviour and ensure predictable execution.
3. A **test harness**, which executes shaders and checks their outputs.
4. A **reduction driver**, for performing test case reduction on a given shader that exhibits a bug, using a pluggable reducer.
5. A WGSL **parser**, which is used by the reconditioner during reduction and by the harness for extracting I/O information.

These tools are all written primarily in Rust, along with a small amount of C++ and Bash shell scripting. Rust provides language features such as sum types (called `enums` in Rust) which are useful for modelling abstract syntax trees (ASTs). More importantly, as wgpu is also written in Rust it is trivial to integrate with, and thanks to Rust's straightforward FFI (foreign function interface) capabilities it is also simple to integrate with Dawn's C/C++ codebase.

Figure 4.1a shows how the fuzzing process works in WGSLsmith. The generator produces test cases that are processed through the reconditioner before being passed to the test harness. This will execute the shader against multiple configurations and check the results. The filter stage is configured by the user to determine which shaders are interesting – this decides whether the shader should be saved or not based on the execution result.

Given the availability of multiple compiler implementations (Tint and Naga), WGSLsmith uses differential testing (section 2.4.1) to provide a test oracle. As WGSL compilers are currently in active development, there are likely to be frequent changes and fixes, so testing the latest versions of compilers is desirable. Additionally, the WebGPU API does not currently expose an interface for controlling the optimization level of shader compilation, making a cross-optimization approach infeasible. Thus, WGSLsmith primarily uses the cross-compiler strategy for differential testing, but is also able to test multiple backends for the different languages supported by Tint and Naga.

WGSLsmith's reduction driver is outlined in figure 4.1b. The shader is passed to an external reducer program (such as C-Reduce) along with an interestingness test (section 2.5). The reducer will incrementally generate reduced candidates and invoke the interestingness test to determine whether the reduced shader remains interesting.

The following sections discuss in detail the design and implementation of each of these components.

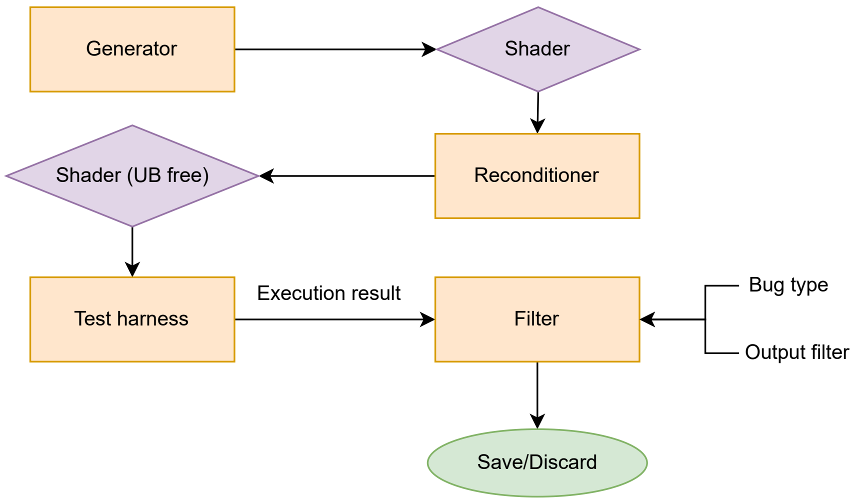
4.1 Generator

WGSLsmith uses a random generator to produce test cases, as this can enable greater program diversity (section 2.2.1). Shaders produced by the generator are guaranteed to be both syntactically well-formed and well-typed, but may behave unpredictably or fail to terminate (this is solved by the reconditioner described in section 4.2).

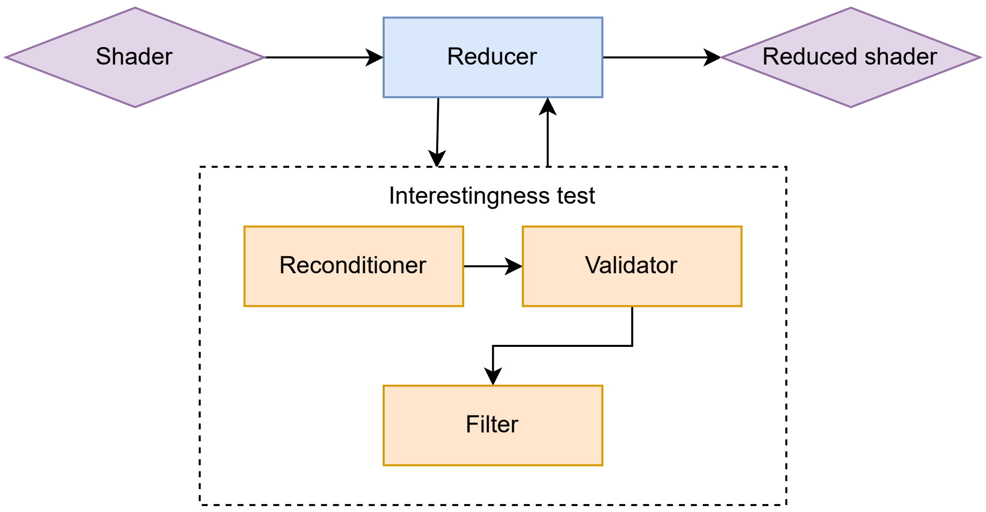
The generator works by constructing an abstract syntax tree (AST), which can later be pretty printed as concrete WGSL syntax. This is useful when using the generator as a standalone tool; however, it is also possible to invoke it as a library and apply further transformations on the AST directly, as is done by the fuzzing driver for reconditioning.

The overall generation strategy is split into two phases. First, a number of supporting constructs including types, global variables and buffer variables are generated. After this, the entrypoint function is generated. Additional functions are generated *on demand*, using a top-down approach similar to Csmith [5]. This process operates recursively, starting from a request to generate a particular construct (such as a function) and recursing until leaf constructs are reached (typically literal expressions or identifiers).

At each stage where a choice of possible constructs that may be produced is available, preconditions for each choice are checked to determine which choices can be generated successfully. A construct type is then selected randomly from this reduced set, using an appropriate distribution. This ensures that there is no need for backtracking as the



(a) Fuzzing



(b) Reduction

Figure 4.1: Fuzzing and reduction workflows in WGSLsmith

generator will never fail to produce a result for a particular request.

4.1.1 Types

WGSL supports user-defined types in the form of structs (section 3.3.1). The process for generating structs is largely straightforward. First, a member count is selected randomly based on configurable bounds, after which types are selected randomly for each member. The allowed types include built-in types, as well as previously generated structs to enable support for nested structs. An example of structs generated by WGSLsmith is shown in listing 4.1.

Notably, when generating types for buffer variables, the set of allowed member types is restricted as WGSL does not allow certain types such as `bool` within host-shareable types. One additional restriction is made by WGSLsmith, which does not currently support nested structs in host-shareable types due to complexities in calculating the alignment and padding, which is necessary for examining buffer values during differential testing. It will be possible to remove this restriction in future.

```
struct Struct_1 {
    a: vec4<f32>,
    b: i32,
}

struct Struct_2 {
    b: vec2<u32>,
    c: vec3<bool>,
    d: i32,
}
```

Listing 4.1: Example of structs generated by WGSLsmith

4.1.2 Global Variables

A collection of global variables is generated early on in the generation process, to enable their use when subsequently generating functions. The number of global variables to generate is selected at random, with user-defined upper and lower bounds. These variables are able to reference types that were defined in the previous stage. Additionally, WGSLsmith will randomly decide to generate or omit an initialization expression for the variable. When generating initialization expressions for global variables, expressions are limited to those that are available in a const context, as required by WGSL.

4.1.3 Expressions

WGSLsmith supports generating most expression types available in WGSL. The expression generator is supplied with a target data type, and recursively generates a suitable expression. Complexity is bounded by a configurable maximum depth – once this limit is reached only leaf nodes (literals and variable identifiers) may be generated.

To generate integer literals, WGSLsmith is able to sample from two different distributions. Firstly, it can sample uniformly from a set of interesting edge cases: for `i32` this is the set {0, 1, -1, `i32::MAX`, `i32::MIN`} and for `u32` this is {0, 1, `u32::MAX`}. These values may be more likely to trigger interesting behaviour where edge cases have not been handled correctly.

The second is sourced from a binomial distribution. Values are sampled from

$$B(\texttt{i32::MAX} \times 2, 0.5)$$

and then shifted by `i32::MAX`, resulting in a distribution centred around 0. This favours generating smaller values, with the aim of reducing the likelihood that operations on these values will be immediately reconditioned away to avoid overflow (section 4.2).

Floating-point operations can be difficult to test, due to differences in behaviour across implementations (section 3.3.4). WGSLsmith uses a very similar approach to [12] to test floating-point operations, by limiting the range of floating-point values and restricting the set of operations. Floating-point literals are generated similarly to integers, to favour smaller values. Details of floating-point support are discussed further in section 4.2.2.

