Universität Zürich[UZH]

Department of Informatics

Bachelor's Thesis

Fuzzing Playground: Easy-to-Use Web-Based Tool to Demonstrate Fuzzing

Oliver Kamer
16-921-009
oliver.kamer@uzh.ch

supervised by
Prof. Dr. Alberto Bacchelli
David Ackermann
Zurich Empirical Software Engineering Team

September 14, 2021

Abstract

Fuzzing describes the fully automatic testing of software for bugs. While fuzzing has become more popular in recent times, the process of setting up a fuzzer for learning purposes is complicated and the output of the fuzzer is hard to understand. This report presents a Fuzzing Playground that gives the user the possibility to easily and quickly start a fuzzing process and see what is happening under the hood. This is being done by running the fuzzing process in-browser and by having precompiled fuzzing targets, ready for the user to pick. The output visualizes the processes of the fuzzer and presents the user with the real fuzzed data.

Zusammenfassung

Fuzzing beschreibt das vollautomatische Testen von Software auf Fehler. Fuzzing ist in letzter Zeit immer beliebter geworden, der Prozess der Einrichtung eines Fuzzers zu Lernzwecken ist aber kompliziert und die Ausgaben des Fuzzers sind schwer zu verstehen. In diesem Bericht wird ein Fuzzing Playground vorgestellt, der einem Anwender die Möglichkeit gibt, einfach und schnell einen Fuzzing-Prozess zu starten und zu sehen, was hinter dem Vorhang passiert. Dies wird ermöglicht, indem der Fuzzing-Prozess im Browser ausgeführt wird und vorkompilierte Fuzzing-Targets zur Verfügung stehen, aus welchen der Anwender auswählen kann. Der Playground visualisiert die Vorgänge des Fuzzers und präsentiert dem Benutzer die Daten, welche durch den Fuzzer generiert wurden.

Acknowledgements

Thanks to Prof. Dr. Alberto Bacchelli for allowing me to explore the field of fuzzing in-depth, for making pragmatic decisions in the right moments, and for giving feedback to my monthly updates.

Thanks to David Ackermann for looking after me and the work, for patiently answering my many questions about the topic and about thesis work in general, and for giving me thoughtful and often new ideas to pursue. He also offered a holistic view of the project and reminded me to think about challenges in the larger context. I would like to further thank him for keeping me motivated and engaged with the topic, and picking me up when frustration about the speed of progress sometimes got the better of me.

And finally, thanks to my family and friends that received a sometimes unwanted introductory class in software testing to be able to comprehend my work and who endured this process with me, offering their support, open ears, and love at every point.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Fuzzing is the fully automatic testing of software. The first attempts at fuzzing were done early on in 1990 when Miller et al. [50] generated "a stream of random characters to be consumed by a target program" for various Unix programs.

Today, there are many specialized tools to fuzz different programs. Some are unstructured (and therefore quite similar to the original fuzzer described by Miller et al. [50]), while others that are aware of the structure of the program that is to be fuzzed. Modern, structure-aware fuzzers include AFL [78, 23] and libFuzzer [37] and hongfuzz [27].

Fuzzing is being used widely to find bugs and other errors in software, and has become increasingly important to ensure high quality in software. A wide know example of that is the Heartbleed bug, which was discovered by some security researchers through fuzzing [5, 62, 72].

More recently, Google announced that they would continuously fuzz open-source projects they deemed important [2] and have since found and "reported over 9,000 bugs" [59] found in the projects alone. Another computer giant, Microsoft has announced their fuzzing framework in recent times, to address some difficulties of fuzzing, such as the ability to "harness, execute, and extract information" [10] from the fuzzing process. Campbell and Walker [10] further mention, that the complexity of fuzz testing in part "required dedicated security engineering teams to build and operate fuzz testing capabilities".

Metzman and Ali [46] further underlines the importance of Fuzz testing. They explain that in "an ideal world, fuzzing should be as ubiquitous and simple as writing a unit test" and that developers should "fuzz your code because if you don't, someone else will". Google has developed ClusterFuzz, specifically to be able to Fuzz test software on a large scale [3, 4].

Even today, "simple black-box techniques" [49] are being used for fuzzing. The conclusion by Miller et al. is that "after more than thirty years, it appears that there is still a place for this type of basic fuzz testing". Having high-quality software is not only more convenient, useful, and less stress-inducing for users, but bad software quality is a major cost as well. Krasner [34] estimate that "the cost of poor quality software in the US in 2018 is approximately $2.84 trillion", as such, private businesses demand high software quality.

The process of starting to fuzz a piece of software by oneself can be quite hard and challenging. While a simple demo effect can easily be achieved by someone who has set up fuzzing tools beforehand, actually starting to fuzz is quite a challenge for novices. It requires one to instruct and compile the target, run the fuzzer, and then also be able to understand the outputs the fuzzer provides. With the importance of fuzz testing and the difficulty of setting it up and understanding it, fuzzing can often look like a difficult and daunting task to newcomers, that is only performed by big companies with security teams or specialized security companies.

The Playground fuzzer attempts to be an easy-to-use education tool, where anyone, no matter the background and technical understanding, can start a fuzzing process and see the work. It runs in most modern browsers [53] without any installation needed. As such, the time between loading the site and being able to start fuzzing is minimal, and people with little to no technical understanding can use it. The tool also provides visual feedback to the user while the fuzzer is running and shows the user the fuzzing corpus, so the user can see how many elements are in the corpus. Further, the user can also download these test cases.

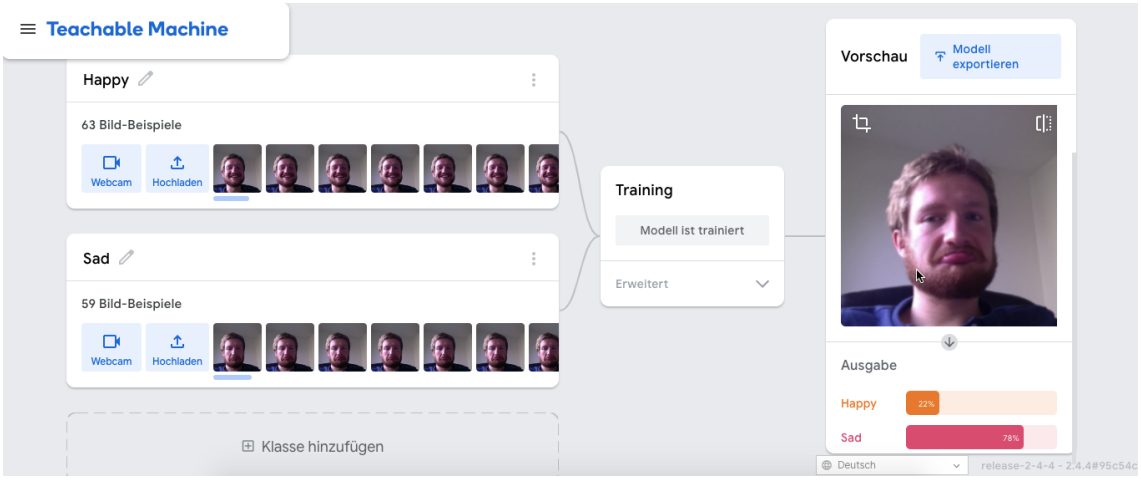


Figure 1: Screenshot of the TEACHABLE MACHINE [12, 26] trained to detect a happy or a sad face.

2 PREVIOUS WORK

While there are many fuzzers [50, 17, 23, 27, 37] of different kinds, most run in a terminal, and most of the computing power invested in fuzzing is probably in the cloud [59]. As such, they are often hard to understand and not easily set up.

Usability is often not one of the main concerns for developers of fuzzers. In fact, Plöger et al. [57] state that "to the best of [their] knowledge, there are no studies concerning the usability of fuzzers or a usability comparison of static analysis and fuzzing". Particularly, LIBFUZZER is widely understood to be difficult to use. Plöger et al. found that there was "no step in the libFuzzer process that did not cause our participants severe problems" when using participants that were familiar with LINUX.

While the FUZZING PLAYGROUND doesn't address the usability of LIBFUZZER, it does serve as an educational tool where potential users can get familiar with fuzzing in a very low-barrier-of-entry environment and strives to have high usability.

2.1 Web-based tools illustrative tools

Modern complex technical concepts are often hard to understand and harder to explain. One approach to bridge the understanding gap is to get people wanting to learn about the difficult technical concepts to try it by themselves [15, 13, 63]. This approach, while good for learning, is often hard to do, as it takes a lot of time and the barrier of entry can be quite high for many of these modern technical tools.

One of these difficult concepts in recent times in computer science is machine learning. While machine learning "is increasingly prevalent in daily life" [12], it is often difficult to grasp. As such, the setup to produce small machine learning outputs takes a lot of time, even for people familiar with technical systems. Carney et al. [12] approached this by developing a simple-to-use machine learning classifier, that uses transfer learning under the hood. The complexity that arises from using machine learning, as well as retraining data, is "hidden from users, who simply benefit by needing fewer data and training time to create useful, accurate models" [12]. Figure 1 shows an example where the TEACHABLE MACHINE is trained to recognize exaggerated facial expressions, in this case *Happy* and *Sad*. A setup similar to the example takes less than two minutes to produce, and the output is reasonably accurate.

