GLeeFuzz: Fuzzing WebGL Through Error Message Guided Mutation

Hui Peng
Purdue University

Zhihao Yao
UC Irvine

Ardalan Amiri Sani
UC Irvine

Dave (Jing) Tian
Purdue University

Mathias Payer
EPFL

Abstract

WebGL is a set of standardized JavaScript APIs for GPU accelerated graphics. Security of the WebGL interface is paramount because it exposes remote and unsandboxed access to the underlying graphics stack (including the native GL libraries and GPU drivers) in the host OS. Unfortunately, applying state-of-the-art fuzzing techniques to the WebGL interface for vulnerability discovery is challenging because of (1) its huge input state space, and (2) the infeasibility of collecting code coverage across concurrent processes, closed-source libraries, and device drivers in the kernel.

Our fuzzing technique, GLeeFuzz, guides input mutation by error messages instead of code coverage. Our key observation is that browsers emit meaningful error messages to aid developers in debugging their WebGL programs. Error messages indicate which part of the input fails (e.g., incomplete arguments, invalid arguments, or unsatisfied dependencies between API calls). Leveraging error messages as feedback, the fuzzer effectively expands coverage by *focusing mutation on erroneous parts of the input*. We analyze Chrome's WebGL implementation to identify the dependencies between error-emitting statements and rejected parts of the input, and use this information to guide input mutation. We evaluate our GLeeFuzz prototype on Chrome, Firefox, and Safari on diverse desktop and mobile OSes. We discovered 7 vulnerabilities, 4 in Chrome, 2 in Safari, and 1 in Firefox. The Chrome vulnerabilities allow a remote attacker to freeze the GPU and possibly execute remote code at the browser privilege.

1 Introduction

The increasing demand for high-performance 2D/3D graphics in web applications culminated in the creation of WebGL (Web Graphics Library) [28], a standardized set of JavaScript APIs supported by all modern browsers. Going beyond browsers, WebGL is also enabled in mobile and desktop application frameworks (e.g., the Android WebView [7] component, iOS WKWebView [9], or Electron [11]). This proliferation and fast adoption of WebGL makes it a lucrative target for attackers because it exposes a large attack surface.

WebGL gives potentially malicious remote users access to the native graphics stack, previously only available locally. Security analysis of the graphics stack is challenging as it often involves closed-source binary blobs developed by OEMs or third-party GPU vendors. Many recent high-severity CVEs in Chrome and Firefox demonstrate this unfortunate situation (e.g., CVE-2020-6492 [41] and CVE-2020-15675 [40]). As a short-term remedy, Chrome and Firefox both implement a deny-list of GPU models, API calls and parameters that would trigger known bugs in the native software stack [31]. This reactive, per-bug-focused solution is not suitable for protecting such a large attack surface of millions of lines of code across user-space and kernel-space.

Unfortunately, applying state-of-the art fuzzing techniques to the WebGL interface for vulnerability discovery is challenging because of (1) its huge input state space, and (2) the infeasibility of collecting code coverage due to the interleaving of multiple processes, libraries, and device drivers. The massive input space of WebGL are two dimensional: (1) ordered sequences of API calls and (2) various arguments passed to each API call. Typically, the execution of one API is dependent on both its arguments and the internal global state, which then sets up the state required by subsequent APIs. The inputs along both dimensions influence code coverage, and thus the effectiveness of a fuzzer.

Given such a huge input space with interleaved dependencies, a naive fuzzer can only explore shallow code paths. To effectively reduce the search space, state-of-the-art fuzzing tools use code coverage as feedback to guide their input mutation [12,25]. However, this approach depends on precise code coverage, which is hard to obtain across the complex WebGL stack as it consists of multiple layered components running in different processes and kernel space. In addition, some components are closed-source and proprietary. Moreover, coverage feedback does not indicate how to effectively extend coverage when performing input mutation due to a priori unknown dependencies between input and code coverage.

We design a novel technique, called *error message guided fuzzing*, which replaces *code coverage* with *error messages* as a feedback mechanism to perform meaningful mutations. The idea of error message guided fuzzing is based on the key observation that during the execution of a WebGL program, when errors are detected, browsers emit meaningful messages to aid developers in correcting their code. Similar to how a programmer fixes bugs in their JavaScript code, the fuzzer mutates the input to get past the error messages or warnings.

We study WebGL error messages and distinguish two types: (1) messages indicating the invalidity of some arguments passed to the current API, and (2) messages indicating the invalidity of the internal WebGL program state caused by unsatisfied dependency between API calls. For type 1 error messages, we compute its target arguments (i.e., which arguments it references); for type 2 error messages, we compute its dependent API set by analyzing which APIs update the internal states on which the checks of the error message is dependent. Using this information, instead of performing random mutation, the fuzzer focuses its mutation on the target argument when a type 1 error message is triggered, or tries to fix the unsatisfied dependency using APIs from its dependent set when a type 2 error message is triggered.

To demonstrate the feasibility of error message guided fuzzing, we evaluate our GLeeFuzz prototype on popular browsers (Chrome, Firefox, and Safari) on both desktop (Linux, Windows, and macOS) and mobile (Android and iOS) OSes. So far we have found 7 vulnerabilities (4 in Chrome, 2 in Safari, and 1 in Firefox). One vulnerability freezes the X-Server on Linux, and another vulnerability enables privilege escalation and possibly remote code execution. GLeeFuzz is available at https://github.com/HexHive/GLeeFuzz. Our main contributions are as follows:

1. A new fuzzing technique that leverages feedback from error messages (instead of code coverage) to guide the fuzzer and to perform meaningful input mutations.
2. A fuzzer for WebGL, GLeeFuzz, that implements our proposed error message guided fuzzing technique.
3. Applying GLeeFuzz to major web browsers across multiple platforms, and responsibly disclosing 7 vulnerabilities so far. We have responsibly reported all our findings to their respective developers.

2 Background

2.1 WebGL Interface

WebGL [28] is a set of JavaScript APIs for 2D and 3D rendering with output to a HTML5 Canvas element. It is designed to closely conform to OpenGL ES API (a subset of OpenGL API tailored to embedded systems) [27], with similar constructs and using the same shading language (GLSL). There are two versions of WebGL specifications: WebGL v1 exposes 163 APIs and WebGL v2 exposes 333 APIs.

WebGL implementations build on the native graphics stack to take advantage of GPU acceleration provided by the user's device. For example, the Chrome browser uses the ANGLE [49] component to transparently translate the WebGL calls to the hardware-supported APIs available on the underlying OS (e.g., OpenGL/OpenGL ES on Unix-like systems and Direct 3D on Windows), which in turn leverages the underlying GPU drivers for GPU accelerated graphics. Note the underlying native graphics stack is usually provided by GPU vendors and mostly in closed-source binary format.

Support for WebGL has gone far beyond browsers in desktops and mobile devices. Application frameworks such as Electron [11] and CORDOVA [52] are built on top of the browser stack; and the GUI of ChromeOS [53] fully builds on the chrome browser. In addition, mobile app frameworks of Android and iOS contain web gadgets (i.e., Android Webview [7] and iOS WKWebView [9]) to allow developers to load and execute web programs in their apps directly. WebGL is by default enabled in all these frameworks and gadgets.

2.2 WebGL Security

The graphics stack has grown over many years and has now reached high feature and code complexity. Historically, applications running on this stack were trusted as they were distributed by well-known software vendors. Moreover, operating systems have traditionally required explicit consent from users to install applications. However, WebGL changes this landscape: it exposes the underlying legacy graphics stack to potential remote adversaries unbeknown to the user. Therefore, its security implication is a great concern for browser vendors. Some vendors (e.g., Microsoft) consider WebGL overly permissive and were highly reluctant in supporting it [6]. The security concern with WebGL is rooted in the complex and potentially vulnerable graphics stacks provided by OEM or third-party GPU vendors, which are not fully prepared to defend against remote attackers.

To mitigate security concerns, multiple dynamic security defense mechanisms are deployed in modern browsers. For example, consider Chrome. First, Control-Flow Integrity

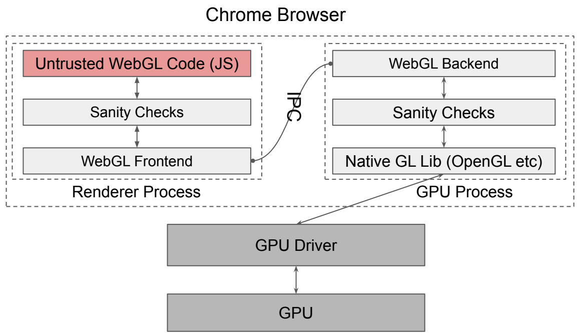


Figure 1: Chrome's WebGL implementation.