In addition to generating simple variable expressions, WGSLsmith is also able to generate accesses to members for vectors and structs, as well as array element accesses, as shown in listing 4.2. For each type `T`, the generator maintains a set of *accessible types* which refers to the types that can be transitively accessed through `T`. For example, the accessible set of a `vec3<u32>` is {`u32`, `vec2<u32>`, `vec3<u32>`} since `T` is trivially accessible from `T`, `u32` can be obtained by accessing individual members, and `vec2<u32>` can be obtained through vector swizzling. This is used when generating variable declarations, to maintain a mapping from types to a list of variables through which the type may be accessed. Thus, the expression generator can simply perform a lookup in this map to obtain a list of possible variables that can satisfy a request for an expression of a target data type.

```
var var_1 = global1[arg_0 + 6u];
var var_2 = vec3<i32>(36, 8).xz;
var var_3 = Struct_4(1, 13, vec2<u32>()).b;
```

Listing 4.2: Member and array element accesses in WGSLsmith

4.1.4 Functions

WGSLsmith supports both built-in functions and user defined auxiliary functions. The function generation process is driven by the expression generator, which can choose to generate a function call expression. The generator maintains a mapping from types to lists of functions, similarly to the mapping for variables described above, where the type represents the function's return type. An example of a function declaration is illustrated in listing 4.3.

When generating a function call expression, the generator will perform a look-up to determine which functions are available. If no suitable function is found, a new function can be generated. Additionally, if existing functions are available, a new one may still be generated with a smaller probability.

Return statements are supported at arbitrary scope levels within a function. WGSL forbids additional statements within a block, after a return statement, so the generator will immediately finish the current block once a return statement is generated. Also, functions that return a value will always contain a return statement as the final expression.

```
fn func_1(arg_0: bool, arg_1: vec3<i32>) -> i32 {
    return select(arg_1.x, arg_1.y, arg_0);
}
```

Listing 4.3: Example of a function declaration. Additional statements are never included after a return statement, as this is forbidden by WGSL.

WGSLsmith supports all the logical and integer built-in functions that are currently implemented in both Tint and Naga. Additionally, certain functions that are only supported in Tint (at the time of writing) are supported through user-configurable options, for crash testing Tint on its own. For floating-point functions, support is limited to a small set of functions with predictable behaviour. Examples of function call expressions are shown in listing 4.4. The full list of supported built-in functions is available in appendix A.

```
var var_1 = reverseBits(abs(-18151 | var_0.x));
var var_2 = !(select(!vec3<bool>(global2.x), !vec3<bool>(true)));
var var_3 = Struct_1(~(-clamp(abs(arg_1.b.a), u_input.c << vec3<u32>(429u))));
```

Listing 4.4: Function calls generated by WGSLsmith

4.1.5 Pointers

Generating code containing pointers is supported behind an opt-in flag. WGSL allows a restricted form of pointers as described in section 3.3.3. WGSLsmith can generate function parameters of pointer type, as well `let` declarations containing pointers, which allow giving new names to memory locations. Address-of and indirection expressions are supported to construct and dereference pointers. An example is shown in listing 4.5.

```
fn func_1(arg_0: ptr<private, i32>) -> i32 {
    return *arg_0 + 36;
}
```

Listing 4.5: Function with pointer parameters and pointer dereferencing.

The generation process for pointers in WGSLsmith is straightforward. Checking the safety conditions of pointer operations is performed during the reconditioning stage and is described in section 4.2.5.

4.1.6 Statements & Control-Flow

WGSLsmith supports `let` and `var` declaration statements as well as assignments. The standard control-flow constructs available in WGSL are also supported, including `if`, `loop`, `for` and `switch` statements. Additionally, `break` and `continue` statements are supported in loops, and `fallthrough` is supported in `switch` statements.

With `for` loops, WGSLsmith can generate both idiomatic and arbitrary loop headers. Normally, the header consists of a loop variable, optionally initialized with a fixed value, a termination condition involving the loop variable, and an update statement which updates the loop variable by incrementing, decrementing or reassigning it. With some probability, any of these statements can be omitted, or an arbitrary initialization expression, loop condition or update statement may be used.

4.2 Reconditioner

The reconditioner ensures that generated shaders are free of UB and have predictable execution (section 2.5.1). While WGSL does have certain runtime error cases that must be avoided, it has little UB as the semantics of many operations that are typically undefined in other languages are explicitly defined in WGSL (section 3.3.5). However, certain features and checks may not be implemented in compilers yet as they are still in development. In fact, some of the behaviour that the WGSL spec specifies, such as wrapping on overflow for integer operations, is not currently enforced in Tint and Naga. Therefore, to produce useful test cases, we need to ensure that these unimplemented behaviours are avoided in addition to handling other issues such as loop termination.

Previous work has focused on reconditioning as a way to remove undefined behaviour from a program [1, 12]. Reconditioning in WGSLsmith also includes:

- An analysis phase that is able to reject certain shaders. This is useful for avoiding some behaviours that may be difficult to recondition away.
- Workarounds for known compiler bugs that can otherwise cause issues in testing. This is particularly useful when testing languages in active development.

The reconditioner is implemented as a two-step process. The first step involves an analysis of the AST – this is specifically for the detection of invalid pointer aliasing (discussed further in section 4.2.5). The second step is implemented as a transformation of the AST, which removes undesired behaviour. The AST is traversed recursively, and each node is mapped to a new reconditioned node.

4.2.1 Arithmetic Wrappers

The behaviours of basic arithmetic operations including addition, subtraction, multiplication, division and remainder are defined in WGSL. However, Naga and Tint do not currently implement the necessary checks to ensure correct behaviour on all backends in case of issues such as overflow and division-by-zero. Indeed, it is possible to observe undefined behaviour in certain cases where different backends will produce different results for the same operation.

For example, the program in listing 4.6 executes a multiplication that will overflow. Both Tint and Naga will compile this directly to the equivalent HLSL, without safety checks. The overflow behaviour is undefined in HLSL and the result of the multiplication will not simply wrap as expected. In contrast, when executing the compiled SPIR-V code with Vulkan, it wraps as expected since the behaviour in SPIR-V is defined.

```
@group(0) @binding(0)
var<storage, read_write> out_buf: i32;

@compute @workgroup_size(1)
fn main() {
    out_buf = 2147483647 * 2147483647;
}
```

Listing 4.6: Tint and Naga compile this to HLSL that exhibits undefined behaviour.

Thus, to ensure consistent behaviour, these checks must be implemented by WGSLsmith. The checks are performed by various arithmetic wrapper functions, which are automatically inserted during reconditioning. As the AST is traversed, any subexpressions that could produce undesired behaviour are transformed to an appropriate function call with the operands passed as arguments to the function. Additionally, the use of the wrapper is registered with the reconditioner state along with the argument data types. The wrapper implementations are generated at the end of the reconditioning process. All arithmetic wrapper functions are independently usable.

Wrappers are named using the convention "_wgslsmith_{wrapper name}_{data type}". The data type must be included as WGSL does not support function overloading, and wrappers for an operation typically consist of a number of variants for combinations of scalar, vector, signed and unsigned operands. Examples of wrapper functions are shown in listing 4.7.

Most wrapper implementations consist of an expression representing the original operation, a safe fallback expression, and a safety condition. The condition is checked to determine whether the original expression or fallback should be returned. For vectors,

```
fn _wgslsmith_div_vec3_i32(a: vec3<i32>, b: vec3<i32>) -> vec3<i32> {
  return select(a / b, a / vec3<i32>(2),
    a[0] == -2147483648 && (b[0] == -1) || (b[0] == 0) ||
      (a[1] == -2147483648 && (b[1] == -1) || (b[1] == 0)) ||
      (a[2] == -2147483648 && (b[2] == -1) || (b[2] == 0)));
}

fn _wgslsmith_add_i32(a: i32, b: i32) -> i32 {
  return select(a + b, a,
    b > 0 && (a > (2147483647 - b)) || (b < 0 && (a < (-2147483648 - b))));
}

// original
fn func_4() {
    var var_2 = vec3<i32>(23, 11, 4) / vec3<i32>(4 + var_1);
}

// safe version
fn func_4() {
  var var_2 = _wgslsmith_div_vec3_i32(vec3<i32>(23, 11, 4),
    vec3<i32>(_wgslsmith_add_i32(4, var_1)));
}
```

Listing 4.7: Examples of arithmetic wrapper functions and their usage.

the safety condition typically involves applying the same check to each component.

In addition to the basic arithmetic operators, certain built-in functions also have associated wrappers. This includes the `dot` and `clamp` functions. The `dot` function is implemented in terms of multiplications and additions; thus, it has similar overflow behaviour which must be checked. For `i32` operands, the wrapper is implemented by clamping the operands between

$$-\left\lfloor\sqrt{\frac{\texttt{i32::MAX}}{\texttt{n}}}\right\rfloor \quad \text{and} \quad \left\lfloor\sqrt{\frac{\texttt{i32::MAX}}{\texttt{n}}}\right\rfloor$$

where `n` is 1 if the operand is a scalar, otherwise the size of the vector. The `u32` variant is similar with the lower bound replaced with 0. The clamping is performed component-wise for vectors. The bounds are constants, so can be precomputed as an optimization. This guarantees that no overflow will occur, and ensures that the dot operation is always executed to improve the chances of finding bugs. However, it does limit the range of possible values that the function will be applied to.

```
fn _wgslsmith_f_op_f32(v: f32) -> f32 {
    return select(v, f32(10.0),
        abs(v) < f32(0.1) || (abs(v) >= f32(16777216.0)));
}
```

Listing 4.8: Wrapper for scalar floating-point operations.

This is an interesting example of the challenge of implementing wrappers for complex operations. An alternative implementation could do a more advanced check to determine whether the specific combination of operands will overflow. However, this would be significantly more complex and result in a large amount of additional code which could mask bugs. This shows the trade-off in terms of complexity and precision when implementing these wrappers.