In a similar vein, Smilkov et al. [64] developed a web tool where users can try out a Neural Network called TENSORFLOW PLAYGROUND. The tool allows the user to choose one of four pre-defined datasets, the features, and the number of hidden layers among other input factors. It also

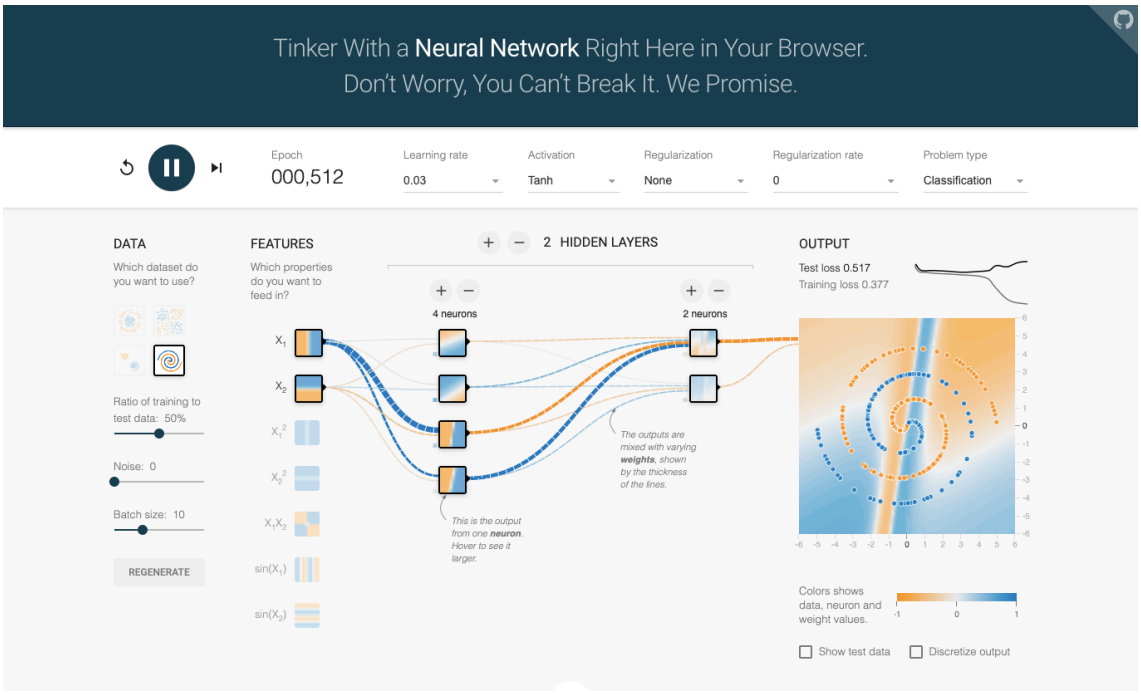


Figure 2: Screenshot of the TensorFlow Playground [64, 70]

has a big play button that starts the training and various data points, such as the neurons of the hidden layers, are available to see through hovering of them. Figure 2 shows the TensorFlow Playground during the execution.

Both Smilkov et al. [64] and Carney et al. [12] have highlighted the educational use of their respective tools. Further, Carney [11] highlighted the usage of Teachable Machine "to understand what AI really is and how to apply it to their domain".

Inspired by the success of the TensorFlow Playground as an education tool [60, 64], other researchers and developers followed suit. The usability and ability to quickly demonstrate more complicated technical concepts, especially in the fields of artificial intelligence [64, 12] is often inspired by the TensorFlow Playground. For example, Norton and Qi [56] presented a playground styled tool called Adversarial-Playground that visualized "machine learning systems in adversarial environments" that is inspired by the TensorFlow Playground playground.

2.2 Visualizing fuzz testing

Some attempts to visualize fuzz testing were previously done. Some of these visualization had the purpose to be able to see the overall progress more closely, while others were done to be able to interact with the code and the relating code coverage.

Vainio [71] describes obtaining "performance metrics [...] along with data output from the fuzzer, and pack them conveniently in some format" to be able to display various performance measurements collected from the machines that are running fuzzers. The system setup "consisted of a monitoring server and a remote client that were connected via network", where the monitoring server collected and aggregated the data from hosts running fuzzers. This allowed for more in-depth monitoring of the fuzzing system.

Zhou et al. [80] developed VisFuzz "to help test engineers to locate the boundary of unexplored regions, understand the semantic context around the bottleneck, and intervene the fuzzing process [...] to achieve higher coverage". Using the tool, they were able to increase coverage and find more unique crashes.

Fioraldi and Pileggi [22] developed FuzzSplore to explore the different coverage points produced by different fuzzers. By producing plots, the user can "select a subset of fuzzers that explore different program points if the points related to each fuzzer in the graphs are clustered". The user can then see "when there is a huge increment of the number of edges". By "selecting testcases in the graph, the user can see if the testcases are similar in the scatterplot in order to understand the ability of the mutator to generate similar or different derived inputs". As such, FuzzSplore gives the user immediate feedback for coverage data that the user is able to interact with.

2.3 Web based fuzzing tools

The first found instance of web-based fuzzing is a talk and implementation by Metzman [42]. Metzman states the goal of being able to run the OSS-Fuzz [42] project at home. The project demonstrates a) the ability to run a fuzzer in-browser, b) shows the potential problems, such as slower speeds when displaying the output of the fuzzer in some way. Further, the fuzzer cannot be controlled in any meaningful way, such as stopping it or restarting it after a crash. Metzman also provides a Dockerfile to build a Docker image [44] where one can compile the SQLite [45] example. As of May 2021, the image cannot be built, as dependencies are outdated and the selected image of Ubuntu 16.04 has reached the end of support for the image [61].

SSLab at Georgia Tech [65] developed Fuzzcoin, a combination of fuzzing and Bitcoin. As such, it is a "is a public fuzzing network inspired from group-mining of BitCoin" [67]. While the thought process behind the development of Fuzzcoin is not clear, one can assume that the wastefulness of bitcoin mining [76] might have been an inspiration to use the proof-of-work concept for something more interesting and productive. The stated goal of Fuzzcoin is to "beat Google's computing power together" [65] of over 25000 cores [4].

While the in-browser fuzzer works, the recommended setup is to use a custom docker container [67]. The project still works, it doesn't appear to have been updated since July 2020 [66].

3 Background

3.1 Types of fuzzers

Most fuzzers work fundamentally the same by following two major principles: they mutate or generate inputs and run with little to no human input [50, 48, 8, 32, 23]. While traditional unit tests attempt to ensure the correctness of certain software components, mostly small parts of code, fuzzing is only looking for bugs [69, 36]. Bugs that do not result in a hang or crash, but rather in an incorrect output by the software cannot be detected by fuzzers, as it would require the fuzzers to have an understanding of correct outputs. Most of the time fuzzers will, however, be able to find certain bugs that normal tests do not cover and on average they achieve much higher code coverage [33, 28] than manual testing through developers writing test cases. No open-source project survey by Zhai et al. [79] reached as high of code coverage as the best fuzzers surveyed by Google [28].

3.1.1 Black-box fuzzer

The first generation of fuzzers were black-box fuzzers, as described by Miller et al. [50, 48]. Black-box fuzzers cannot see inside the program that is being fuzzed and are unaware of the structure of the program or source code.

Sutton et al. [68] describes black-box fuzzing as "[y]ou as the end user control the input that goes into the black box, and you can observe the output that emerges from the other end, but you do not have knowledge of the inner workings of your target".

Miller et al. [50, 48, 51, 49] have over time repeated their experiment using simple black box fuzzer and found in their 2020 paper that they "are still seeing failure rates from 12% to 19% with the original simple methods" [49].

They also comment that "[s]some of the errors that we found have been present in the code for many years, as far back as 1994 for `checknr`, `ctags`, `dc`, and `indent`; 1997 for `spell`; and 1998 for `gdb` and `ptx`" and state that "[m]ore frequent application of the basic fuzz tests could help to avoid this situation" [49].

As such, black-box fuzzer, while not being the exciting new thing, can still find a relatively high number of bugs and be potentially of use to find issues in software.

3.1.2 White-box fuzzer

In contrast to black-box fuzzing, white-box fuzzing requires full access to all source code and the fuzzing process is not as random and directly based on the code. Sutton et al. [68] describe the information necessary for white-box fuzzing as follows: "If you were taking a pure white box approach to breaking into a home, you would have full access to all information about the home before breaking in. This might include blueprints, a list of lock manufacturers used, details of the construction materials used in building the home, and more."

Bounimova et al. [8] describe the success at Microsoft using white-box fuzzing to find bugs in Windows 7. Further Godefroid et al. [25] propose "an alternative whitebox fuzz testing approach inspired by recent advances in symbolic execution and dynamic test generation".

An example of a widely used white-box fuzzer is KLEE [9] that gets "100% coverage on 16 tools and over 90% coverage on 56 while the developer tests get 100% on a single utility" when fuzzing the GNU Coreutils suite.

3.1.3 Grey-box fuzzer

Grey-box fuzzers are fuzzer that combine some white-box and some black-box approaches. As such, there is no need for access to the source code, but the fuzzers do look at certain signals

during the execution, mostly coverage of the executed code [23, 37].

The logic behind coverage-guided fuzzers is to cover as much code as possible in the execution to find new or well hidden bugs. This is done by mutating the inputs and storing the interesting ones that produce new coverage to the corpus. Coverage-guided fuzzers generally use "instrumentation to determine a unique identifier for the path that is exercised by an input" [7].

Grey-box fuzzers generally sacrifice in-depth code analysis for fast, lightweight information about the execution [41]. This means execution is much faster, which leads to more inputs being tried for a program. On the other hand, this sometimes leads to finding less uncovered parts of the program, especially in comparison to white-box fuzzing. Bohme et al. [7] writes that "coverage-based fuzzers tend to visit certain paths with high frequency, generating too much fuzz that exercises the same few paths". Real-life tests however also show that grey-box fuzzers generate high code coverage, with all fuzzers getting coverage of more than 84%, while libFuzzer and AFL reached coverages of more than 90% [28, 47].

Modern coverage-guided grey-box fuzzers have enjoyed great success in finding security vulnerabilities and software bugs [32, 36]. They are also widely used internally by companies to find bugs and security issues with their software and systems [52, 2].