(CFI) is deployed to protect C++ virtual calls and C indirect calls [54] from being hijacked. Second, a multi-process design [5] is employed to compartmentalize the untrusted WebGL code in a renderer process and the graphics stack code in a service GPU process (see Figure 1). The renderer process is sandboxed without direct access to system resources, while the GPU process is launched with access to GPU drivers. Moreover, invocation of the underlying graphics stack from the WebGL code (running in the renderer process) is redirected to the GPU process through an IPC mechanism. Third, a diverse set of sanity checks is used to vet the arguments passed to WebGL API calls and internal program state transitions. Most of the sanity checks are derived from the OpenGL ES specification [27]. Other checks include filters introduced by browser vendors in reaction to new vulnerabilities found in the underlying drivers before they are fixed by GPU vendors. A comprehensive study on the types of deployed WebGL checks in Chrome is presented in Milkomeda [60].

Chrome deploys sanity checks in both the renderer and GPU processes. As the checks for defending against known vulnerabilities may be driver-specific and thus need to access the graphics stack, they are deployed only in the GPU process. The checks derived from OpenGL ES specification are deployed in both the renderer process and GPU process. The deployment of checks in the renderer process avoids the performance penalties introduced by IPC with the GPU process. The deployment of checks in the GPU process is based on the consideration that the integrity of the renderer process is uncertain. More specifically, as in Chrome's threat model [5], the execution of the renderer process may be under the control of an adversary due to potential vulnerabilities in it.

Even with these mitigations, the exposed attack surface remains significant. Deployed CFI systems demonstrate poor effectiveness because of the large permissible target sets over-approximated by practical static analysis techniques [14, 36]. Although compartmentalization avoids direct access to the underlying graphics stack, indirect access is still unsafe, as demonstrated by recently disclosed CVEs in Chrome [13, 42]. The deployed checks only serve as a passive approach to defend against known vulnerabilities, not targeting zero-days.

3 Threat Model

Our threat model consists of an adversary trying to gain control of a machine by exploiting a vulnerability in the target's WebGL software stack. To carry out the attack, the adversary sends their malicious WebGL program as a drive-by-download payload [2]. For example, the attacker may encode their exploit as an online advertisement that is then downloaded and executed automatically on the victim machine.

Aligning with assumptions by Google [1, 3], we refer to vulnerabilities as defects that may be exploited by an attacker to compromise the confidentiality, integrity, or availability of the browser or beyond (e.g., the X-Server or desktop environment). Other (less critical) defects are referred to as bugs.

4 Design Overview of GLeeFuzz

WebGL is a complex, high-risk interface as it allows remote adversaries to access a privileged, complex, obscure, and largely untested interface. The risks exposed by WebGL and the severity of potential consequences of attacks motivate our urge to uncover anomalies in the underlying graphics stack that can be directly triggered through WebGL APIs. In particular, we target the components running in the GPU process, including the WebGL backend implementation, and the GL libraries, as well as the GPU driver. As the WebGL implementation adopts a complex multi-process design that involves binary-only components (close-source GL libraries and GPU drivers), it is challenging to apply static analysis across all these different layers. Instead, we leverage fuzz testing to dynamically analyze the security of the WebGL interface.

4.1 Motivation and Intuition

Algorithm 1: Syzkaller Fuzzing algorithm

```
1  $counter \leftarrow 0, corpus \leftarrow \emptyset$
2  while $true$ do
3      if $corpus = \emptyset$ or $counter \ \% \ gen\_period = 0$ then
4          $i \leftarrow gen\_random()$
5      else
6          $i \leftarrow select\_input(corpus)$
7          $random\_mutate(i)$
8      $execute\_input(i)$
9      if $i$ triggered new coverage then
10         $corpus \leftarrow corpus \bigcup \{i\}$
11     $counter \leftarrow counter + 1$
```

A key metric for WebGL fuzzing is to generate inputs that bypass deployed security checks and cover the underlying layers of the software stack. Two key challenges when targeting WebGL are: (C1) generating proper values for API arguments; and (C2) handling dependencies between API calls.

As a program interface, WebGL is similar to system calls. Several fuzzers already target exposed APIs at the OS or hypervisor level [12, 25, 32, 35, 45]. As shown in Algorithm 1, Syzkaller uses two complementary approaches to generate inputs: (i) random new seed generation using `gen_random` in line 4; (ii) random mutation on existing seeds in corpus using `random_mutate` in line 7. Random new seed generation aims to explore code that cannot be covered through mutations of existing seeds while random mutation expands coverage of existing seeds in the corpus. Note that random seed generation and mutation are based on carefully manually crafted templates, which capture the signatures and argument types of system calls. While it is tempting to apply coverage-guided fuzzing on the WebGL interface, it is challenging to collect code coverage. Also, coverage feedback is not indicative on how to mutate existing seeds. These two limitations make coverage-guided fuzzing unsuitable for WebGL fuzzing.

Challenge in collecting code coverage. WebGL code runs in a multi-process, multi-stack environment. As mentioned in Section 2.2, most GL libraries and some GPU drivers are close-source. Coverage collection for binary code is possible through dynamic instrumentation or static binary rewriting, however, they either incur prohibitive runtime overhead or rely on brittle techniques, which are detrimental to the applicability of the technique. Even for open-source browsers and graphic stacks, precise coverage collection remains challenging due to the multi-process design. With this design, the execution of a WebGL API spans across the renderer process that invokes the API and a shared GPU process, which serves the entire browser's rendering and composition through an asynchronous IPC interface used by various browsers and web app renderer processes, resulting in a needle-in-a-haystack scenario where the WebGL-induced coverage is overwhelmed by large amounts of coverage noise. Thus collecting precise code coverage triggered by WebGL APIs is infeasible without a nontrivial engineering effort in overhauling the target.

Coverage-guided feedback is futile. Coverage feedback is not indicative of where and how to mutate the input to effectively expand coverage. In coverage-guided fuzzers like Syzkaller, coverage feedback serves as a metric to judge input quality, based on which the mutating seeds are selected (see `select_input` function in Algorithm 1) in the fuzzing process. For a given input, coverage feedback lacks information about which input parts are erroneous; therefore coverage-guided Syzkaller simply resorts to random mutations.

Algorithm 1 highlights Syzkaller's approach to WebGL fuzzing. To address C1, Syzkaller randomly chooses some APIs from the input and mutates their arguments (as part of `random_mutate` in line 7). To address C2, it randomly adds APIs at random locations to hopefully fix dependencies between APIs. Random mutation is inefficient to expand code coverage as the mutator has no intuition on what to mutate to explore new code areas. Appendix A shows an example.

Error Messages. To aid WebGL program debugging, WebGL implementations provide error messages as feedback when errors are detected in the input. We distinguish two types of such error messages. The first type (type 1) flags issues with the arguments (e.g., an argument is not of the expected format) and the second type (type 2) flags issues with internal state (e.g., an internal variable has not been set up correctly). Section A.1 discusses this in more detail.

Error messages carry human understandable information regarding which part of the input was rejected. This information helps a WebGL developer locate and fix errors. Error messages are manually written by browser developers, are not standardized and differ from browser to browser. In Chrome, the error message also contains a field indicating the type of the error message (in Listing 1 and Listing 2 `GL_INVALID_ENUM` or `GL_INVALID_VALUE` passed to the first argument of `SynthesizeGLError` indicates a type 1 error, while `GL_INVALID_OPERATION` indicates type 2).[1]

Error Message Guided Fuzzing. Considering the challenges in collecting precise code coverage from WebGL and the inefficiency of coverage-guided fuzzing, we design an error message guided fuzzing technique that leverages the error messages as feedback to guide input mutation. More specifically, if a type 1 error message is emitted when executing some API in the input, the fuzzer focuses its mutation on the argument indicated by the message, instead of mutating random arguments of randomly chosen APIs. Similarly, if a type 2 error message is emitted, the fuzzer tries to fix the unsatisfied dependency by choosing an API from a computed dependent API set, instead of randomly.

Compared with coverage-guided fuzzing, a key feature of our error message guided fuzzing technique is that it performs mutation on the parts of the input indicated by the error messages, thus improving fuzzing efficiency by avoiding meaningless mutations in other parts of the API chain (as done by Syzkaller or other error-agnostic approaches).

4.2 Research Challenges and Approaches

Our goal is to build a fuzzer that generates dependent API sequences using error message guided mutation, instead of random approaches used in coverage-guided fuzzers like Syzkaller. To build an error message guided fuzzer, the key challenge is to locate errors in the input when a certain error message is emitted. For a human analyst who is familiar with the WebGL specification, it might be straight-forward to infer such information simply from the error messages, but it is difficult for an automated approach.

A feasible solution is to analyze all possible error messages for each API, infer the target arguments for type 1 error messages and a dependent API set for type 2 error messages beforehand. During fuzzing, the fuzzer programmatically applies mutations on the target arguments or using the dependent API set when error messages are emitted. To construct an error message guided fuzzing technique using this solution, the following research problems need to be addressed.