4.2.2 Floating-Point

WGSLsmith uses a very similar approach to GLSLsmith [12] for reconditioning floating-point operations. The generator restricts floating-point literals to integer values within the range -16777216.0 to 16777216.0 (excluding 0.0), and does not support floating-point values in input buffers. The reconditioner then only needs to ensure that the results of operations on floats remain in this set.

To implement this, WGSLsmith includes the "_wgslsmith_f_op_{data type}" family of wrapper functions (listing 4.8). These wrappers check that the argument value is within the safe set, and if not replace it with a fixed safe value. All floating-point operations including arithmetic operations and built-in function calls are wrapped, to ensure that at each point in the program, it only operates on safe float values.

In addition to the `+`, `-` and `*` operators supported by GLSLsmith, WGSLsmith can also recondition floating-point division (`/`) by observing that the concrete value returned by an operation is irrelevant; if there is a difference in results for the correct and incorrect cases, this is sufficient to detect a bug. The wrapper implementation is shown in figure 4.9. This checks whether the absolute value of the result is less than the dividend, which should hold as the absolute value of the divisor must be greater than one (fractional literals are not generated in WGSLsmith). Also note that division-by-zero is avoided as zero literals are not produced by the generator. In both cases a fixed safe value is returned. However, this approach is limited as it only performs a basic check on the division, and makes composing operations involving division impossible.

Certain floating-point functions are also supported (full list in appendix A). These functions are guaranteed to be correctly rounded in WGSL, and give exact results when applied to whole numbers, making them safe to use similarly to `+`, `-` and `*`.

```
fn _wgslsmith_div_f32(a: f32, b: f32) -> f32 {
    return select(f32(42.0), f32(-123.0), abs(a / b) > abs(a));
}
```

Listing 4.9: Wrapper for scalar floating-point division.

4.2.3 Loop Limiters

WGSLsmith guarantees that loops (and by extension shaders) will terminate. Shaders that fail to terminate may be forcefully terminated or cause other issues such as crashes. In WGSL, certain cases where infinite loops can be statically detected are considered shader-creation errors and mean that the program will fail validation by the compiler. To enforce loop termination, WGSLsmith uses loop limiters [12, 13] as shown in listing 4.10. Two strategies were considered for loop limiters: *global* limiters and *local* limiters.

Local limiters use local scoped variables to store a loop counter. This means that the counter is effectively reset at the exit of each loop, meaning that if the loop is executed again later (such as through a different invocation of the function, or a different iteration of an outer loop), the counter will restart from 0.

The alternative strategy of global limiters relies on using global variables for each loop. This means that each syntactic loop has a globally fixed upper bound on the number of iterations throughout the execution of the program. The main advantage of this strategy is that it can reduce the execution time of the shader.

After experimenting with local limiters, it was found that these significantly increase execution time and often result in shaders that must be forcefully terminated. Thus, global limiters have been implemented in WGSLsmith. For each syntactic loop in the program, WGSLsmith creates an associated counter, stored in a global array, which is incremented at the start of each loop iteration. A check is additionally inserted at the start of the loop to compare the counter against a maximum value. Once the maximum count is reached, the loop is terminated. This places an upper bound on the number of iterations of each loop.

4.2.4 Array Bounds Checking

Array accesses must be made safe as WGSL allows out-of-bounds accesses to return an arbitrary value of the element type. In WGSL, arrays usually have a fixed size specified as part of the type. Runtime-sized arrays are possible for host-shareable buffers but are not currently supported in WGSLsmith.

Array indexes are reconditioned using the remainder operator (%) to enable all indexes to have a similar probability of being used. Alternatively clamping could be used, but this

```
var<private> LOOP_COUNTERS: array<u32, 17>;
fn func_6() {
    loop {
        if (LOOP_COUNTERS[10u] >= 5u) { break; }
        (LOOP_COUNTERS)[10u] = LOOP_COUNTERS[10u] + 1u;
        ...
    }
}
```

Listing 4.10: Loop limiters in WGSLsmith.

would mean that the extremal values (`0` and `arrayLength-1`) would occur with higher probabilities. For signed integers, it is necessary to ensure that the index is positive (since negative values can behave unpredictably with `%`). This is done using the `abs` function. While `abs` is well-defined for `i32::MIN` in WGSL, it is equivalent to the identity function and will produce a negative value; this case must be checked for separately. WGSLsmith uses a wrapper function for all indexing operations, as shown in listing 4.11.

```
fn _wgslsmith_index_i32(index: i32, size: i32) -> i32 {
    return select(abs(index) % size, 0, index == -2147483648);
}

fn _wgslsmith_index_u32(index: u32, size: u32) -> u32 {
    return index % size;
}

fn main() {
    var a = array<u32, 4>();
    _ = a[_wgslsmith_index_i32(-1234, 4)];
    _ = a[_wgslsmith_index_u32(1234u, 4u)];
}
```

Listing 4.11: Array bounds checking in WGSLsmith.

4.2.5 Pointer Aliasing

WGSL has limited support for pointers, as described in section 3.3.3. While most of the rules relating to pointers are statically enforced and can be avoided by construction, invalid pointer aliasing (defined below) results in a dynamic error which may behave unpredictably at runtime. Approaches to making pointer operations safe have been used in other languages. For example, Csmith uses a combination of runtime checks and static analysis to avoid invalid pointer operations [5].

Dynamic checks are not possible in WGSL, due to its limited pointer capabilities – it is not possible to check pointer values at runtime as the language does not allow comparisons involving pointers. Thus, WGSLsmith relies on static analysis, inspired by previous work on pointer analyses [40, 41]. In contrast to the previous reconditioning techniques, which use local information to apply transformations, this requires global information about the program. The pointer analysis used is context-insensitive and flow-insensitive [41], meaning that it is insensitive to control-flow structures, which loses some precision but reduces implementation complexity.

Informally, the aim of WGSL's aliasing rule is to prevent multiple accesses to the same memory location through different aliases, where at least one of the accesses is a write operation. Below, a more precise explanation of this rule is provided which introduces some key terminology, before describing the analysis.

Memory locations and aliasing

Here, we define the concept of a *memory location* as a memory region that is introduced by a `var` statement (a variable). The variable identifier that introduces a memory location is called the *originating variable*. By definition, each memory location has exactly one originating variable. While function parameters and `let` declarations may also be implemented using memory, they are not considered memory locations according to this definition, for the purpose of the aliasing analysis. This distinction is made as pointers can only point to variables, since they are the only reference types in WGSL (section 3.3.1 & 3.3.3).

As described in section 3.3.1, Pointers can be named in two ways. Firstly, a `let` declaration can be used to assign a name to a pointer expression (i.e. an address-of expression or an existing identifier of pointer type). Secondly, a function may accept a parameter of pointer type. Unlike in other languages such as C, pointers are not *storable*; thus, they cannot be stored in memory locations (as defined above).

Memory locations can be accessed in expressions using identifiers. Additionally, each access to a memory location is performed through a *root identifier*, which is defined as follows. **Within the scope of a single function**, for a given identifier v that refers to a memory location, there are three cases:

1. v is a reference type and refers to an originating variable.
2. v is a pointer type and is a function parameter.
3. v is a pointer type and was named by a `let` declaration.

For (1) and (2), the root identifier is trivially v. For (3), the root identifier is defined as the root identifier of the address-of expression or identifier expression that was used to declare v. Note that we only consider the scope of a single function – if, for example, a `let` is declared in another function and passed as a parameter, the parameter is still the root identifier for all its usages within the callee.

Each distinct root identifier is considered an *alias* of the memory location that the identifier references. From this, it is clear to see that a program that does not contain pointers will only ever have a single alias for each memory location, namely the originating variable. Listing 4.12 illustrates an example of memory locations, originating variables and root identifiers in a WGSL program.

```
// x is an originating variable, and defines a new memory location.
var<private> x: i32;

fn f(p: ptr<private, i32>) -> i32 {
    // x is accessed through root identifier p.
    return *p;
}

fn g() {
    // x is accessed through the originating variable.
    x = 1;
    let q = &x;
    // x is accessed through q, but the root identifier is still x as q is a
    // 'let' declaration.
    *q = 2;
    // f accesses x through p. However, in the context of g the access is
    // performed through x (as p is replaced with &x).
    f(&x);
}
```

Listing 4.12: Illustration of memory locations and root identifiers.

Finally, invalid aliasing occurs when **all** the following conditions are satisfied:

- Multiple aliased root identifiers are used to access a memory location.
- The accesses occur within the same function invocation.
- At least one of these accesses executes a memory write operation.

Detecting invalid aliasing

From the definition of invalid aliasing, it follows that to analyse a function for invalid aliasing, it is necessary to identify which memory locations it accesses and which root identifiers each access is performed through. Therefore, a pointer analysis of the function parameters (of pointer type) is required to determine the possible memory locations they could point to.

The general process for detecting invalid aliasing involves three stages:

1. Collecting information about parameters, memory accesses and nested function calls for each function.
2. Performing an expansion of the collected information.
3. Finding invalid accesses using the expanded information.

The first two stages build the required information as stated above, while the third performs the actual detection.

First, we define a memory access as a pair $\langle \sigma, \omega \rangle$ where $\sigma \in \{\mathit{read}, \mathit{write}\}$ is the type of the access, and ω is the root identifier through which the access is performed. Additionally, each variable declaration and function parameter is assigned a globally unique ID that labels its memory location.