3.2 Modern coverage-guided fuzzers

Two fuzzers were evaluated for this product: AFL [78] and libFuzzer [37]. They are quite similar in that they are both coverage-based grey-box fuzzers. They are also often described as being the most popular fuzzers currently around [36, 80, 6].

Both of them are widely in use and are often used side by side. They both have some internal mechanisms to keep track of coverage found and produce a corpus with interesting seeds. AFL has many newer and improved iterations and generally performs better, finding more coverage after a set amount of time [28].

One major difference is the process of starting the fuzzer. With AFL one instruments the fuzzing target using a special flag that inserts the executable with the code coverage signals at various points. One then has to compile AFL and run the fuzzer against the instructed binary. On the other hand, libFuzzer is an in-process fuzzer. This means one first has to write a harness, that takes the inputs from the fuzzer and then runs this input through the target software. In a second step, the target, harness, and fuzzer have to be compiled together into a single fuzzing executable, that can then be run. For general fuzzing applications, this means that libFuzzer will stop after having found a single crash, while AFL will simply take note of the crash and keep fuzzing until stopped [32].

For this playground, using the in-process fuzzer from libFuzzer means that one has to only compile a single executable that is the fuzzer and fuzzing target all in one. Attempts using AFL were unsuccessful, as linking an executable that is compiled through Emscripten to AFL was not achieved. Further compilation of AFL to WebAssembly was also not achieved. In fact, instrumentation of WebAssembly targets is on a list of possible future ideas for AFL++ [1].

Compilation of the in-process fuzzer libFuzzer was shown by Metzman [42]. The disadvantages of using the less popular fuzzer are, in this case, negligible, as the playground intends to be an educational tool and not a fuzzer to find bugs. It generally not expected that the fuzzer finds any crashes, especially considering the targets are already well fuzzed through OSS-Fuzz [2]. Secondly, the lower coverage finding rate of libFuzzer in comparison to AFL++ [28] also doesn't matter, since the performance loss from running the fuzzer in the WebAssembly virtual machine, already makes the in-browser fuzzer unattractive as direct competition to normal fuzzing that is done for bug discovery purposes instead of educational ones.

3.3 Emscripten

Emscripten is a compiler that allows to compile C or C++ code to code that can be executed in browser. More precisely, it compiles Low Level Virtual Machine (LLVM) assembly code to JavaScript and WebAssembly [74]. Emscripten "allows compiling a very large subset of C and C++ code into JavaScript, which can then be run on the web" [77].

Emscripten was conceived to be able to run interactive code more easily in browsers. While many approaches to run code in the browser have existed and do still exist, the only platform that is available to pretty much all browsers is JavaScript. As such Zakai [77] first wrote a compiler from LLVM assembly code to JavaScript, as "JavaScript is present in essentially all web browsers, by compiling one's language of choice into JavaScript, one can still generate content that will run practically everywhere" [77].

The early JavaScript implementation of Emscripten faced the interesting challenge of having to compile a low-level language, LLVM assembly code, to a higher level language, JavaScript. This meant Emscripten had to address some specific problem, such as recreating control statements native to JavaScript to ensure the interpreter of the resulting code can use optimizations native to JavaScript. For this purpose, Emscripten introduces the Relooper algorithm, which "generates high-level loop structures from low-level branching data, and prove its validity" [77].

Emscripten compiled its code into a specialized subset of JavaScript called asm.js. Speeds of executing this JavaScript subset are described to be anywhere between 1/10th [77], 1/2nd [31] and 2/3rd [24] of native execution speed. It is assumed, that execution speed is affected by the JavaScript implementation of the browser. In fact, Firefox even implemented specific optimizations to increase the speed of the asm.js execution [73].

Based on the asm.js specification, WebAssembly was then developed to address the speed, safety, portability, and compactness of the code that is executed in browser [30]. WebAssembly executes much faster, with benchmarks showing it to be "33.7% faster than asm.js" and nearly all the benchmarks being "within 2× of native".

As Emscripten is specifically made for LLVM assembly code, it is an almost perfect fit for libFuzzer, which is part of the LLVM Toolchain. This allows fuzzing processes that are compiled to WebAssembly to be executed in the browser at pretty fast speeds, although certainly some components, including the frontend, slow down the fuzzing process additionally.

WebAssembly and therefore compilation through Emscripten is supported by most modern browsers, and these browsers should also be able to execute the compiled code successfully [16]. It is noted, that certain WebAssembly features are not supported in all browsers and the specification, as well as the support thereof, is evolving [75].

3.4 Preact

Preact is a frontend framework that describes itself as a fast, lightweight React alternative [58]. It allows the use of the wide React ecosystem and helps with quickly developing interactive web applications. To display the information, the input is first received, stored as states in the frontend, which then automatically triggers re-rendering from Preact.

4 Implementation

4.1 Goals

The following six goals are defined for the playground. These goals ensure that the playground is of educational value:

G1 The playground should be so easy to run that anyone without prior technical knowledge or specialized tools can run it within a reasonable amount of time and without reading lots of documentation.

G2 The user should be able to control the process through simple-to-use controls and abstractions thereof. The controls should be pretty much self-explanatory.

G3 The playground should provide reasonable defaults and simple to use, clearly understandable options.

G4 The visualization should indicate to the user when the fuzzer is running, in order for the user to understand the current state and be able to interact with it accordingly.

G5 The visualization should help the user understand what the fuzzer is doing and provide some additional insights into the current process of the fuzzer.

G6 The fuzzer should implement a state-of-the-art fuzzer that is running and not simply show results that were calculated in advance or simulations of the actions of the fuzzer.

4.2 Build process

There are two main parts of the build process. One is to compile to projects that are to be fuzzed, and the other is to build the Preact [58] frontend.

The compilation of the frontend is rather simple. One just needs to install all the packages from npm and then run the command `npm run build`. This then produces a `build` folder, which can be deployed to a server or shown locally in the browser as well.

Compilation of the targets is a bit more complicated. To ensure consistent and reproducible build, Docker was chosen, as it allows to clearly define via the `Dockerfile` which packages to use and is a complete system-in-a-box that allows for reproducible builds. Building the fuzzer in Docker is also the chosen approach by OSS-Fuzz [2] and Metzman [42].

4.2.1 Build fuzzing targets for WebAssembly

The following section describes the process to build the fuzzing targets for the WebAssembly build target. Emscripten provides some drops in compiler commands to replace common build tools such as `emcc` [20] as a replacement to calling the Clang Compiler.

The `Dockerfile` prepares the environment to be able to build the fuzzer to Emscripten. It is largely inspired by the `Dockerfile` provided by Metzman [43] and the respective `Dockerfile` available from OSS-Fuzz [29]. It does the following to be able to do that (see Listing 8 in Appendix C).

Line 17 Grab the Docker image for the most recent Ubuntu LTS version.

Lines 20-23 Update all package lists and packages. Install the necessary build tools to compile `c` and `c++` code.

Lines 25-28 Create folder to work in, including the output folder. The output folder can then be mounted into the local system to get the output back from the Docker container.

Line 21 Install more packages to build LLVM.

Lines 32-34 Clone the Chromium clang version, as shown by Metzman [43]. This has multiple reasons: Firstly, libFuzzer states that "you will need the current (or at least a very recent) version of the Clang compiler" [37] and the version provided in Ubuntu 20.04 is currently outdated two major versions (10.0 [40] in Ubuntu against the current version 12.0.1 [38]) and the Chromium version is "just upstream clang built at a known-good revision that we bump every two weeks or so" [14], ensuring a relatively current stable version.

Lines 36-37 Grab the current version of LLVM, getting an up-to-date libFuzzer.

Lines 42-46 Build the compiler-rt runtime libraries, clang and the LDD linker and installs them.

Line 48 Downloads the latest version of emsdk which allows a specific version of the Emscripten Toolchain to be installed and activated.

Line 50 Install the python package, which is needed by Emscripten even though it is the same as the python3 package installed earlier.

Lines 51-52 Install and activate a specific Emscripten version.

Lines 53-57 Activate the `emsdk` and overwrite the configuration to ensure getting the current LLVM build, that was built just earlier.

Lines 60-62 Copy the custom `compile_libFuzzer.sh` file, overwriting the upstream one. Then run the libFuzzer compilation and compile the fuzzer using `emcc` as a drop-in replacement compiler.

Lines 65-70 Download and install the packages needed to build the target. Then clones the target's source code and copies the fuzzing harness and build script into the Docker container.

After having built the Docker container, one can then run the container using the following command: `docker run -v ~/out:/out -it`. The `-v` option for volume is needed to get the compiled files out of the docker container again. The `out` folder can be mounted anywhere to the host.

Once in the container, the build script which was previously copied to the container can simply be run with `bash build.sh`. The build script can be seen in Listing 10 in Appendix C.3. Table 1 explains the different options set to the `emcc` command.

The program that is to be fuzzed (`lodepng.cpp`), the fuzzing harness (`lodepng_fuzzer.cpp`) and libFuzzer are then compiled together into the output file specified with `-o`. It should be noted that not only the JavaScript file is generated, but also a corresponding WebAssembly file that is called from the generated JavaScript file.

4.3 Layers

The main stack has three distinct layers. All three layers run in the user's browser and are responsible for different tasks. Figure 3 shows an overview of the different layers, their communication streams and their shared storage.

Frontend The frontend that the user sees and interacts with is written in JavaScript using the Preact framework. This is where the user can select the inputs and sees the visualization and the corpus of the fuzzer.