C1. Computing possible error messages: computing all possible error messages for each API is the first step. We leverage static analysis (Section 5.1) to identify all call statements to `SynthesizeGLError` function (the function for emitting graphics error messages in Chrome) that are reachable from the entry of the native function corresponding to the WebGL API under analysis. This

[1] This field is not available in Firefox, but as we show next, we focus on analyzing error messages from Chrome and then generalize to other browsers.

gives us a set of call statements from which we can extract all error messages.

C2. Inferring target arguments for type 1 error messages: inferring target arguments is the process of identifying the arguments whose invalidity is indicated by an error message, so the fuzzer can focus its mutation on the identified arguments of the API emitting the message, instead of randomly chosen arguments of random APIs. We identify the target arguments by performing a backward taint analysis on the path condition of the call statement emitting the error message (Section 5.2).

C3. Inferring the dependent API set for type 2 error messages: the dependent API set is an approximated set of APIs that may fix the invalid dependency indicated by an error message. GLeeFuzz uses the computed dependent APIs, instead of randomly chosen ones, to fix dependency when the error message is emitted. To identify the dependent APIs, we look for any API that updates the internal variables of the path condition of the call statement emitting the error message (Section 5.3).

5 GLeeFuzz Design

Figure 2 depicts the overall workflow of GLeeFuzz. From a high-level point of view, GLeeFuzz is divided into two phases: a pre-processing phase and a fuzzing phase. In the pre-processing phase, all error messages that may be emitted by each API are computed (Section 5.1). For each error message that may be emitted by an API, depending on the type of the message, either its target arguments (Section 5.2) or the dependent API set (Section 5.3) are computed. In the fuzzing phase, GLeeFuzz iteratively generates random inputs, executes the generated inputs on target browsers, takes the error messages from Chrome browser as feedback, and performs error message guided mutation on the fuzzing inputs based on computed results from the pre-processing stage.

GLeeFuzz also features a multi-browser execution technique, which enables us to apply our error message guided fuzzing technique based on analysis on the source code of Chrome to other browsers (Section 5.6).

5.1 Computing Error Messages

To build our error message guided fuzzing technique, the first step is to compute all error messages that may be emitted by an API at runtime. To this end, there are two options: dynamic analysis and static analysis. Considering that it is difficult to trigger a complete set of error messages using dynamic analysis, we use static analysis in this step.

Chrome emits error messages using `SynthesizeGLError`, thus all error messages of an API can be enumerated by identifying the calls to `SynthesizeGLError` reachable from the entry point of the native function of a WebGL API. Note WebGL APIs are implemented as C++ methods and accessed using JavaScript. We refer to these C++ methods as native functions or native implementation of WebGL APIs.

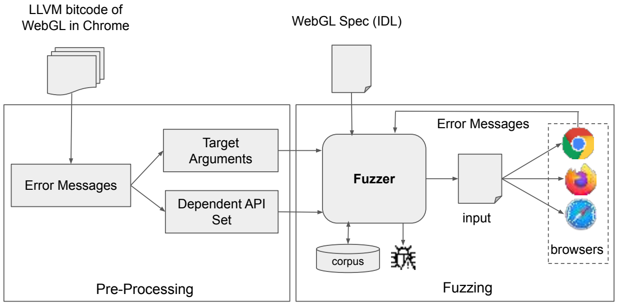


Figure 2: GLeeFuzz design. In the pre-processing stage, GLeeFuzz identifies error messages, and computes their target arguments and the dependent API set, which the fuzzer uses for error message guided input mutations.

To search all call statements to `SynthesizeGLError`, we use a worklist-based algorithm to traverse all code paths in the control-flow graph from the entry point of the native function of the API (see Section 6). This gives us a list of call statements, from which we can extract the error message and its type. Other analysis in the following subsections are based on these identified call statements.

We conduct static analysis on Chrome's source code. These results guide the fuzzing of other WebGL-compliant browsers using our multi-browser execution. As the WebGL APIs and their error conditions are standardized, repeating static analysis on other browsers would not add new information. We discuss the multi-browser execution in Section 5.6.

5.2 Inferring Target Arguments

For type 1 error messages, GLeeFuzz analyzes the target arguments of the corresponding API calls during the preprocessing phase, and focuses mutation on the target arguments, if the error message is triggered by the input.

To infer the target arguments, a naive approach is to *manually* analyze the messages. This approach is clearly not scalable as the total number of log messages is substantial (more than 3,000 messages in Chrome, see Section 7.1), and it requires nontrivial effort in understanding the internal state machine of the WebGL specification. Another plausible approach we tried is to use natural language analysis on the log messages. However, as our efforts showed, this approach still needs large efforts to label log messages as there are no ground facts to train the data set.

We observe that the call statement of a type 1 error message is conditioned on sanity checks on the values that are relevant to its target arguments. For example, in the native function of `bufferData` shown in Listing 1, the call statement of the "invalid target" message is guarded by a sanity check on the `target` argument. Note that we simplified the native function shown in Listing 1. In the actual code, the argument is passed to a helper function to perform the sanity check.

Based on this observation, we use a backward taint analysis to infer the target argument. More specifically, we search in the taint sources of the path condition (of the call statement of the error message) for arguments of the native function of the API under analysis. The algorithm and implementation details are presented in Section 6.

Algorithm 2: Inferring Target Arguments

```
Input: API: native function of the API under analysis
Input: PC: path condition of the error emitting
       statement
Output: ARGS: detected target arguments
$VFG \leftarrow build\_vfg(API)$
$pc \leftarrow get\_node\_from\_VFG(PC)$
$work\_list \leftarrow \{pc\}$
$visited \leftarrow \{\}$
while $work\_list \neq \varnothing$ do
    $v \leftarrow work\_list.pop()$
    if $v \notin visited$ then
        for $p \in v.pred$ do
            $work\_list \leftarrow \{p\} \cup work\_list$
        if $v$ is an argument of $API$ then
            $ARGS \leftarrow ARGS \cup \{arg\,\#\,of\ v\}$
        $visited \leftarrow \{v\} \cup visited$
```

5.3 Inferring Dependent API Set

For type 2 error messages, GLeeFuzz pre-computes its dependent API set in the pre-processing phase. In the fuzzing phase, when mutating a given input, the fuzzer tries to fix invalid dependencies using the dependent API set if a type 2 error message is triggered by an API in the input.

We observe that call statements of type 2 error messages are guarded by checks on some internal variable, which is updated by the execution of its dependent APIs. For example, the call statement for the message “no valid shader program in use” in Listing 2 is nested in an `if` statement conditioned on a check on an internal variable `currnet_program_`, which is updated by the native function of `useProgram` in Listing 3.

Accordingly, in GLeeFuzz, we identify the dependent API set of a type 2 error message as those updating the dependent internal variables of the call statement emitting the error message, which entails solving two sub-problems: (P1) analyzing the internal variables an API may update; (P2) analyzing the internal variables an error message is dependent on.

P1 is close to the problem of computing error messages discussed in Section 5.1, which can be solved by searching for all update (assign) statements to WebGL internal variables in all code paths using a classical worklist-based algorithm. P2 is close to the problem of computing target arguments in Section 5.2, which can be solved by searching for internal variables in the taint sources of the path condition of the call statement emitting the error message. Here we use the following heuristic rule to identify internal variables of WebGL: all variables of types defined in the WebGL namespace or fields of such variables are considered as WebGL internal variables.

Based on solutions to P1 and P2, we compute the dependent API set D_M of error message M in 3 steps. (1) For each API X, we compute the set of internal variables U_X that X updates, by solving P1. (2) We compute the set of internal variables DI_M the call statement of M is control dependent on, by solving P2. (3) We iterate through all APIs and add API X to D_M if DI_M is a subset of U_X; i.e., $DI_M \subset U_X$.

5.4 Collecting Error Messages as Feedback

As shown in Figure 2, in the fuzzing phase, GLeeFuzz relies on the error messages emitted by executed APIs to perform error message guided mutations based on on the results computed in Section 5.2 and Section 5.3. To this end, the fuzzer needs to access the error messages emitted by APIs.

The error messages emitted by WebGL APIs are written to the browser console, which is not directly accessible from the fuzzer code. To access the error messages from the fuzzer code, one naive option is to save the output of the browser console to a file and extract the error messages from it. This approach is used in a related work [4] to extract log messages emitted by some APIs from the system log. As the browser console contains a lot of unrelated messages concurrently emitted by other modules in the browser, it is challenging to identify which message was emitted by which API during the execution of an WebGL program.

GLeeFuzz relies on precise error message feedback from the target (e.g., which error message was triggered by which API). We therefore add a new JavaScript API to Chrome that allows the fuzzer to retrieve error message triggered by the previously executed API. Using multi-browser execution in the fuzzing phase, GLeeFuzz relies on the customized Chrome browser to perform error message guided mutation, and tests other browsers by replaying mutated inputs.

