WasmView: Visual Testing for WebAssembly Applications

Alan Romano and Weihang Wang
alanroma,weihangw@buffalo.edu
Department of Computer Science and Engineering
The State University of New York at Buffalo

ABSTRACT

WebAssembly is the newest language to arrive on the web. It features a binary code format to serve as a compilation target for native languages such as C, C++, and Rust and allows native applications to be ported for web usage. In the current implementation, WebAssembly requires interaction with JavaScript at a minimum to initialize and additionally to interact with Web APIs. As a result, understanding the function calls between WebAssembly and JavaScript is crucial for testing, debugging, and maintaining applications utilizing this new language. To this end, we develop a tool, WasmView, to visualize function calls made between WebAssembly and JavaScript in a web application. WasmView also records the stack traces and screenshots of applications. This tool can help in supporting visual testing for interactive applications and assisting refactoring for code updates. The demo video for WasmView can be viewed at https://youtu.be/kjKxL7L7zxI and the source code can be found at https://github.com/wasmview/wasmview.github.io.

ACM Reference Format:
Alan Romano and Weihang Wang. 2020. WasmView: Visual Testing for WebAssembly Applications. In *42nd International Conference on Software Engineering Companion (ICSE '20 Companion), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377812.3382155

1 INTRODUCTION

WebAssembly is a new bytecode designed for the web. It is designed to efficiently perform the computationally-intensive operations unsuited for JavaScript. All major browsers including Chrome, Firefox, Safari, and Edge support WebAssembly on their desktop and mobile browsers since November 2017 [9]. In December 2019, WebAssembly became an official web standard by the World Wide Web Consortium (W3C) [15].

WebAssembly provides a compilation target for languages such as C, C++ and Rust. Programs are ported to the web by compiling to a WebAssembly binary using a compiler toolchain (e.g., Binaryen [7]). Then, JavaScript WebAssembly APIs are used to run the WebAssembly module alongside JavaScript. In its current form, WebAssembly cannot directly access the Web APIs (e.g., the DOM, Canvas, WebSockets, and WebWorkers API). Instead, JavaScript functions are passed as import functions. Additionally, WebAssembly relies on JavaScript for module initialization. As a result, WebAssembly applications routinely make cross-language calls (between WebAssembly and JavaScript) to perform singular operations. This essentially introduces problems on both worlds. Specifically, a bug in an application could span across functions in both languages. For instance, JavaScript libraries that manipulate `window`-level functions could unexpectedly affect imported JavaScript functions in WebAssembly modules. To support resolving dependencies from code updates, a tool that can clarify the function calls of WebAssembly programs and JavaScript programs is highly desirable. In addition, WebAssembly is commonly used in interactive applications such as video games. A tool that enables visual testing to record screenshots of the interactive apps and the stack traces can facilitate testing and offline analysis.

Currently, WebAssembly debugging is limited to the development tools built into browsers such as Chrome and Firefox. These include functionalities such as viewing the stack and local variables during execution. However, cross-language interaction must be manually reconstructed through source inspection.

To this end, we develop WasmView to facilitate testing, debugging, and maintenance of WebAssembly applications. WasmView traces function calls between WebAssembly and JavaScript to construct visual call graphs that clarify function interactions for a simplified refactoring process. It supports visual testing of these applications by logging the stack traces and screenshots of application execution for future offline analysis.

2 BACKGROUND

WebAssembly is a statically typed language that defines four value types: `i32`, `i64`, `f32`, and `f64`. `i32` and `i64` are 32-bit and 64-bit integers while `f32` and `f64` are 32-bit and 64-bit floating points. The language is assembly-like, so all data structures are composed of the four primitive types. Fig. 1 shows the compilation of a C++ code snippet to WebAssembly. The C++ code on the left performs variable assignments and an addition. The WebAssembly text format in the middle is designed to be friendly for reading. The text syntax presents the stack-based nature of the language through nested expressions and shows examples of WebAssembly instructions, such as `i32.const` and `i32.add`. The text format is meant for debugging while the binary format on the right is how it is delivered to and compiled by browsers.

3 WASMVIEW

WasmView is a visual testing tool for WebAssembly applications that features (1) *visual call graphs* that capture WebAssembly and JavaScript function calls, and (2) *trace logs and screenshots* for interactive applications to support offline analysis and reproducing

ICSE '20 Companion, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7122-3/20/05.
https://doi.org/10.1145/3377812.3382155

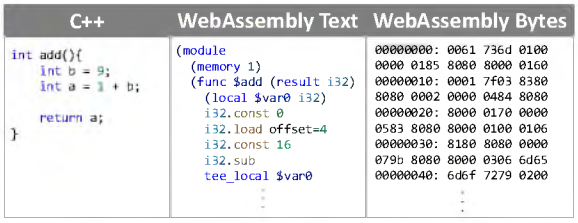


Figure 1: WebAssembly code example.

defects. Fig. 2 shows screenshots of the tool. First, to use WasmView, a WebAssembly application developer enters a URL into the system as shown in Fig. 2(a). Fig. 2(b) shows the interactive browser displayed to the app developer so that the developer can interact with the app. Fig. 2(c) shows an example of the stack traces collected during a 30-second scan. Finally, Fig. 2(d) shows the visual call graph generated for the page.