Option	Purpose
`-s ERROR_ON_UNDEFINED_SYMBOLS=0`	Turn off link-time errors as not all symbols are defined.
`-s ALLOW_MEMORY_GROWTH_=1`	Ensure fuzzer isn't aborted with error when trying to allocate more memory.
`-s EXIT_RUNTIME=1`	Properly exiting the WebAssembly runtime after successful finish of the fuzzing iterations. Also prevents warning printed about fuzzer being finished but not terminated.
`-s TOTAL_MEMORY=GB1`	An alias for INITIAL_MEMORY and setting the initial memory to 1 GB.
`-O2`	Run as many optimizations as possible to ensure low file-size without running the JavaScript Closure Compiler that would change JavaScript variable names and therefore not accept `Module` input anymore.
`-fsanitize-coverage=inline-8bit-counters`	Increment a counter of every edge hit, [39]. This option is passed to Clang.
`-lidbfs.js`	To activate the persistent IndexedDB storage

Table 1: Options set for `emcc` to compile fuzzing target to WebAssembly

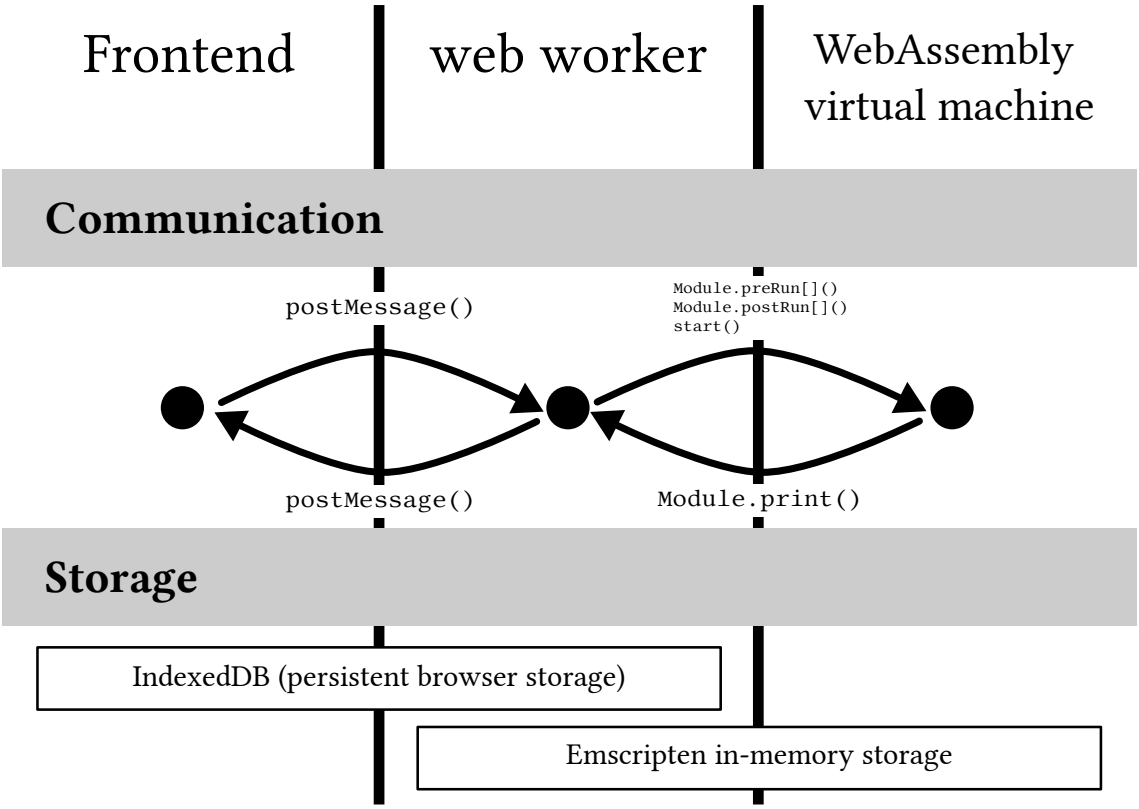


Figure 3: Overview of the layer model with communications and storage

JavaScript Web Worker A web worker who is started from the frontend that controls the execution of the WebAssembly virtual machine and handles its outputs.

Web Assembly A WebAssembly Virtual Machine in which the actual fuzzing process takes place.

4.3.1 Communication

Each of the layers can communicate with its neighboring layers directly and often through some command also with the other layer indirectly via the middle layer. For example, the start button pressed by the user in the frontend sends a message to the web worker, which in turn calls the start function in the web worker to start the execution of the fuzzer in the WebAssembly virtual machine.

4.3.1.1 Frontend and web worker

The communication between the frontend and the web worker is exclusively through messages, both the frontend and the web worker implement a `onMessage` method that is used to handle the messages sent by `postMessage` [55]. Throughout the messages, a JavaScript object is used that defines an action and often additional data depending on the action. Such objects being sent to the worker or the front end can be seen in Listings 1 and 2. The communication via message is shown in Figure 3 in the communications level.

This allows to use the single available communication channel efficiently and ensure that both the frontend and the web worker can deal with the incoming messages in a good and targeted way, as shown in Listing 3.

After every run of the fuzzer, the worker is terminated as shown in line two of Listing 4 before potentially starting a new web worker as shown in lines 8-18.

```
    let newWorker = new Worker();

    // Set the selected seed
    newWorker.postMessage({ action: "setSeed", seed: currentSeed.name });

    // Start the fuzzing process
    newWorker.postMessage({ action: "start" })
```

Listing 1: Setting the seed and starting the fuzzing process from the main frame

```
    // Send filenames of currentFiles
    postMessage({ action: "currentFiles", files: fileNames });

    // Web worker sending the message that the current fuzzing iterations if
        finished
    postMessage({ action: "runFinished" });
```

Listing 2: Web worker sending messages to the frontend

```
    newWorker.onmessage = function (e) {
      const action = e.data.action;
      // deal with output
      if (action == "console") {
        const outputLine = e.data.message;
        [...]
      }

      // File list is sent
      if (action == "fileList") {
        setCurrentFilesList(e.data.files);
      }

      // Run finished massage sent
      if (action == "runFinished") {
        console.log("Run finished");
        setCurrentRunBaseIterations((prevState) => prevState + runs);
        setCurrentRunIterations(0);
        setRunning(false);
      }
    };
```

Listing 3: Frontend dealing with incoming messages from web worker depending on the action attribute of the transmitted object

```
    // Terminate worker
    worker.terminate();

    // Reading of files from IndexedDB
    [...]

    // Potentially restart worker
    setTimeout(async () => {
      if (keepRunning) {
        console.log("Keep running is active, starting another run.");
        setCurrentRuns(runs);
        const startedWorker = await startWorker(newRuns);
        setWorker(startedWorker);
        setRunning(true);
      } else {
        console.log("Keep running is inactive, stopping.");
      }
    });
```

Listing 4: Frontend terminating the web worker which has run the fuzzer before start a new web worker depending on the `keepRunning` variable

4.3.1.2 Web worker and WebAssembly virtual machine

The web worker controls the execution of the fuzzer in the WebAssembly virtual machine through the usage of the `Module` object [19]. The `Module` object defines any potential arguments that are passed to the executable, which is used here to define the folder the corpus is saved to and the max executions for the specific iteration. The `Module` object defines functions that are run before, during, and after execution of the WebAssembly code. Figure 3 shows the communications paths between web worker and WebAssembly virtual machine.

Before the execution, the functions for `preRun` are being called. In this case, during the prerun, the file system is being set up and potentially already available files in the IndexedDB are being synced to the Emscripten in-memory file system. Further, if any initial seeds have been selected, these are also put in the file system for the fuzzer to find them.

During the execution, whenever the fuzzer prints any output, the function `print` is being called when something is printed to output by libFuzzer in the WebAssembly virtual machine. This print function mostly acts as a forward to the frontend, where certain information is extracted, and the lines are displayed in the terminal.

When the execution of a certain number of iterations is finished, the fuzzer exits gracefully and the `postRun` functions are called by Emscripten. In the Playground Fuzzer this has two main tasks. Firstly and foremost, the files that are in the in-memory file system need to be synced to the persistent IndexedDB. Subsequently, a list of all files is generated using the `FS.readdir()` method. That list is then sent to the frontend, so the reading from the IndexedDB is a bit easier later. Finally, the signal that the run is finished is also sent to the frontend. This allows for the Playground Fuzzer to ensure that the program has finished gracefully and there won't be any damage or loss of data by terminating the web worker, as we are sure the files are synced to persistent storage, by only posting the message that the run is finished in the callback function of the `FS.syncfs()` method.

4.3.2 Storage

Every fuzzer needs some kind of memory and storage [78, 37]. The memory is mostly used to hold short-lived artifacts, such as current test cases and their relating coverage, while persistent storage is used to keep the corpus. This is especially important, so one can interrupt the fuzzing process and resume later without losing access to the coverage already discovered. It further

(a) Start button (b) Stop button (c) Stopping indication

Figure 4: Control buttons in different states

enables the ability to run the fuzzer in a distributed environment, where many machines are fuzzing a target, but they share or sync the corpus from time to time to ensure a low amount of duplicate work.

In the browser, memory and storage are often a bit more complicated, as the browsers sandbox the execution. While modern browsers often support some kind of access to the file system, for example through the File System Access API [35], Emscripten does not currently implement this as of August 2021. Therefore, this feature could not be used. Instead, a mixture of the JavaScript stack and the browser API for IndexedDB [54] is used to store the corpus files first in-memory and later persistently in the IndexedDB. It should be noted that the IndexedDB is not a complete file system with folders and files, but rather a form of key-value storage. Emscripten abstracts its internal, in-memory file system to the IndexedDB by using the full path as the key and then store the object as the value [18, 21].

The JavaScript web worker and the WebAssembly virtual machine both have access to the Emscripten in-memory file system [18] symbolized by the `FS` object. This means that the web worker can preload files into the file system in the `preRun` functions and sync the file system to the IndexedDB in one of the `postRun` functions [19].

The in-memory file system understands folders and must be activated through the build flag `FORCE_FILESYSTEM=1`. It's not persistent. When the web worker is terminated, so is the corresponding file system. This leads to the `FS` object being destroyed, as "all files exist strictly in-memory, and any data written to them is lost when the page is reloaded" [18].