5.5 Error Message Guided Mutation

Test case generation in GLeeFuzz is inspired by Syzkaller. Similar to Syzkaller, GLeeFuzz generates random inputs and mutates seeds chosen from a corpus following the API signatures and their argument types defined in the specification. However, unlike Syzkaller, API signatures and argument types are parsed **automatically** from the specification where these API information are defined in the WebGL IDL (Interface Definition Language) sources [29, 30].

The major difference to Syzkaller is that instead of using random mutation on seeds chosen from the corpus, GLeeFuzz leverages error messages to infer erroneous parameters. This information subsequently guides input mutation. As code coverage collection is challenging in WebGL, instead of using code coverage, we use the number of successfully executed APIs and the number of unique error messages as a metric to evaluate the quality of an input. Based on our study of the source code, the execution of a WebGL API returns with an error message if the arguments or the internal state fail to pass sanity checks. Although we cannot corroborate that returning without error message always indicates successful execution, it is reasonable to assume that it has bypassed deployed sanity checks and considered to be executed successfully.

Given an input consisting of a list of APIs, GLeeFuzz iteratively applies mutation on the input based on the error messages triggered by each API in the input. We now detail the mutation rules for each type of error message.

Mutating target arguments. During fuzzing, when a type 1 error message is triggered, our mutation **focuses** on the target argument of the API by which the error message was emitted. The mutation on the identified target argument is random in general, with two optimizations: (1) when mutating argument of `enum` type, as the value space is small, the chance of generating the same value (which results in meaningless mutation) using a random approach is high, so we always generate a value that is different from the current one. (2) Certain integer arguments are used as indexes to access WebGL objects in memory buffers. Sanity check failures on this type of argument result in error messages complaining that the value is too large or too small (e.g., “xxx is out of bound”). When mutating this type of argument, we introduce heuristics using a **manually** curated list of 12 keywords (e.g., “too large”, “too small”), to detect messages of the form “Parameter XXX is too large” or “Parameter XXX is too small”. We generate a random index number that is either smaller or larger than the current one as indicated by the error messages.

Fixing API dependency. When a type 2 error message is triggered, GLeeFuzz mutates the inputs based on the result computed in Section 5.3. To fix the dependency using the dependent API set, GLeeFuzz checks the APIs before the API emits the error message, and performs mutation on the input based on the following heuristic rules. (1) If none of the dependent APIs are used before the current API, we choose a random dependent API, randomly generate its argument values, and then insert it before the error-emitting API. The inserted API may fix the dependency by properly setting up the internal variables for the next test case or future test cases. If not, we continue this process in the iteration. (2) If any one of the dependent APIs has already been used before the error message emitting API, but its execution failed in the previous execution with its own error message emitted (which indicates that fixing of dependency was not attempted at all), we do not perform any additional mutation to the input because the prior mutations based on the feedback of the dependent APIs may have already fixed the erroneous inputs and its successful execution may fix the dependency in the current mutant. (3) If any dependent API have been used and its execution succeeded (without triggering any error message), another randomly chosen dependent API is inserted before the current API with the intuition that the new API might fix the dependency in the resulting mutants, because the dependent API that is already in the input was not (yet) able to fix the dependency.

Our input generation only leverages the dependencies of error emitting statements to infer “where to mutate” (i.e., target argument and dependent API) except for the mentioned optimizations. As error messages carry human-understandable information regarding *how the input is invalid* they indicate “how to mutate”. While it is tempting to perform smarter mutations based on the meaning of error messages, our attempt was fruitless. Training NLP models to infer better mutators failed as the models lacked domain knowledge of WebGL. In our current implementation, we only employ the mentioned optimizations based on our study on the error messages. Thus, “interpreting” more informative error messages may potentially improve the effectiveness of these optimization.

5.6 Multi-Browser Execution

GLeeFuzz relies on static analysis on the source code of WebGL implementation in the Chrome browser and collection of error messages as feedback. These requirements make it difficult to apply our error message guided fuzzing on closed-source browsers (e.g., Safari).

The WebGL specification defines error scenarios, such as invalid inputs and disallowed states [29]. Since WebGL implementations in different browsers follow the same standardized specification, GLeeFuzz uses multi-browser execution to apply fuzzing, guided by error messages collected from our instrumented Chrome. More specifically, in addition to executing newly generated inputs on our customized Chrome (which allows the fuzzer to collect error message, see Section 5.4), GLeeFuzz relays the same inputs to other browsers (e.g., Firefox or Safari). As shown in Figure 2, input mutation is based on the error message collected from the customized Chrome, whereas, the generated inputs are replayed on all browsers instances. As a result, GLeeFuzz allows for testing WebGL implementations in different browsers based on the analysis on the source code of Chrome—implementing a form of cross-pollination across different targets.

6 Implementation Details

The implementation of GLeeFuzz is divided into two parts: (1) a set of static analysis tools used in the pre-processing phase; (2) a dynamic fuzzer used in the fuzzing phase. In this section we present their technical details.

6.1 Static Analysis

The static analysis described in Section 5 is performed on the LLVM bitcode of WebGL implementation in Chrome. To get the IR code of the WebGL component in Chrome, we modified the Chrome build system to generate LLVM bitcode using wllvm [55].

Our static analysis leverages the SVF framework [51]. First, we use SVF to build the inter-procedural control flow graph (ICFG) for native functions of WebGL APIs, which we use to determine emitted error messages (see Section 5.1) and dependent API calls (see Section 5.3). During ICFG construction, we discovered unresolved virtual calls. Without knowing the target, static analysis cannot compute error message emitted in callees of virtual calls. The callee of a virtual call is loaded from the vtable of the API's C++ class. We statically resolve the callees of a virtual call to the set of possible implementations, i.e., the target class and its subclasses. This callee set is an over-approximation, thus the computed error messages may include ones that cannot be triggered at runtime. However, as the error messages merely guide the input mutation when they are triggered, the imprecision in callee resolution does not impact fuzzer performance. Second, to perform taint analysis as mentioned in Section 5.2 and Section 5.3, we use SVF to build a value flow graph (VFG), which is a directed graph capturing the tainted relationship of variables and expressions in a program.

Based on the computed ICFG and VFG, we implement our static analyses as a graph traversal from the root node to search for nodes satisfying certain conditions based on a worklist algorithm. Algorithm 2 shows the implementation of a backward taint analysis based on traversing the VFG to search for the target arguments of an error message.

6.2 Fuzzer

Due to the uniqueness of our proposed approach (error message guided fuzzing) and target (to the best of our knowledge, this is the first work targeting WebGL), there is no existing baseline for reuse or customization, we implemented the fuzzer from scratch. Input generation described in Section 5.5 is written in Python. To support executing inputs (i.e., sequences of WebGL API calls) in browsers, we implemented an executor module consisting of an HTML page and a Javascript function which parses and executes inputs generated by the fuzzer. To execute an input, GLeeFuzz loads the executor onto a target browser, and calls the Javascript function (passing the input as argument) using the standardized WebDriver [57] interface in form of RPC. The implementation of our execution engine is based on Selenium [50], a Python wrapper of the WebDriver interface. To test browsers on Android and iOS devices via the WebDriver interface, Appium [48] is used as a proxy between our fuzzer and the browsers running on the device side. As discussed in Section 5.4, GLeeFuzz relies on a customized Chrome for error message collection. Our executor module uses the additional API to collect error messages if available.

7 Evaluation

We conduct extensive experiments to evaluate our GLeeFuzz prototype on a broad set of GPUs ranging from Intel, NVIDIA, and AMD desktop GPUs to mobile systems on ARM such as Adreno, Mali, or PowerVR along with the Apple GPU. At the same time we evaluate across five operating systems and three widely used browsers (Chrome, Firefox and Safari) on both desktop and mobile systems on x86 and ARM. Table 1 shows the different configurations.

7.1 Static Analysis

In this section, we present the total number of error messages identified by our static analysis and the runtime of the preprocessing phase. The experiments in this section were performed on Ubuntu 20.04 LTS running on an x86-64 desktop equipped with an Intel i7-8086K CPU and 32GB of memory.

Error message distribution. Table 2 summarizes the number of error messages detected by our static analysis tool. Column 2 and 3 show the total number of type 1 and type 2 error messages in all APIs of WebGL v1 and v2 interface implementations. As we over-approximate the callees of virtual calls (see Section 6.1) the detected error messages may contain false positives. In total there are 998 and 2934 error messages in WebGL v1 and v2 respectively. This large number calls for an automatic approach to identify the error messages.

Pre-processing time. Analyzing 12.3 MB of LLVM bitcode for the WebGL implementation in Chrome is divided into the following two steps: (1) construction of ICFG and VFG and (2) static analysis on the ICFG and VFG to compute error messages and infer target arguments or dependent API set. We measured the total time spent on each step when analyzing the native implementations of all APIs in WebGL v1 and v2. The construction of ICFG and VFG takes 20.4 seconds for both versions, as the process is the same (SVF builds ICFG and VFG for all functions in the input). The static analysis on the ICFG and VFG takes 40.7 and 120.1 seconds respectively.