For each function f, we will store:

1. A mapping from pointer parameters of f to a set of possible memory locations that the pointer could refer to.
2. A set of memory accesses, $\mathit{accesses}(f)$, that occur statically within f.
3. A set of function calls, $\mathit{calls}(f)$, that occur statically within f, storing the location of the call, function name and mapping of pointer parameters to root identifiers.

Statically is used to mean a syntactic occurrence within the function, though it may not be executed at runtime (e.g. it may be dead code, or executed conditionally based on inputs). There are three AST nodes of interest during this process: function calls, expressions that read memory, and assignment statements which write to memory.

Below, $\mathit{rootIdentifier}(e)$ denotes the root identifier of an expression e that contains an identifier, $\mathit{memLocs}(e)$ denotes the set of memory locations that $\mathit{rootIdentifier}(e)$ could refer to, and $\mathit{pointsTo}(p)$ denotes the set of possible memory locations that p could point to, where p is a function parameter of pointer type.

When visiting a call to function g with parameters $p_1, ..., p_n$ and arguments $e_1, ..., e_n$, the call is recorded in $\mathit{calls}(f)$ as described above. For each e_i, if p_i is of pointer type then p_i is associated with $\mathit{rootIdentifier}(e_i)$ for the call. Additionally, for each p_i, $\mathit{memLocs}(e_i)$ is added to $\mathit{pointsTo}(p_i)$.

For an expression e that performs a memory read, $\langle \mathit{read}, \mathit{rootIdentifier}(e) \rangle$ is added to $\mathit{accesses}(f)$. Similarly, for an assignment to the memory location represented by expression e, $\langle \mathit{write}, \mathit{rootIdentifier}(e) \rangle$ is added to $\mathit{accesses}(f)$.

Once this is complete, for each f, $\mathit{calls}(f)$ stores the set of calls in f, and $\mathit{accesses}(f)$ stores the set of memory accesses in f. Additionally, $\mathit{pointsTo}(p)$ stores the set of memory locations that pointer-type parameter p could point to, for each p in the entire program.

It is important to note that WGSL does not support recursion. This means that the call graph forms a directed acyclic graph, enabling the analysis to be done in a single iteration. The traversal must start at the root of the call graph (i.e. the `main` function) and use a breadth-first approach, since analysing a function requires having previously built the $\mathit{pointsTo}$ set for each of its parameters (and thus visiting all calls to the function).

The next step involves effectively *inlining* function calls to expand the set of memory accesses within a function. For each function f, consider a call c to function g from f with argument root identifiers $\omega_1, ..., \omega_n$. Then, for each access $\langle \sigma, \omega \rangle \in \mathit{accesses}(g)$, there are two cases:

1. ω is an originating variable.
2. ω is a pointer parameter p_i of g.

(1) is trivial as $\langle \sigma, \omega \rangle$ is added to $\mathit{accesses}(f)$. For (2), $\langle \sigma, \mathit{argRootIdent}(c, p_i) \rangle$ is added to $\mathit{accesses}(f)$, where $\mathit{argRootIdent}(c, p_i)$ is the root identifier of the argument e_i passed for p_i in call c (this information was stored for each call in the previous step). The intention here is to replace memory accesses through pointer parameters in g with the corresponding root identifiers that f passed as arguments when calling g.

After inlining calls, a further expansion step involves resolving pointer parameters to concrete memory locations used for each access. We define a concrete memory access as a pair $\langle m, \psi \rangle$, where m is a memory access and ψ is the ID of a memory location. Then for each access m of the form $\langle \sigma, \omega \rangle$ in function f:

1. If ω is an originating variable then add $\langle m, \omega \rangle$ to $\mathit{concreteAccesses}(f)$.
2. If ω is a pointer parameter p then add $\langle m, \psi \rangle$ to $\mathit{concreteAccesses}(f)$, for each $\psi \in \mathit{pointsTo}(p)$.

This results in a set, $\mathit{concreteAccesses}(f)$, storing the access type, root identifier, and a memory location to which the root identifier could point for each transitive memory access in f.

The final step involves examining $\mathit{concreteAccesses}(f)$ to find invalid aliasing accesses. This can be done simply by looking through the set for memory locations where there are multiple accesses using different root identifiers and at least one access is a write.

Implementation in WGSLsmith

Several implementation details have been omitted above. In particular, details of handling `let` declarations have not been provided. This is considered an implementation detail of the *rootIdentifier* function. In WGSLsmith, this is implemented using a data structure to represent the scope, which is maintained through the AST traversal and maps normal identifiers to their root identifiers. This is used to enable mapping identifiers declared with `let` statements back to the root.

WGSLsmith also performs a small optimization in the final step, storing a map of memory locations to sets of accesses, rather than a single flat set of accesses. This means that accesses to the same memory location are already grouped together.

As mentioned previously, this detection is performed before the main reconditioning stage and rejects shaders that fail the check. This is mainly to simplify the implementation, and in practice has been found to reject approximately 30% of shaders when enabled, which is considered to be acceptable. However, it may be possible to avoid this in future through approaches such as proper reconditioning or avoiding invalid aliasing by construction in the generator.

4.3 Test harness

The test harness is responsible for executing shaders within a WebGPU pipeline, optionally with provided input data, and checking the output buffers to detect potential bugs. The harness is designed to be a general purpose tool for executing WGSL compute shaders, and is capable of handling shaders with an arbitrary number of uniform and storage buffers. For each uniform buffer, the user is able to supply an input buffer in the form of a JSON byte array, which will be used to initialize the buffer. The process of executing a shader through the harness is described in figure 4.2, and involves the following stages:

1. Reflection
2. Preprocessing
3. Execution
4. Buffer checking

To support differential testing, the harness is able to execute shaders against one or more *configurations* specified by the user. A configuration is defined as the combination of a WebGPU implementation (such as Dawn or wgpu) and a graphics adapter. The graphics adapter is identified by its backend type (D3D12, Metal, Vulkan) and a platform-specific device ID. For example, the string `dawn:vk:9348`[1] identifies a configuration that is executed with Dawn using the Vulkan backend, on device ID 9348. The device ID is consistent across WebGPU implementations and backends, and is usually provided by the device driver to uniquely identify the physical device (or software device if e.g. SwiftShader [42] is being used) on the system. For example on Windows, it may correspond to a PCI

[1]This is a real configuration ID for the Vulkan backend used with an Nvidia RTX 3070 GPU on Windows.

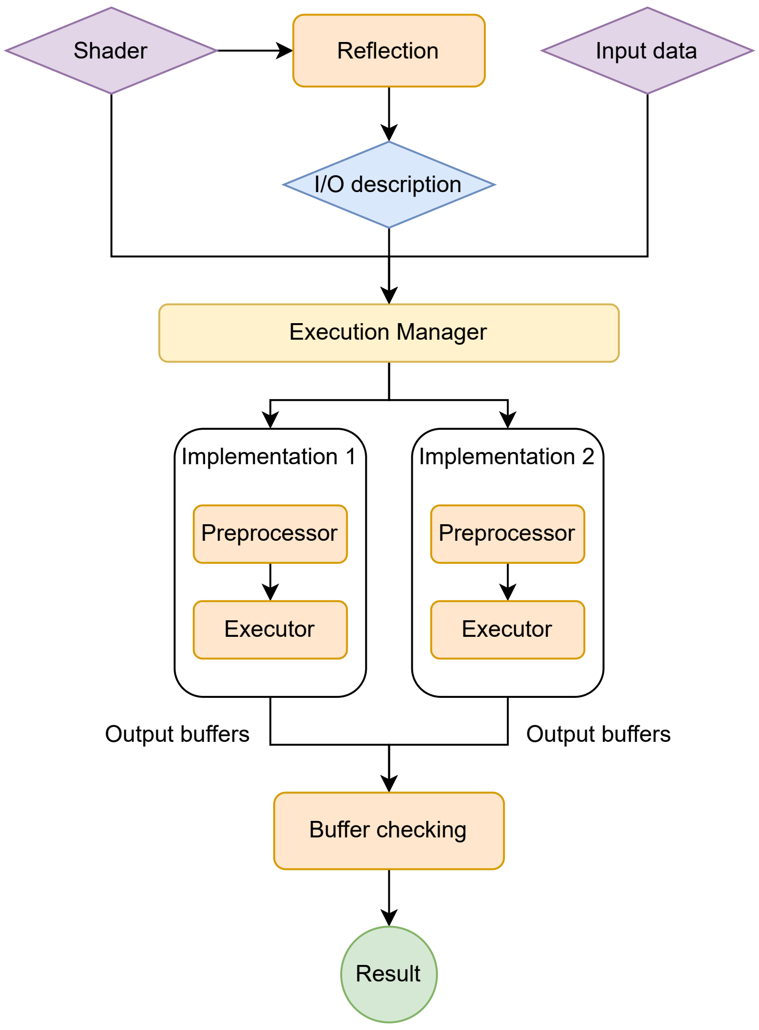


Figure 4.2: Overview of harness execution process.

device ID. This mechanism allows the harness to be used with multiple hardware and software platforms, enabling the user to specify which specific configuration to test.

The harness is also able to support a client-server model, where shaders can be sent to a server process for execution. This is designed to support a remote testing model, such as to perform testing on a mobile phone while driving the process from a desktop machine.

4.3.1 Reflection

To execute a compute shader, it is necessary to set up a WebGPU pipeline (section 3.1). Part of this involves a description of the shader's input and output buffers, including the sizes of the buffers and the slots that they must be bound to. In the harness, the reflection stage is responsible for extracting this information from the shader source code.