The frontend and the web worker both have access to the IndexedDB. The frontend never writes to the IndexedDB but extracts the corpus from it after the web worker synced the in-memory file system to the IndexedDB.

The web worker, acting as the intermediary layer between frontend and WebAssembly virtual machine, has access to both the in-memory storage and the persistent IndexedDB. The web worker does not have access to the memory of the WebAssembly virtual machine and can therefore not extract data during the execution.

4.4 Product

4.4.1 Controls

The running and stopping of the playground are controlled by two buttons: a start/stop button and a reset button. Figure 4 show the buttons in the different states.

When the fuzzer is not running, the start button (shown in Figure 4a) is shown. By clicking on the button, the fuzzing process is started from the current state.

During the execution, the stop button (shown in Figure 4b) is visible. Clicking the button will however not immediately stop the process as to not lose the data from the fuzzer. It will set a flag to not start any new process and change to the button shown in Figure 4c to indicate to the user that the stop signal has been received. The fuzzing process will finish the predefined number of iterations before gracefully stopping. After having stopped, the user is once again shown the state as in Figure 4b and could restart from the current state.

Should the user want to restart with a clean slate, the reset button will reset and remove all data from the previous fuzzing run, recreating a state that is identical to when the Fuzzing

Playground is first loaded. This is useful if one wants to compare between running the fuzzer for a while with different seeds. Selecting a different target resets the fuzzer automatically.

4.4.2 Fuzzing process

While most fuzzers are normally run for a set amount of time or even more likely indefinitely, this could not be replicated in this implementation. Continuous fuzzing is possible in the WebAssembly virtual machine, but then access of the data in the in-memory data system is not possible. To access that data, one needs to be able to use the `FS.syncfs()` command to sync the files to IDBFS file system that was mounted beforehand through `FS.mount()` [18].

One could also run libFuzzer continuously and use the JavaScript web worker to control the execution, which would allow the fuzzer to be stopped pretty quickly at any point. Unfortunately, this would not be a soft termination but rather a hard termination of the web worker, which would result in all the data stored in the in-memory file system of Emscripten and the web worker being lost. Because of the abrupt nature of the termination, syncing the files to the IDBFS file system would also not be possible. It would also further not be possible to execute a command in the web worker running the WebAssembly file. This is because of the asynchronous architecture of JavaScript, which means the execution of the command, which could potentially stop the execution more graceful, gets delayed until the execution is finished. A continuous fuzzer would not stop unless there is a crash.

Therefore, the Playground does not do continuous fuzzing, but rather a set amount of fuzzing executions that are repeated until the user decided to stop or pause the process.

When the user starts the process, the following takes place:

1. A new JavaScript web worker is started with the input values from the user.
2. A set number of fuzzing iterations is performed, during which the direct outputs are sent to the JavaScript frontend.
3. After the set number of iterations, the files are synced from the in-memory storage of the WebAssembly virtual machine to the persistent storage in the browser, the IndexedDB.
4. The files are then displayed in the JavaScript frontend for the user to see.
5. The JavaScript web worker and corresponding WebAssembly virtual machine is terminated.
6. If the user has not paused or stopped the execution, the JavaScript frontend will automatically repeat step 1, with the generated corpus as the initial seed for the new process.

The number of iterations is increased, as it is assumed that the longer the user leaves the fuzzer running, the less likely the user wants to immediately stop the execution. The number of iterations starts off being 1000, are then doubled every run (which should lead to approximately always double the execution time) and finally stall out at 100,000 iterations per run.

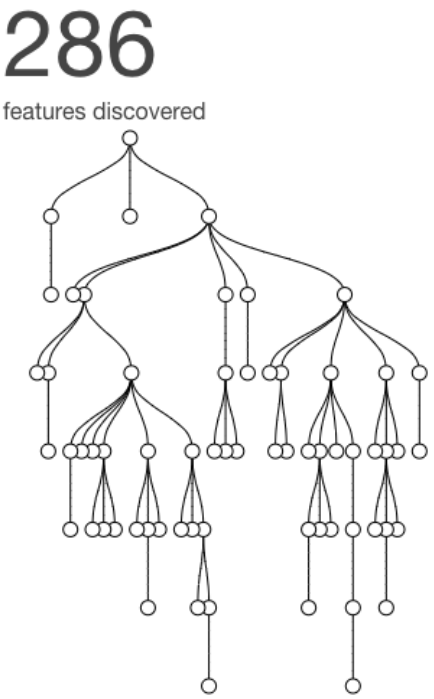
4.5 Visualization

4.5.1 Running indication

There are two main visualizations to indicate the fuzzer is currently running. As the data in the fuzzer is not available during execution, these indicators are generated to show the user what is happening under the hood in a simulation, without actually being able to look under said hood.

The numbers displaying how many inputs were tried, how many features were found and the number of items and the file size of the corpus are read directly out of the terminal output

(a) Illustration of the current input by mutation of the original image



(b) Illustration of the graph tree by generating an approximate graph tree and adding node and leaves when newly covered features are found.

Figure 5: The two illustrations simulating the mutations and coverage that LIBFUZZER uses internally when fuzzing LODEPNG.

of LIBFUZZER and displayed for the user to see. This gives the user the ability to quickly discern what the fuzzer is doing, for example, how quickly it is generating new test cases or how many test cases get moved to the corpus.

To illustrate what the fuzzer is doing, there is a simulated look under the hood. One that has an input is repeatably manipulated at random places to simulate what is happening in the system. Ideally, one would be able to access actual inputs and actual coverage data from LIBFUZZER, but as this is not available during execution and coverage is not stored by the fuzzer, this approximation gives a good indication of what is happening.

Figure 5a shows an ever-changing for the mutating input when fuzzing LODEPNG. That image that often looks glitched or broken. This is achieved by randomly mutating certain parts of the image while keeping the file structure of the image intact. To not be overwhelming and to not crash the site, the image is generated every second by the code shown in Listing 6 in Appendix A.2.

Finally, Figure 5b illustrates how the fuzzer covers more of the code. While the tree is only a visualization and doesn't show the real coverage data found by LIBFUZZER it does grow at the same pace as the fuzzer finds more coverage. This allows the user to quickly see if the fuzzer is finding more coverage or not. It also allows for the comparison with empty seeds and non-empty ones, seeing a much faster growing tree if a seed is provided initially.

4.5.2 Terminal

When running LIBFUZZER normally, there is two main outputs one can have a look at the terminal indicating inputs tried and features found among other indicators and the output directory if one has provided one to the fuzzer. A terminal is also provided in the FUZZING PLAYGROUND to display

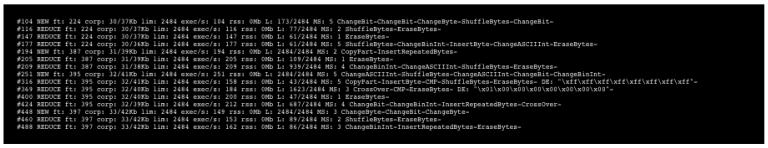


Figure 6: The terminal display the real output of LIBFUZZER.

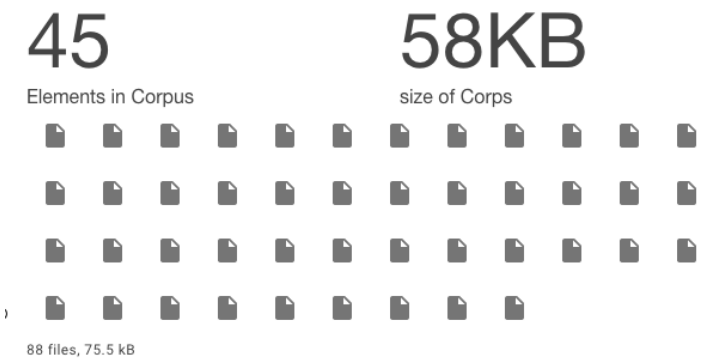


Figure 7: Corpus display during execution with number of elements, size, and illustration of the corpus.

the terminal output by LIBFUZZER.

The small terminal in the FUZZING PLAYGROUND serves two main purposes. On the one hand, it is another indicator that the fuzzer is currently running, and it gives the user a similar view as if they are running the compiled LIBFUZZER on their machine. As such, the user can look into the real running fuzzer and, for example, check which mutations are currently being applied among some other outputs by the fuzzer.

On the one hand, this is also a great way for more advanced users to see the difference when choosing different fuzzing configurations. The terminal also contains interesting information on startup, such as the seed and how many elements are already in the corpus, and the max file size calculated from the corpus.

After having finished the execution, LIBFUZZER also displays some additional information as the time taken for the number of iterations and the recommend dictionary, as shown in Listing 7 in Appendix B.

4.5.3 Corpus

The corpus is displayed while LIBFUZZER is running, as well as when it is paused. During execution, one cannot access the files produced by LIBFUZZER, as they are not available. The reason one cannot access the files is that they first need to be synced to a persistent storage location. The display of the corpus can be seen in Figure 7.

Once the execution of the fuzzer has stopped, the files are loaded by the frontend by opening the INDEXEDDB storage of the browser and loading the files. The files that were synced to the INDEXEDDB before in the post-run step of the web worker are displayed in the frontend.

After the sync, the user can hover over the files and see the filename and size of the file. Often the files are rather small, but differences in file sizes are very much apparent. Especially when comparing the fuzzer that is being run with an empty initial seed, where most of the interesting inputs found early on are tiny, while runs with some interesting seeds, will often produce larger results, mostly similar in size to the seeds provided in addition to the small files.

(a) Seed file

(b) Generated file

Figure 8: File preview for a seed file that was provided to the fuzzer and a file the fuzzer generated and stored in the corpus.

5 Discussion

This section will first evaluate the playground proposed in this work according to the goals set out in Section 4.1, list the limitations, and propose further work or alternative approaches to solving the problem.

