Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

WGSmith

Randomised smart call sequence generation for WebGPU

Author:
Daniel Simols

Supervisor:
Prof. Alastair Donaldson

Second Marker:

January 24, 2024

Abstract

WebGPU is an exciting new standard for a JavaScript API that provides GPU-accelerated graphics and compute on the web. 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.

We introduce a tool that intensively tests the WebGPU API by smart call sequence generation.

Acknowledgements

WGSLSmith, fuzz-d, RustSmith?

Contents

1 | Introduction

WebGPU is an exciting new standard for a JavaScript API that provides GPU-accelerated graphics and compute on the web. It succeeds WebGL to enable use of modern GPU capabilities, better compatibility with modern GPUs, support for general purpose GPU computations, and faster operations.

Particularly, we look at Google's Dawn WebGPU implementation. Chrome uses sandboxing to create layered security boundaries between high privilege Chrome internals and the low privilege web content APIs exposed to web developers and users. However, Dawn's rapid evolution, the high amount of complexity required in the implementation, their need to interface directly with the GPU and its drivers, and their implementation in memory-unsafe languages mean that bugs in Dawn are especially useful for bypassing this security boundary.

There are two new potential attack vectors arising from WebGPU:

- the WebGPU API implementation
- the WGSL shader compiler for compiling GPU workloads

WGSL compilers have been investigated in the past by "WGSLsmith: a Random Generator of WebGPU Shader Programs" by Hasan Mohsin. WGSL is not the focus here. We focus on testing the WebGPU API by smart call sequence generation.

The study of compiler testing has proven itself effective at finding bugs in compiler implementations. We utilise current state-of-the-art techniques in order to build a tool that builds random JavaScript programs that call into the WebGPU API in interesting, intricate ways.

1.1 Contributions

This project makes the following contributions:

1. We develop a fuzzing tool to test the WebGPU API.
2. Something unique about fuzzing tool...
3. Any bugs found...

2 | Program Correctness

2.1 Overview

2.1.1 Program Correctness

Our goal is to test the WebGPU API. So, we start by looking at methods that show program correctness. Figure 2.1 shows a high level overview of ways to show that a program is correct. We can either formally verify that a program is correct, or we may use any of a plethora of testing techniques to validate the program up to a certain extent. Note that we can only be sure that a program works as it should if we use formal verification - testing techniques only show the presence of bugs for certain test cases, but trying all possible test cases is impossible, thus we cannot prove that a program is correct by testing it. However, formal verification has proven to be a hefty task even for subsets of certain programming languages. As an example, it is generally accepted that the most well-known C compilers, like GCC, do not fully conform to the C standard. This is because compilers are complex and large systems, and it is hard enough to write them, let alone verify them. (CompCert?)

Because formal verification is so hard, 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. The standard approach is to write many test cases for your internal functions by hand, on a unit testing framework. 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.

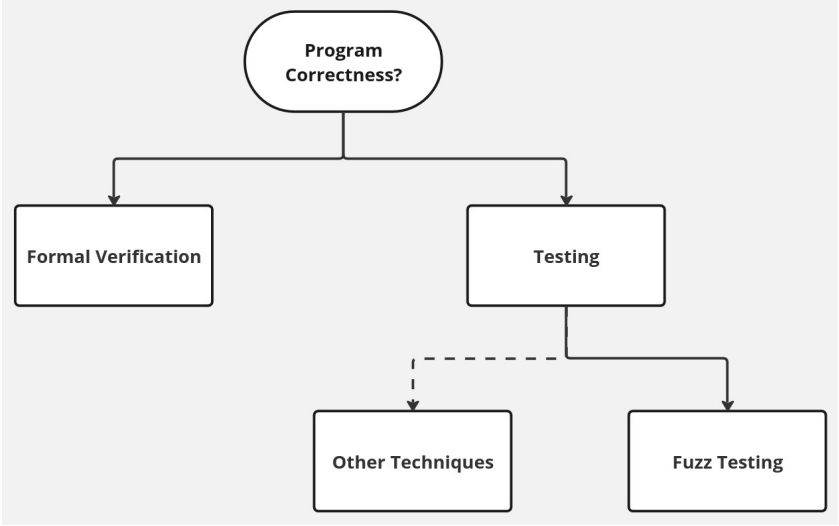

Figure 2.1: General overview of ways to check program correctness

2.1.2 Fuzz Testing

Fuzzing means providing random inputs to a program in order to find bugs. There are many ways to check whether the system under test (SUT) performs as expected, and this is called the 'Test Oracle Problem'. The most common 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. Fuzzing 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. We will delve into relevant parts in the following chapters. Fuzzing solves the problems of test case creation being time-consuming, and of human bias.