This is done by parsing the shader, and traversing the top-level declarations to collect all global variable declarations that are associated with the `uniform` or `storage_buffer` address spaces. Each such buffer variable is required to have a `@group` and `@binding` attribute, which is extracted to determine the slot that the buffer should be bound to. The data type is also inspected to determine the minimum buffer size that must be used.

One important detail to consider is the alignment and padding requirements for host-shareable types in WGSL. This is crucial for two reasons. Firstly, it is important to consider padding within a struct, to correctly compute the minimum buffer size required. If the buffer is too small, the type will not fit and the shader will fail to execute. Secondly, for much of the duration of WGSLsmith's development and testing, the treatment of padding in host buffers has been poorly defined and a point of active discussion for language designers. Thus, it has been necessary to avoid observing padding when performing buffer comparisons for differential testing, as compilers may behave differently. This has recently been finalized in the spec [43], but illustrates the challenges of working on a language in active development.

Another consideration is that in WGSL, a shader's resource interface consists only of buffer variables that are statically accessed within the program. This means that if the shader does not contain any uses of a buffer variable (including in unreachable code), then the variable is not considered part of the shader's interface. Thus, attempting to include the unused buffer in the pipeline may fail. While this is not a problem for executing shaders produced by WGSLsmith (which always use all defined buffer variables), it may result in failure to remove irrelevant variables during test case reduction, and is also inconvenient for manual debugging. The harness thus performs a simple analysis of the program to determine which variables are accessed at any point. Buffer variables that are not accessed are removed from the previously computed list of buffers.

4.3.2 Preprocessing

Another challenge of WGSL's active development status is that it occasionally experiences breaking syntax changes. Often, these changes are purely syntactic and do not affect behaviour in any way. Different compilers may adopt these changes at different paces, resulting in a situation where the latest version of one compiler requires a new syntax while another compiler does not support it. While the compiler developers could provide a deprecation period, this does not always happen; not doing so is justifiable as WGSL is still in early development. However, this is unfortunate as there may be other changes such as bug fixes or new features that are useful to incorporate in WGSLsmith.

The syntactic preprocessing step is introduced in the harness as a method for applying implementation-specific syntactic transformations, in order to patch the shader syntax to enable it to work across compiler implementations. For example, as shown in figure 4.13, the syntax for shader stage attributes was modified in the WGSL specification to be more concise [44]. This was adopted by Naga almost immediately as a breaking change [45], but was only implemented in Tint much more recently [46].

```
// old syntax
@stage(compute)
fn main() { ... }

// new syntax
@compute
fn main() { ... }
```

Listing 4.13: A recent version of the WGSL spec has modified the syntax of shader stage attributes.

The generator produces shaders using the old syntax to remain compatible with Tint, and the preprocessor is able to patch the attribute syntax for executions with wgpu, but not with Dawn. Currently, the preprocessor is implemented as a text-based transformation, though it can apply more complex AST transformations if necessary.

4.3.3 Execution

The shader is compiled during the execution stage, and a pipeline is set up using the appropriate backend API as specified by the configuration. Buffers are created according the information extracted by the reflection process, and initialized with user supplied input data if available. The input data is optional, and defaults to zero-initialized buffers if not supplied.

Additionally, it is possible to provide input data that has a different size to the actual reflected buffer size. If the input data is smaller than the buffer, the trailing bytes in the buffer are zero-initialized. If the input data is larger, then only the leading bytes of the input are used. This is important when performing test case reduction as it allows the reducer to change the sizes of buffers (by removing struct fields or modifying types, for example), without causing the execution to fail.

Shaders are currently executed using a workgroup size of 1 (this means that the shader is essentially single-threaded), as parallelism is not yet supported.

If the execution fails, the harness will immediately terminate with an appropriate exit code signifying a crash. This is useful when testing for crash bugs rather than miscompilations. On a successful execution, storage buffers are mapped on the host (uniform buffers are read-only so will never change) and read back.

In WGSLsmith, separate executors are currently implemented for each of Dawn and wgpu. There has been ongoing effort to standardize a WebGPU C header [47]. This would enable code sharing between conforming implementations. Both Dawn and wgpu have implemented parts of this header; however, certain limitations, particularly regarding asynchronous operations [48], currently require implementation-specific code making this approach infeasible for now in WGSLsmith.

4.3.4 Buffer Checking

The final stage involves comparing the output buffers to find mismatches. As discussed in section 4.3.1, padding bytes in buffers must be ignored since current compiler implementations do not always have consistent behaviour [49] (though this should be rectified in future). The data ranges to examine are computed for each buffer during reflection; this information is then used during buffer checking to only inspect the parts of the buffer that contain data.

If the buffers match across configurations, the harness will exit successfully, indicating that the test case is not interesting. If a mismatch is detected, the harness will exit with an appropriate status code to indicate a buffer mismatch.

4.3.5 Server Mode

The harness supports running in a server mode, to facilitate remote execution workflows. In this mode, it accepts requests over a TCP socket. Requests contain a list of the configurations to execute, as well as the shader source code and optional input data, and are encoded using the Bincode format [50]. For each request, the harness is spawned as a child process (for resilience against crashes) and executed with the provided inputs. The server returns a response containing the exit status code as well as the stderr from the process, for matching against when performing crash testing.

The server also supports controlled parallelism, as it creates a threadpool of worker threads to handle requests. This is particularly useful for test case reduction, where reducers can typically test multiple candidates in parallel. Controlling the size of the threadpool enables controlling the number of shaders that can be executed in parallel.

During this project, the server has been used to enable a workflow where the tools are driven from Windows Subsystem for Linux [51] (WSL, essentially a Linux virtual machine) while executing shaders natively on Windows, as some development and testing tools are easier to use on Linux. In the future, this could be used to enable testing on physically remote devices such as Android smartphones.

4.4 Reduction Driver

The reduction driver is responsible for controlling the test case reduction process. This involves spawning and managing the reducer child process, and providing an interestingness test that allows the reducer to determine whether a candidate shader remains interesting. WGSLsmith includes support for reducing shaders using C-Reduce [26], C-Vise [52], Perses [27] and Picire [25].

4.4.1 Interestingness Test

The interestingness test is a script/program that is invoked by the reducer on a given shader. It must return an exit code of 0 to signify that a shader is interesting (i.e. it continues to reproduce the bug), while a non-zero exit code indicates that the shader should be discarded.

In WGSLsmith, the interestingness is split into two parts. The main test logic is written in Rust and built into the `wgslsmith` program to be exposed through the `test` command. In addition, a wrapper shell script invokes the `test` command with appropriate arguments. Implementing the majority of the logic in Rust allows for sharing code with other components, and enables linking Tint and Naga directly to avoid the need to depend on external executables.

The shell script is still necessary because some reducers strictly require the interestingness test to be a shell script containing a Unix style shebang. The script is parsed to determine which shell to use when invoking the script – presumably, this is used to simplify supporting non-Unix platforms.

The interestingness test supports multiple modes for reducing test case. It is possible to reduce both crashes and buffer mismatches. For mismatches, the test harness is used to execute shaders and compare the output buffers. For crashes, in many cases they are triggered by the backend language compiler (e.g. HLSL or MSL) due to bugs or miscompilations from the WGSL compiler. It is possible to reduce these using the harness, but WGSLsmith also supports calling language-specific validation tools directly. For example, it can invoke Microsoft's FXC [53] compiler to validate the HLSL produced by Tint and Naga. This is often faster than doing a full execution of the shaders, and allows testing on platforms where execution may not be possible (e.g. Metal shaders cannot be executed on Windows, but Apple does provide a version of the Metal compiler for Windows).

When reducing crash bugs a regular expression must also be supplied. The output from the harness/validator is matched against the regular expression to determine whether it is interesting. This helps to avoid bug slippage (section 2.5.2).

Chapter 5

Evaluation

WGSLsmith has been evaluated continuously through the project, to find both crash and miscompilation bugs in Tint and Naga. This has proven to be quite successful with 33 new bugs found in total, to date. In particular, crashes due to WGSL compilers producing incorrect code for the target language have been found to be very common. Currently, 21 bugs have been reported to Naga and 8 have been reported to Tint. 11 of the reported bugs have been fixed so far, with another fix currently pending.

Section 5.1 explores examples of bugs that WGSLsmith has been able to find and highlights features of WGSLsmith that enabled finding these bugs. Section 5.2 describes the bug reporting process and illustrates some of the issues that must be considered in large bug-reporting campaigns. Finally, section 5.3 evaluates the performance of multiple reducers when applied to WGSL, and validates the use of reconditioning as an effective method for avoiding UB during reduction.

5.1 Bugs Found

WGSLsmith has found several types of bugs in Tint and Naga, including compiler crashes as well as miscompilations causing unexpected runtime behaviour. Table 5.1 presents a summary of bugs that have been found for each combination of compiler and backend that has been tested, grouped into crashes and miscompilations. Table 5.2 shows the current statuses of bugs. Note that the counts are exclusive (i.e. confirmed does not include reported and fixed does not include confirmed).

HLSL bugs appear to have been the most lucrative, particularly in Naga's HLSL backend. This is likely to be because the language is not as well-specified as MSL and SPIR-V in some cases, and often contains edge cases and compiler bugs which can be overlooked.