G1 The playground is easy to run. In-fact, any reasonably modern browser can be pointed at it, and it is ready to go. There is no need to ready any documentation, and some additional information is provided to the user in the frontend interface. The setup time is non-existent and the usage of the tool doesn't require previous technical knowledge.

G2 The controls are easy to use, and the user should be able to use them quite comfortably. The start and stop button is clear and easy to use, as is the reset button. The frontend reacts very fast to the user's input, therefore providing immediate feedback to the user.

The fact that the fuzzer keeps running for a bit after having pressed the stop button might be confusing to some users, although the indicator saying *Stopping…* does alleviate that to a certain degree.

G3 The playground provides a reasonable configuration to start with. This is similar to starting fuzzing with an empty seed, which is a pretty normal configuration. The default is very much ready to go and in case the user doesn't want to change anything, the user can simply press the start button and begin the fuzzing process.

G4 The indication of the fuzzer running is present in three places to clearly show that the fuzzer is running. The first indicator of running is the changing button that changes to a stop button when the fuzzer is running. The second indicator is the small terminal which displays the real output by the fuzzer. The third indicator is the visualization and tree, providing simulated feedback of what is happening under the hood.

G5 The three main visualizations, the inputs tried, the features found, and the corpus indicate what is happening under the hood and provide visualizations of what is happening. However, the current input and the tree are unfortunately not real data from the fuzzer, as the data generated by the fuzzer is not available during execution. As such, the visualization is here but doesn't use the actual fuzzer. The corpus files show the corpus growing and the different sizes that are generated quite well, but is only available for inspection when the fuzzer is not running. The corpus files use the data generated by the fuzzer directly.

G6 There is a real fuzzing process running in the browser of the user. The chosen fuzzer, LibFuzzer is a modern fuzzer and the data and output the fuzzer produces are displayed to the user in real-time. LibFuzzer is generally a good fuzzer that performs reasonably well, but other fuzzers might achieve better results in benchmarks [28].

5.1 Limitations

There are some limitations present in the Fuzzing Playground. Most of these limitations stem from the fact that the entire fuzzing process is run in the browser, and as such, some sacrifices were made.

Without any restraints, AFL [1] would have been favored over libFuzzer, as AFL is generally more popular and is thought to find more coverage quicker. As AFL doesn't run in-process, this was deemed not doable with the Emscripten setup.

The fuzzer isn't as performant as running the same fuzzer in a local machine. While the overhead from Emscripten is probably quite low and in theory could achieve the promised near-native speed [30], the outputs and visualizations do produce some overhead that probably slow down the fuzzing process some more. While execution speed is not that important for an educational tool, it is generally important when fuzzing.

Another limitation is that the user can only choose from some preselected targets and seeds. Allowing the user to bring their target is infeasible since this would either require the ability to compile code in the browser through the WebAssembly virtual machine which is currently not possible, or it would require the user to compile the target themselves which would then greatly detract from the simplicity of the playground and would be a major barrier of entry to use the playground.

The visualizations are still quite limited, mostly owed to the fact that the Emscripten file system is not available during execution, but only after. Better visualizations could perhaps further help the understanding of the fuzzing process. An attempt was made to use the `Module.print` method to extract the corpus whenever something new was found. This however slowed down the web worker too much while the WebAssembly code was still executed at normal speed, which lead to the outputs being mangled and unusable. In the end, this resulted in a crash of the application.

Finally, the Fuzzing Playground also doesn't deal with bugs being found through the fuzzer in the browser. This scenario is very unlikely, as the chosen targets are all extensively fuzzed in OSS-Fuzz [59], but finding bugs and recreating the corresponding inputs nevertheless is an important part of fuzzing.

5.2 Future work

One important aspect that the Fuzzing Playground doesn't address is the actual finding of bugs. Interesting work could be done by reproducing certain high-profile bugs, such as Heartbleed [5], by providing specific outdated targets for the user to choose from. This could help to illustrate the real-world application of fuzzers to users and move fuzzing further away from a theoretical concept to a real-world tool actively being used to find important security issues and bugs.

Many of the limitations could be addressed by moving away from in-browser execution. A smart queue-based system, where users could enter their desired configuration into a queue and get the corresponding data from the server, would offer the possibility to combine installationless fuzzing and greater access to the internal data to the fuzzer. This would, however, require some good hardware for the server and would not scale with more people using it.

By performing a trade-off between the ease of use and interesting insights provided, one could also try to replace the in-browser fuzzing with a simple to use Docker container, in which the fuzzing and corresponding visualization could take place. This would, of course, mean trading some convenience for more data, as starting a Docker container is definitely more effort than simply opening a website, but is still straightforward, especially for people with technical experience and an interest in fuzzing. Running in Docker would provide a pretty easy setup, with greater access to internal data and the file system in real-time. This would also allow for usage of a non in-process fuzzer, such as AFL++ [23] or hongfuzz [27].

Rapid advances in Emscripten, WebAssembly and fuzzers could allow for more flexibility and greater access to low-level systems in the browser. This could further allow for better visualizations and more in-depth looks into the fuzzer. In fact, AFL developers are looking into the possibility of being able to instrument WebAssembly targets [1].

Finally, comparing different fuzzers against each other, such as in FuzzBench [47] could also provide interesting, albeit more advanced insights into the fuzzing process. Trying different fuzzers or different seeds for the same fuzzer in an easy-to-use, minimal setup could help the understanding of more advanced fuzzing users, trying to quickly compare different fuzzers and find their strengths and weaknesses.

6 Conclusion

Fuzzing is important and widely used as a tool to find critical bugs and security issues early to address them promptly [46]. The entry hurdle to using most fuzzers is quite high and the usability of fuzzers, even when used by technical users, leaves a lot to be desired [57].

Advances in web technologies support a wide range of applications that can run complex systems and tools such as a fuzzer in browser [77, 30, 74]. While advancements are quite impressive and having the ability to run compiled code in the browser at fast speeds is exciting, some low-level integration, often the ones necessary for certain types of fuzzing are not always implemented or implementation progress varies across browsers [75].

The Fuzzing Playground provides an easy-to-use, installation-free tool to immediately start a fuzzing process in the user's browser. The fuzzing process takes place on the user's machine entirely and runs a real, modern fuzzer: libFuzzer. The user can control the fuzzing configuration in limited ways, but can quickly compare different configurations against each other. Users can also discern the current state of the fuzzer as well as some additional information. Further, the user can inspect the interesting inputs, the inputs found to provide additional coverage in the corpus.

References

[1] AFL++ Contributors. Ideas for AFL++, 2021. URL https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/ideas.md.

[2] M. Aizatsky, K. Serebryany, O. Chang, A. Arya, and M. Whittaker. Announcing OSS-Fuzz: Continuous fuzzing for open source software, 2016. URL https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html.

[3] A. Arya and C. Neckar. Fuzzing for Security, 2012. URL https://blog.chromium.org/2012/04/fuzzing-for-security.html.

[4] A. Arya, O. Chang, M. Barbella, and J. Metzman. Open sourcing ClusterFuzz, 2019. URL https://opensource.googleblog.com/2019/02/open-sourcing-clusterfuzz.html.

[5] H. Böck. How Heartbleed could've been found, 2015. URL https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html.

[6] M. Böhme and B. Falk. Fuzzing: on the exponential cost of vulnerability discovery. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 713–724, New York, NY, USA, nov 2020. ACM. ISBN 9781450370431. doi: 10.1145/3368089.3409729. URL https://dl.acm.org/doi/10.1145/3368089.3409729.

[7] M. Bohme, V. T. Pham, and A. Roychoudhury. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2019. ISSN 19393520. doi: 10.1109/TSE.2017.2785841.

[8] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 122–131. IEEE, may 2013. ISBN 978-1-4673-3076-3. doi: 10.1109/ICSE.2013.6606558. URL http://ieeexplore.ieee.org/document/6606558/.

[9] C. Cadar, D. Dunbar, D. Engler, C. Cadar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, USA, 2008. USENIX Association. doi: 10.5555/1855741.1855756.

[10] J. Campbell and M. Walker. Microsoft announces new Project OneFuzz framework, an open source developer tool to find and fix bugs at scale, 2020. URL https://www.microsoft.com/security/blog/2020/09/15/microsoft-onefuzz-framework-open-source-developer-tool-fix-bugs/.

[11] M. Carney. Using Teachable Machine in the d.school classroom, 2019. URL https://medium.com/@michellecarney/using-teachable-machine-in-the-d-school-classroom-96be1ba6a4f9.

[12] M. Carney, B. Webster, I. Alvarado, K. Phillips, N. Howell, J. Griffith, J. Jongejan, A. Pitaru, and A. Chen. Teachable Machine: Approachable Web-Based Tool for Exploring Machine Learning Classification. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–8, New York, NY, USA, apr 2020. ACM. ISBN 9781450368193. doi: 10.1145/3334480.3382839. URL https://dl.acm.org/doi/10.1145/3334480.3382839.

[13] D. Christozov, J. Galletly, V. Karagiozov, and S. Bonev. Learning by Doing – the Way to Develop Computer Science Professionals. In *Informatics Education Europe II Conference*, pages 53–59, 2007.

[14] Chromium Contributors. Clang, 2021. URL `https://chromium.googlesource.com/chromium/src/+/HEAD/docs/clang.md`.

[15] L. E. Colson and J. F. Sullivan. Hands-on Engineering: Learning by Doing in the Integrated Teaching and Learning Program. *International Journal of Engineering Education*, 15(1):20–31, 1999.

[16] A. Deveria. WebAssembly, 2021. URL `https://caniuse.com/wasm`.

[17] D. Dunbar, C. Cadar, D. Engler, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, USA, 2008. USENIX Association. doi: 10.5555/1855741.1855756.

[18] Emscripten Contributors. File System API, 2015. URL `https://emscripten.org/docs/api_reference/Filesystem-API.html`.