7.2 Effectiveness of Error Messages

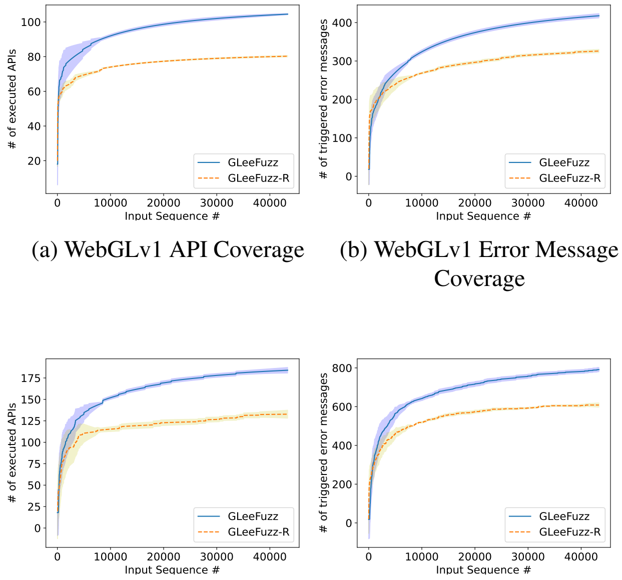
To validate the relative effectiveness of our error message guided fuzzing technique, we implemented a version of GLeeFuzz, referred to as GLeeFuzz-R, in which inputs are mutated randomly, following the Syzkaller strategy (see Section 4.1). A random mutating fuzzer for WebGL did not exist, and it involves a non-trivial effort to overhaul a fuzzer designed for

OS	Platform	Browsers	GPU
Windows	x86-64 Desktop	Chrome, Firefox	Intel, NVIDIA, AMD
Linux	x86-64 Desktop	Chrome, Firefox	Intel, NVIDIA, AMD
macOS	MacMini	Chrome, Firefox, Safari	Intel, Apple M1
Android	Android Smartphones	Chrome	Adreno, Mali, PowerVR
iOS	iPhone	Safari	PowerVR, Apple GPU

Table 1: Selected targets used in GLeeFuzz evaluation. On x86-64 platforms, we evaluate Chrome and Firefox with graphic stacks of Intel UHD 630, NVIDIA GeForce GTX 980 and AMD Radeon RX 550 on Windows 10 and Ubuntu 18.04. On Mac mini, we evaluate Chrome, Firefox and Safari with graphic stacks of Intel UHD 630 and Apple M1 on macOS 11.6. On OnePlus 9 Pro, OPPO Reno2 and OPPO Reno5, we evaluate Chrome with graphic stacks of Qualcomm Adreno 660, PowerVR 9446 and Mali G77 on Android 11. On iPhone 6s plus and iPhone X, we evaluate the Safari with stack of PowerVR 7660 and Apple GPU on iOS 11.4.

Version	Type 1	Type 2	Total #
WebGL v1	706	292	998
WebGL v2	2028	906	2934

Table 2: The total number of error messages detected in Chrome's WebGL implementation.



(a) WebGLv1 API Coverage

(b) WebGLv1 Error Message Coverage

(c) WebGLv2 API Coverage

(d) WebGLv2 Error Message Coverage

Figure 3: Coverage comparison between GLeeFuzz and GLeeFuzz-R in metrics of the number of executed APIs and the number of triggered error messages.

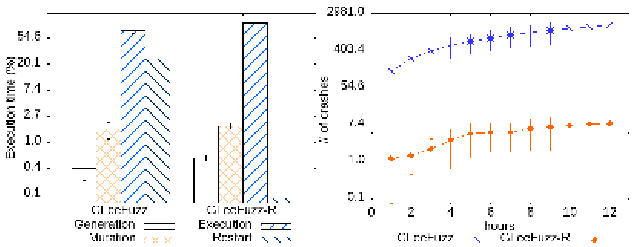
another target to fuzz WebGL. Therefore, we implement the random mutation approach on top of our system.

Due to the challenge in collecting precise coverage information from the targets (as mentioned in Section 4.1 and Section 5.5), we use the number of successfully executed APIs (i.e. APIs that return successfully without triggering any error messages) and the number of error messages as an alternative metric for coverage in comparison.

Fuzzers must restart the browser whenever a tab crash is triggered. This is a heavy-weight operation, introducing noise into the measurement process. To compare fuzzers, we record generated inputs over time and replay non-crashing inputs in sequence to measure time. We run each evaluation five times for 12 hours, each targeting Chrome on an x86-64 desktop running Ubuntu 18.04 with 32GB of memory. Since we only select non-crashing inputs to minimize noise, fuzzing time is spent entirely on executing the test cases, and none on browser restarts. We break down the execution time including the cost of browser restarts, in Section 7.3.

Figure 3 shows the progression of the number of successfully executed APIs and triggered error messages in WebGLv1 and WebGLv2 over five runs, the shaded area shows the standard deviation. From Figure 3, we learn that GLeeFuzz outperforms GLeeFuzz-R in terms of both the number of successfully executed APIs and triggered error messages. Starting from an empty corpus, the inputs generated by both GLeeFuzz and GLeeFuzz-R are able to extend coverage (number of logs and APIs, as defined in Section 5.5), by triggering more error messages and bypassing the sanity checks to make more successful API execution. However, under the guidance of emitted error messages, GLeeFuzz was able to expand coverage at a much faster pace, by generating inputs triggering more error messages and more successfully executed APIs.

Table 3 summarizes the total number of triggered error messages and successfully executed APIs averaged across 5 runs. In 12-hour evaluations, GLeeFuzz triggered 1,208 (417 in v1 and 791 in v2) error messages and successfully executed 288 (104 in v1 and 184 in v2) APIs, while GLeeFuzz-R triggered 933 (325 in v1 and 608 in v2) error messages and successfully executed 213 (81 in v1 and 132 in v2) APIs. Compared to GLeeFuzz-R, GLeeFuzz triggers 28.3% more error messages and successfully executes 28.4% more APIs (by generating inputs that bypass the sanity checks) in WebGLv1, while in



(a) Execution Time Breakdown (b) Number of Crashes

Figure 4: Execution time and crash comparison between GLeeFuzz and GLeeFuzz-R. The y-axes show log scale.

WebGLv2 GLeeFuzz triggers 30.0% more error messages and successfully executes 39.3% more APIs.

On average, there are 581 and 2,134 error messages not triggered in WebGL v1 and v2 respectively. The number of untriggered error messages is the difference between all detected (Table 2) and triggered (Table 3) error messages. Untriggered fall into 3 categories: (1) error messages that cannot be triggered at all due to the overapproximated virtual callees; (2) error messages detected in APIs that were not tested by the fuzzer; (3) error messages detected in APIs that were tested by the fuzzer, but its error emitting statement was not executed because the arguments or internal state failed to bypass earlier sanity checks.

7.3 Breakdown of Execution Time

To demonstrate the performance of GLeeFuzz, we have broken down the execution time into launching a fuzzing target, generating a test program, executing a program on the target, and mutating the program. GLeeFuzz mutates the program based on the collected error messages, whereas GLeeFuzz-R simply mutates the program randomly. We conduct this experiment for five times, 12 hours each, targeting Chrome on x86-64 desktops running Ubuntu 18.04 with 16GB of memory. The results are shown in Figure 4.

Figure 4a shows the average percentage of execution time spent on each of the fuzzing stages, and Figure 4b shows the average number of crashes (over five runs, with standard deviation) in our 12-hour experiments. Figure 4a, we observe that, on average, GLeeFuzz spends 217× wall time on launching the browser than GLeeFuzz-R, which is explained by 202× counts of browser crashes. The high number of crashes incurred by GLeeFuzz showcases the effectiveness of our error message guided mutation. The breakdown of execution time also shows that GLeeFuzz's test program generation and mutation do not incur significant overhead. Both GLeeFuzz and GLeeFuzz-R, on average, spend about 2% of their time generating and mutating inputs.

	GLeeFuzz-R		GLeeFuzz			
WebGL	err msg	API	err msg	Δ	API	Δ
v1	325	81	417	28%	104	28%
v2	608	132	791	30%	184	39%

Table 3: Average number of successfully executed APIs and triggered error messages and by GLeeFuzz-R and GLeeFuzz.

> Our error message guided fuzzing technique improves fuzzing performance over random mutation as used in coverage-guided fuzzers like Syzkaller. Focusing mutation on parts of the input indicated by error messages, GLeeFuzz reaches higher coverage faster (in terms of the number of triggered error messages and APIs that returned successfully) and triggers a significantly higher number of crashes than a random fuzzer.