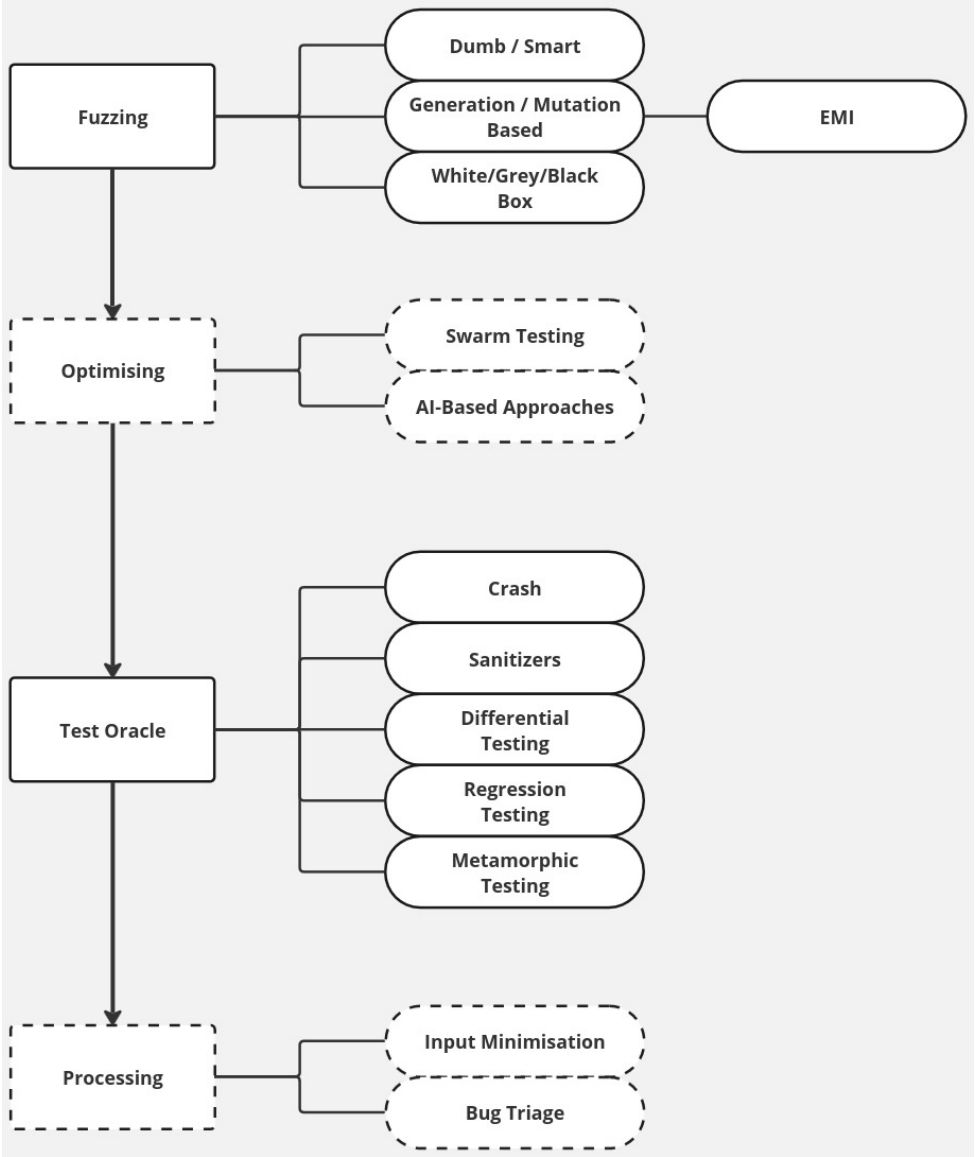

Figure 2.2: General overview of fuzz testing

2.2 Fuzzing

2.2.1 Compiler Fuzzing

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 our use case. We would like to input JavaScript programs that use the WebGPU API in random, interesting ways in order to find bugs in the WebGPU implementation. We would like to generate at least syntactically valid, random, JavaScript + WebGPU programs (smart fuzzing). In order to do so, we must decide on an algorithm to do this. Generally, there are two ways available: Generation, and Mutation Based Fuzzing.

2.2.2 Generation Based Fuzzing

Generation Based Fuzzing means generating an input completely from scratch. In our case, we may generate random API calls that adhere to some basic type-checking rules, etc. There are two main generation-based approaches: Grammar-directed and Grammar-aided. We explore these two approaches below.

Grammar-directed

Grammar-directed generation takes a grammar as an input and generates a random string following the rules of that grammar. 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. So a context-free grammar would, for example, test the part of our JavaScript engine that carries out semantic checks, over WebGPU's API calls.

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

Grammar-aided

2.2.3 Mutation Based Fuzzing

Mutation Based Fuzzing generates new inputs by modifying an existing corpus of inputs. 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.

2.3 Optimising Fuzzing

2.3.1 Swarm Testing

2.4 Test Oracles

2.4.1 Sanitizers

2.4.2 Differential Testing

2.4.3 Metamorphic

2.5 Processing

2.5.1 Input Minimisation

3 | WebGPU

3.1 Introduction

3.2 Programming

3.3 Dawn

4 | WGSmith

5 | Evaluation

6 | Conclusion

A | First Appendix

Bibliography