Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

WebGPU API Fuzzing

Randomised smart call sequence generation for WebGPU

Author:
Daniel Simols

Supervisor:
Prof. Alastair Donaldson

Second Marker:
Dr. Jamie Willis

June 17, 2024

Abstract

WebGPU is an exciting new standard for a JavaScript API that provides access to the GPU on the web for rendering and general computation. However, bugs in the WebGPU implementation can expose new attack surfaces that can be abused by malicious websites. Thus, it is critical to test the API extensively to limit exploitable bugs.

This paper introduces *wg-fuzz*, a tool that intensively tests the WebGPU API by smart call sequence generation. An exploration of how to implement a general fuzzer for a library API like WebGPU is carried out. This paper subsequently introduces the concept of fuzzy conditions in the context of the *wg-fuzz* fuzzer.

Finally, *wg-fuzz* is run on Dawn, Chrome's implementation of WebGPU. 14 distinct bugs are found and reported, 6 of which cause a fatal crash in an otherwise normal JavaScript WebGPU program. Overall, 4 of the bugs have since been fixed. Lastly, a fuzzing campaign is run that lasts for 1,100h compute-hours and enables an evaluation of the effectiveness of *wg-fuzz*'s features.

Acknowledgements

I would like to thank Prof. Alastair Donaldson for the opportunity to work on such an exciting project. His feedback and encouragement has been absolutely indispensable. Additionally, Amber Gorzynski brought crucial support with her updates to WGSLsmith, which *wg-fuzz* uses.

I thank Hasan Mohsin for his WGSLsmith tool, which *wg-fuzz* uses to create more interesting programs.

I thank the Dawn developers for their swift responses and updates to the Dawn code-base following my bug reports.

Finally, I thank my friends and family, for their ever unwavering support.

Contents

1 | Introduction

The new WebGPU JavaScript API [1, 2] on browsers like Chrome and Firefox provides a way to access the GPU for rendering and general computation workloads. WebGPU is the successor to WebGL, which has so far been the go-to for rendering needs.

However, WebGL does not support general computation workloads directly. It is also based on OpenGL, which is no longer under active development, in favour of the succeeding native GPU API Vulkan. Thus WebGL does not benefit from any new features or modern GPU capabilities. WebGPU, however, will have new features added to it going forward, alongside better GPU compatibility and faster operations.

However, this opens up a serious potential for misuse of native resources via the web. Thus it is crucial to implement WebGPU carefully so as to not allow malicious uses of it, or any possible breach of privacy. This is especially critical as implementations such as Google's Dawn are implemented in memory-unsafe languages [3], allowing for the typical pitfalls of undefined behaviour or memory misuse.

There are two new potential attack vectors [4] arising from WebGPU implementations:

- the WebGPU API implementation
- the internal WGSL shader compiler

WGSL compiler testing has been investigated in the past by "WGSLsmith: a Random Generator of WebGPU Shader Programs" [5] by Hasan Mohsin. This paper focuses not on WGSL, but on testing the WebGPU API by smart call sequence generation.

The field of fuzzing has proven itself effective at finding bugs in various tools and compilers [6]. Much of the research into fuzzing focuses on the specific application of compiler fuzzing, because of its effectiveness at finding bugs in these critical and complex pieces of software. Many of the techniques and concepts in fuzzing are useful in this paper's goal of testing the WebGU API.

1.1 Contributions

This project makes the following contributions:

1. *wg-fuzz*, a fuzzing tool that intensively tests the WebGPU API
2. 14 distinct bugs are found and reported, 6 of which cause a fatal crash, 4 of which have since been fixed
3. An analysis of *wg-fuzz*'s features following a fuzzing campaign of 1,100h compute-hours

This paper introduces *wg-fuzz*, an automated testing tool for the WebGPU API via smart call sequence generation. *wg-fuzz* works by generating a JavaScript program with a large number of random WebGPU calls that build on the previous calls. It automatically runs this against a locally compiled Dawn executable in order to test Chrome's implementation of WebGPU, Dawn.

There are roughly 102 different fields and methods of the WebGPU API to cover [2] and *wg-fuzz* covers all the 100 that running on Node allows for. Two methods depend on an HTML document image / video, which are unavailable on Node, which is the JavaScript engine *wg-fuzz* runs on. These API calls build up a resource tree in the JavaScript program with complex restrictions for API calls.

wg-fuzz implements and explores the effectiveness of adding a probabilistic 'override' variable to the API restrictions to possibly create calls that should get invalidated by the WebGPU implementation. *wg-fuzz* calls this 'fuzzy conditions'.

A testing campaign is run on some servers in parallel, with various sanitizers in order to find bugs in Dawn, Chrome's implementation of WebGPU [3]. 14 bugs are found and reported, 6 of which crash without even the need for sanitizers. Overall, Dawn was found to be more buggy than one would expect. It is possible that implementing it in a memory-safe language would solve some fraction of the bugs found. Many of the bugs have since been fixed.

2 | Fuzz Testing

2.1 Overview

This paper's goal is to test the WebGPU API. So this paper starts by studying methods that show program correctness. Figure 2.1 shows a high level overview of ways to show that a program is correct. It is possible to either formally verify that a program is correct [7], or to use any of a plethora of testing techniques to validate the program up to a certain extent.

Note that it is only possible to be sure that a program works as it should through the use of formal verification - testing techniques only show the presence of bugs for certain test cases, but trying all possible test cases is impossible, thus a program cannot be proven correct by testing it. However, formal verification has proven to be a hefty task even for subsets of certain programming languages [8]. This is because many programs, such as compilers, are complex and large systems, and it is hard enough to write them, let alone verify them.

Because formal verification is so hard, empirical testing techniques have been utilised and have proven time-effective and efficient at helping to find the bugs that really matter to software developers and users [9]. The standard approach is to write many test cases for your internal functions by hand, on a unit testing framework, possibly along with some simple integration tests. While this is effective, it is time consuming, and nonetheless prone to biases by the human writing them. A less widely known, but rapidly growing, testing technique is fuzzing.

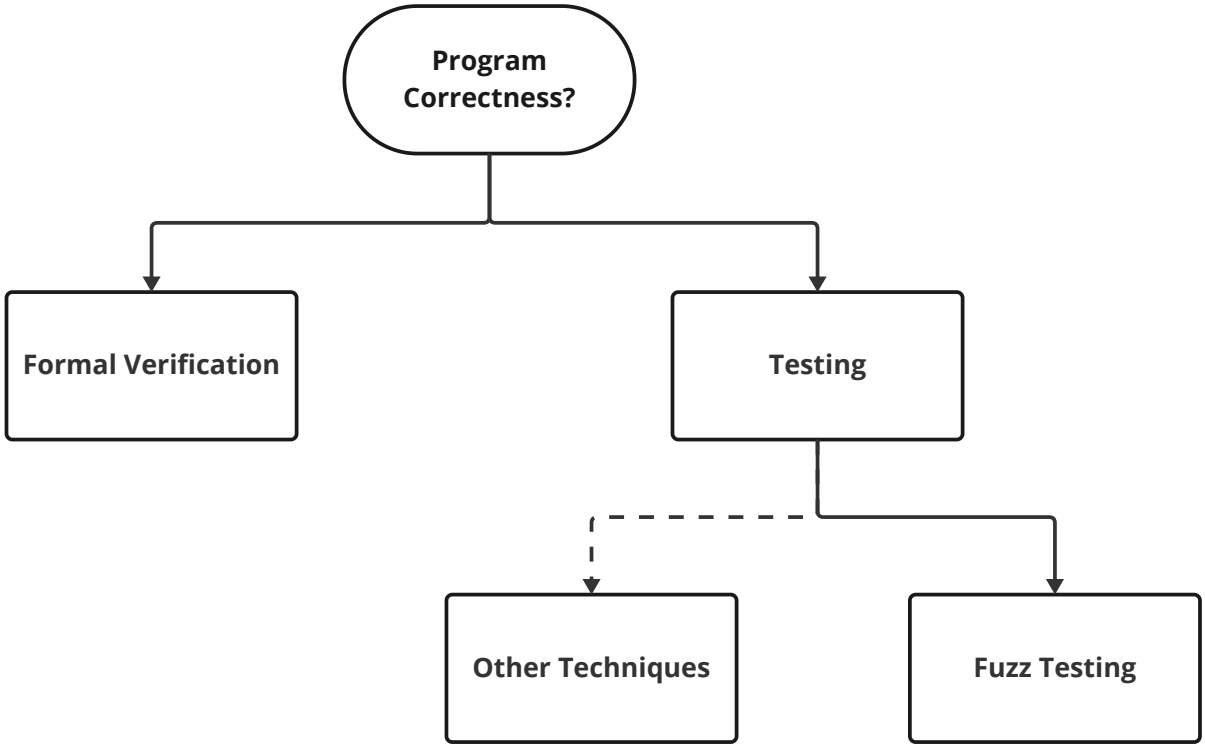

Figure 2.1: General overview of ways to check program correctness

2.1.1 Fuzz Testing

Fuzzing means providing random inputs to a system under test (SUT) in order to find bugs [10]. In many cases, the inputs provided are quite dumb, meaning they do not necessarily conform to the SUT's input grammar, and get rejected. This is not always a problem, as such inputs are rejected quickly in the parsing stage of the SUT, and fuzzers are typically run for a while. This also avoids any human error in modelling the input domain. The foundational fuzzing tool was a dumb fuzzer, and found many crashes in basic UNIX utilities [11, 12].

You may also try at providing strictly syntactically valid, random inputs to the SUT. This is called smart fuzzing [13], in contrast to dumb fuzzing. This is more efficient than dumb fuzzing provided that the fuzzer's generator implementation is correct. Fuzzers meant for testing specific tools or libraries usually opt for smart fuzzing.

In order to generate new random inputs, the fuzzer may opt to generate them from scratch, either dumbly or according to some input grammar, or by mutating existing known inputs from a corpus. As an example, two mutation approaches available to the AFL fuzzing tool are random bit flips, or splicing inputs together [14]. The Equivalence Modulo Input (EMI) technique [15] tries to mutate inputs according to some input grammar that should give the same output from the SUT. For example, if given a calculator app and that the result of sin(x) is already known, sin(π - x) should therefore give us the same output from the calculator, regardless of x.

Knowing how much of the code-base of the SUT has been covered (executed during at least one of the test inputs) is an important metric in fuzz testing. It allows the evaluation of the robustness of an SUT compared to how many bugs were found. Coverage information can also help us direct future test inputs. This process is called grey-box testing [16]. As an example, this is what AFL does when it adds an input to its corpus if it manages to cover a new path [14]. In contrast, black-box testing does not use any coverage information to direct future test inputs, and white-box testing is much more akin to formal verification techniques.

Lastly, there are many ways to check whether the SUT performs as expected, and this is called the 'Test Oracle Problem'. The simplest way is to check for crashes - i.e. feed random inputs into the program and monitor for crashes due to bugs in the program source code. However, there are other, more effective oracles.

The advantage of fuzzing is that it is done in an automated way so as to find as many bugs as possible. Figure 2.2 shows an overview of the entire fuzzing process. Relevant parts will be explored in the following chapters. Fuzzing solves the problems of human bias, and allows for great scalability without needing much more time invested in the future once it is set up.

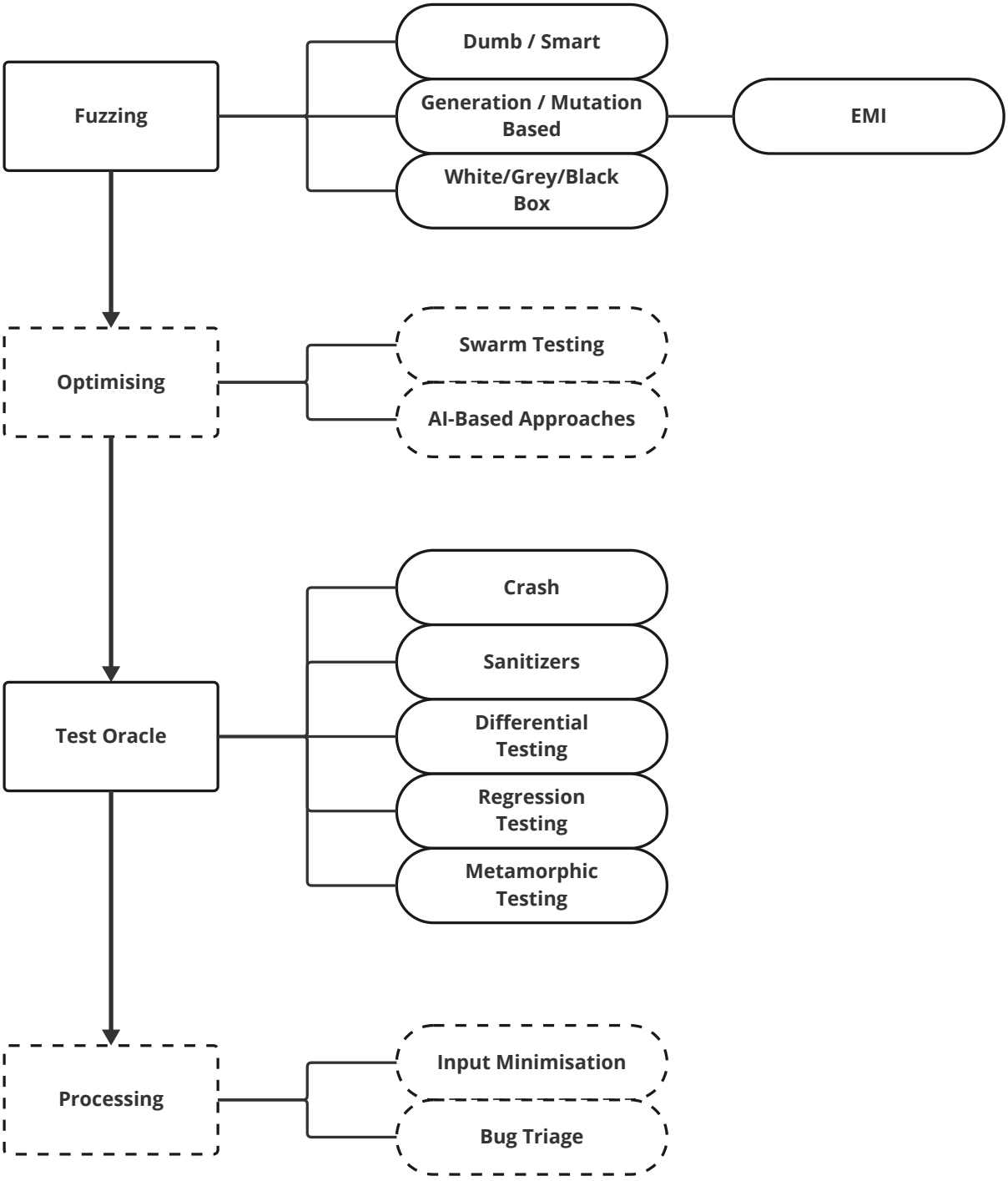

Figure 2.2: General overview of fuzz testing

2.2 Generation

A popular use-case for fuzzing is fuzzing compilers. In the case of compilers, their input is a program in the form of source code. This is not entirely unlike fuzzing library APIs. The goal of this paper is to create JavaScript programs, but with a specific restriction on using the WebGPU API in random ways.