7.4 New Vulnerability Findings

We evaluate GLeeFuzz's effectiveness in discovering previously unknown vulnerabilities in commodity browsers. We run GLeeFuzz on the most widely used browsers (e.g., Chrome, Firefox, or Safari) on both desktop (Windows, MacOS, Linux) and mobile (Android and iOS) OSes using the setup shown in Table 1. Since an implementation of the WebGL software stack consists of both the browser code and underlying native graphic stack, it may form a diverse set of combinations with a lot of options. Therefore, we focus our selection of targets on the most widely used ones, selecting common hardware components (e.g., Intel, NVIDIA, AMD, or Apple). Finding vulnerabilities in such common environments is harder as the software stacks are generally better tested than those of unknown niche hardware. For instance, there are only 87 CVEs assigned across all WebGL implementations since WebGL was first available in 2011 [44].

Our fuzzing platform has been running intermittently for more than two months and has triggered a large amount of crashes. Crash analysis and bug deduplication is challenging and we are continuously working on triaging the crashes. So far we have reduced over a thousand crash detections to identify and reproduce 7 previously **unknown** vulnerabilities (shown in Table 4): 2 vulnerabilities in Safari, 4 vulnerabilities in Chrome and 1 vulnerability in Firefox.

Out of these 7 vulnerabilities, two were triggered in the GPU driver in the kernel space: one in Chrome on Linux with an Intel GPU, the other in Safari iOS on a PowerVR GPU. The inputs generated by our fuzzer triggered GPU resets. Although the GPU reset on iOS is properly handled and only interrupts systems graphics, the reset triggered on the Intel GPU cannot be handled by the X-Server running on Ubuntu, resulting in a system freeze. After triggering this vulnerability, a hard reboot (shutdown and then boot) is required to restore GPU functionality. We suspect that it compromises state in the GPU firmware that is persistent cross soft reboots. In our evaluation, we realized that once this vulnerability is triggered, the fuzzer remains blocked and cannot proceed without restart. This issue is also hardware dependent, although we can reproduce it with the fuzzer generated input on our machine, it cannot be reproduced by the Chrome developers on a similar but slightly different model of the GPU. After we reported it to the developers, it was fixed in recent versions of Ubuntu.

GLeeFuzz automatically recovers from browser crashes and system reboots. However, minimal manual intervention is required when the machine needs a hard reset. On average, GLeeFuzz takes 2.98 seconds to relaunch a browser after a browser crash, and 36.40 seconds to restart the fuzzing session after a system reboot.

> Discovering two kernel-level vulnerabilities demonstrates GLeeFuzz's capability in generating inputs that can bypass all sanity checks deployed in the upper layers and trigger exceptions in the lowest level of the software stack.

In addition, GLeeFuzz finds three vulnerabilities in Chrome: one assertion failure in the renderer process, and two vulnerabilities that crash the browser's privileged GPU process due to memory corruptions and subsequent assertion failures. Assertions are written by developers in the source code to check for preconditions expected by the code following it. In debug builds, assertion statements are enabled and assertion failures terminate the process. In release builds assertions are disabled, thus executing the code beyond the assertion statement. The crash in the renderer process is raised by an assertion placed on the link counter of a shader program to prevent it from being double-linked (by calling `linkProgram`). The impact of the assertion failure in the renderer process remains opaque in release builds; consequently, the developers evaluated this issue as a low-severity security bug. The other two vulnerabilities in the GPU process have confirmed security impact since the GPU process is shared among the browser components and has access to GPU drivers. One vulnerability is a null pointer deference followed by multiple assertion failures in the restarted GPU process, allowing denial of service attacks. Another vulnerability is memory corruption in the GPU process, potentially allowing remote code execution. These two vulnerabilities in the GPU process remain unpatched as of this writing. One (memory corruption in the GPU process) has received high severity ratings.

The Firefox vulnerability triggers an unbounded heap memory allocation. We have received a security rating from Mozilla, but this bug remains under embargo and is unpatched, awaiting final confirmation. The reason for the tab crash in Safari remains unclear due to the lack of source code. Although we have not received any feedback from Apple for our report, the vulnerability has been patched in recent versions.

> Finding vulnerabilities on non-Chrome browsers demonstrates the effectiveness of our multi-browser execution technique mentioned in Section 5.6. Prior to GLeeFuzz, our evaluation of these browsers using a naive fuzzing technique did not find any of these vulnerabilities.

7.5 Finding Known Vulnerabilities

To validate GLeeFuzz's relative effectiveness in bug finding capability in comparison to random mutation, we evaluate it against a set of known ground truth vulnerabilities reported in WebGL of Chrome in the past years.

For this evaluation we created a ground truth benchmark by porting known bugs to a recent version of Chrome. This allows reproduction of multiple bugs on a single system and circumvents dependencies issues due to outdated libraries in older versions of the browser. We selected all bugs in the browser layer reported between 2012 and 2021 that are labeled as security issues. Table 5 shows the list of the selected known vulnerabilities. We studied their patches and found that all these selected vulnerabilities are fixed by introducing new security checks before the execution reaches the crash site. To "reintroduce" these vulnerabilities into a recent version of Chrome, we extract the triggering conditions of these issues from their respective security checks, and add conditional assertions to signal that an issue is triggered.

The evaluation on this benchmark was performed on an x86-64 Desktop running Ubuntu 18.04 with 32G memory. For comparison, on a version of Chrome with our selected security bugs ported, we run GLeeFuzz and GLeeFuzz-R 5 times, each lasting 12 hours. Table 6 shows the detected vulnerabilities for each run. In total, GLeeFuzz detected 3 of the issues (145544, 765469, and 784183), while GLeeFuzz-R only detected one (765469). Issue 765469, which was detected by both GLeeFuzz and GLeeFuzz-R, was detected in 4 runs of GLeeFuzz, while in only 2 runs of GLeeFuzz-R.

We investigated the triggering conditions of the vulnerabilities that were not detected by either GLeeFuzz or GLeeFuzz-R. Issue 848914 requires a WebGL v2 object that is then passed to a WebGLv1 context [19]. In the implementation level, as WebGL v1 and v2 APIs are exported through different Javascript objects and it is only allowed to export either a v1 or v2 object on one HTML Canvas element, our fuzzer currently does not generate cross-version fuzzing inputs and this issue therefore remains undetected. Issue 1149204 can only be triggered in the constructor of a WebGL context object (a Javascript object through which WebGL APIs are exported) with certain incompatible parameters [22], and issue 1219886 can only be triggered from an exported API of a WebGL extension object. Our fuzzer detected neither as it only tests the

Bug Descriptions	GPU	Platform	Browser	Bug Location	Severity
GPU hang	Apple GPU	iOS	Safari	GPU Driver	*Not set*
GPU hang; X-Server freeze	Intel	Ubuntu	Chrome	GPU Driver	Medium
Nullptr dereference in GPU process	N/A	N/A	Chrome	Browser	*Not set*
Memory corruption in GPU process	N/A	N/A	Chrome	Browser	High
Assertion failure	N/A	N/A	Chrome	Browser	Low
OS memory leak	Intel	macOS; Ubuntu	Firefox	Browser	Low
Tab crash	N/A	macOS	Safari	Browser	*Not set*

Table 4: New Vulnerabilities found by GLeeFuzz

Issue	Bug Description
774174 [17]	Fail to check for a disallowed GL value type in the argument of TexSubImage2D
145544 [15]	Fail to check for integer overflow in the arguments of TexSubImage2D
1219886 [23]	The check for out-of-bound array access is incomplete without considering drawcount+offset
848914 [19]	WebGL context version confusion leads to operations on invalid context
784183 [18]	Insufficient security check leads to integer overflow in ValidateTexImageSubRectangle
1149204 [22]	WebGL does not support offscreen canvas, but it can still happen due to the lack of validation
765469 [16]	Fail to check for a disallowed GL format and type combination in WebGLImageConversion

Table 5: List of known and fixed WebGL vulnerabilities used in our evaluation.

APIs exported through a WebGL context object.

	GLeeFuzz-R					GLeeFuzz				
Issue	1	2	3	4	5	1	2	3	4	5
774174										
145544						✓			✓	
1219886										
848914										
784183							✓		✓	
1149204										
765469		✓	✓			✓	✓	✓		✓

Table 6: Comparison of vulnerability detection between GLeeFuzz GLeeFuzz-R on our manually collected benchmark.

While issue 774174 can be triggered from `texImage2D` API, its condition is guarded by around 10 sanity checks on input. Our investigation shows that in each of the 5 runs, there are some generated inputs by GLeeFuzz containing calls to `texImage2D`, with arguments that bypass some sanity checks, but unfortunately not all of them within 12 hours.

The ground truth evaluation results show that error message guided fuzzing performs better in finding vulnerabilities than random mutation approach used in coverage-guided fuzzers.