In typical compiler testing, **crashes** refer specifically to those crashes that occur during

Compiler	Crashes	Miscompilations	Total
Tint-HLSL	3	4	7
Tint-MSL	4	3	7
Tint-SPIRV	0	1	1
Naga-HLSL	10	6	16
Naga-MSL	5	3	7
Naga-SPIRV	0	2	2
Total	22	19	41
Total Distinct	20	13	33

Table 5.1: Summary of bugs affecting Tint and Naga. Several bugs affect both Tint and Naga, possibly across multiple backends. Results for *Total Distinct* show the deduplicated counts across compilers and backends.

Status	Count
Unreported	6
Reported	2
Confirmed	13
Pending Fix	1
Fixed	11

Table 5.2: Statuses of bugs that have been found across all compilers and backends. Rows are exclusive (e.g. reported does not include confirmed).

execution of the compiler under test, while crashes that may occur when running the compiled program are considered **miscompilations**. WGSL is an interesting case, since the compilation process is split into multiple stages. A first-stage compiler such as Tint and Naga will translate WGSL code into an appropriate language for the target graphics API, while a second-stage compiler is invoked later when creating a WebGPU pipeline. Here, we will refer to a crash that occurs during either compilation stage as a **crash** bug, while a bug that causes incorrect behaviour during actual shader execution is referred to as a **miscompilation**.

5.1.1 Smallest integer literals in Metal

A group of related bugs have been found involving operations on the smallest integer literal (`-2147483648`). The metal compiler treats this literal as a 64-bit integer rather than a 32-bit integer. It is likely that the compiler parses it as a unary negation operation applied to a positive literal, rather than a negative literal, but positive `2147483648` is not representable as a 32-bit signed integer.

In Naga, this is not handled correctly as the compiler naively translates the literal and the related operations to their MSL equivalents, without inserting appropriate checks or transformations. This can cause a crash when the MSL (Metal Shading Language) code is later compiled. Listing 5.1 shows an example of a left shift operation applied to `INT_MIN`, the result of which is stored as a value of type `i32`.

```
struct Struct_1 {
    a: i32,
}

fn f(a: u32) {
    _ = Struct_1(-2147483648 << a);
}
```

Listing 5.1: This code performs a left shift of an INT_MIN literal. This is compiled by Naga to MSL that is rejected by the Metal compiler.

WGSLsmith was able to discover these bugs thanks to its literal generation process (section 4.1.3). This involves sampling from multiple distributions, including a uniform distribution that contains specific values likely to be edge cases, such as `INT_MIN`.

5.1.2 Pointer bugs

WGSLsmith has been able to identify two issues related to pointers so far, thanks to its support for generating code involving pointers. This includes a crash bug affecting Naga, and a miscompilation resulting in unexpected runtime behaviour that affects both Tint and Naga.

The crash is caused by generating incorrect HLSL code for functions that accept pointers to an array. Naga translates the parameter as a pointer to the element type, rather than to the array itself. This is rejected by Microsoft's FXC [53] compiler when compiling the resulting HLSL, due to the incorrect types.

The miscompilation is an interesting case of pointer aliasing, where an alias exists as a function parameter but is not actually used to perform a memory access. An example is illustrated in listing 5.2, where the `flag` variable is only accessed through a single alias throughout the program.

Though listing 5.2 illustrates a simple and arguably contrived example, a more complex scenario could involve conditionally accessing different pointer arguments of a function, which may be more representative of code that a programmer would write. While such an example is likely to be rejected by WGSLsmith's conservative pointer analysis, this shows that the analysis is still able to allow related cases that can highlight the same problem, and is liberal enough to enable finding interesting bugs.

```
@group(0) @binding(1)
var<storage, read_write> s_out: i32;
var<private> flag: bool;

fn func(p: ptr<private, bool>) { flag = true; }

@stage(compute) @workgroup_size(1)
fn main() {
    func(&flag);
    if (flag) { s_out = 1; }
    else { s_out = 2; }
}
```

Listing 5.2: `func` accepts a pointer `p`, which is an alias of `flag` at runtime. However, it does not read or write `p`, so no memory access is performed through `p` and there is no invalid aliasing occurring. This program results in unexpected behaviour when executed with the DirectX backend (using HLSL).

Both issues above have been reported to and confirmed by the Naga developers. Since pointer support was added at a very late stage of the project and has not seen as much testing as other features, it is possible that further issues will be discovered in future.

5.1.3 FXC issues affecting WGSL compilers

FXC [53] is the legacy tool for compiling HLSL, and has been superseded by the newer DXC compiler [54], which is open-source and based on LLVM infrastructure. While FXC is generally not maintained any longer (other than for major bug fixes), Tint and Naga still aim to retain compatibility with it for now, due to issues such as availability and compatibility of DXC on older systems. Indeed, wgpu does not include support for DXC at this time. This means that workarounds for issues with FXC must be implemented in WGSL compilers, where FXC rejects or miscompiles seemingly valid HLSL.

WGSLsmith has been able to identify a number of these FXC related issues, including 3 crash bugs where FXC rejects valid code, and 4 cases of unexpected behaviour at runtime. Two of the miscompilations affected both Tint and Naga and have been fixed in both, and one of the crashes affecting Naga has also been fixed.

Listing 5.3 shows a WGSL program that safely wraps a division by zero. However, the HLSL code generated by Naga for this is rejected by FXC, despite being valid, as FXC believes there to be a statically-detectable division by zero. This particular case is interesting as the WGSL specification requires certain division semantics that must be checked at runtime. Naga does not yet implement the check, but Tint implements it as a ternary operation in HLSL wrapping the division (resulting in two checks), which has the side effect of appeasing FXC too. This is shown in listing 5.4.

```
fn divide(a: i32, b: i32) -> i32 {
    if (b == 0) { return a / 2; }
    else { return a / b; }
}

@stage(compute) @workgroup_size(1)
fn main() {
    _ = divide(0, 0);
}
```

Listing 5.3: Naga compiles this WGSL to valid HLSL code. However, FXC complains about a possible division by zero, despite the division operation being checked at runtime.

```
// WGSL
fn _wgslsmith_div_i32(a: i32, b: i32) -> i32 {
  return select(a / b, a / i32(2), ((a == -2147483648) && (b == -1)) || (b == 0));
}

// HLSL
int _wgslsmith_div_i32(int a, int b) {
  bool tint_tmp_1 = (a == -2147483648);
  if (tint_tmp_1) { tint_tmp_1 = (b == -1); }
  bool tint_tmp = (tint_tmp_1);
  if (!tint_tmp) { tint_tmp = (b == 0); } // <- WGSLsmith's b == 0 check
  return ((tint_tmp) ? (a / int(2)) : (a / (b == 0 ? 1 : b))); // <- Tint's check
}
```

Listing 5.4: HLSL code generated by Tint for WGSLsmith's `i32` division wrapper. The second `b == 0` check inserted by Tint is able to prevent FXC from throwing a division by zero error for the case in listing 5.3.

5.1.4 Switch statements

Three bugs have been found by WGSLsmith related to Naga's handling of `switch` statements in its HLSL backend. One of these is due to an FXC issue where it incorrectly identifies an invalid fallthrough between `case` blocks, while another is due to a bug in Naga's implementation of the `fallthrough` statement in WGSL (listing 5.5). The final issue is caused by FXC rejecting uses of the `continue` statement in `switch` statements, even if the `switch` is contained within a loop.

```
// WGSL
fn main() {
    switch (1) {
        case -1: { fallthrough; }
        default: { let var_2 = i32(11632); }
    }
}

// HLSL
void main() {
    switch(1) {
        case -1: {
            {}
            { int var_2_ = int(11632); } break;
        }
        default: { int var_2_1 = var_2_; break; }
    }
}
```

Listing 5.5: Naga generates HLSL that duplicates the contents of the `default` block in the first `case` block, to simulate a `fallthrough`. However, the `default` block then attempts to reference a variable defined in the `case` block, which is invalid as it is not in scope.

5.1.5 Implicit array initialization

As described in section 4.1.2, WGSLsmith is able to omit the initialization expression when generating global variables. This was able to detect an issue in Naga's SPIR-V backend, where variables in the `private` and `function` address spaces were not being correctly initialized when lacking an explicit initializer expression. In certain cases, this could result in unexpected runtime behaviour.

5.2 Bug Reporting

Throughout the project, bugs that have been found by WGSLsmith have been reported to compiler developers, through Naga's GitHub repository and Tint's issue tracker. The developers have been receptive to reported bugs, with Naga developers in particular often providing quick responses that have been helpful for validating issues. Several of the bugs that have been reported have received fixes, and one currently has a pending pull request.

There are important social issues that must be considered when reporting bugs in bug-finding campaigns such as this [55, 56]. Submitting many bug reports in a short period of time can be overwhelming for developers who will have to investigate and validate the reports. Particularly with WGSL compilers, there are some issues that may be related to functionality which has simply not been implemented yet. In such a case the developers may not be interested in a report, so this is an additional consideration when deciding to report a bug.

Care has been taken to locally investigate and validate all of my bug submissions, where possible. In addition, bug reports detail reproducible code examples, as well as intermediate code where this is relevant, and provide details of error messages and expected outcomes.

It is important to maintain a good relationship with developers, and it is possible to come across as a nuisance if simply dumping large numbers of bug reports. Thus, I have also contributed fixes for some of the issues found by WGSLsmith [57–59]. The intent of WGSLsmith is to assist WGSL compiler developers with ensuring their compilers are robust and reliable, and contributing fixes can help to ease the burden on maintainers. In addition, it can help with further bug-finding as existing bugs can mask other bugs.