Depending on what the objective is to test, varying degrees of correctness could be sought in the final produced program. For example, to test the JavaScript interpreter (in this case Node), a generator could opt for slightly invalid JavaScript syntax, or JavaScript semantics, to see if the oracle picks up something interesting. However, the focus of this paper is not on Node, but on WebGPU, so the goal of this paper is to produce fully valid JavaScript, both syntactically and semantically.

Running the WebGPU Conformance Test Suite on Dawn, it is trivially found that the methods and fields defined for the different resources in the API are as expected. So, it does not make sense to focus on syntax on the WebGPU API. The API provides various validation checks for various API calls in the specification. A simple example is `(GPUQueue).writeBuffer(...)` [17]. One validation restriction is "The buffer's `GPUBuffer.usage` includes the `GPUBufferUsage.COPY_DST` flag". It is worthwhile to test if any errors crop up as a result of messing with these internal validation checks. Hence, the focus is on fully valid JavaScript, semantically partially or completely valid WebGPU calls.

So, here are the ways the case of testing a low-level library API differ to compiler testing:

- The abstract program representation is flat as opposed to an abstract syntax tree
- There is a bigger focus on semantic restrictions than on complex syntax

However, an approach to generating the program must still be chosen. Generally, there are two approaches available: Generation, and Mutation-Based Fuzzing.

2.2.1 Generation Based Fuzzing

Generation Based Fuzzing means generating an input or syntax tree completely from scratch based on a specific grammar [18].

Grammar-based generation takes a grammar as an input and generates a random string following the rules of that grammar. Below is an example grammar:

```
<program> := 'begin' <stat> 'end'
<stat>    := <type> <ident> '=' <assign-rhs> | <stat> ';' <stat> | ...
<type>    := <base-type> | <array-type> | <pair-type>
...
```

The challenge is, when using a context-free grammar, to express context-sensitive features, e.g. declaring a variable at some point before using it.

Csmith is an example of one the foundational fuzzing tools that uses grammar-aided generation [19, 20]. Csmith's grammar is based on a subset of C. It works to obtain context-sensitive features by maintaining a *global environment* that holds top-level types, variables, and functions, and a *local environment* that contains the current call chain, effect information on objects that may have been read or written to, and a pointer table for data on the pointers and what they point to.

The global environment is extended as new types, variables, and functions are defined on the top level of the program during program generation. This, along with the local environment, is how Csmith solves context-sensitivity in grammar-based program generation [19].

To make a choice for its next grammar production, Csmith randomly selects an allowable production based on a probability table and filter function pair for that specific point in the program [19]. There is a probability table / filter function pair for statement, another pair for expressions, etc.

The probability table assigns a probability to each of the options [19]. The filter enforces C semantics such as a 'continue' only being available inside a loop, as well as limits imposed by the user on the maximum statement depth and number of functions. If the filter fails on the selected production, Csmith simply loops back until one succeeds.

If the selected production requires a specific target, like a variable or a function, then Csmith chooses one at random, or creates a new one on demand [19]. Similarly, Csmith chooses types at random, given the restriction (e.g. integer type). Along the way, it updates the local environment, and executes some safety checks.

If Csmith creates a call to a not-yet-defined function on demand, it suspends construction of the current function until the new function is generated [19]. When the top-level function is generated, Csmith is done. It then converts the internal representation to a string and exits.

2.2.2 Mutation Based Fuzzing

Mutation Based Fuzzing generates new inputs by modifying an existing corpus of inputs [21]. The initial corpus of inputs can either be provided by the user, or the fuzzer may initially generate the inputs from scratch. There are many approaches to mutation-based fuzzing.

AFL (American fuzzy lop) [22] is a prime example of a fuzzer that does mutation-based fuzzing well. It mutates inputs from a corpus without understanding the grammar necessary for a valid input. It instruments the system under test so that it knows when new ground is covered inside the SUT. In this way, its corpus is guided by the information it gets about program coverage. This approach has found bugs in many libraries.

AFL keeps track not of basic block coverage in the SUT, but rather of state transitions [14]. For example, state transitions `A -> B -> C` and `A -> C -> B` would be distinguished in AFL. The pairs `(A, B)`, `(B, C)` would be stored, and if AFL came across `(A, C)`, it would store the new input file for additional processing later on. If no new pairs are encountered, AFL will not store the input, even if its overall execution flow is unique.

If a mutated test case produces new state transitions, it is used as a starting point in future rounds of fuzzing [14]. These supplement the already existing corpus. Every now and then, AFL culls the corpus using some metrics that preserve non-redundant aspects of the test cases. It also utilises a trimmer that trims input file sizes by attempting to remove random blocks of data from the input and checking if the executed trace map checksum is the same.

The strategies used to mutate the inputs files in the corpus early in the process include the following [14]:

- Sequential bit flips with varying lengths and stepovers
- Sequential addition and subtraction of small integers
- Sequential insertion of known interesting integers (0, 1, `INT_MAX`, etc)

Later on, AFL turns to stacked bit flips, insertions, deletions, arithmetic operations, and splicing test cases together.

Other than AFL, there are also smart fuzzer based approaches. This includes parsing the input file, and performing either semantics-preserving transformations, or semantics-changing transformations [15].

For example, in a C input program, you could insert trivial no-op statements or e.g. add '`+ 0`' to integer expressions or '`&& true`' to boolean expressions. This method originates from the methodology *equivalence modulo inputs* (EMI) [15], and you expect the same output once run.

You could also mutate the program without preserving semantics, in order to generate new test cases for your oracle. This may be useful if you know your SUT's input grammar, and you want to utilise feedback from your fuzzer to inform future inputs.

2.3 Library API Fuzzing

As *wg-fuzz* fuzzes the WebGPU API, the current state-of-the-art in library API fuzzing is surveyed. Fuzzing many different library API functions gives substantially different problems than fuzzing a single tool, or a compiler.

2.3.1 GraphFuzz

The GraphFuzz paper goes into some detail on the problems inherent in Library API fuzzing [23]. Firstly, it recognises that model-based (input structure-aware) fuzzers are gaining popularity as unstructured fuzzers simply get stuck trying to brute force input after input (as fuzz targets generally send a random array of bytes as input parameters to the API regardless of any requirements imposed by the API).

Despite the practice of fuzzing gaining traction, the authors of GraphFuzz did not find many systems capable of fuzz testing C/C++ libraries. Existing fuzzers such as libFuzzer are well suited to fuzzing a small number of endpoints, but require significant manual effort to scale to many endpoints, such as in the case of libraries [23]. Additionally, fuzzing individual endpoints is insufficient to find all bugs in a library. Some bugs only arise out of the internal interaction from calling several endpoints in sequence.

GraphFuzz identifies four categories of API fuzzing [23]:

- Harness
- Code-gen
- Harness-gen
- Dynamic

The most crude method of API fuzzing is the Harness. This is when a standard unstructured grey-box fuzzer like libFuzzer is hooked up to a while loop and switch statement, choosing a random endpoint to test at each iteration [23]. However, as mentioned before, unstructured fuzzers are very inefficient at finding bugs in library APIs due to input restrictions and bugs in intricate endpoint interactions.

Code-gen is the method of generating and executing program source code that use the API [23]. This is most useful for script-based languages like JavaScript and Ruby, where compilation is not an expensive step in execution.

Harness-gen attempts to automate the process of writing fuzz targets for each API endpoint - i.e. the step of translating a random sequence of bytes to a valid input to the endpoint [23]. One way you might do this is by analysing existing codebases that use the API, or analysing logs of uses of the API.

Finally, the Dynamic approach augments the harness-gen method by additionally randomly choosing a sequence of API calls at fuzz-time, too [23]. Alternatively, you can think of it like Code-gen, but without the need to run it through an interpreter / compiler.

The GraphFuzz tool itself explores a model of data-flow graph-based mutations to a sequence of API calls, designed to fuzz C/C++ libraries [23]. It uses a configurable schema defined in the YAML format that contains a list of the API endpoints and object types available in a Library API.

2.3.2 RULF

RULF (RUst Library Fuzzing) [24] is a tool that attempts to solve the problem of lack of fuzz targets for Rust libraries. It does this by a novel API graph traversal approach. As different API calls produce and consume resources common between calls, it builds up a graph of these endpoints and resources, then tries to find a minimal covering set of sequential API calls. Each of these sequential API calls then form their own fuzz target which receives random input from a tool such as AFL++.

One example of what RULF might generate is given below. Notice how (b) and (c) overall cover every function defined in library (a).

```rust
struct S1;
struct S2;
fn f1(a: i16) -> S1;
fn f2(b: u32) -> S2;
fn f3(c: &[u8]) -> S2;
fn f4(s1: S1, s2: &mut S2) -> S2;
fn f5(s2: &S2, d: &str);
```

(a) A toy Rust crate `toylib`.

```rust
fn fuzz_target_1_for_toylib(data: &[u8]) {
    if data.len() < 3 {return;}
    let a = to_i16(data, 0);
    let c = to_slice::<u8>(data, 2, data.len());
    let s1 = toylib::f1(a);
    let mut s2 = toylib::f3(c);
    let _ = toylib::f4(s1, &mut s2);
}
```

(b) Fuzz target 1.

```rust
fn fuzz_target_2_for_toylib(data: &[u8]) {
    if data.len() < 5 {return;}
    let b = to_u32(data, 0);
    let d = to_str(data, 4, data.len());
    let s2 = toylib::f2(b);
    let _ = toylib::f5(&s2, d);
}
```

(c) Fuzz target 2.

Figure 2.3: Example RULF generated fuzz targets taken from [24]

This approach seems to work quite well for RUFL, however, it may be limited by the fact that it does not take requirements from specific API endpoints into account. For example, if the initial endpoint call takes a sequence of bytes, perhaps it is meant to encode an ASCII string. In this case, it is likely that most randomly generated inputs will have an invalid character, and thus be rejected. There is no guarantee that endpoints later in the sequence necessarily accept any resource produced by earlier endpoint calls, given it is at least the required type.

RULF's main advantage is that it tries to solve the problem of lack of fuzz targets for libraries in an automatic way, meaning it is general in its application to theoretically any Rust library crate. There are likely many library endpoints which do not have very strict requirements. RULF has been able to find many bugs in various Rust libraries because of this.

There may also be bugs that only occur in a certain sequence of API endpoint calls, even if there are two endpoints that produce the same resource. RULF possibly ignores these cases, and so is not completely general to all possible behaviours that can be exhibited by a user of the API.

2.3.3 DeepREL

DeepREL [25] attempts to help fuzz Deep Learning libraries, as bugs in these libraries can lead to serious consequences, potentially even threatening human lives. DeepREL operates on the basic assumption that a DL library may contain a number of endpoints sharing similar input parameters and outputs. As these endpoints are assumed to be potentially similar, DeepREL borrows inputs from one endpoint to test related endpoints. DeepREL is run on PyTorch and TensorFlow to find a number of bugs.

DeepREL is motivated by the fact that prior work has found a number of equivalent endpoints in older software systems (e.g. Java programs) [25]. It hypothesises that Deep Learning libraries may also have this property. So, if two endpoints should implement the same functionality, they should give the same output for any given input (value equivalence).

It also uses the fact that endpoints with vaguely similar functionality should give the same exit status (status equivalence). For example, the same data fed into `AdaptiveAvgPool3d` and `AdaptiveMaxPool3d` should give the same exit status. This can be used directly in differential testing between endpoints.

In order to obtain information on endpoint relations in an API, DeepREL reads in the API syntactically and finds all possible endpoint pairs based on that information. It then finds a set of sample valid inputs to each endpoint, traced from prior sample endpoint executions, and runs it against these possible endpoint pairs to check which relations hold [25].

Lastly, DeepREL uses mutation-based fuzzing to generate a diverse set of inputs, and runs them against the valid endpoint pairs to find potential bugs [25].

This approach is surprisingly effective on Deep Learning libraries, as many contain a number of endpoints that have similar interfaces. However, this is mostly limited to Deep Learning libraries, as this does not account for custom resources created between endpoints and the sequencing of endpoint calls.

2.4 Optimising the Testing Process

If your fuzzer is smart, i.e. it works based on a known input grammar of the SUT, it is usual to enable all possible features to be used in the generated input. For example, if you were testing a stack implementation, you would enable both "push" and "pop" calls to be made in all inputs.

In fact, if you do not implement swarm testing [26], it is common to try to find a single optimal configuration for the probabilities of different features in your generator, as well as other configuration options available for your tool. E.g. some research shows how to use genetic algorithms to find this optimal configuration [26].

However, this has the possibility of restricting certain bugs from being found. For example, if "push" and "pop" were both enabled in a stack implementation at equal probability, the likelihood of you finding a bug in the handling of a stack overflow at 100 items would be quite small. Conversely, if you optimise for more "push" operations, you may not find a bug if, say, a certain string of calls leads to a bug in the stack underflow code.

2.4.1 Swarm Testing

Swarm testing is a technique to improve the diversity of the test cases generated [26]. Wider diversity of test cases leads to better coverage and more bugs found.

In swarm testing, some features in the input generation are omitted in random configurations from run to run. For example, from the earlier example, omitting the "pop" operation would enable us to realistically find the bug in the stack overflow handling, as "push" is the only operation available while generating an input. Additionally, different features must compete for space in the input program, potentially limiting the depth that any one feature can be tested in a program [26].

Swarm testing appreciates that *possibility* is not the same as *probability*. Many times, it is desirable to reach an edge case in the input program, e.g. 128 calls to the same function overflows a resource and crashes the program. This is highly unlikely to happen if there are only have a limited amount of calls to make in a single test program, or it is highly order dependent.

There is a potential disadvantage to being unable to find certain bugs which require many, if not all, of the features to be available to trigger it. However, swarm testing does not restrict us to having a strict subset of the originally available features. It is possible to randomise the configurations in a way that it is also possible that all features are available in a given run. Swarm testing simply enables fuzzers to find more diverse programs.

2.5 Test Oracles

In automated software testing, the test oracle problem [27] is the problem of how to know whether your SUT behaved correctly given a certain test input. The most basic example of an oracle might be a human. In most cases, as a human has decided what "correct" means for a certain SUT, they should be able to tell whether the SUT behaved as expected given a certain input. However, this is not perfect, for example in the case where the input is so large that it is incomprehensible to the human, they will not be able to say whether the SUT acted as expected.