8 Related Work

8.1 Runtime Mitigations

The graphics stack is a lucrative target of attack since it often runs in a privileged context. To protect the graphics stack, many defense mechanisms are proposed to detect attacks or reduce the risks of attacks. Milkomeda [60] leverages existing browser security checks to safeguard the mobile graphics stack. To compartmentalize the graphics stack, Sugar [59] runs the WebGL graphics stack in an isolated environment leveraging GPU virtualization. To further protect the rendered content, recent related work runs the graphic stack in trusted environments. E.g., VButton [38], TruZ-View [61] and Rushmore [46] produce secure framebuffers and display secure content with a self-contained graphics stack running in the secure world of ARM TrustZone.

Defense mechanisms protect the users from potential exploits at runtime, while GLeeFuzz complements the defense mechanisms by finding and fixing vulnerabilities.

8.2 Interface Fuzzing

Fuzzing plays an important role in unveiling software and hardware vulnerabilities. Recent work in fuzzing interfaces employs diverse program analysis techniques and has shown promising results. E.g., to apply coverage guided fuzzing on system calls, kAFL [56] leverages a hypervisor and Intel PT (Processor Trace) to obtain code coverage from an unmodified OS. Syzkaller [25] applies generational fuzzing in system calls based on manually crafted interface specification, its input generation is also guided by code coverage which is obtained by static instrumentation (through kCov). To address explicit dependencies among system calls, IMF [32] tracks API return values and the parameters passed to other APIs in strace logs. MoonShine [45] improves syzkaller's input generation distilling seeds from real-world programs. FuzzGen [34] and APICRAFT [62] automate fuzz driver generation by static/dynamic data flow analysis on the target. USBFuzz [47] and FuzzUSB [37] target I/O interface between drivers and their hardware devices, addressing the challenge of injecting random device input through device emulation. GraphicsFuzz [10,24] is an automated fuzzing system that targets shader compilers by generating metamorphic inputs, i.e., semantically equivalent shader programs. Aafer et al. [4] analyze input validation logs to infer specifications of Android media APIs in TV Boxes and to guide input generation.

Compared to these works, GLeeFuzz targets WebGL (where code coverage collection is hard), leveraging our novel error messages guided fuzzing technique. In contrast to coverage guided fuzzing techniques, error message guided fuzzing can perform mutations on input parts indicated by the feedback and avoid meaningless (i.e., low quality) mutations.

8.3 Browser Fuzzers

Web browsers are an attractive target with multiple attack surfaces, for which there are varying fuzzing tools, each targeting a different part with their own approach.

Domato [26] and BFuzz [39] target HTML parsing and generate input from templates. FreeDom [58] further applies coverage guided fuzzing to test DOM parsing. Funfuzz [43] targets the Javascipt engine with generational fuzzing (based on manually crafted templates). Grammarinator [33] tests JavaScript engines by generating inputs following the grammars of the JavaScipt language. Favocado [8] targets the PDF, MOJO, and DOM binding layer in the JavaScript engine and generates inputs based on semantic API information.

Among these tools, Favocado is most relevant to GLeeFuzz. GLeeFuzz also generates inputs following the API specification. However, GLeeFuzz uses its novel error message guided fuzzing to perform tailored input mutation.

9 Discussion and Future Work

Improving crash detection. In the current implementation, our crash detection relies on feedback from the WebDriver. Sometimes, as shown by our evaluation, the WebDriver feedback is insufficient or incomplete, e.g., when a tab crashes it only returns "tab crash". A simple extension is to obtain browser logs through side channels, such as redirecting stderr over the network. This approach does not apply to all devices because some operating systems, such as iOS, do not allow log redirection. Without access to more information about the crash such as a stack trace or at least crash location, the fuzzer cannot tell if the detected crash is a duplicate or not. De-duplication requires manual work, and thus makes bug triaging harder. In the future, we plan to make the stack trace accessible in the WebDriver and implement crash de-duplication based on the stack traces of crashes.

Extending error-guided fuzzing technique to other targets. The design and implementation of error message guided fuzzing presented in this work targets the Chrome WebGL implementation. However, our technique applies to a much wider range of targets, since it only relies on indicative feedback which pinpoints faults in the input. To facilitate application using this technique, on one hand, without aggressive code instrumentation for code coverage, developers can include such information in the error code; on the other hand, on targets where there is no such clear feedback, static analysis (e.g., taint analysis) and instrumentation can be applied on the target to make it amenable to error message guided fuzzing. We leave error-guided fuzzing on other targets as future work.

Supporting the WebGPU interface. WebGPU is the upcoming successor of WebGL [21]. In addition to rendering, WebGPU is designed to further expose GPU accelerated computational interfaces to web apps. In August 2021, Chrome was the first browser to release a beta test version of WebGPU, which does not yet fully support the promised features, and the initial specification is still in a draft stage [20]. Because WebGPU does not yet have a finalized specification, we wait for its final roll-out. The information available so far indicates that WebGPU implementations in major browsers will continue to provide web developers with informative error messages to facilitate web app debugging. Therefore, GLeeFuzz will be able to use these error messages to extend our support to future WebGPU implementations.

10 Conclusion

WebGL is a widely supported, security-critical interface exposing the native graphic stack to remote users. Collecting code coverage from WebGL is both challenging (code is spread across processes, libraries, kernel, and GPU firmware) and inapt (code coverage lacks a straight-forward mapping to the API sequences with complex arguments that trigger bugs). Fuzzers such as Syzkaller exclusively rely on code coverage therefore resort to semi-random mutation and highly depend on starting seeds and templates to find bugs.

To mitigate these challenges and provide more informed and tailored fuzzing, we present a novel error message guided fuzzing technique, which uses the triggered error messages, instead of code coverage, as feedback to guide input mutation. Our mutators are API aware and mutate the parts of the input indicated by the observed error messages. So far, GLeeFuzz, our prototype, has discovered 7 new vulnerabilities in widely used browsers: 4 in Chrome, 2 in Safari and 1 in Firefox. Source code is available at https://github.com/HexHive/GLeeFuzz.

Acknowledgements

We thank Adrian Herrera and the anonymous reviewers for their insightful feedback, and Google for acting quickly and professionally on our bug reports. We also thank Apple for silently patching our reported bugs but would have appreciated an acknowledgement. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), DARPA HR001119S0089-AMP-FP-034, US AFRL FA8655-20-1-7048, SNSF PCEGP2_186974, and NSF awards CNS-1846230, CNS-1953932, and CNS-2145744.

References

[1] Chromium Sandbox. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/design/sandbox.md.

[2] Drive-by download. https://en.wikipedia.org/wiki/Drive-by_download.

[3] Post-Spectre Threat Model Re-Think. https://chromium.googlesource.com/chromium/src/+/master/docs/security/side-channel-threat-model.md.

[4] Yousra Aafer, Wei You, Yi Sun, Yu Shi, Xiangyu Zhang, and Heng Yin. Android smarttvs vulnerability discovery via log-guided fuzzing. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[5] Adam Barth, Collin Jackson, Charles Reis, TGC Team, et al. The security architecture of the chromium browser. In *Technical report*. Stanford University, 2008.

[6] Microsoft Security Response Center. Webgl considered harmful. https://msrc-blog.microsoft.com/2011/06/16/webgl-considered-harmful/, 2021.

[7] Android Developers. Webview | android developers. https://developer.android.com/reference/android/webkit/WebView, 2021.

[8] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases.

[9] Apple Developer Documentation. Wkwebview | apple developer documentation. https://developer.apple.com/documentation/webkit/wkwebview, 2021.

[10] Alastair F Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. Automated testing of graphics shader compilers. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

[11] Electron. Electron | build cross-platform desktop apps with javascript, html, and css. https://www.electronjs.org/, 2021.

[12] Jesse Hertz etc. Project triforce: Afl + qemu + kernel = cves! (or) how to use afl to fuzz arbitrary vms. https://raw.githubusercontent.com/nccgroup/TriforceAFL/master/slides/ToorCon16_TriforceAFL.pdf, 2018.

[13] Sergiu Gatlan. Google chrome 85 fixes webgl code execution vulnerability. https://www.bleepingcomputer.com/news/security/google-chrome-85-fixes-webgl-code-execution-vulnerability/, 2020.

[14] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy*, pages 575–589. IEEE, 2014.

[15] Google. Chromium issue 145544. https://bugs.chromium.org/p/chromium/issues/detail?id=145544, 2012.

[16] Google. Chromium issue 765469. https://bugs.chromium.org/p/chromium/issues/detail?id=765469, 2017.

[17] Google. Chromium issue 774174. https://bugs.chromium.org/p/chromium/issues/detail?id=774174, 2017.

[18] Google. Chromium issue 784183. https://bugs.chromium.org/p/chromium/issues/detail?id=784183, 2017.

[19] Google. Chromium issue 848914. https://bugs.chromium.org/p/chromium/issues/detail?id=848914, 2018.

[20] Google. Access modern gpu features with webgpu. https://web.dev/gpu, 2021.

[21] Google. Chromium blog: Chrome 94 beta: Webcodecs, webgpu, scheduling, and more. https://blog.chromium.org/2021/08/chrome-94-beta-webcodecs-webgpu.html, 2021.