[19] Emscripten Contributors. Module object, 2015. URL `https://emscripten.org/docs/api_reference/module.html`.

[20] Emscripten Contributors. Emscripten Compiler Frontend (emcc), 2015. URL `https://emscripten.org/docs/tools_reference/emcc.html`.

[21] Emscripten Contributors. library_idbfs.js, 2021. URL `https://github.com/emscripten-core/emscripten/blob/2.0.26/src/library_idbfs.js`.

[22] A. Fioraldi and L. P. Pileggi. FuzzSplore: Visualizing Feedback-Driven Fuzzing Techniques. feb 2021. URL `http://arxiv.org/abs/2102.02527`.

[23] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. AFL++: Combining incremental steps of fuzzing research. *WOOT 2020 - 14th USENIX Workshop on Offensive Technologies, co-located with USENIX Security 2020*, 2020.

[24] N. Gibbins. asm.js and WebAssembly. URL `http://edshare.soton.ac.uk/20638/5/03d-WebAssembly.pdf`.

[25] P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *NDSS'2008 (Network and Distributed Systems Security)*, pages 151–166, 2008.

[26] Google. Teachable Machine. URL `https://teachablemachine.withgoogle.com/`.

[27] Google. honggfuzz, 2021. URL `https://honggfuzz.dev/`.

[28] Google. 2021-04-11 report, 2021. URL `https://www.fuzzbench.com/reports/2021-04-11/index.html`.

[29] Google. OSS-Fuzz: Continuous Fuzzing for Open Source Software, 2021. URL `https://github.com/google/oss-fuzz`.

[30] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien. Bringing the web up to speed with WebAssembly. *ACM SIGPLAN Notices*, 52(6):185–200, 2017. ISSN 15232867. doi: 10.1145/3062341.3062363.

[31] D. Herman, L. Wagner, and A. Zakai. frequently asked questions, 2014. URL http://asmjs.org/faq.html.

[32] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. *Proceedings of the ACM Conference on Computer and Communications Security*, pages 2123–2138, 2018. ISSN 15437221. doi: 10.1145/3243734.3243804.

[33] P. S. Kochhar, D. Lo, J. Lawall, and N. Nagappan. Code coverage and postrelease defects: A large-scale study on open source projects. *IEEE Transactions on Reliability*, 66(4):1213–1228, 2017. ISSN 00189529. doi: 10.1109/TR.2017.2727062.

[34] H. Krasner. The cost of poor quality software in the US: A 2018 report. *Consortium for IT Software Quality (CISQ)*, 2018.

[35] P. LePage and T. Steiner. The File System Access API: simplifying access to local files, 2021. URL https://web.dev/file-system-access/.

[36] J. Liang, M. Wang, Y. Chen, Y. Jiang, and R. Zhang. Fuzz testing in practice: Obstacles and solutions. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, volume 2018-March, pages 562–566. IEEE, mar 2018. ISBN 978-1-5386-4969-5. doi: 10.1109/SANER.2018.8330260. URL http://ieeexplore.ieee.org/document/8330260/.

[37] LLVM Project. libFuzzer – a library for coverage-guided fuzz testing., 2021. URL https://llvm.org/docs/LibFuzzer.html.

[38] LLVM Project. LLVM Download Page, 2021. URL https://releases.llvm.org/download.html.

[39] LLVM Project. SanitizerCoverage, 2021. URL 2021-09-14.

[40] LLVM Project, LLVM Packaging Team, M. Klose, and S. Ledru. Paket: clang (1:10.0-50~exp1), 2021. URL https://packages.ubuntu.com/focal/clang.

[41] V. J. M. Manes, H. Han, C. Han, sang kil Cha, M. Egele, E. J. Schwartz, and M. Woo. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering*, pages 1–20, nov 2018. ISSN 0098-5589. doi: 10.1109/TSE.2019.2946563. URL https://ieeexplore.ieee.org/document/8863940/http://arxiv.org/abs/1812.00140.

[42] J. Metzman. Your Browser is my Fuzzer : Fuzzing Native Applications in Web Browsers, 2019. URL https://github.com/jonathanmetzman/wasm-fuzzing-demo/blob/master/meetup-Fuzzing-Native-Applications-in-Browsers-With-WASM.pdf.

[43] J. Metzman. Demos From My Talk on Fuzzing Native Code in Web Browsers using WASM, 2019. URL https://github.com/jonathanmetzman/wasm-fuzzing-demo.

[44] J. Metzman. Dockerfile, 2019. URL https://github.com/jonathanmetzman/wasm-fuzzing-demo/blob/master/Dockerfile.

[45] J. Metzman. SQLite Demo, 2019. URL https://jonathanmetzman.github.io/wasm-fuzzing-demo/sqlite-fast/sqlite.html.

[46] J. Metzman and A. Ali. Developers are Buzzing on Fuzzing, 2021. URL https://thenewstack.io/developers-are-buzzing-on-fuzzing/.

[47] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya. FuzzBench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1393–1403, New York, NY, USA, aug 2021. ACM. ISBN 9781450385626. doi: 10.1145/3468264.3473932. URL https://dl.acm.org/doi/10.1145/3468264.3473932.

[48] B. Miller, D. Koski, C. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz Revisited - A re-examination of the reliability of UNIX utilities and services. *October*, 1525(October 1995):1–23, 1995. URL http://www.eecs.northwestern.edu/~robby/courses/395-495-2009-fall/fuzz-revisited.pdf.

[49] B. Miller, M. Zhang, and E. Heymann. The Relevance of Classic Fuzz Testing: Have We Solved This One? *IEEE Transactions on Software Engineering*, (February):1–11, 2020. ISSN 19393520. doi: 10.1109/TSE.2020.3047766.

[50] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, dec 1990. ISSN 0001-0782. doi: 10.1145/96267.96279. URL https://dl.acm.org/doi/10.1145/96267.96279.

[51] B. P. Miller, G. Cooksey, and F. Moore. An empirical study of the robustness of MacOS applications using random testing. In *Proceedings of the 1st international workshop on Random testing - RT '06*, page 46, New York, New York, USA, 2006. ACM Press. ISBN 159593457X. doi: 10.1145/1145735.1145743. URL http://portal.acm.org/citation.cfm?doid=1145735.1145743.

[52] A. Mockus, N. Nagappan, and T. T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 291–301. IEEE, oct 2009. ISBN 978-1-4244-4842-5. doi: 10.1109/ESEM.2009.5315981. URL http://ieeexplore.ieee.org/document/5315981/.

[53] Mozilla. WebAssembly, 2021. URL https://developer.mozilla.org/en-US/docs/WebAssembly.

[54] Mozilla. IndexedDB API, 2021. URL https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API.

[55] Mozilla. Web Workers API, 2021. URL https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API.

[56] A. P. Norton and Y. Qi. Adversarial-Playground: A visualization suite showing how adversarial examples fool deep learning. In *2017 IEEE Symposium on Visualization for Cyber Security (VizSec)*, volume 2017-Octob, pages 1–4. IEEE, oct 2017. ISBN 978-1-5386-2693-1. doi: 10.1109/VIZSEC.2017.8062202. URL http://ieeexplore.ieee.org/document/8062202/.

[57] S. Plöger, M. Meier, and M. Smith. A Qualitative Usability Evaluation of the Clang Static Analyzer and libFuzzer with CS Students and CTF Players. In *USENIX Symposium on Usable Privacy and Security (SOUPS) 2021.*, pages 553–572, 2021. ISBN 9781939133250.

[58] Preact Contributors. preact, 2021. URL https://preactjs.com/.

[59] M. Ruhstaller and O. Chang. A new chapter for OSS-Fuzz, 2019. URL https://opensource.googleblog.com/2019/01/a-new-chapter-for-oss-fuzz.html.

[60] K. Sato. Understanding neural networks with TensorFlow Playground, 2016. URL https://cloud.google.com/blog/products/ai-machine-learning/understanding-neural-networks-with-tensorflow-playground.

[61] SCS Computing Facilities Carnegie Mellon University. Ubuntu 16.04 - End of Life, 2020. URL https://computing.cs.cmu.edu/news/2020/eol-ubuntu-1604.

[62] M. Silic and A. Back. The Influence of Risk Factors in Decision-Making Process for Open Source Software Adoption. *International Journal of Information Technology and Decision Making*, 15(1):151–185, 2016. ISSN 02196220. doi: 10.1142/S0219622015500364.

[63] M. Skirpan and T. Yeh. Beyond the Flipped Classroom: Learning by Doing Through Challenges and Hack-a-thons. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 212–217, New York, NY, USA, feb 2015. ACM. ISBN 9781450329668. doi: 10.1145/2676723.2677224. URL https://dl.acm.org/doi/10.1145/2676723.2677224.

[64] D. Smilkov, S. Carter, D. Sculley, F. B. Viégas, and M. Wattenberg. Direct-Manipulation Visualization of Deep Networks. 2017. URL http://arxiv.org/abs/1708.03788.

[65] SSLab at Georgia Tech. Fuzzcoin - let's find bug, 2020. URL https://fuzzcoin.kr/.

[66] SSLab at Georgia Tech. Project Roadmap, 2020. URL https://fuzzcoin.kr/roadmap.

[67] SSLab at Georgia Tech. Are you a bug hunter? Try out FuzzCoin., 2020. URL https://fuzzcoin.kr/readme.

[68] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*, volume 148. Pearson Education, Upper Saddle River, NJ, 1 edition, 2007. ISBN 0-32-144611-9.

[69] Tao Xie, D. Marinov, and D. Notkin. Rostra: a framework for detecting redundant object-oriented unit tests. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pages 196–205. IEEE, 2004. ISBN 0-7695-2131-2. doi: 10.1109/ASE.2004.1342737. URL http://ieeexplore.ieee.org/document/1342737/.