The simplest possible general and automated oracle may be the crash. Normally, no SUT is ever expected to crash. If the input is not valid according to the SUT's input grammar, the SUT is expected to print an appropriate error message, and possibly exit elegantly. If the SUT crashes, it means there is an urgent problem in its code. This is how the original fuzzing white-paper found bugs in basic UNIX utilities [11, 12].

One more possible simple oracle is to instrument your SUT with assertion statements on what state you expect your program to be in at each stage. This is an excellent halfway between human intention and automated testing, as assertions provide a general way to ensure certain assumptions are true throughout your program's lifetime, while allowing you to test your program in a highly scalable way [28].

2.5.1 Sanitizers

A sanitizer helps to detect bugs in your SUT in various forms of suspicious behaviour. For example, some sanitizers available on the Clang C compiler are:

- AddressSanitizer
- MemorySanitizer
- ThreadSanitizer
- UndefinedBehaviourSanitizer

AddressSanitizer (ASan) asserts that no behaviour such as out-of-bounds accesses to heap/stack/globals, use-after-free, use-after-return, use-after-scope, double-free, invalid free, or memory leaks happen in your SUT.

MemorySanitizer (MSan) asserts that no uninitialised reads occur in your SUT.

ThreadSanitizer (TSan) asserts that there are no data races in your SUT.

UBSan (UBSan) asserts that none of the various forms of C undefined behaviour occur in your SUT.

Compiling your SUT with sanitizers can help you to lower the threshold for when the oracle flags something as a bug, and help you find more bugs in your system through the fuzzing process.

2.6 Processing

A stage that is just as important in fuzz testing as the others is the post-processing of potential bugs found. In the end, any potential bugs found are pointless if it is unclear where the problem occurs, or if it is not fixed.

2.6.1 Input Minimisation

During the process of fuzz testing, the created programs are usually very large and highly obscure. It may be hard to tell which part of the input is causing the issue. This sort of input test case is useless to the tester - a minimal form that still reproduces the same bug is desirable.

To this end, it is possible to run an automated test case reducer. The process essentially repeatedly removes random parts of the input and checks if the same bug is still hit through an "interestingness test".

Delta Debugging is a technique that works towards this end. It uses an extended binary search algorithm to reduce a failing-inducing input to its minimal form, in computationally quick time.

John Regehr et al. introduce three approaches to reducing bug-inducing C source files. The first two instrument Csmith, a random C program generator.

The first approach is using their tool *Seq-Reduce*, which integrates into Csmith. Seq-Reduce first tells Csmith to dump the representative random number sequence of the generated C program. It then modifies this randomly, then runs it through Csmith again and checks if it produces the same bug. This is a simple implementation that is random and embarrassingly parallel, but has trouble with reducing cases that come up late in Csmith's execution, and has no obvious termination condition.

The second approach uses *Fast-Reduce*, which also integrates directly into Csmith. It performs a number of different transformations while Csmith runs. Some representative example include: Dead-code elimination, testing for wrong path divergence, and in-lining the effect of a function.

The last and most used tool is *C-Reduce*. C-Reduce is a general reducer able to work source-to-source without any need for Csmith. It essentially implements 50+ general transformations, and iterates through them until no more work, while seeing which ones successfully keep the bug. Some examples of transformations include changing integers to 0 or 1, removing braces, removing unused functions, etc.

2.6.2 Bug Triage

Another important aspect of bug processing is bug triage. This is the process of trying to group failure-inducing bugs by their root cause, so as to not have redundant bug reports. It is also useful to flag them according to their perceived severity / importance. In some cases, developers do not care about fuzzer-found bugs as they only cause an unimportant issue given an exceedingly unlikely user input.

3 | WebGPU

3.1 Introduction

WebGPU is a public standard for a JavaScript API that provides access to the GPU on the web for rendering and general computation. Its first public working draft was released on 18 May 2021, so it is still very new, and different browser vendors are still in the experimental stage of implementing it.

WebGPU is the successor to WebGL. WebGL is a port of the native GPU API OpenGL, allowing web-pages to render to a canvas very efficiently. It has been the go-to for intensive rendering needs on the web so far. However, there are several issues with it.

Since WebGL's debut, several new native GPU APIs have been released. Namely, Microsoft's Direct3D, Apple's Metal, and The Khronos Group's Vulkan (used in Linux). These are continuously updated with new features, and keep up with the latest features available in modern GPU hardware. OpenGL, on the other hand, is no longer in active development, in favour of the new Vulkan API, since 2017. Hence, WebGL does not receive any more updates for the latest features available in modern GPU hardware.

Additionally, WebGL is based solely around the rendering of graphics to a HTML canvas. It does not have first-class support for general-purpose GPU computations.

As 3D graphics web applications continue to become more elaborate and demanding in terms of the features, and due to the AI boom in recent years, general computation workloads have begun to have significant importance.

WebGPU solves these issues with a new and more general-purpose architecture allowing for the use of GPU APIs like Direct3D, Metal, and Vulkan. It is cheaper on the CPU side, and supports new rendering features like depth-of-field simulation, and it can handle expensive workloads like culling and skinned model transformation on the GPU. It also provides first-class support for general-purpose GPU workloads through its API.

3.2 API

3.2.1 General Model

It makes sense to begin with a general overview of the WebGPU API. There are several layers of abstraction in its implementation.

At the lowest level, you have a physical GPU on your computer. This GPU is accessible via a low-level driver that it provides. However, there are many different drivers for different GPU architectures, and it is unrealistic for applications to write calls intended for each possible driver. For this reason, cross-platform GPU APIs like Vulkan exist. There are different possible cross-platform GPU APIs that might be available on a given system - Direct3D, Metal, and Vulkan being the most prominent examples.

The entry-point for WebGPU is to request an adapter. You can request a "low-power", "high-performance", or unspecified adapter. The adapter effectively represents the computer's underlying GPU.

Finally, you can request a logical device from your adapter. This abstraction is intended to provide compartmentalised access to the GPU for each web-app. As a physical GPU may be used by many web-app concurrently, each web-app should access the GPU in a sand-boxed way for security and ease of programming.

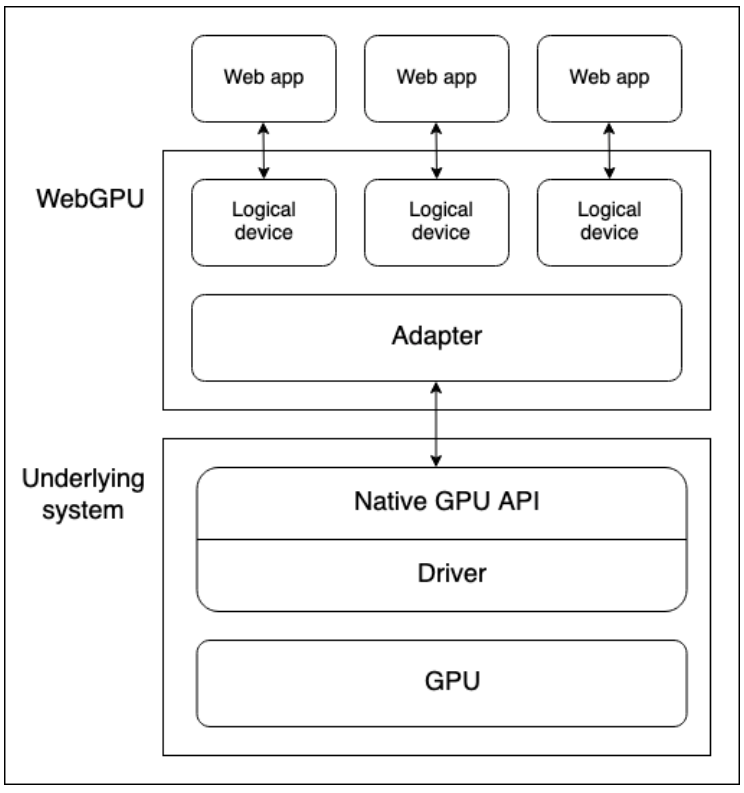

Figure 3.1: General Architecture of WebGPU (Credit to MDN)

3.2.2 Complexity

The concrete API is exceedingly large and complex, since GPU rendering is exceedingly complex. There are almost 100 different API calls available in WebGPU.

3.2.3 Entrypoint

First, you start off by accessing a device as explained in the General Model subsection. This is fairly straightforward, and can be done like so:

```
async function init() {
  if (!navigator.gpu) {
    throw Error("WebGPU not supported.");
  }

  const adapter = await navigator.gpu.requestAdapter();
  if (!adapter) {
    throw Error("Couldn't request WebGPU adapter.");
  }

  const device = await adapter.requestDevice();

  //...
}
```

Note that credit for this and the following code-blocks goes to MDN. As the code is straight-forward, they have been re-purposed in explaining the API.

3.2.4 Pipelines

In order to run something on the GPU, you need to prepare either a render pipeline, intended to render some graphics to a canvas (or more generally a texture object), or a compute pipeline, intended for general purpose GPU workloads.

Compute pipelines are simpler than render pipelines, simply taking in some data, running some user-defined shader code in parallel across a specified number of workgroups, and returning the result in a buffer.

Render pipelines have two stages: a vertex stage, and a fragment stage. In the vertex stage, the shader code takes in some positioning data fed in through a buffer, and uses it to position a series of vertices in 3D space by applying effects like translation, rotation, and perspective. The vertices are assembled into triangles, a basic shape in rendered graphics, followed by a rasterization phase, where the GPU decides which pixels are covered by which triangle surfaces.

In the fragment stage, a fragment shader computes the colour of each pixel from the vertex stage. Usually, this involves images in the form of textures. These essentially provide the surface details of the objects being rendered. It also computes the position and colour of virtual lights.

3.2.5 Setting up Resources

In order to run a workload, there is a fair amount of set up required by the API. An overview of the setup required for a render workload is explained as it is more complex than a compute workload, and so a compute workload can then be understood relatively simply.

To start off, you are required to provide the shader code that you would like to run. WebGPU implements its own Rust-like shader language called WGSL (WebGPU Shader Language). This is provided as a string to the WebGPU API. Below is an example render shader:

```
const shaders = `
struct VertexOut {
  @builtin(position) position : vec4f,
  @location(0) color : vec4f
}

@vertex
fn vertex_main(@location(0) position: vec4f,
               @location(1) color: vec4f) -> VertexOut
{
  var output : VertexOut;
  output.position = position;
  output.color = color;
  return output;
}

@fragment
fn fragment_main(fragData: VertexOut) -> @location(0) vec4f
{
  return fragData.color;
}
`;
```

You load it into a WebGPU shader module like so:

```
const shaderModule = device.createShaderModule({
  code: shaders,
});
```

Next, you decide where you want the render output to go. The destination is usually to some kind of texture. The most common texture to output to is an HTML canvas texture - i.e. to output the render result on the screen. If you want to output to an HTML canvas, you must specially configure it with information on the WebGPU device object that the information will come from, along with the format the textures will have and the alpha mode to use for semi-transparent textures. Below is an example snippet:

```
const canvas = document.querySelector("#gpuCanvas");
const context = canvas.getContext("webgpu");

context.configure({
  device: device,
  format: navigator.gpu.getPreferredCanvasFormat(),
  alphaMode: "premultiplied",
});
```

Next up, some input data is required. For the rendering workload, the goal is to render a single triangle with different colours at each of its vertices. A self-defined format is used for the position and colour data that will work with the earlier shader code. A JavaScript Float32Array is created with 8 floats per triangle vertex - X, Y, Z, and W for position, and R, G, B, and A for colour. A triangle has 3 vertices, hence 24 floats overall:

```
const vertices = new Float32Array([
  0.0, 0.6, 0, 1, 1, 0, 0, 1,
  -0.5, -0.6, 0, 1, 0, 1, 0, 1,
  0.5, -0.6, 0, 1, 0, 0, 1, 1,
]);
```

However, this data is currently inaccessible to the GPU. The GPU uses its own local buffers for its high-speed processing, and they are not readily accessible in the computer's current environment. A buffer must be created via a WebGPU call and specially written to. Size and usage flags are specified on creation:

```
const vertexBuffer = device.createBuffer({
  size: vertices.byteLength, // make it big enough to store vertices in
  usage: GPUBufferUsage.VERTEX | GPUBufferUsage.COPY_DST,
});
```

The data is then written into it, like so:

```
device.queue.writeBuffer(vertexBuffer, 0, vertices, 0, vertices.length);
```

3.2.6 Setting up a Pipeline

Next, all of these resources are put together by defining a render pipeline. Firstly, a descriptor object must be created explaining how the vertex and colour data is laid out in the buffer:

```
const vertexBuffers = [
  {
    attributes: [
      {
        shaderLocation: 0, // position
        offset: 0,
        format: "float32x4",
      },
      {
        shaderLocation: 1, // color
        offset: 16,
        format: "float32x4",
      },
    ],
    arrayStride: 32,
    stepMode: "vertex",
  },
];
```

Now, a pipeline descriptor object is created. For both the vertex and fragment stages, the shader module that the shaders can be found in is specified, along with the name of the function to invoke. In the case of the fragment stage, the vertexBuffers object is provided, explaining the layout of the data in the buffer. In the case of the fragment stage, the rendering format is provided.

The type of primitive that the render pipeline will be drawing is also specified, in this case triangles, and additionally a layout. The layout provides information on all of the resources (buffers, textures, etc) used in the shaders. Specifying this information is optional in order to allow the GPU to figure out the optimal way to run the pipeline ahead of time, but the value “auto” can be used to let it figure it out from the shader code.

```
const pipelineDescriptor = {
  vertex: {
    module: shaderModule,
    entryPoint: "vertex_main",
    buffers: vertexBuffers,
  },
  fragment: {
    module: shaderModule,
    entryPoint: "fragment_main",
    targets: [
      {
        format: navigator.gpu.getPreferredCanvasFormat(),
      },
    ],
  },
  primitive: {
    topology: "triangle-list",
  },
  layout: "auto",
};
```

Finally, the render pipeline object is created like so:

```
const renderPipeline = device.createRenderPipeline(pipelineDescriptor);
```

3.2.7 Running a Rendering Pass

Lastly, a command encoder is required to encapsulate the commands to run on the GPU. First, you create one:

```
const commandEncoder = device.createCommandEncoder();
```

The goal is to run a rendering pass, so a render pass is begun on the command encoder. The only mandatory field for beginning a render pass is the colorAttachments field specifying the texture view to render to, how to load the view (i.e. clear it using a certain colour first), and the store operation for the render pass (store it):

```
const clearColor = { r: 0.0, g: 0.5, b: 1.0, a: 1.0 };

const renderPassDescriptor = {
  colorAttachments: [
    {
      clearValue: clearColor,
      loadOp: "clear",
      storeOp: "store",
      view: context.getCurrentTexture().createView(),
    },
  ],
};

const passEncoder = commandEncoder.beginRenderPass(renderPassDescriptor);
```

Now the earlier defined pipeline and buffer can be encoded in this render pass. As there is data for 3 vertices in the buffer, the render pass is set to draw 3 vertices:

```
passEncoder.setPipeline(renderPipeline);
passEncoder.setVertexBuffer(0, vertexBuffer);
passEncoder.draw(3);
```

Finally, the command encoder is signalled that the render pass is done, and subsequently also that the command encoder is done. The command encoder becomes a command buffer, which can be sent to the logical device to be run:

```
passEncoder.end();

device.queue.submit([commandEncoder.finish()]);
```

3.3 Dawn

While the WebGPU specification exists, browser vendors have the choice of implementing it into their own browsers, or not. Google has already opened some support for WebGPU into their Chrome browser since version 113.