[22] Google. Chromium issue 1149204. https://bugs.chromium.org/p/chromium/issues/detail?id=1149204, 2021.

[23] Google. Chromium issue 1219886. https://bugs.chromium.org/p/chromium/issues/detail?id=1219886, 2021.

[24] Google. How it works: metamorphic testing using glsl-fuzz. https://github.com/google/graphicsfuzz/blob/master/docs/glsl-fuzz-intro.md, 2021.

[25] Google. Syzkaller - kernel fuzzer. https://github.com/google/syzkaller, 2021.

[26] google project zero. Dom fuzzer. https://github.com/googleprojectzero/domato, 2021.

[27] Khronos Group. Opengl es overview. https://www.khronos.org/opengles/, 2021.

[28] Khronos Group. Webgl overview. https://www.khronos.org/webgl/, 2021.

[29] Khronos Group. Webgl overview. https://www.khronos.org/registry/webgl/specs/latest/1.0/, 2021.

[30] Khronos Group. Webgl overview. https://www.khronos.org/registry/webgl/specs/latest/2.0/, 2021.

[31] The Khronos Group. Blacklists and whitelists. https://www.khronos.org/webgl/wiki/BlacklistsAndWhitelists, 2022.

[32] HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2345–2358, 2017.

[33] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT international workshop on automating TEST case design, selection, and evaluation*, pages 45–48, 2018.

[34] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. {FuzzGen}: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2271–2287, 2020.

[35] Dave Jones. Trinity: Linux system call fuzzer. https://github.com/kernelslacker/trinity, 2018.

[36] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-sensitive control flow integrity. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 195–211, 2019.

[37] Kyungtae Kim, Taegyu Kim, Ertza Warraich, Byoungyoung Lee, Kevin RB Butler, Antonio Bianchi, and Dave Jing Tian. Fuzzusb: Hybrid stateful fuzzing of usb gadget stacks. pages 632–649, 2022.

[38] Wenhao Li, Shiyu Luo, Zhichuang Sun, Yubin Xia, Long Lu, Haibo Chen, Binyu Zang, and Haibing Guan. Vbutton: Practical attestation of user-driven operations in mobile apps. In *Proceedings of the 16th annual international conference on mobile systems, applications, and services*, pages 28–40, 2018.

[39] Dhiraj Mishra. Fuzzing browsers. https://github.com/RootUp/BFuzz, 2021.

[40] MITRE. Cve-2020-15675. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-15675, 2021.

[41] MITRE. Cve-2020-6492. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-6492, 2021.

[42] Mozilla. Mozilla foundation security advisory 2020-42. https://www.mozilla.org/en-US/security/advisories/mfsa2020-42/, 2020.

[43] MozillaSecurity. Javascript engine fuzzers. https://github.com/MozillaSecurity/funfuzz, 2021.

[44] National Institute of Standards and Technology. Search results for webgl. https://nvd.nist.gov/vuln/search/results?form_type=Basic&results_type=overview&query=WebGL, 2022.

[45] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing {OS} fuzzer seed selection with trace distillation. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 729–743, 2018.

[46] Chang Min Park, Donghwi Kim, Deepesh Veersen Sidhwani, Andrew Fuchs, Arnob Paul, Sung-Ju Lee, Karthik Dantu, and Steven Y Ko. Rushmore: securely displaying static and animated images using trustzone. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pages 122–135, 2021.

[47] Hui Peng and Mathias Payer. Usbfuzz: A framework for fuzzing {USB} drivers by device emulation. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 2559–2575, 2020.

[48] Appium project. Appium: Mobile app automation made awesome. http://appium.io/, 2021.

[49] Chromium Project. Angle - almost native graphics layer engine. https://chromium.googlesource.com/angle/angle, 2021.

[50] Selenium Project. Seleniumhq browser automation. https://www.selenium.dev/, 2021.

[51] SVF Project. Static value-flow analysis framework for source code. https://github.com/SVF-tools/SVF, 2021.

[52] The Cordava Project. Apache cordava. https://cordova.apache.org/, 2021.

[53] The Chromium Projects. Chromium os - the chromium projects. https://www.khronos.org/opengles/, 2021.

[54] The Chromium Projects. Control flow integrity. https://www.chromium.org/developers/testing/control-flow-integrity, 2021.

[55] Tristan Ravitch. wllvm: A wrapper script to build whole-program llvm bitcode files. https://github.com/travitch/whole-program-llvm, 2021.

[56] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for {OS} kernels. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 167–182, 2017.

[57] W3C. Webdriver-w3c working draft 24 august 2020. https://www.w3.org/TR/webdriver/, 2021.

[58] Wen Xu, Soyeon Park, and Taesoo Kim. Freedom: Engineering a state-of-the-art dom fuzzer. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 971–986, 2020.

[59] Zhihao Yao, Zongheng Ma, Yingtong Liu, Ardalan Amiri Sani, and Aparna Chandramowlishwaran. Sugar: Secure gpu acceleration in web browsers. *ACM SIGPLAN Notices*, 53(2):519–534, 2018.

[60] Zhihao Yao, Saeed Mirzamohammadi, Ardalan Amiri Sani, and Mathias Payer. Milkomeda: Safeguarding the mobile gpu interface using webgl security checks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1455–1469. ACM, 2018.

[61] Kailiang Ying, Priyank Thavai, and Wenliang Du. Truzview: Developing trustzone user interface for mobile os using delegation integration model. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, pages 1–12, 2019.

[62] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, and Yang Liu. Apicraft: Fuzz driver generation for closed-source {SDK} libraries. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, pages 2811–2828, 2021.

A Concrete WebGL example program

To illustrate, assume the sample WebGL program shown in Figure 5 is selected by a coverage-guided fuzzer for mutation. This WebGL program is a partially valid input generated by the fuzzer with two errors: (1) in line 6, the first argument passed to `bufferData` is invalid and its execution will stop at the `return` statement in line 10 in Listing 1; (2) in line 10, although all the arguments passed to `drawArrays` are valid, as the program is in an invalid internal state (no shader program is properly set up by calling `useProgram`) its execution will stop at the `return` statement in line 5 in Listing 2. To effectively expand the coverage to execute the code not triggered in `bufferData` in this input, ideally, the fuzzer needs to generate a valid value for its first argument; similarly, to expand the coverage of `drawArrays` the fuzzer needs to insert a call to `useProgram` to set up the shader program. Unaware of which part of the input is erroneous, the random approach taken by the `random_mutate` function is highly unlikely to choose the right argument of the right API to mutate, or to select the right API to fix the dependency.

```
1.   canvas = document.createElement("canvas");
2.   gl = canvas.getContext("webgl");
3.   shader = gl.createShader(gl.VERTEX_SHADER);
4.   buffer = gl.createBuffer();
5.   // ......
6.   gl.bufferData(gl.ALPHA, 100, gl.STATIC_DRAW);
7.   program = gl.createProgram();
8.   // ......
9.   // gl.useProgram(program);
10.  gl.drawArrays(gl.POINTS, 100, gl.STATIC_DRAW);
```

Figure 5: A sample WebGL program

```
1  void bufferData(GLenum target, int64_t size,
       GLenum usage) {
2    // ......
3    switch (target) {
4      case GL_ELEMENT_ARRAY_BUFFER:
5        buffer = bound_vertex_array_object_->
       BoundElementArrayBuffer();
6        break;
7      // ......
8      default:
9        SynthesizeGLError(GL_INVALID_ENUM,"
       invalid target");
10       return;
11   }
12   // ......
13 }
```

Listing 1: The native function of `bufferData`

```
1  void drawArrays(GLenum mode, GLint first,
       GLsizei count) {
2    // ......
3    if (!current_program_) {
```

```
4      SynthesizeGLError(GL_INVALID_OPERATION, "
       no valid shader program in use");
5      return;
6    }
7    // ......
8  }
```

Listing 2: The native function of `drawArrays`

```
void useProgram(WebGLProgram* program) {
  // ......
  current_program_ = program;
  // ......
}
```

Listing 3: The native function of `useProgram`

A.1 Type 1 and type 2 error messages

For example, in Figure 5, when `bufferData` is executed at runtime, an error message “invalid target” will be emitted by calling `SynthesizeGLError` (line 9 in Listing 1) when the `target` argument is detected to be invalid. This message is representative of a category of error messages in WebGL that indicates that some argument passed to the executed API is invalid. We denote those collectively as type 1 error messages. The argument whose invalidity is indicated by a type 1 error message is called the *target argument* of it. Similarly, the implementation of `drawArrays` emits an error message “no valid shader program in use” (in line 4 of Listing 2) when the internal variable `current_program_` is not set up. This message is representative of another category of error messages that indicates an invalid internal state caused by unsatisfied dependencies. We denote this category of error messages collectively as type 2 error messages. The set of APIs that are able to fix the invalid internal state indicated by this type of message is called `dependent API set`.