5.3 Reducer Evaluation

The aim of program reconditioning is to enable existing general-purpose program reducers to work with languages such as WGSL, which existing reducers cannot automatically be applied to while maintaining correctness of the reduced programs (section 2.5.1). This section evaluates the effectiveness of different reduction tools when combined with reconditioning, for WGSL compiler testing. It also provides a general comparison of reducer performance on WGSL programs.

Reducers have been tested using a collection of 17 test cases found with WGSLsmith, that exhibit crash bugs. This is split into 8 shaders with bugs affecting the HLSL backends and 9 shaders with bugs affecting MSL backends. These have been selected as they are all reproducible with the same compiler versions. Crash bugs are used for this experiment as they are easy to reproduce across multiple systems.

Four reducers have been tested, including C-Reduce, C-Vise, Perses and Picire. The tests have been conducted on a machine with a 12-core 3.8GHz AMD Ryzen 9 3900X CPU and 32 GB of RAM, running Ubuntu 20.04. Each of these reducers supports parallelism, so both single-threaded and multithreaded reductions have been tested.

Reduction has been performed using WGSLsmith's special crash reduction mode. As described in section 4.4.1, this avoids using the harness to execute shaders, and instead compiles WGSL shaders directly to the target language and uses language-specific validation tools to test for the presence of crashes. This is done to avoid making these tests platform-specific.

The speed of reduction is measured by recording the total reduction time per test case, as well as the number of calls made to the interestingness test. This is measured by sending a signal from the test script to the reduction driver process to increment a counter, rather than through other methods such as the filesystem, to ensure that each test script call remains isolated when multithreading is enabled. The level of reduction achieved is also measured, by comparing the size in bytes of the reduced file to the original.

Note that most reducers will continue running until a fixpoint is reached, before terminating. However, Picire appears to favour fast reduction times at the cost of not maximizing the level of reduction possible. Thus, a similar approach to [1] has been used by running Picire on successively reduced shaders three times to allow Picire to perform further reduction if possible.

Table 5.3 summarizes the reduction percentages achieved by each reducer. The results are given for multithreaded reductions, though the single-threaded results are largely identical. Table 5.4 details the full results including sizes and timing comparisons between reducers.

All reducers have been able to achieve reasonable levels of reduction on average, including Picire, though it has poor worst-case performance. C-Reduce, C-Vise and Perses in particular have all been able to achieve similar levels of reduction. Reduction times are more varied, with Perses demonstrating the fastest mean and median reduction times.

	Min	Max	Mean	Median
C-Reduce	92.93%	99.69%	97.52%	97.61%
C-Vise	90.77%	99.66%	96.96%	97.25%
Perses	92.63%	99.58%	97.68%	98.15%
Picire	30.41%	97.86%	74.63%	83.36%

Table 5.3: Percentage reduction achieved by each reducer. Results are given for multithreaded reductions, but single-threaded results are largely identical.

It is interesting to see that C-Reduce and C-Vise can benefit quite significantly from multithreading, with C-Reduce achieving 56% and 40% improvements for mean and median times respectively, while Perses does not see similar improvement. Comparing the number of test calls between C-Reduce and Perses in single and multithreaded modes shows an increase in calls for C-Reduce and a small decrease for Perses with multithreading. This suggests different parallelization strategies between the reducers, where Perses may be able to use parallelism to rule out certain paths more quickly, while C-Reduce appears to test as many candidates as possible.

In practice, all reducers have been able to achieve a reasonable level of reduction with

		Min	Max	Mean	Median
Original	Size (bytes)	5785	17496	11592.5	11749
C-Reduce (s)	Time (m:s)	00:42	07:48	02:07	01:17
	Test calls	1965	26435	8230.7	6737
	Size (bytes)	46	973	288.6	256
C-Reduce (m)	Time (m:s)	00:14	03:12	00:56	00:46
	Test calls	2686	30069	10151.2	8819
	Size (bytes)	46	973	288.6	256
C-Vise (s)	Time (m:s)	00:38	09:19	02:30	01:43
	Test calls	1387	26018	7653.7	6050
	Size (bytes)	51	1270	357.6	288
C-Vise (m)	Time (m:s)	00:26	04:20	01:28	01:01
	Test calls	2161	28242	8774	6969
	Size (bytes)	51	1270	357.6	288
Perses (s)	Time (m:s)	00:14	01:21	00:30	00:31
	Test calls	90	960	301.2	222
	Size (bytes)	59	1014	266.4	158
Perses (m)	Time (m:s)	00:14	01:14	00:29	00:30
	Test calls	87	904	295.3	232
	Size (bytes)	59	1014	266.4	158
Picire (s)	Time (m:s)	00:16	09:34	01:40	01:06
	Test calls	145	2691	1799.2	2067
	Size (bytes)	158	10257	3117	2103
Picire (m)	Time (m:s)	00:09	04:40	01:09	00:32
	Test calls	172	6423	3900.8	3699
	Size (bytes)	158	10263	3124.6	2103

Table 5.4: Comparison of reduction performance across multiple reducers. (s) indicates single-threading and (m) indicates multithreading with 8 threads.

acceptable run-times, and reconditioning is able to ensure that the reduced programs contain predictable behaviour. Perses has proven to be particularly effective at performing fast reductions of WGSL programs. This is as expected since it uses a language grammar to ensure that reduction candidates are syntactically valid, allowing it to avoid testing large groups of candidates that other reducers such as C-Reduce will try.

Chapter 6

Conclusion

WGSLsmith is one of the first tools for large-scale testing of WGSL compilers and has proven to be an effective bug-finding tool, having found 33 bugs in Tint and Naga to date. Most of these have been reported to and confirmed by developers and 11 have been fixed so far, with another fix currently pending. This shows that WGSLsmith is able to find important bugs that developers want to fix. In addition, WGSLsmith's novel support for testing pointers in a shader language has enabled finding two bugs relating to pointers in WGSL.

Applying reconditioning to WGSL has worked well to ensure correct and predictable behaviour of generated test cases, enabling effective automatic test case reduction. A number of existing reducers have been shown to be effective for WGSL, with Perses in particular providing fast reduction times and high levels of reduction. Reconditioning has also been particularly useful in the context of WGSL, by providing a way to implement fixes and mitigations for current compiler bugs and missing UB checks. This is very useful for testing languages in active development, such as WGSL.

Working on a language in flux has been challenging, as there have been both syntactic and behavioural changes to the language over the course of the project, which have been adopted by compilers at varying paces. Syntactic preprocessing (section 4.3.2) and reconditioning techniques have helped to mitigate some of these problems, by smoothing out small language differences between compilers.

6.1 Future Work

WGSLsmith is able to generate code using a variety of WGSL language features; however, there are still several features that are currently missing. In particular, WGSLsmith does not support matrices or bitcast operations, and lacks some of WGSL's control-flow constructs such as `continuing` (not to be confused with `continue`) and `break-if` statements. This is due to time constraints and these features are good candidates for future extensions. More complex extensions to WGSLsmith could include support for concurrency in shaders (using barriers and atomics) and the ability to test graphics shaders.

Additionally, floating-point support is currently limited to a restricted set of values and operations. No floating-point related bugs have been identified so far, although the feature has not seen as much testing as other features. As floating-point operations in WGSL usually specify the possible rounding error that could occur, it may be interesting to see if this could be used to make checks more precise and to relax some of the above restrictions, which could prove more effective at finding bugs.

While WGSLsmith supports pointers with promising initial results, the analysis contains some limitations. It is currently context-insensitive, meaning that it does not properly consider control-flow which can result in false positives. Further work could be done to make the analysis more precise, allowing it to accept more shaders. Alternative approaches could also be explored that do not require rejecting shaders at all.

Finally, the platform support for WGSLsmith could be extended to enable testing on other devices. As WGSLsmith supports remote shader execution, it should be possible to perform testing on mobile devices such as Android smartphones with minimal effort. iOS could be supported similarly, though may require more work due to platform restrictions.

6.2 Ethical Considerations

This project aims to develop tools to improve the reliability and security of compiler implementations used in web browsers. Web browsers are a key target for attackers thanks to their massive audience and the complexity of the modern web which entails a wide attack surface. Thus, there is potential for this work to improve the security of web users' personal devices and safeguard private data.

However, there is also potential for fuzzing tools to be misused. It is possible for this work to be used to find bugs in a browser by a malicious attacker. If these bugs are security critical, an attacker may be able to exploit them to attack users' devices. However, the risk of exploitable bugs exists regardless of whether tools to find these bugs exist or not; thus, it is important that browser vendors patch security issues in a timely fashion to avoid attacks.

Another consideration for fuzzing tools is the environmental impact of a large fuzzing campaign. Fuzzing can be described as a "brute-force" approach as it involves generating and testing a very large number of test cases in the hope that a small fraction will trigger bugs. This can result in high energy usage, so improving the efficiency of the process can have beneficial environmental impacts.

Appendix A

Built-In Functions

Type	Function

Logical	`all`
	`any`
	`select`

Integer	`abs`
	`clamp`
	`countOneBits`
	`reverseBits`
	`firstLeadingBit`
	`firstTrailingBit`
	`min`
	`max`
	`countLeadingZeros`
	`countTrailingZeros`
	`extractBits`
	`insertBits`
	`dot`

Floating-point	`ceil`
	`floor`
	`round`
	`sign`
	`trunc`
	`max`
	`min`
	`step`

Table A.1: Built-in WGSL functions supported by WGSLsmith.