However, they are still actively developing their implementation, with not all extension features laid out in the specification supported yet. Additionally, the WebGPU specification is still in flux, so it cannot be relied upon as a completely stable API yet.

3.3.1 Security Concerns

Bugs in the WebGPU implementation are particularly serious, as WebGPU exposes a completely new attack surface that may be abused by malicious parties.

Particularly, Google's Dawn is implemented in C/C++, which are memory-unsafe languages, making it especially useful to bypassing security boundaries through undefined behaviour and memory misuse.

WebGPU exposes two different attack surfaces:

1. The WebGPU API
2. The WGSL shader compiler

This paper focuses on the WebGPU API specifically. There are several classes of security bugs in particular to consider.

One example is CPU-based undefined behaviour. Typically, the GPU APIs available on a system do not guarantee any outcome if its validation rules are not followed. This undefined behaviour must be avoided as it can be exploited by an attacker to breach memory boundaries, or execute malicious code. The WebGPU API thus defines some validation rules on its endpoints. For example, its `copyBufferToBuffer()` call cannot hold the same buffer in its source and destination, as this may lead to undefined behaviour in the GPU API call. These validation rules must be implemented without any bugs.

Furthermore, WebGPU plans to later support multi-threaded use via Web Workers. While it is designed not to open users to timing attacks, some objects, like GPUBuffer, have shared state that can be simultaneously accessed, which may allow race conditions.

There are also privacy concerns. Fingerprinting is a major concern on the web, as it can be used to track users across websites. WebGPU must expose a lot of detail about the machine's GPU and its capabilities out of necessity.

WebGPU tries to mitigate any fingerprinting that is possible from this by imposing default GPU limits that are accessible to most users, yet provide enough power for most applications. There are also some differences in computational artefacts that can be observed across different GPUs. However, these are mostly identical for the same GPU vendor.

4 | Design of *wg-fuzz*

wg-fuzz is a tool designed to test any given implementation of WebGPU, given it is possible to run on Node. It does this through a process of fuzzing, or generating random JavaScript files that call into the WebGPU API in random ways.

Its design is such that it is more flexible in terms of the sequences of API calls that it is able to test compared to some of the state-of-the-art library API fuzzers that are studied in this paper. However, it is also restricted enough in what it generates in order to test the relevant parts of the Dawn implementation, and not, say, focusing too much on testing the basic WebGPU validation code for the API call parameters by being too general in what is allowed, but also including interactions between multiple validated WebGPU resources.

The trade-off for these benefits is the need for much work in order to set up a fuzzer that is specific to WebGPU's almost 100 different available API calls, and their interactions. This work has been provided in the implementation of *wg-fuzz*, with an open-source codebase that should be clear enough to be easily maintainable in the future should the WebGPU specification update.

There are two main resources that are built up over the course of generating a WebGPU program:

- An Abstract Call Vector
- An Abstract Resource Tree

The Abstract Call Vector effectively represents a WebGPU program in the internal state of *wg-fuzz*, before it is converted to JavaScript. This allows the generated program to be easily manipulated inside the tool. For example, to write a custom reducer inside *wg-fuzz*, it could be achieved more easily thanks to the ACV (Abstract Call Vector).

The Abstract Resource Tree is also a hugely critical part of the generation, building up an internal record of the WebGPU resources that are available at any given point, as well as their state. For example, what flags a buffer has, or whether it has been destroyed, allowing us to know, for example, whether it would be a valid operation to copy data into it.

Given these two types of resource, *wg-fuzz* has several components in its WebGPU program generator:

- The Available Calls Driver
- Resource Simulation
- Conversion to Source Code

The Available Calls Driver is the most complex part of the generation, climbing through the Abstract Resource Tree, and checking which from the roughly 100 API calls are available given the internal state of each resource. It is also subject to swarm testing in order to keep the calls chosen varied, and *wg-fuzz*'s self-defined *fuzzy conditions*, allowing possible breaches of WebGPU validation rules with a given probability. It returns a vector of available API calls, from which one is chosen at random.

In order to help the Available Calls Driver know what API calls are available in the future, and to help keep the programs generated interesting, *wg-fuzz* uses Resource Simulation. *wg-fuzz* updates the Abstract Resource Tree appropriately given the API call that it end up choosing.

Finally, *wg-fuzz* converts the Abstract Call Vector to source code. This is also the stage where it outputs WGSL shader code files to be read by the WebGPU JavaScript program. *wg-fuzz* utilises WGSLsmith to write randomised WGSL compute shaders (it does not support render shaders), and interface with those shaders accordingly. To limit the scope of this project, *wg-fuzz* uses representative render shader code. This is also the stage where *wg-fuzz* might generate various random arguments to the calls that do not necessarily affect the Abstract Resource Tree in an important way, like the data in a JavaScript array.

wg-fuzz is written in pure Rust apart from where JavaScript / WGSL snippets are used, or e.g. a BASH interestingness test for reducers. The memory safety, yet fine control provided by Rust has made writing *wg-fuzz* easier and more robust overall. Its enum and struct types give the flexibility needed for the Abstract Resource Tree. As a bonus, it integrates with WGSLsmith easily, as it is also written in Rust.

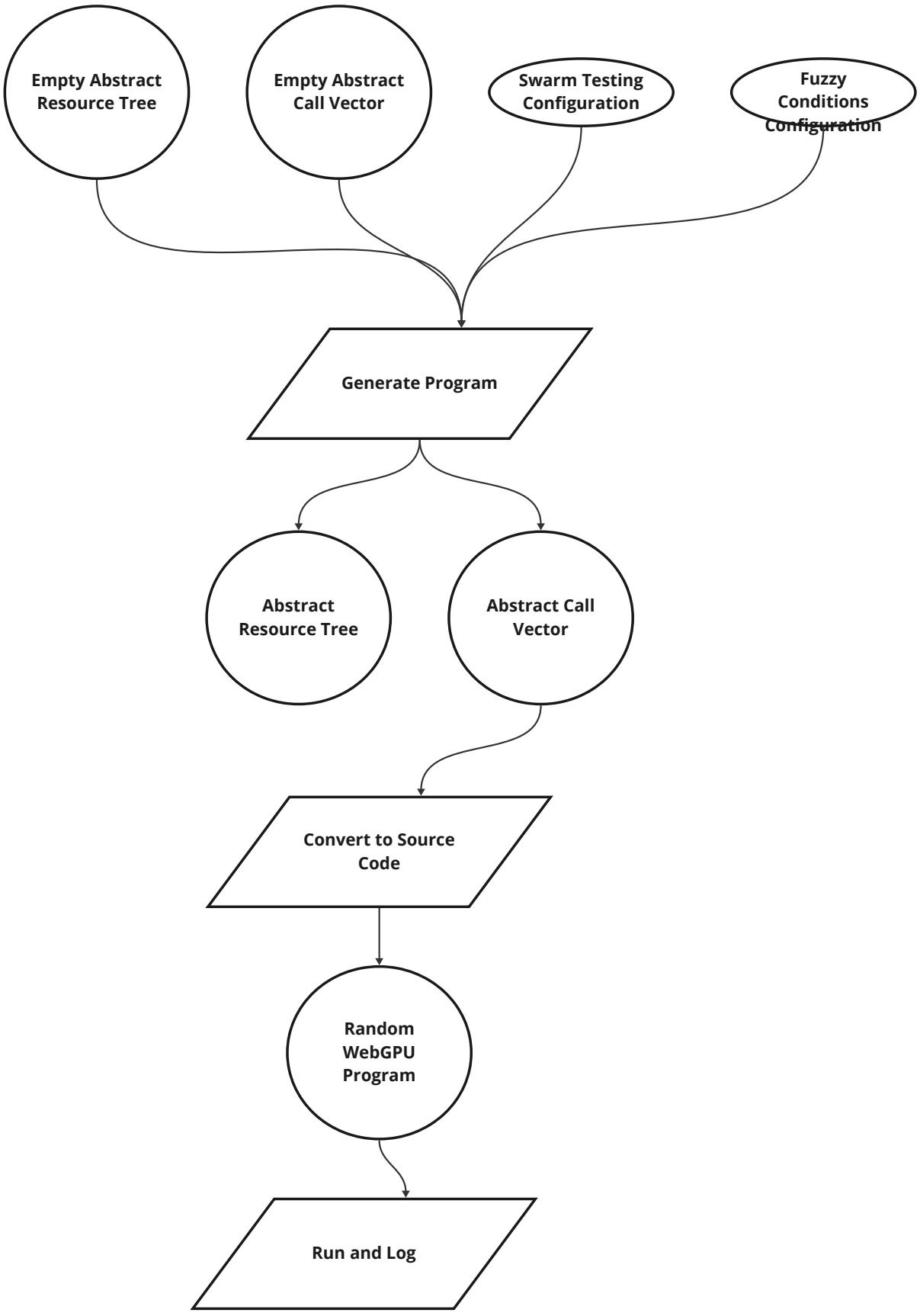

Figure 4.1: General Architecture of *wg-fuzz*

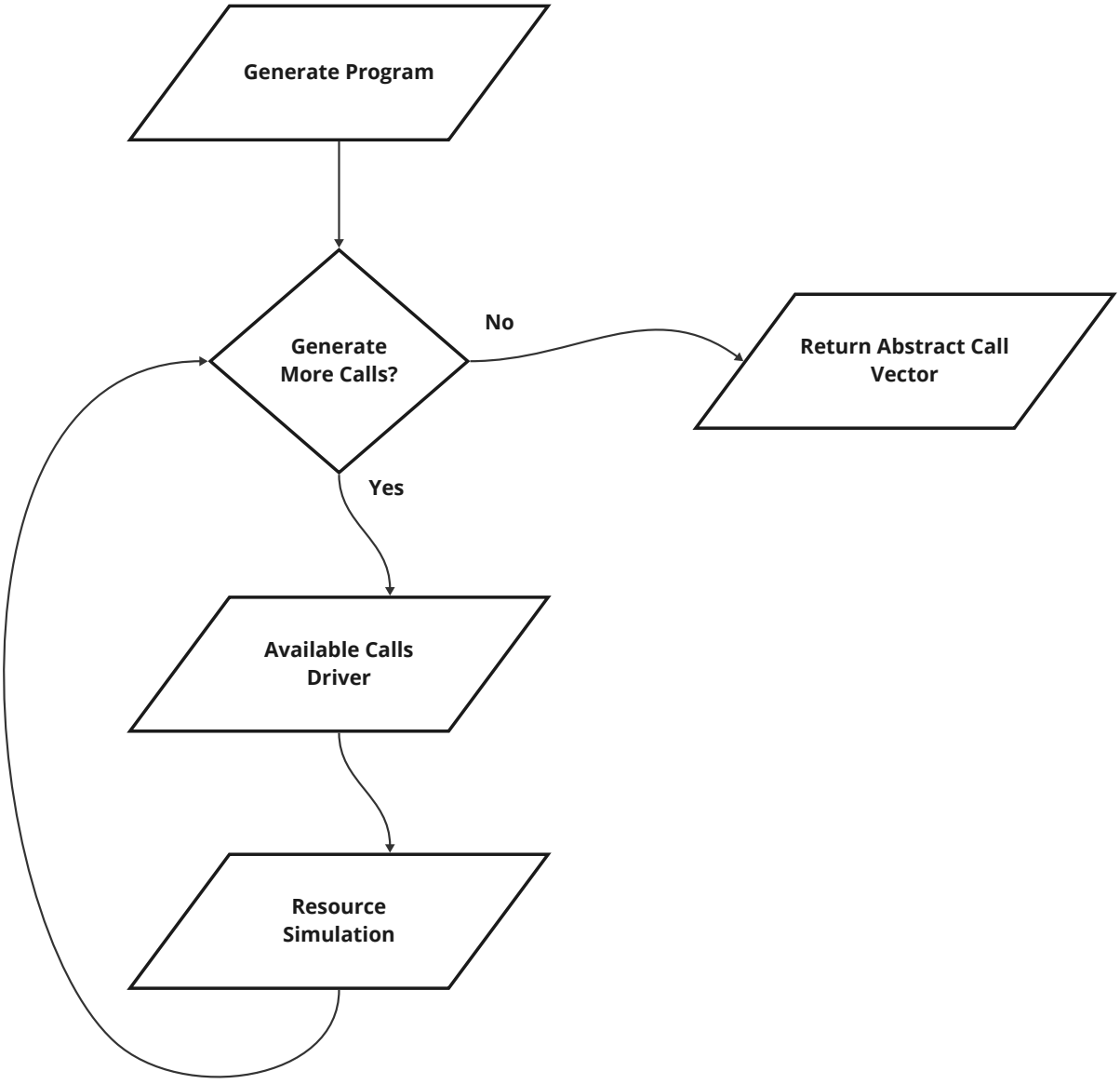

Figure 4.2: Generation Algorithm

4.1 Approach

Recall the problems inherent in library API fuzzing:

- Each endpoint has a specific interface and validation rules, requiring work to translate a random array of bytes to an interesting argument in a fuzz target
- It is desirable to test the interaction of different API calls in sequence

The Harness and Harness-gen methods that were studied earlier in this paper have the limitation of not generalising to testing sequences of different calls to the API. Instead, they test each endpoint individually. This has the potential of missing out on bugs that only appear when API calls are used in a certain sequence with certain arguments.

The Dynamic approach solves this problem by additionally encoding sequences of API calls in its random sequence of bytes. However, in this case, implementations of WebGPU such as Dawn are very large and complex projects, and it would require significant development effort in order to interface and compile directly to the WebGPU endpoints in the codebase. Additionally, this makes *wg-fuzz* less general, as it relies on interfacing with a very specific code-base / implementation of Dawn.

wg-fuzz uses the Code-gen approach, as this allows it to cover the most general search space for finding bugs by allowing randomised sequences of API calls, and allowing it to integrate with any WebGPU implementation with minimal effort, provided it supports running on Node. This also opens up the possibility of differential testing across WebGPU implementations, though effort has been focused on reporting bugs to the Chrome developers.

wg-fuzz enjoys these benefits in return for significant development effort, and requiring the tool to specialise specifically in testing the WebGPU API. On the other hand, it is possible that some development effort may have been saved by, instead of implementing the validation rules for each of the $\tilde{1}00$ API endpoints separately for separate fuzz targets, *wg-fuzz* is able to link all of these rules in one complex algorithm in the Available Calls Driver, possibly saving some setup redundancies.