[70] TensorFlow. A Neural Network Playground, 2021. URL https://playground.tensorflow.org/.

[71] J. Vainio. *The Use of Data Visualization in Fuzz*. Master's thesis, University of Oulu, 2014.

[72] R. Vamosi. The Fuzzing Files: The Anatomy of a Heartbleed, 2020. URL https://forallsecure.com/blog/the-fuzzing-files-the-anatomy-of-a-heartbleed.

[73] L. Wagner. asm.js in Firefox Nightly, 2013. URL https://blog.mozilla.org/luke/2013/03/21/asm-js-in-firefox-nightly/.

[74] L. Wagner. Turbocharging the web. *IEEE Spectrum*, 54(12):48–53, dec 2017. ISSN 0018-9235. doi: 10.1109/MSPEC.2017.8118483. URL http://ieeexplore.ieee.org/document/8118483/.

[75] WebAssembly Contributors. Roadmap, 2021. URL https://webassembly.org/roadmap/.

[76] S. Williamson. Is Bitcoin a Waste of Resources? *Review*, 100(2):107–115, 2018. ISSN 00149187. doi: 10.20955/r.2018.107-15. URL https://research.stlouisfed.org/publications/review/2018/02/13/is-bitcoin-a-waste-of-resources.

[77] A. Zakai. Emscripten: An LLVM-to-JavaScript compiler. *SPLASH'11 Compilation - Proceedings of OOPSLA'11, Onward! 2011, GPCE'11, DLS'11, and SPLASH'11 Companion*, pages 301–312, 2011. doi: 10.1145/2048147.2048224.

[78] M. Zalewski. american fuzzy lop (2.52b), 2017. URL https://lcamtuf.coredump.cx/afl/.

[79] H. Zhai, C. Casalnuovo, and P. Devanbu. Test Coverage in Python Programs. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, volume 2019-May, pages 116–120. IEEE, may 2019. ISBN 978-1-7281-3412-3. doi: 10.1109/MSR.2019.00027. URL https://ieeexplore.ieee.org/document/8816791/.

[80] C. Zhou, M. Wang, J. Liang, Z. Liu, C. Sun, and Y. Jiang. VisFuzz: Understanding and intervening fuzzing with interactive visualization. *Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*, pages 1078–1081, 2019. doi: 10.1109/ASE.2019.00106.

A Frontend code

A.1 Read files from IndexedDB

```
    // Generate request to open indexedDB
    const DBOpenRequest = window.indexedDB.open(currentDir, 21); // 21 from
        Emscripten

    // Deal with error (hopefully not)
    DBOpenRequest.onerror = function (event) {
      console.log("Error loading database");
    };

    // Open the database and start on transaction to fetch files
    DBOpenRequest.onsuccess = async function (event) {
      console.log("Database initialized");

      const db = event.target.result;

      // Must be more than 0, otherwise no files.
      if (db.objectStoreNames.length > 0) {
        let transaction = db.transaction(db.objectStoreNames, "readwrite");
        let object_store = transaction.objectStore(db.objectStoreNames[0]);
        let request = object_store.openCursor();
        request.onerror = function (event) {
          console.err("error fetching data");
        };

        let filesNew = [];
        request.onsuccess = function (event) {
          let cursor = event.target.result;
          if (cursor) {
            const key = cursor.primaryKey;
            const value = cursor.value;

            // Do something
            const fileName = key.split("/")[2];
            if (currentFilesList.includes(fileName)) {
              const file = {
                filename: fileName,
                timestamp: value.timestamp,
                contents: value.contents,
                bytelength: value.contents.byteLength,
              };
              filesNew.push(file);
            } else {
              /* nothing */
            }
            cursor.continue();
          } else {
            console.log("Database finished");
            setCurrentFiles(filesNew.reverse());

            // Potentially restart worker
            setTimeout(async () => {
              if (keepRunning) {
                console.log("Keep running is active, starting another run.");
                const newRuns =
                currentRuns * 2 > 100000 ? 100000 : currentRuns * 2;
                setCurrentRuns(newRuns);
                const startedWorker = await startWorker(newRuns);
                setWorker(startedWorker);
```

```
                setRunning(true);
              } else {
                console.log("Keep running is inactive, stopping.");
              }
            });
          }
        };
      }
    };
```

Listing 5: Code to read the files from IndexedDB

A.2 Generate image input visualization

```
    let image = new Image();
    visualInterval = setInterval(() => {
      image.onload = function () {
        glitch({
          seed: Math.floor(Math.random() * 100), // integer between 0 and 99
          quality: Math.floor(Math.random() * 100), // integer between 0 and 99
          amount: Math.floor(Math.random() * 100), // integer between 0 and 99
          iterations: Math.floor(Math.random() * 40), // integer
        })
        .fromImage(image)
        .toDataURL()
        .then(function (dataURL) {
          setCurrentStylisedImage(dataURL);
        });
      };

      image.src = oli;
    }, 1000);
```

Listing 6: Code that generated the image visualization

B Example libFuzzer output

```
    ###### Recommended dictionary. ######
    "\xff\xff\xff\xff\xff\xff\xff\xff" # Uses: 19
    "\xff\xff\xff\xff" # Uses: 14
    "\xff\xff" # Uses: 15
    "\x01\x00\x00\x00\x00\x00\x00\x00" # Uses: 13
    "\x01\x00\x00\x00" # Uses: 14
    ###### End of recommended dictionary. ######
```

Listing 7: A example of a recommended dictionary by libFuzzer

C Docker build

C.1 Dockerfile

```
# Copyright 2019 Google Inc.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
```

```
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
#
    ################################################################################


FROM ubuntu:20.04
ENV DEBIAN_FRONTEND noninteractive

RUN apt-get update && \
    apt-get upgrade -y && \
    apt-get install -y libc6-dev binutils libgcc-7-dev && \
    apt-get autoremove -y

ENV SRC /src
ENV OUT /out
ENV WORK /work
RUN mkdir $SRC $OUT $WORK

# Checkout, build and install llvm
RUN apt-get install -y build-essential make cmake ninja-build git g++-multilib
    python3 python-is-python3
RUN mkdir $SRC/chromium_tools
WORKDIR $SRC/chromium_tools
RUN git clone https://chromium.googlesource.com/chromium/src/tools/clang

ENV LLVM_SRC $SRC/llvm-project
RUN git clone https://github.com/llvm/llvm-project.git $LLVM_SRC

RUN mkdir -p $WORK/build
RUN cd $WORK/build

ENV TARGET_TO_BUILD "host;WebAssembly"

ENV PROJECTS_TO_BUILD "compiler-rt;clang;ldd"
RUN cmake -G "Ninja" -DLIBCXX_ENABLE_SHARED=OFF -
    DLIBCXX_ENABLE_STATIC_ABI_LIBRARY=ON -DLIBCXXABI_ENABLE_SHARED=OFF -
    DCMAKE_BUILD_TYPE=Release -DLLVM_TARGETS_TO_BUILD="$TARGET_TO_BUILD" -
    DLLVM_ENABLE_PROJECTS=$PROJECTS_TO_BUILD $LLVM_SRC/llvm
RUN ninja install

RUN git clone https://github.com/emscripten-core/emsdk.git /src/emsdk
WORKDIR /src/emsdk
RUN apt-get install -y python
RUN ./emsdk install 2.0.26
RUN ./emsdk activate 2.0.26
RUN printf "LLVM_ROOT = '/work/build/bin'\nBINARYEN_ROOT = '/src/emsdk/upstream
    '\nEMSCRIPTEN_ROOT = '/src/emsdk/upstream/emscripten'\nNODE_JS = '/src/
    emsdk/node/14.15.5_64bit/bin/node'\nTEMP_DIR = '/tmp'\nCOMPILER_ENGINE =
    NODE_JS\nJS_ENGINES = [NODE_JS]" > /src/.emscripten-llvm-override

# Activate the emsdk and don't let it overwrite the .emscripten file we need to
# point to our LLVM build.
RUN echo "/src/emsdk/emsdk activate && source /src/emsdk/emsdk_env.sh &> /dev/
    null && cp /src/.emscripten-llvm-override /root/.emscripten" >> /root/.
    bashrc
```

```
58 # RUN echo "PATH=$PATH:/src/emsdk:/src/emsdk/upstream/emscripten:/src/emsdk/
       node/12.9.1_64bit/bin" >> /root/.bashrc
59
60 WORKDIR /src/llvm-project/compiler-rt/lib/fuzzer
61 COPY compile_libfuzzer.sh /src
62 RUN bash /src/compile_libfuzzer.sh
63
64 #lodepng part
65 WORKDIR /
66 RUN apt-get update && apt-get install -y make autoconf automake libtool
67 RUN git clone --depth 1 https://github.com/lvandeve/lodepng.git lodepng     #
       or use other version control
68 WORKDIR lodepng
69 COPY lodepng_fuzzer.cpp ./
70 COPY build.sh .
```

Listing 8: Dockerfile for the environment to build the fuzzing target to Emscripten

C.2 compile_libFuzzer.sh

```
1 #! /bin/bash
2
3 source /src/emsdk/emsdk_env.sh
4 cd /src/llvm-project/compiler-rt/lib/fuzzer
5 CXX=emcc bash build.sh
```

Listing 9: Script to compile libFuzzer with Emscripten

C.3 build.sh

```
1 emcc -s ERROR_ON_UNDEFINED_SYMBOLS=0 -s ALLOW_MEMORY_GROWTH=1 -s EXIT_RUNTIME=1
      -s TOTAL_MEMORY=1GB -O2 -fsanitize-coverage=inline-8bit-counters -lidbfs.
     js lodepng.cpp lodepng_fuzzer.cpp /src/llvm-project/compiler-rt/lib/fuzzer/
     libFuzzer.a -o $OUT/lodepng.js
```

Listing 10: Using the Emscripten drop in command emcc to build the fuzzing target