Bibliography

[1] Alastair F. Donaldson and Bastien Lecoeur. Program reconditioning: Avoiding undefined behaviour during compiler test case reduction. Unpublished, 2022.

[2] Dzmitry Malyshau, Kai Ninomiya, Brandon Jones, and Justin Fan. WebGPU, 2022. URL `https://www.w3.org/TR/2022/WD-webgpu-20220617/`. Accessed: 2022-06-20.

[3] David Neto, Myles C. Maxfield, and Dan Sinclair. WGSL, 2022. URL `https://www.w3.org/TR/2022/WD-WGSL-20220617/`. Accessed: 2022-06-20.

[4] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 294–305, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. doi: 10.1145/2931037.2931074. URL `https://doi.org/10.1145/2931037.2931074`.

[5] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306638. doi: 10.1145/1993498.1993532. URL `https://doi.org/10.1145/1993498.1993532`.

[6] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020. doi: 10.1145/3428264. URL `https://doi.org/10.1145/3428264`.

[7] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[8] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 216–226, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327848. doi: 10.1145/2594291.2594334. URL `https://doi.org/10.1145/2594291.2594334`.

[9] Christian Lindig. Random testing of C calling conventions. In *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging*, AADEBUG'05, page 3–12, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930507. doi: 10.1145/1085130.1085132. URL `https://doi.org/10.1145/1085130.1085132`.

[10] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7): 107–115, jul 2009. ISSN 0001-0782. doi: 10.1145/1538788.1538814. URL `https://doi.org/10.1145/1538788.1538814`.

[11] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017. doi: 10.1145/3133917. URL `https://doi.org/10.1145/3133917`.

[12] Bastien Lecoeur. GLSLsmith: a Random Generator of OpenGL shader programs. Master's thesis, Imperial College London, 2021.

[13] 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)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:29, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. ISBN 978-3-95977-154-2. doi: 10.4230/LIPIcs.ECOOP.2020.22. URL `https://drops.dagstuhl.de/opus/volltexte/2020/13179`.

[14] Alastair F. Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 1017–1032, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454092. URL `https://doi.org/10.1145/3453483.3454092`.

[15] Google. Tint, n.d. URL `https://dawn.googlesource.com/tint`. Accessed: 2022-06-20.

[16] Rust Graphics Mages. Naga, n.d. URL `https://github.com/gfx-rs/naga`. Accessed: 2022-06-20.

[17] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. *SIGPLAN Not.*, 50(10):386–399, oct 2015. ISSN 0362-1340. doi: 10.1145/2858965.2814319. URL `https://doi.org/10.1145/2858965.2814319`.

[18] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, page 849–863, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450344449. doi: 10.1145/2983990.2984038. URL `https://doi.org/10.1145/2983990.2984038`.

[19] Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. *Closer to the Edge: Testing Compilers More Thoroughly by Being Less Conservative about Undefined Behaviour*, page 1219–1223. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450367684. URL `https://doi.org/10.1145/3324884.3418933`.

[20] W.E. Howden. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, SE-4(4):293–298, 1978. doi: 10.1109/TSE.1978.231514.

[21] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. *ACM Comput. Surv.*, 53(1), feb 2020. ISSN 0360-0300. doi: 10.1145/3363562. URL `https://doi.org/10.1145/3363562`.

[22] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, 1998.

[23] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-7, page 253–267, Berlin, Heidelberg, 1999. Springer-Verlag. ISBN 3540665382.

[24] Ralf Hildebrandt and Andreas Zeller. Simplifying failure-inducing input. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, page 135–145, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581132662. doi: 10.1145/347324.348938. URL `https://doi.org/10.1145/347324.348938`.

[25] Renáta Hodován. Picire, n.d. URL `https://github.com/renatahodovan/picire`. Accessed: 2022-06-20.

[26] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI

’12, page 335–346, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312059. doi: 10.1145/2254064.2254104. URL https://doi.org/10.1145/2254064.2254104.

[27] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE ’18, page 361–371, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381. doi: 10.1145/3180155.3180236. URL https://doi.org/10.1145/3180155.3180236.

[28] Josie Holmes, Alex Groce, and Mohammad Amin Alipour. Mitigating (and exploiting) test reduction slippage. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, A-TEST 2016, page 66–69, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450344012. doi: 10.1145/2994291.2994301. URL https://doi.org/10.1145/2994291.2994301.

[29] Microsoft. DirectX graphics and gaming, 2022. URL https://docs.microsoft.com/en-gb/windows/win32/directx. Accessed: 2022-06-20.

[30] Apple. Metal, n.d. URL https://developer.apple.com/metal/. Accessed: 2022-06-20.

[31] Khronos Group. OpenGL, n.d. URL https://www.opengl.org/. Accessed: 2022-06-20.

[32] Khronos Group. Vulkan, n.d. URL https://www.vulkan.org/. Accessed: 2022-06-20.

[33] Microsoft. High-level shader language (HLSL), 2021. URL https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl. Accessed: 2022-06-20.

[34] Apple. Metal shading language specification version 3.0, 2022. URL https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf. Accessed: 2022-06-20.

[35] Khronos Group. SPIR-V Specification, 2021. URL https://www.khronos.org/registry/SPIR-V/specs/unified1/SPIRV.html. Accessed: 2022-06-20.

[36] John Kessenich, Dave Baldwin, and Randi Rost. The OpenGL® Shading Language, 2014. URL https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.40.pdf. Accessed: 2022-06-20.

[37] Google. Dawn, n.d. URL https://dawn.googlesource.com/dawn. Accessed: 2022-06-20.

[38] Rust Graphics Mages. wgpu, n.d. URL `https://github.com/gfx-rs/wgpu`. Accessed: 2022-06-20.

[39] IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008. doi: 10.1109/IEEESTD.2008.4610935.

[40] Vlas Zyrianov, Christian D. Newman, Drew T. Guarnera, Michael L. Collard, and Jonathan I. Maletic. Srcptr: A framework for implementing static pointer analysis approaches. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, page 144–147. IEEE Press, 2019. doi: 10.1109/ICPC.2019.00031. URL `https://doi.org/10.1109/ICPC.2019.00031`.

[41] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, page 54–61, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 1581134134. doi: 10.1145/379605.379665. URL `https://doi.org/10.1145/379605.379665`.

[42] Google. SwiftShader, n.d. URL `https://swiftshader.googlesource.com/SwiftShader`. Accessed: 2022-06-20.

[43] Alan Baker. Disallow accessing padding, 2022. URL `https://github.com/gpuweb/gpuweb/pull/2987`. Accessed: 2022-06-20.

[44] Jim Blandy. Use concise forms for entry point stage attributes., 2022. URL `https://github.com/gpuweb/gpuweb/pull/2652`. Accessed: 2022-06-20.

[45] Igor Shaposhnik. [wgsl-in] update entry point stage attributes, 2022. URL `https://github.com/gfx-rs/naga/pull/1833`. Accessed: 2022-06-20.

[46] Dan Sinclair. Issue 1503: concise shader stage attributes, 2022. URL `https://bugs.chromium.org/p/tint/issues/detail?id=1503`. Accessed: 2022-06-20.

[47] Correntin Wallez, Kai Ninomiya, Austin Eng, ShrekShao, Rajveer Malviya, Dzmitry Malyshau, Westerbly Snaydley, and Jiawei Shao. webgpu-headers, n.d. URL `https://github.com/webgpu-native/webgpu-headers`. Accessed: 2022-06-20.

[48] Tianqi Chen. [discuss] synchronization or lightweight sync to async abstraction, 2020. URL `https://github.com/webgpu-native/webgpu-headers/issues/47`. Accessed: 2022-06-20.

[49] Alastair F. Donaldson. Issue 1544: Issue with padding in storage buffer struct, 2022. URL `https://bugs.chromium.org/p/tint/issues/detail?id=1544`. Accessed: 2022-06-20.

[50] Ty Overby and Trangar. Bincode, n.d. URL `https://github.com/bincode-org/bincode`. Accessed: 2022-06-20.

[51] Microsoft. Windows subsystem for linux documentation, 2021. URL `https://docs.microsoft.com/en-us/windows/wsl/`. Accessed: 2022-06-20.

[52] Martin Liška. C-Vise, n.d. URL `https://github.com/marxin/cvise`. Accessed: 2022-06-20.

[53] Microsoft. Effect-Compiler Tool, 2020. URL `https://docs.microsoft.com/en-us/windows/win32/direct3dtools/fxc`. Accessed: 2022-06-20.

[54] Microsoft. DirectX Shader Compiler, 2022. URL `https://github.com/microsoft/DirectXShaderCompiler`. Accessed: 2022-06-20.

[55] John Regehr. Responsible and effective bug finding, 2020. URL `https://blog.regehr.org/archives/2037`. Accessed: 2022-06-20.

[56] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, feb 2010. ISSN 0001-0782. doi: 10.1145/1646353.1646374. URL `https://doi.org/10.1145/1646353.1646374`.

[57] Hasan Mohsin. [hlsl-out] Fix countOneBits and reverseBits for signed integers, 2022. URL `https://github.com/gfx-rs/naga/pull/1928`. Accessed: 2022-06-20.

[58] Hasan Mohsin. Fix generated hlsl for writes to scalar/vector storage buffer, 2022. URL `https://github.com/gfx-rs/naga/pull/1903`. Accessed: 2022-06-20.

[59] Hasan Mohsin. Implement reverseBits and countOneBits for SPIR-V, 2022. URL `https://github.com/gfx-rs/naga/pull/1897`. Accessed: 2022-06-20.