In comparison to library APIs for applications such as Deep Learning, the WebGPU API is much more varied in its interfaces, and contains many more validation rules for arguments. This has the implication that fuzzing with conventional grey-box tools like libFuzzer may be inefficient as most inputs will simply be rejected by the many validation rules for one endpoint, let alone allowing for the testing of interactions between multiple validated API calls. Because of these complex validation rules and the many types of resources returned by the API, in order to do any effective testing of the API, *wg-fuzz* requires implementations such as the Abstract Resource Tree and Available Calls Driver.

4.2 Abstract Resource Tree

The crux of the Abstract Resource Tree is to simulate the resources available in the WebGPU program as *wg-fuzz* goes along, so it can feed this information to its Available Calls Driver. For the most part this resembles the hypothetical resource tree available in the WebGPU API, but some nodes may be emitted if they are not needed in the Available Calls Driver.

The Abstract Resource Tree must store not only the objects available, but their state and current configuration, too, insofar as it is useful for the Available Calls driver. For example, a buffer has a field that stores its usage flags that the WebGPU API takes on the creation of a buffer. This is useful so *wg-fuzz* knows what operations are permissible according to the WebGPU specification (copying data to the buffer, destroying it, etc.). Consequently, on the buffer struct in the Abstract Resource Tree implementation, in the call to create a new buffer node, the constructor takes the buffer's usage as an argument.

Inside the various implementations of WebGPU, it is possible that they implement a similar resource tree. In the case they do, if one had knowledge of the entire code-base, it would hypothetically be possible to write a fuzzer similar in structure to *wg-fuzz* alongside the actual implementation of WebGPU. In this way, one would be able to cut out the redundancy of re-writing the abstract syntax tree of WebGPU and have the Available Calls Driver look at the internal implementation directly.

However, in this case, the implementations are exceedingly complicated, with a large code-base maintained by a large number of people from large companies such as Google and Mozilla, with documentation that is not always fully available, in order to enable rapid internal development. In cases such as this, to test a highly complex and restrictive API, an effective fuzzer like *wg-fuzz* without the redundant Abstract Resource Tree would likely only be possible by people in the internal development team of the API.

Additionally, re-implementing the Abstract Resource Tree ourselves enables us to test various implementations of a specification, and does couple us with any particular implementation.

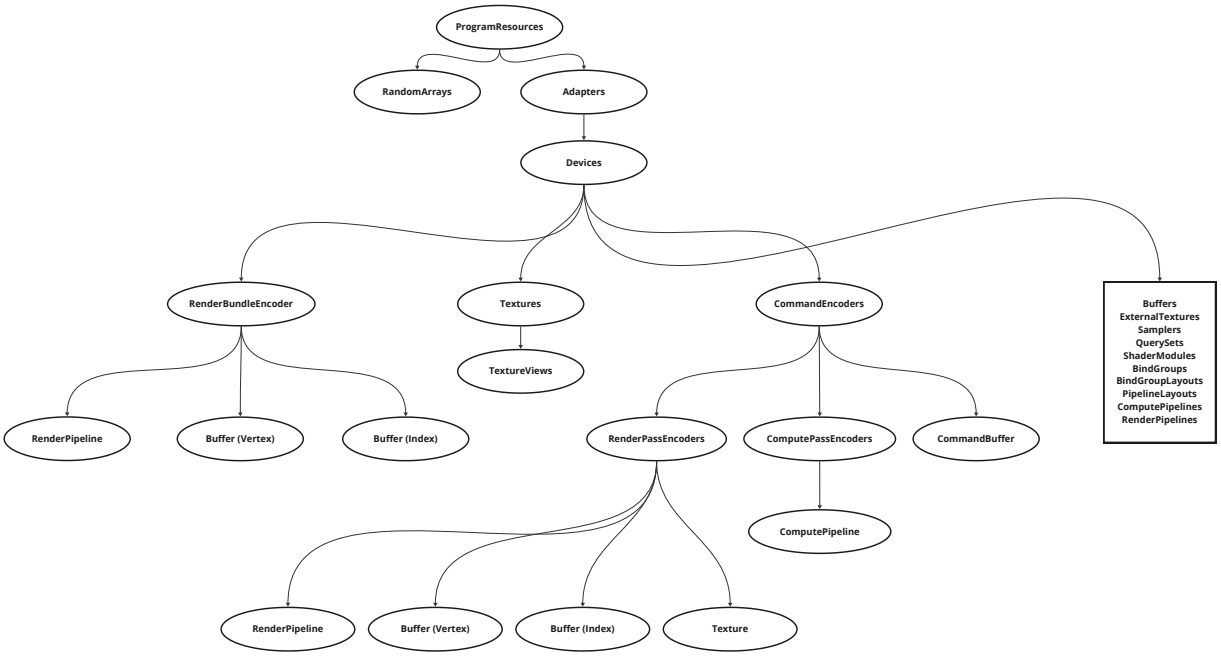

Figure 4.3: Overview of Abstract Resource Tree (additional leaf nodes on rightmost box to save space)

4.3 Abstract Call Vector

The purpose of the Abstract Call Vector is to encode the generated WebGPU program internally. As its name suggests, it is a vector, and it is comprised of API call enum variants. The API call enum does not correlate to the WebGPU API one-to-one, though it is close.

In particular, this enables us to easily write *wg-fuzz*'s own reducing algorithm, if necessary, through transformations such as checking for redundant calls, or removing enum variants purely for print statements.

Custom enum variants are added for purposes like printing out a resource's fields, with some enum variants possibly encoding several WebGPU calls. For example, the `ReadMappedBuffer()` variant uses WebGPU calls `mapAsync()`, `getMappedRange()`, `slice()`, and `unmap()`, as these are all quite low level and are usually used together. For the most part, one enum variant encodes one WebGPU API call.

wg-fuzz currently supports 90 different enum variants. Each possible WebGPU API call is available for *wg-fuzz* to make. Some example enum variants include `CreateAdapter()`, `CopyTextureToBuffer(GPUCommandEncoder, GPUTexture, GPUBuffer)`, and `WaitSubmittedWork(GPUDevice)`.

A limitation of *wg-fuzz* is that, while it covers each possible WebGPU call at least once, it does not necessarily support every possible optional argument in each WebGPU call.

You may also note that the conversion to JavaScript source code is defined on the Abstract Call Vector enums.

4.4 Available Calls Driver

The Available Calls Driver is the main and most complex part of *wg-fuzz*. It is what decides which API calls are available at the current point in the program generation by climbing through the current Abstract Resource Tree.

It is particularly important to get this right, as making it too permissive would mean *wg-fuzz* only gets basic validation errors from the WebGPU API, it is possible that no bugs are ever found this way as no valid WebGPU programs are ever truly tested. Conversely, if *wg-fuzz* is too strictly bound to the most valid of WebGPU programs, it may not find bugs that crop up with partially incorrect WebGPU programs.

From this, the goal is to write an Available Calls Driver that returns valid WebGPU API calls by default. In principle, at its base implementation, it should never suggest an invalid WebGPU API call, as this extreme is more useful than the converse, which would make completely random and probably invalid API calls and probably only test the basic front-leaning validation code in WebGPU.

It is desirable to insert invalid uses of the API only here and there. So, *wg-fuzz* introduces fuzzy conditions. *wg-fuzz* uses fuzzy conditions in its Available Calls Driver code that determines which WebGPU API calls are available to be used next. This enables *wg-fuzz* to make invalid API calls sometimes.

When the Available Calls Driver returns a vector of available API calls, the API call that *wg-fuzz*'s generator uses is chosen at random on a uniform distribution. This call is passed to the Resource Simulation part of the code, and added to the Abstract Call Vector. *wg-fuzz* then iterates to find another API call. A slightly abstracted version of the algorithm used is given below:

```
for _ in 0..num_calls {
    let mut available_api_calls = available_api_calls(resources);
    let api_call = available_api_calls.pick_random();
    update_program_resources(resources, &api_call);
    program.calls.push(api_call);
}
```

wg-fuzz also optionally accepts a swarm testing configuration, and a fuzzy condition configuration to the Available Calls Driver when it is called. If these are at their default value, it should simulate the algorithm as if swarm testing and fuzzy conditions are not implemented at all.

4.4.1 Number of API Calls

One quite important hyper-parameter and approach to consider is the number of API calls that *wg-fuzz* makes in its randomly generated WebGPU program. In other words, how large the resultant WebGPU program is.

Because of how general *wg-fuzz*'s generated WebGPU programs are, and the WebGPU API's validation restrictions, it may take quite a few calls before, say, *wg-fuzz* randomly creates a buffer that has a flag set for writing into it, and additionally randomly creates a buffer that has a flag set for reading from it. Perhaps a certain value is required to be set up in the read buffer, and when this value is present and the program tries to copy the contents of the read buffer into the write buffer, a bug is hit. The actual WebGPU setup before running a real GPU workload takes a combination of many specific API calls, too. *wg-fuzz*'s generator needs to be set up to write a large enough number of API calls in its resultant WebGPU program to be able to hit these cases.

However, if the programs are too large, there is a risk of overshooting how many API calls are truly required before hitting a bug, and seriously slowing down the bug finding process at generation. Indeed, during the later bug finding campaign, most bugs were found from relatively small WebGPU programs, once reduced.

As such, the limit in *wg-fuzz* is set at 1000 API calls, which has been experimentally checked to be hitting most API calls through the whole WebGPU process up to the point of submitting multiple compute and render pipelines to be actually processed by the GPU. In case the number of API calls has an effect, e.g. if the absence of a certain API call by the end of the program is what triggers a bug, the number of API calls has been set up to be chosen at random from a uniform distribution between 1 and 1000. *wg-fuzz* also has some "program termination" code in its algorithm that adds some API calls in case a program that is fully valid is desirable and it needs to run any final API calls so all resources end in a valid state.

4.4.2 Algorithm

With generating a sequence of API calls, there are several options for the order and algorithm with which to achieve this. Below are three possible options:

- In-order, i.e. from beginning to end
- Backwards, i.e. from the end of the program to the beginning
- Random inserts

In-order is the first algorithm that comes to mind, and is the simplest one to implement. This is what *wg-fuzz* uses. The algorithm is essentially how *wg-fuzz* is architected - i.e., keep a resource tree, and decide on a random API call from the ones available to be called next. This approach should theoretically achieve the full search space of WebGPU programs. As new WebGPU resources are generated from random API calls, they are not necessarily used later on. However, empirically, it seems that most of the resources from programs generated using *wg-fuzz* do get used later on, though not all.

Another possible approach would be to start with the API calls made at the end of the generated program, and work backwards, keeping a list of required resources. This approach seems to be simpler on the surface, requiring that the fuzzer creates all of the necessary resources by the end of the generation. However, this approach would likely require a more complex algorithm and mental model in order to create programs that are general like in-order generation achieves. In order to have as general API interactions as possible, you would need to keep track of the resources that you have available, potentially essentially creating an Abstract Resource Tree but based on resources you should create as you go along. The mental model of in-order seems clearer and simpler to implement.

A last possible approach might be to insert API calls at random places in the generated program. This approach seems to largely neglect the fact that the API calls have complex validation rules, so it does not seem like it would be very effective as a fuzzer nor easy to implement (e.g. how would the fuzzer know which buffers are available for a buffer-to-buffer write at a random point in the program? It would seem to need to re-implement the Abstract Resource Tree approach from the in-order algorithm).

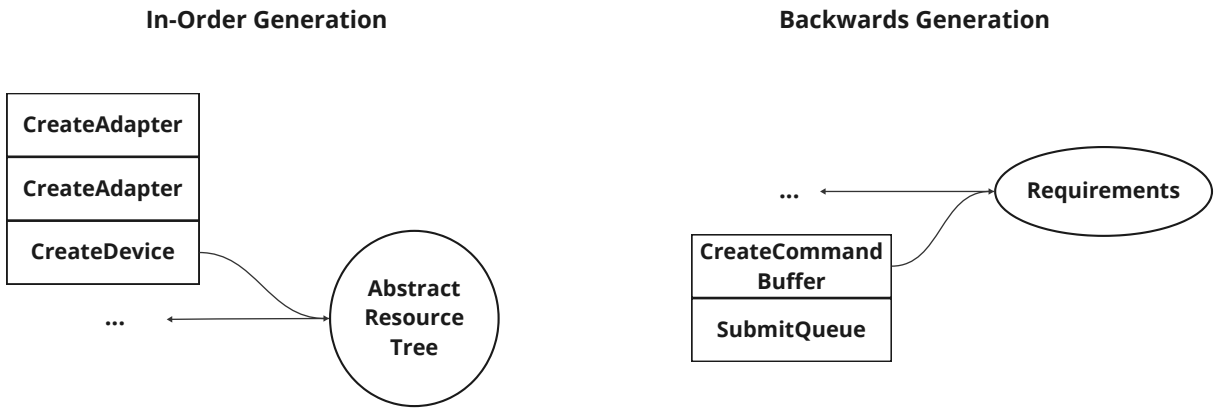

Figure 4.4: Comparison of in-order generation with backwards generation

4.4.3 Termination

It is possible for *wg-fuzz*'s generator to reach the end of its allowed number of API calls, and some resources to be left in a half-done state. For example, beginning a render pass encoder, but not finishing it before the end of the program. Since the initial objective in the implementation of *wg-fuzz* is for it to produce fully valid WebGPU programs, it is desirable to end in a valid state, too. Because of the randomised approach to the number of API calls made in each program generated, there is twice the uncertainty on when it will end.

For this reason, after the main generation loop ends, *wg-fuzz* invokes a second, similar, loop, but specifically only for adding any API calls that would deal with any unfinished state of the resources in the Abstract Resource Tree. Essentially, it finds all available API calls like usual, but applies a hard-coded filter to which API calls make it through.

For example, `CreateRenderPass()` would be filtered out and unavailable, while specifically the calls that are required in order to finish the render pass off, such as setting the pipeline, drawing, and ending the render pass, are kept.

It is important that the API calls that are kept in are guaranteed to not be available on the next call to the Available Calls Driver, otherwise the generator may get stuck inside an infinite loop, as it keeps generating these terminating API calls until there are none left.

For this reason, the availability of certain API calls like the ones for the render pass are implemented in a strictly sequential order, where it is guaranteed to no longer be available after it is called.

An abstracted version of the 2nd loop used for termination is shown below:

```
let mut terminating_api_calls = available_terminating_api_calls(resources);
while terminating_api_calls.len() > 0 {
    let api_call = terminating_api_calls.pick_random();
    let new_resource = update_program_resources(resources, &api_call);
    program.calls.push(api_call);

    terminating_api_calls = available_terminating_api_calls(resources);
}
```

4.4.4 Swarm Testing

The use of swarm testing is vital, as it keeps *wg-fuzz*'s programs random, but also interesting. It gives *wg-fuzz* the potential to create programs that extensively test a specific part of the WebGPU API. Even though it does not increase *possibility*, it hugely increases the *probability* of certain combinations of API calls. For example, by default, *wg-fuzz* would be hugely unlikely to find a bug that occurs after 100 invocations of the same API call (perhaps some buffer overflows internally), as it would have so much choice between API calls that there would be path explosion.

With swarm testing, from run to run (*not* API call to API call), *wg-fuzz* randomly creates a filter for which API calls are available during this run of program generation. This is randomly decided using a configurable probability argument to *wg-fuzz*.

Since the filter applied is completely random in which API calls it chooses, it is possible to filter out a call that is vital in setting up a resource in a group of related calls. However, this is desirable, as this enables intensive testing of a particular API call from any point in WebGPU.

Choosing an appropriate probability used in the creation of the swarm filter is vital in being able to find bugs. For example, the likelihood of finding a bug diminishes exponentially as the probability of keeping an API call in the swarm filter goes down. At the very least, *wg-fuzz* needs to invoke certain calls like `requestAdapter()` and `requestDevice()` as the entrypoint in the program for using WebGPU. If the probability is too low, *wg-fuzz* may bottleneck itself too much for later API calls. At the extreme end, filtering out all the calls is completely useless.

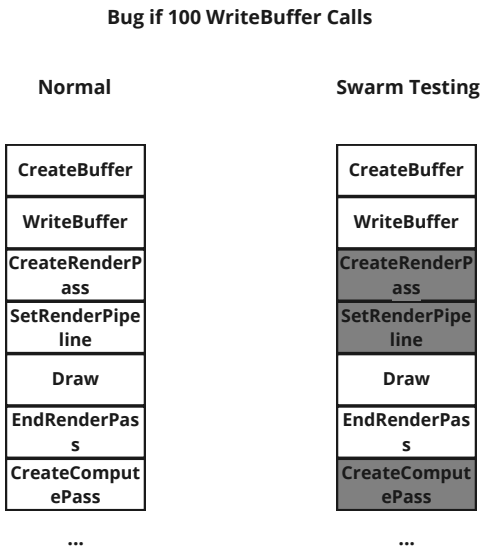

Figure 4.5: Swarm Testing in WebGPU

4.4.5 Fuzzy Conditions

At its base implementation, *wg-fuzz* should only output strictly valid WebGPU programs. This is useful as there may be bugs in the interaction of WebGPU objects, and they may only interact if each is valid, and the interaction call is valid. This drives *wg-fuzz* to encode the validation rules and restrictions in the Available Calls Driver implementation.

However, there may also be bugs in WebGPU API calls that are not completely valid according to the specification. In fact, some may be quite far into the WebGPU program, only accessible after successfully creating many different resources in the setup necessary for it. This gives us a subtle problem of needing slightly invalid WebGPU programs, sometimes, in order to find the most bugs.

To this end, *wg-fuzz* uses what it calls fuzzy conditions. The actual implementation is quite straightforward. In the core Available Calls Driver algorithm, for each if statement that checks whether certain validity rules in the Abstract Resource Tree resources hold before adding an available API call to the list, you add an 'or' condition that may override it with a certain probability defined by the fuzzy conditions probability configuration. An example implementation is given below:

```
pub fn fuzzy(rng: &mut ThreadRng, ratio_incorrectness: f64) -> bool {
    rng.gen_bool(ratio_incorrectness)
}
```

And here is an example usage in the Available Calls Driver code:

```
if !query_set.destroyed || fuzzy(&mut rng, fuzzy_prob) {
    available_api_calls.extend([PrintQuerySet(query_set)]);
}
```

In this way, a careful balance between valid and invalid API calls is struck in an attempt to find the most bugs possible, or to be as efficient in fuzzing as possible. However, the probability cannot be too high, as programs that are completely valid are still desired, in case they show up bugs. Particularly, if there are around 1000 API calls in a generated program, and many if statements in the Available Calls Driver, the probability of overriding the Available Calls Driver's conditions should be quite low.

4.5 Resource Simulation

Once the API call to be added is chosen, its effect must be simulated on the Abstract Resource Tree in order to inform future iterations in the Available Calls Driver of what resources are available. The API call enum is effectively de-structured via a match statement, and the appropriate update to the Abstract Resource Tree is made.

For example, if `CreateTexture(device)` was randomly chosen as the next API call, it would be passed into the Resource Simulation code, which would direct it to the appropriate handling code, which would update the Abstract Resource Tree by adding a new Texture object on the WebGPU device that is passed in through the API enum's field.

Additionally, when a new object is created on the Abstract Resource Tree, a JavaScript identifier is also automatically decided, which will be used when it gets converted to source code. This is information that the source code converter needs, so a copy of this new resource is additionally returned from the Resource Simulation code, where the top level generator code stores it alongside the API call that it decided on, inside the Abstract Call Vector.

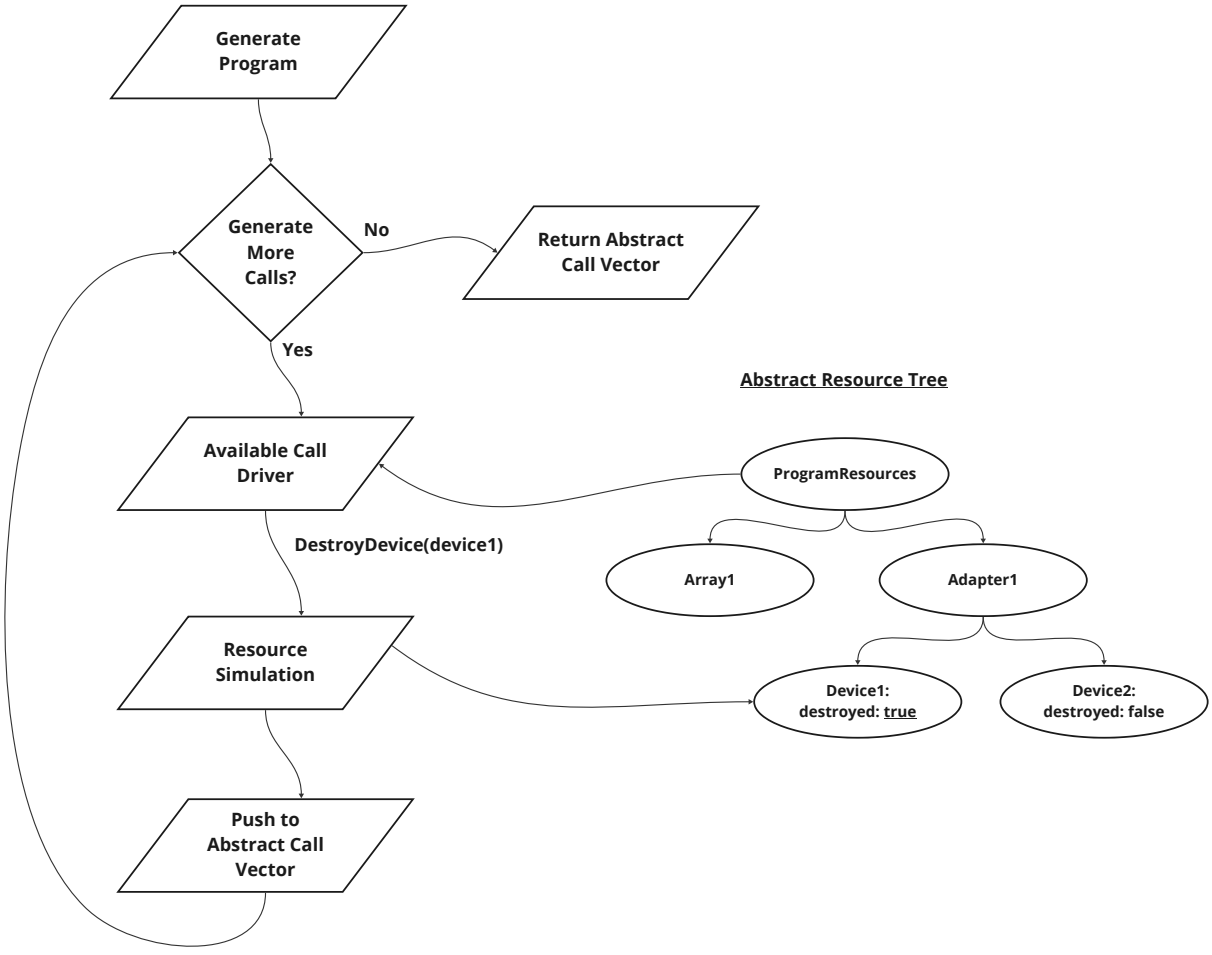

Figure 4.6: Updated Generation Algorithm

4.6 Converting to Source Code

Once the generator has finished creating an internal representation of the WebGPU program through the Abstract Call Vector, it needs to convert this to JavaScript so that *wg-fuzz* can run this through Node hooked up to a WebGPU implementation.

Each internal API enum variant implements a `to_javascript()` method, so *wg-fuzz* simply iterates through the Abstract Call Vector, calling `to_javascript()` on each API call held inside. A program prelude string is appended at the very beginning of the output program, for purposes such as importing the WebGPU implementation into Node, and starting the program main function definition. A postlude string is also appended, to finish off the function defined above, and to call it.

There are three possible types of sources for the custom objects and options needed in a `to_javascript()` call:

- The API enum variant's fields
- If the call being converted is the type that creates a resource, the new resource returned from the Resource Simulation
- For basic options that are not very important or restrictive, randomly generated inside the `to_javascript()` call itself.

An example showing the use of the first two is shown below:

```
match api_call {
    CreateRandomBuffer(device) => {
        if let Resource::GPUBuffer(buffer) = created_resource {
            format!("\
    const {} = {}.createBuffer({{
        label: \"{}\",
        size: 400,
        usage: {}
    }});",
                buffer.var_name, device.var_name, buffer.var_name, buffer.
   use_case)
        } else {
            panic!("created_resource for CreateRandomBuffer() call is not a
   buffer!")
        }
    }
    ...
}
```

It is also required at times to generate some data randomly inside the `to_javascript()` method itself. For example, when defining a new Float32Array with randomly initialised data for use by WebGPU. This data is not very restricted in its values, and it is desirable to have as broad a range as possible to test the API. While not all options available in WebGPU are covered by *wg-fuzz* (as there are many that interact in different ways), *wg-fuzz* achieves complete API call coverage for the calls that are available in Node.

4.6.1 WGSLsmith

One particularly special case of the `to_javascript()` method calls is the one that defines a new shader module. While the focus is not on WGSL (the shader language), it is possible that there are bugs in the API that only appear with certain shader files. This treads carefully on the thread of complexity explosion, so *wg-fuzz* has not focused too hard on expanding what kind of WGSL shader files are generated.

Luckily, a previous project, WGSLsmith by Hasan Mohsin, has built a WGSL fuzzer, built in Rust. While it only supports compute shaders, *wg-fuzz* makes use of it for this purpose. This enables *wg-fuzz* to build more general WebGPU programs than would otherwise be possible. For render shaders, *wg-fuzz* reuses some representative shader code that contain the main features sought to be tested in the API, such as buffer bindings.

The use of this tool has shown how quickly the WebGPU specification is evolving, as another research student, Amber Gorzynski, has had to create her own fork with updates to it, as the original version from 2 years ago no longer works on more recent implementations of the more recent specification.

It includes a reconditioner that ensures termination of the shader code, and removes undefined behaviour. The termination condition especially is quite useful for *wg-fuzz*, as having multiple different shaders being executed in a program that *wg-fuzz* generates would increase the likelihood of the execution not terminating.

One downside of using WGSLsmith is that is seems to take a few seconds to run for each generated WGSL shader. The main bottleneck seems to be the reconditioner, however, the reconditioner is necessary if *wg-fuzz* is to use WGSLsmith in order to avoid non-terminating WGSL shaders. Especially when *wg-fuzz* generates many shader modules in a WebGPU program, this significantly increases the time it takes to run a test case.

It is unclear if WGSLsmith is useful in finding bugs in the WebGPU API. This is coupled with the complexity explosion of different possible shaders and program combinations. So far, the vast majority of bugs found in the API have been without shader modules. Perhaps an investigation into a more "templated" form of WGSL shaders may make *wg-fuzz* quicker and more effective at finding bugs, as currently it uses the same bind group resources in each compute shader, and in each render shader.

For *wg-fuzz*'s use case, options are passed into WGSLsmith telling it to output a smaller WGSL shader file. Otherwise, execution is too focused on generating WGSL files over WebGPU API sequences. *wg-fuzz* asks for a max block depth of 1 and a max of 2 functions.

4.7 Fuzzing and Processing

While the main challenge in fuzzing WebGPU is creating a random WebGPU program, the main objective is to test any certain WebGPU implementation. For this reason, as soon as a WebGPU program is generated, it is execute using Node and a compiled WebGPU implementation.

In order to find bugs, an oracle is required. The two main oracles *wg-fuzz* has utilised in its fuzzing are crashes and sanitizers. If any given execution via Node crashes, it is most likely because of a bug in the WebGPU implementation. While it is possible for Node to crash itself, it is highly unlikely, especially given that it is very popular in the use case of running servers.

In a singular run of *wg-fuzz*, it removes and creates a temporary directory called `out/`. In here, it generates the `test.js` program, along with any generated WGSL shader files. Additionally, the WebGPU implementation, which has so far been Dawn (i.e. dawn.node), is copied into the `out/` directory. So, all the files required to run the generated test case are self-contained inside the `out/` directory.

Now, the generated test case is run with sanitizer flags. If a crash is detected (exit code non-zero), or the stdout or stderr outputs contain any of some defined key phrases, such as "sanitizer", then the test case is logged as a potential bug.

While *wg-fuzz* is run, it logs potential bugs inside a `generated_bugs/` directory under folders named after the timestamp that it was found. Now, since the `out/` directory was self-contained along with the compiled version of WebGPU that was used, it is easy to reproduce exactly what *wg-fuzz* found, even if the WebGPU implementation executable is replaced at the top end. Additionally, `stdout.txt`, `stderr.txt`, and `exitcode.txt` files are included to cross check what *wg-fuzz* saw with what is seen when the program is re-run, as a bug may be non-deterministic (such as based on timing or race conditions).

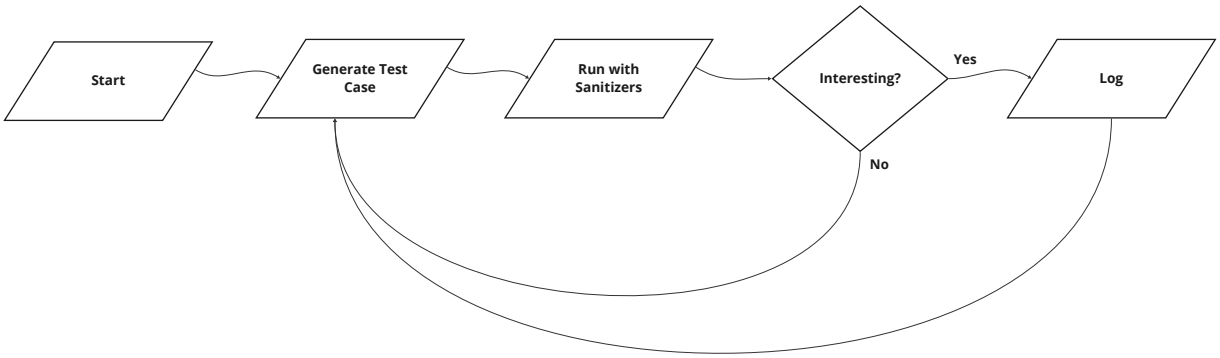

Figure 4.7: The Overall Fuzzing Process

4.7.1 Program Reduction

As the programs generated are quite large, it can be hard to know where the bug is coming from, and reporting bugs in files of this size would not be very helpful to the developers. Hence programs are reduced to their minimal bug-inducing form.

The *wg-fuzz* open source repository contains instructions on how to use C-Reduce to reduce a program to its minimal form. *wg-fuzz* includes an interestingness test for use by C-Reduce. However, unfortunately, this takes a very long time to run - so long that it is infeasible to use it practically.

This is likely because C-Reduce has to run each tentative change it makes through Node, since this is what *wg-fuzz*'s interestingness test specifies to do. Since Node is a JavaScript interpreter, oftentimes it does not detect something as basic as an error in the source code syntax until it is about to run it. This is exacerbated by the fact that WebGPU programs in particular can take a while to run, as potentially has to go through running multiple pipelines on the GPU on each interestingness test invocation.

This is a limitation of the way *wg-fuzz* is designed to run. It may be possible to do some free transformations inside of *wg-fuzz*'s implementation using the Abstract Call Vector, such as removing API calls that simply print some fields, or detecting resource creation calls which are not used later and seeing if removing them keeps the bug in the program, but this approach will still have the same issue of invocations of the program being expensive in the end. Creating a minimal version of the test file would still take too long.

It was found be to effective to reduce any bugs found by hand. This actually does not take very long, as removing the latter half of the calls in the file and testing this against Node, or removing as much as you can without the bug escaping you, results in a fairly quick process. It also helps to understand when the bug is exhibited. For example, some bugs were found to happen with a combination of a certain API call, and then one of several possible calls just after. This might not have been discovered with an automatically reduced file taken at face value. This is also quite useful to report in the bug report for developers to further understand the root cause of a bug.

5 | Evaluation

This project has focused on testing Dawn, Google's implementation of WebGPU, being in frequent contact with the developers. Dawn was continuously tested against WebGPU programs throughout the project, resulting in 14 bug reports to the developers as of today, with 6 of these being fatal crashes without even requiring sanitizers and are open to a normal execution through Node. Additionally, some potential issues are reported. Many of these have since been fixed by the developers.

Overall, Dawn was found to be more buggy than expected. It is clear that it is still under active development, as one challenge of the project was a rapidly changing specification. For example, one call (`requestAdapterInfo()`) was no longer available in the API after a simple update to the code-base. Additionally, not every feature defined in WebGPU is available in Dawn yet, with, for example, adapter objects in current WebGPU programs showing that none of the additional functionality laid out in the specification is implemented yet.

As well as looking for bugs during development of *wg-fuzz*, a large-scale fuzzing campaign is also run on university servers in parallel (through Condor) in order to evaluate the effectiveness of swarm testing and fuzzy conditions at finding potential bugs.

5.1 Bugs Found

A number of bugs have been found, both crashes and other, in the implementation of Dawn. Table 5.1 shows an overview of the state of bug finding so far. Only the useful reports were included in the data (some may come e.g. from a misunderstanding of a recent update to the API specification).

There are several reasons a bug may remain unreported. The most frustrating one is that some bugs are simply too unreproducible due to their nondeterministic nature, disabling any sort of reducing or bug triage that would be possible. For example, *wg-fuzz* found a crash that seemed useful to report, however, it was only possible to reproduce it once after many runs of the same program. The large nondeterminism of this crash made it completely infeasible to reduce the thousands of lines of code that lead to this crash. The developers are unlikely to know what to make of thousands of lines of code that leads to a crash only some of the time if it was to be reported. There was another bug like this in some WGSL shader code, presumably generated by WGSLsmith.

Status	Count
Unreported	4
Reported	14
Crash	6
Other	8
Fixed	4

Table 5.1: Overview of bugs found

The 'Other' category of bugs found includes bugs found through sanitizers UndefinedBehaviourSanitizer and AddressSanitizer, such as a heap-use-after-free bug, and also includes weird error messages produced by the Dawn implementation during execution of a program. As an example, this error message cropped up several times during the course of execution for a few of *wg-fuzz*'s generated programs:

```
Invalid injected error, must be Validation or OutOfMemory
    at APIInjectError (/home/leu/Documents/projects/fyp/dawn/src/dawn/native
   /Device.cpp:1970)
```

Low-priority bugs such as leaks are included in the 'Other' category, as the developers seem to be doing a little fuzzing themselves, so they are likely to already know about them. Additionally, leaks are not as likely to break anything inside the code or to provide any security vulnerabilities.

Some of the bugs found were found during a preliminary stage of playing with different hand-written WebGPU programs to test running the integration of WebGPU into Node, and to test running it with sanitizers. One example of this is leaks. There seemed to be a leak from within Dawn by simply importing it into Node. The number of crashes vs. other bugs found in both stages is given in table 5.2 below.

Found by hand	
Fatal Crash	0
Other	4
Found by *wg-fuzz*	
Fatal Crash	6
Other	4

Table 5.2: Comparison of bugs found by *wg-fuzz*

Evidently, the bugs found in the preliminary stage of testing by hand were fairly trivial in terms of crashes. One bug was a result of using UndefinedBehaviourSanitizer, which gave us an assertion failure at the end of an execution of a program using Dawn, regardless of what the program was. This is not as bad as the crashes found using *wg-fuzz* that crash without even any sanitizers.

Comparatively, the number of crashes found by *wg-fuzz* is relatively high. These are especially useful as they only show up in simple but specific cases, along with the fact that these show up in a regular setup without sanitizers, i.e. given that it is not a crash specific to Dawn's Node bindings, it should show up in Chrome, too (though this has not been specifically verified).

From the crashes found by *wg-fuzz*, 4 are in Dawn's native implementation, and 2 are in its Node bindings. The crashes found in its native implementation would likely impact more end users, as Dawn is integrated directly into the Google Chrome browser without the need for Node. Crashes in a WebGPU program may lead to frustration in users trying to use a web application, and in developers as they try to build a web application using this new API.

However, crashes in the Node bindings might be just as serious. Recall that one of the flagship improvements of WebGPU over its predecessor WebGL is that is supports general purpose GPU computations. Node's package manager, NPM, is the largest software repository in the world with over 800,000 code packages. It is very popular, especially for server-side applications. Coupled with the recent AI boom, more and more developers are seeking to develop machine learning applications, which make heavy use of the GPU. On top of this, there are many image-processing libraries that could make use of the GPU. Therefore, it is desirable that Dawn's Node bindings are as robust as possible in order to support this new avenue of development.

As a final comment, some of the bugs live in one of Google's "third party" libraries, Abseil. Though this is not directly a crash in Dawn's code, it may be a a result of misuse of the library, and it is just as serious nonetheless. The Dawn project still holds the executive decision of which libraries to use, and a crash that happens as a result of the use of one of Google's other developed libraries is not enough to dismiss it.

Below is a discussion of some of the interesting bugs that were encountered in the process of bug finding.

5.1.1 Possible Race Condition

Perhaps the simplest and most interesting crash found was with this simplified WebGPU program:

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

device.queue.onSubmittedWorkDone();

device.destroy();
```

The `onSubmittedWorkDone()` call returns a JavaScript Promise. The Promise is not awaited, which means it is executed as an asynchronous operation. When this program is executed, this text is printed on the standard error channel:

```
FATAL ERROR: Error::New napi_get_last_error_info
----- Native stack trace -----
...

Aborted (core dumped)
```

My best guess is that the Dawn implementation does not take into account this race condition between `device.destroy()` and the asynchronous operation `onSubmittedWorkDone()`. However, note that *wg-fuzz* managed to get the same error with a different WebGPU program where there is no `destroy()` call, and `onSubmittedWorkDone()` is awaited. The main similarity is the presence of `onSubmittedWorkDone()`. It is unclear what the underlying problem is from an outsider's perspective.

5.1.2 Unreachable Statement Hit

Another set of crashes that *wg-fuzz* came across were in the Node bindings. The first one was caught pretty quickly, as it cropped up after simply creating an adapter and device. It was fixed quickly by the developers, which then lead to a second, similar, unreachable bug to be found in this program:

```
const adapter = await navigator.gpu.requestAdapter();
const device1 = await adapter.requestDevice();
const device2 = await adapter.requestDevice();

const buffer = device2.createBuffer({
    size: 100,
    usage: GPUBufferUsage.MAP_READ
});

await buffer.mapAsync(
    GPUMapMode.READ,
    0,
    100,
);
```

Note that this is the minimal form - the bug requires two such devices, and the buffer calls. It is possible to replace the buffer calls with some of the other API calls to exhibit the same bug, but this is how it was originally caught.

For reference, this is the error exhibited when the program is run:

```
/home/leu/Documents/projects/fyp/dawn/src/dawn/node/binding/GPUDevice.cpp
   :182: UNREACHABLE: operator()() WGPUDeviceLostReason_Force32|
   FailedAtCreation
Aborted (core dumped)
```

The implementation seems to hit an `unreachable()` statement, at which point it exits. This second unreachable bug has not yet been fixed, but it has been assigned to a developer to fix once he finishes his other tasks.

5.1.3 Compute Pipeline Creation

To set this crash up, a certain generated WGSL file must be used, along with a WebGPU program that imports the WGSL code into a shader module and creates a basic compute pipeline from it. When the program is executed, it runs for a few minutes (which is unusual), after which is prints `Killed` and exits with an exit code of 137. It is unclear from the stack trace to what extent the crash is a result of the WGSL file, but it mentions Dawn native and the Abseil library at some points.

5.2 Reflection on Bugs Found

wg-fuzz seems to do a fairly good job of finding a high percentage of critical bugs in the WebGPU API, i.e. bugs that cause crashes, or timeouts. While many of the bugs have not been fixed, the bug reports submitted to the developers are high quality, giving information such as environment used, steps to reproduce, expected vs. observed output, a stack trace in the case of crashes, and possibly personal thoughts on the bug's nature. Additionally, any questions that the developers have on the bug report are promptly answered.

This paper has focused on maintaining high quality bug reports and developer relations via email, as it is important to keep in mind that the ultimate objective of this project is to enhance the robustness of WebGPU's implementations. It cannot be neglected that whether this is actually achieved may well also largely be a social factor.

There may be more bugs to be found, but would most likely be as a result of further development of *wg-fuzz* in order to cover more of the options available in the large WebGPU API. This paper has not had time to cover every possible feature, and it is likely that as additional features are supported, the number of possible interactions increases exponentially, which may lead to more bugs being found.

5.3 Feature Effectiveness

In order to find as many bugs as possible, it is desirable to run *wg-fuzz* for as long as possible. Since *wg-fuzz* is embarrassingly parallel, it is easy enough to set up to run in parallel on many different machines as the same time.

wg-fuzz was set up to run in parallel on 110 different machines for 10 hours each. This results in 1,100 hours of fuzzing overall. Each machine was automatically configured to use a certain (swarm testing, fuzzy conditions) probability configuration in order to diversify the tests produced, and also to evaluate the usefulness of these features in *wg-fuzz*.

The probability values were split up in step-overs of 0.1. In fuzzy conditions, this results is 11 possible probability values, and in swarm testing, 10, as the edge case of a probability of 0.0 is entirely useless as it does not allow any API calls. Thus, 11 x 10 = 110 configurations, with one machine running one possible configuration.

	0.1	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**	**1.0**	**Total**
0.0	1	1	3	1	1	1	1	1	4	2	16
0.1	1	1	1	1	2	7	5	6	19	13	56
0.2	1	1	0	3	2	0	0	26	4	2	39
0.3	1	1	1	1	0	1	0	0	0	2	7
0.4	1	2	0	1	0	0	1	1	0	3	9
0.5	1	1	0	0	0	0	0	0	0	1	3
0.6	1	0	0	0	4	1	1	0	1	0	8
0.7	0	1	5	1	0	0	0	0	0	0	7
0.8	1	1	1	0	1	2	0	1	0	0	7
0.9	1	0	0	1	0	4	0	0	0	0	6
1.0	4	0	1	0	3	1	0	0	0	0	9
Total	13	9	12	9	13	17	8	35	28	23	167

Table 5.3: Number of *potential* bugs found in the fuzzing campaign. Top row represents the fraction of swarm testing API calls available, left column represents the fuzzy condition probability of overriding validation rules.

5.3.1 Bugs Found

The campaign yielded 3 additional bug reports to be filed to the Dawn developers. There was an effort to filter the bugs that would be useful to report. Additionally, these 3 bugs showed up many times across the potential bugs reported, and were sometimes hard to reproduce. One bug is only reproducible on the university servers used to fuzz in parallel, while another is elusive and simply doesn't show up. It is probable that this is due to the bug's non-deterministic nature.

Additional sources for potential bugs were caused by a sanitizer-internal implementation issue (error log directly points to https://github.com/google/sanitizers/issues/856). Therefore, there are many we cannot report to the developers of Dawn. Additionally, there seems to have been a bug in the namespace of JavaScript unsigned integer arrays that sometimes get declared in generated programs of *wg-fuzz*. Nevertheless, it is useful to analyse this data for insights into the effectiveness of swarm testing and fuzzy conditions. Even if a data point contains bugs we cannot report, it still means that this configuration was more successful at finding these edge cases than other data points, due to the increase in variety of programs that they generate.

5.3.2 Fuzzy Conditions

The more novel feature of *wg-fuzz* is its fuzzy conditions. We would like to evaluate its effectiveness in the context of generating random API sequences with complex restrictions. Note that fuzzy conditions with a probability of 0.0 exactly represents *wg-fuzz*'s implementation before it introduced fuzzy conditions.

Looking at the data, we see that a probability of 0.1 is most effective of the probabilities tested. This is a huge improvement over 0.0, as it jumps from a total of 16 potential bugs found, to 56. This is a 250% increase in the number of potential bugs found. Close to 0.1, a probability of 0.2 also managed to find an increased number of potential bugs, at 39. This is a 143.75% increase. From 0.3 onwards, the number of potential bugs effectively flattens out.

Hence, the optimum seems to be somewhere between 0.0 and 0.2, in this case. This paper hypothesises that the optimum is only slightly above 0.0, and below 0.1, as this would allow for more intricately invalid WebGPU programs to be built. This result also matches this paper's expectations as laid out in the section introducing fuzzy conditions.

It is unclear what the maximum potential gain of fuzzy conditions for *wg-fuzz* is. To get a clear overview of the distribution, and not to assert this paper's assumptions or hypothesis in the analysis, a distribution over the whole range of probabilities from 0 to 1 has been tested. However, it is possible that a greater yield is achievable by a further, more fine investigation into the analysis of fuzzy conditions.

5.3.3 Swarm Testing

Additionally, we include an analysis of the effectiveness of swarm testing. Swarm testing's effectiveness has already been shown in past research, however, it is still useful to check it in the case of *wg-fuzz*. It is important to test all probabilities in case we miss a huge spike in efficiency at a certain configuration, and a malicious party trying to find an attack vector on the web comes upon it and is able to do much more effective fuzzing.

Note that a probability of 1.0 is effectively equivalent to *wg-fuzz*'s implementation before it implemented swarm testing. As might be expected, a slight filter on API calls made does enable an increased number of potential bugs to be found. Going from a probability of 1.0 to 0.9, we go from 23 to 28, or a 21.74% increase in potential bugs found. Similarly, going from 1.0 to 0.8, there is an increase of 52.17%. Below 0.8, the number of potential bugs found levels out more.

5.3.4 Interaction

First of all, it is clear that the optimal configuration is roughly found using a swarm testing probability nearer to 1.0, and a fuzzy condition probability closer to 0.0. However, significant gains can be made be moving slightly away from that point. As the data is highly variable, an effective analysis cannot be made using singular data points, but this observation can be shown by considering two blocks of 2x2 cells with a different in distance from the top right corner of the table. If we find the sum of the closest four cells to the top right corner, we get 38 potential bugs. Moving our 2x2 block one cell down and one cell left, a similar sum gives us 55. This shows us roughly that moving slightly away from the top right corner should give us better results.

More specifically, we can see by comparing the peaks found for fuzzy condition totals vs. swarm testing totals, that the peak for fuzzy conditions is closer to the default value than is the case for swarm testing. This likely means that fuzzy conditions are harder to get right, as it dangerously offers to give invalid WebGPU programs, while swarm testing does not have this effect. However, fuzzy conditions also offer a much greater increase in potential bugs found than swarm testing does. This was seen when finding the percentage increase with the peaks compared to the default values - 250% for fuzzy conditions, 52.17% for swarm testing.

5.4 Analysis of *wg-fuzz*

The implementation approach that *wg-fuzz* has taken has allowed it to generate the most general form of tests for the WebGPU API, while keeping the programs it generates diverse, interesting, and effective for bug finding. Since it takes a code-gen approach, it is also possible to use it in differential testing.

This paper did not have enough time to try out differential testing with Mozilla's implementation of WebGPU, though it seems a worthwhile avenue for future work. This is something that the approach of *wg-fuzz* allows, and is likely one of the next features that *wg-fuzz* would allow. The focus was instead on finding as many bugs in Google's Dawn implementation as possible.

One possible criticism of it is that it requires large development effort to develop and maintain, meaning it has not yet been able to cover the whole API in full. However, this paper maintains that, regardless, it has been able to find bugs that may not have been possible in other implementations that result from specific combinations of API calls. As ensuring robustness of this new avenue of development on the web and Node is important, this may well be worth the effort required.

Another criticism is that it may be slower compared to interfacing directly to a particular implementation of WebGPU, and possibly due to the use of WGSLsmith. However, this allows it the potential to test any WebGPU implementation, and it is simpler for an outside developer to implement. It is possible that replacing WGSLsmith with a custom WGSL generator that cares more about the interface of the program than the operations, and is faster, may have a significant impact on the efficiency of *wg-fuzz*.

Effort has been made to allow users of *wg-fuzz* to pass some options to it through the command line, however, features such as this have not been a major focus of the project as a sole developer and user. If *wg-fuzz* were used by more people, I would be happy to increase its user friendliness.

6 | Conclusion

As far as this paper is aware, *wg-fuzz* is the first tool that has been developed with testing random sequences of WebGPU API calls in mind. It has been effective at finding critical bugs, reporting 14 bugs to the developers of Google's implementation Dawn, 6 of those being fatal crashes. 4 have been fixed so far, with most others having been discussed and assigned to developers already.

This paper additionally applies and evaluates swarm testing and *wg-fuzz*'s concept of fuzzy conditions in the case of fuzzing the WebGPU API. A fuzzing campaign is run over a total of 1,100h compute-hours, with different configurations for fuzzy conditions and swarm testing. Fuzzy conditions in the context of *wg-fuzz* seems effective, enabling a substantial increase in the number of potential bugs found. Swarm testing is also confirmed to increase effectiveness, though not to the same extent as fuzzy conditions.

6.1 Future Work

wg-fuzz achieves 100% coverage of the WebGPU API as far as running it on Node allows, with WebGPU exposing almost 100 possible API calls. However, due to time constraints, this paper has not been able to fully test all of the arguments available in these API calls. It is possible that many of these contain bugs, especially with the explosion of the space of possible WebGPU programs that *wg-fuzz* might be able to generate given the implementation of these arguments. It is unknown how many bugs live in this uncovered program space.

Given more time, it would be worthwhile to additionally implement a differential testing oracle in *wg-fuzz*, and test both Google's and Mozilla's WebGPU implementations. This would make *wg-fuzz* more robust, in the sense that it currently relies on checking for simple phrases such as "sanitizer" in the standard error / output of the program, along with the exit code. Though this catches the most critical bugs, it is not the most robust oracle that is possible - differential testing would be a valuable addition.

Lastly, it is possible that a valuable addition would be to write a custom WGSL program generator for render and compute shaders, with a focus on the API features that it uses, such as resource bindings, as opposed to the specific instructions executed. Currently, WGSLsmith takes several seconds to generate and recondition all of the WGSL shaders that one run of *wg-fuzz* needs. A custom WGSL shader generator with less focus on the instructions would likely be much faster, and speed up *wg-fuzz* immensely.

While this paper has endeavoured to find as many bugs as possible, there is a strong possibility that is it possible to find more, given more fuzzing campaigns and more features implemented in *wg-fuzz*. Of course, it is always possible to work on reporting more bugs to the developers of WebGPU implementations.

6.2 Ethical Considerations

The goal of this paper is to improve the reliability and potentially security of implementations of the WebGPU API. Since the web is a major target for malicious parties thanks to its far-reaching nature and underlying security complications, *wg-fuzz* and the bugs it has found should help to make the web a more robust environment for users and developers.

Since *wg-fuzz* is open-source, it is available to anyone to use to fuzz WebGPU, including to malicious parties, as it is not possible to differentiate them from helpful ones. This opens up the possibility of malicious parties using the tool to find bugs in WebGPU that they might be able to exploit.

However, security is a major concern of the developers for both the WebGPU specification, and the implementations. The WebGPU specification includes a non-normative section on possible security vulnerabilities that an implementation might have, and, for example, Google uses security features like sand-boxing in order to limit any potential security issues, and has a security team that works with the developers to investigate and create reports on the state of the security of the implementation of WebGPU.

As far as it is possible to tell, *wg-fuzz* has not found bugs specifically pertaining to security as opposed to robustness, and this paper suspects that a malicious party would need to be quite knowledgeable in order to outsmart all of the security features that are in place in WebGPU implementations through the use of *wg-fuzz*.

With this in mind, *wg-fuzz* has been used throughout the project to find any obvious bugs, and a fuzzing campaign was run that lasted for 1100 compute-hours by taking advantage of the fact it is embarrassingly parallel. This was across different swarm testing - fuzzy condition configurations, meaning any potential discoverable peak in efficiency should have been covered.

Bibliography

[1] Ninomiya et al. at W3C (13 June 2024). WebGPU W3C Working Draft. Retrieved 17 June 2024 from https://www.w3.org/TR/webgpu/.

[2] MDN (7 July 2023). WebGPU API. Retrieved 17 June 2024 from https://developer.mozilla.org/en-US/docs/Web/API/WebGPU_API.

[3] Google. Dawn Repository. Retrieved 17 June 2024 from https://dawn.googlesource.com/dawn/.

[4] Bookholt et al. WebGPU Technical Report. Retrieved 17 June 2024 from https://chromium.googlesource.com/chromium/src/+/main/docs/security/research/graphics/webgpu_technical_report.md.

[5] Hasan Mohsin (June 2022). WGSLsmith: a Random Generator of WebGPU Shader Programs. Retrieved 17 June 2024 from https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/2122-ug-projects/2122-individual-projects/WGSLsmith---a-Random-Generator-of-WebGPU-shader-programs.pdf.

[6] Junjie Chen et al. (January 2019). A Survey of Compiler Testing. Retrieved 17 June 2024 from https://software-lab.org/publications/csur2019_compiler_testing.pdf.

[7] Robert P. Kurshan (May 2000). Program Verification. Retrieved 17 June 2024 from https://community.ams.org/journals/notices/200005/fea-kurshan.pdf?adat=May%202000&trk=200005fea-kurshan&cat=feature&galt=none.

[8] Blazy et al. (Aug 2006). Formal Verification of a C Compiler Front-end. Retrieved 17 June 2024 from https://inria.hal.science/inria-00106401/document.

[9] Frankl and Iakounenko (1998). Further Empirical Studies of Test Effectiveness. Retrieved 17 June 2024 from https://dl.acm.org/doi/pdf/10.1145/288195.288298.

[10] Zhu et al. (September 2022). Fuzzing: A Survey for Roadmap. Retrieved 17 June 2024 from https://dl.acm.org/doi/pdf/10.1145/3512345?casa_token=oXF1XbZaPIcAAAAA:

aPmkEEnVYUzLTKgXXBWNnNyYI0koGuWmK9Lu2aXhvtkig5kA_zt5jKTfdiUVLzlAxB8NYflhyueWmw.

[11] Miller et al. (1989). An Empirical Study of the Reliability of UNIX Utilities. Retrieved 17 June 2024 from https://www.paradyn.org/papers/fuzz.pdf.

[12] Miller et al. (October 1995). Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Retrieved 17 June 2024 from https://www.paradyn.org/papers/fuzz-revisited.pdf.

[13] Pham et al. (25 August 2016). Model-based whitebox fuzzing for program binaries. Retrieved 17 June 2024 from https://dl.acm.org/doi/10.1145/2970276.2970316.

[14] Zalewski. Technical whitepaper for afl-fuzz. Retrieved 17 June 2024 from https://lcamtuf.coredump.cx/afl/technical_details.txt.

[15] Le et al. (June 11, 2014). Compiler Validation via Equivalence Modulo Inputs. Retrieved 17 June 2024 from https://dl.acm.org/doi/pdf/10.1145/2666356.2594334?casa_token=nodVnCHTRmkAAAAA:UpcQkJh7sW4vPRpzX5C6pGQUpHISfoJ_2PEEie4tMK4PArX7Uw3NCQotGt-htCgfwRBpdhF3UnYqrQ.

[16] Böhme et al. (October 2016). Coverage-based Greybox Fuzzing as Markov Chain. Retrieved 17 June 2024 from https://dl.acm.org/doi/10.1145/2976749.2978428.

[17] MDN (5 February 2024). GPUQueue: writeBuffer() method. Retrieved 17 June 2024 from https://developer.mozilla.org/en-US/docs/Web/API/GPUQueue/writeBuffer#validation.

[18] Godefroid et al. (13 June 2008). Grammar-based Whitebox Fuzzing. Retrieved 17 June 2024 from https://dl.acm.org/doi/pdf/10.1145/1375581.1375607?casa_token=zEAWipdLWzgAAAAA:C_cxEWMg3dkDaENRxC5_MIUkLwJw5lf_Yu2QTwHM6Z0lQhDL0sOZujFC7oe6TMn8GINK-ViCbIfmRQ.

[19] Yang et al. at W3C (June 2011). Finding and Understanding Bugs in C Compilers. Retrieved 17 June 2024 from https://users.cs.utah.edu/~regehr/papers/pldi11-preprint.pdf.

[20] Donaldson et al. (8 July 2022). CsmithEdge: more effective compiler testing by handling undefined behaviour less conservatively. Retrieved 17 June 2024 from https://www.doc.ic.ac.uk/~afd/homepages/papers/pdfs/2022/EMSE.pdf.

[21] Offutt et al. (September 2004). Generating Test Cases for Web Services Using Data Perturbation. Retrieved 17 June 2024 from https://dl.acm.org/doi/pdf/10.1145/1022494.1022529.

[22] Zalewski. American fuzzy lop. Retrieved 17 June 2024 from https://lcamtuf.coredump.cx/afl/.

[23] Green et al. (29 May 2022). GraphFuzz: Library API Fuzzing with Lifetime-aware Dataflow Graphs. Retrieved 17 June 2024 from https://dl.acm.org/doi/pdf/10.1145/3510003.3510228?casa_token=nGPYlpIIO88AAAAA:i0cvT1FsP4moKfP5bjebxRuN_uET9iBLlpjhB_9Ge20qz1aW8Osr2lq0-9CfW6p15hoBuz92NNyVVw.

[24] Jiang et al. (19 November 2021). RULF: Rust Library Fuzzing via API Dependency Graph Traversal. Retrieved 17 June 2024 from https://ieeexplore.ieee.org/document/9678813.

[25] Deng et al. (18 November, 2022). Fuzzing Deep-Learning Libraries via Automated Relational API Inference. Retrieved 17 June 2024 from https://arxiv.org/pdf/2207.05531.

[26] Groce et al. (July 2012). Swarm Testing. Retrieved 17 June 2024 from https://users.cs.utah.edu/~regehr/papers/swarm12.pdf.

[27] Barr et al. (May 2015). The Oracle Problem in Software Testing: A Survey. Retrieved 17 June 2024 from https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6963470.

[28] Howard (22 December 2023). BC and DC notes. Retrieved 17 June 2024 from https://git.gavinhoward.com/gavin/bc/src/branch/master/manuals/development.md#user-content-fuzzing-1.