As shown in Fig. 3, WasmView is comprised of two major components: a *Trace Collector* component and a *Visualizer* component. The Trace Collector is responsible for collecting stack traces, and the Visualizer uses the stack traces to construct visual call graphs for WebAssembly and JavaScript function calls. WasmView also logs the stack traces and screenshots of interactive applications when the app developer interacts with the app. These recorded interaction traces and screenshots can be used to support offline analysis and reproducing defects efficiently.

3.1 Trace Collector

The Trace Collector is responsible for *collecting stack traces* that capture function calls between JavaScript and WebAssembly. To construct call graphs for identified WebAssembly programs across language boundaries, stack traces are used because they contain the chain of JavaScript functions calling exported WebAssembly functions (i.e., JavaScript to WebAssembly calls), as well as the imported JavaScript functions called by WebAssembly (i.e., WebAssembly to JavaScript calls). This is achieved by instrumenting the JavaScript WebAssembly APIs and imported JavaScript functions through the Chromium DevTools controlled by the Puppeteer [6] library. Chromium DevTools is a suite of development tools that allow profiling and modifying a web page as well as controlling the network resources fetched. Puppeteer is a Node.js package that provides an API to interact with the DevTools protocol programmatically.

Fig. 4 shows the Puppeteer connection that accesses the page of a given URL through the Chromium DevTools. In particular, Puppeteer's `evaluateOnNewDocument` method is used to instrument the global `window` object's `WebAssembly` object (①). Internally, the `instantiate` and `instantiateStreaming` methods of the `window.WebAssembly` object are instrumented with code shown in Fig. 5. The stack trace is collected through the `stack` field of `Error` object (i.e., `Error().stack`) when `instantiate` is called (Fig. 5, line 3). An `Error` object is used because it can be accessed from any execution context, including strict-mode scripts [11]. The stack trace is recorded into a storage object in the `window` object (Fig. 5, line 4). When the page finishes executing, the storage object in the `window` object is retrieved using Puppeteer's `evaluateHandle` method (②). Moreover, the `WebAssembly.Instance` object returned from both methods is also instrumented to collect stack traces when any exported methods are invoked (Fig. 5, lines 8-9).

Since WebWorker threads have isolated scopes, the `WebAssembly` object on each Worker's scope is instrumented to collect the same details as done for the `window.WebAssembly` object (③ and ④).

3.2 Visualizer

The Visualizer is responsible for *displaying the call graphs* that capture the WebAssembly and JavaScript function calls. The visual call graphs are constructed by processing the stack trace obtained from the Trace Collector. On the client side, React [13] handles rendering of the page and sending requests to the server, and the mxGraph [8] library draws the constructed call graphs.

4 IMPLEMENTATION AND INSTALLATION

4.1 Implementation

WasmView is a web application running on Node.js [5]. The web server is powered by the Next.js framework [18]. It is implemented utilizing React for the client-side app and mxGraph to draw the graph visualizations.

It starts with a homepage allowing a user to enter the URL of the web application that they would like to analyze. Once the URL is submitted, the web server uses the Puppeteer library to start a new instance of the Chromium browser and inject the instrumentation code before any scripts are run. While the browser visits the URL, the instrumented Chromium instance is shown to the user to interact with the web application in order to trigger WebAssembly actions. The browser stays on the page for 30 seconds. Puppeteer then collects the stored stack traces before the browser instance is closed. The web server sends the traces to the client-side application where the visual call graph is drawn from the traces.

4.2 Installation

The prerequisites for installation are Node.js and Git. The application source can be found at https://github.com/wasmview/wasmview.github.io. To get the tool source code, run the command:

```
$ git clone
    https://github.com/wasmview/wasmview.github.io.git
```

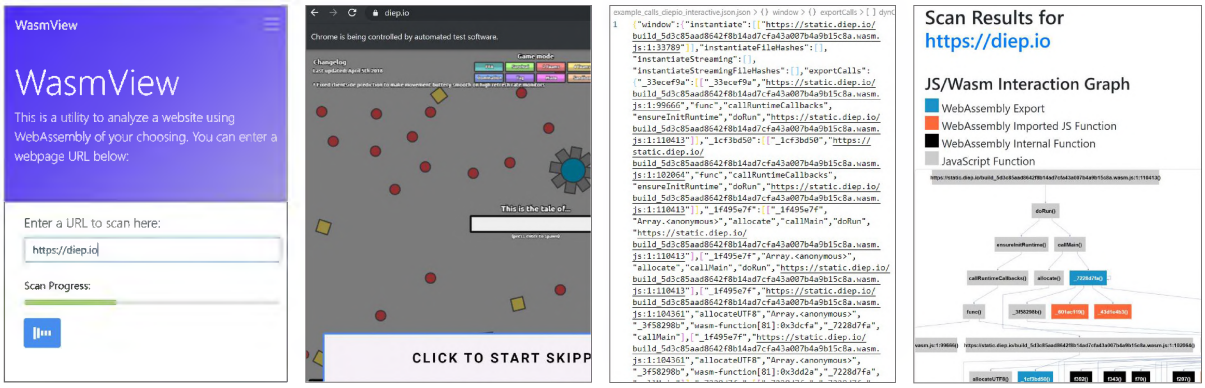
After this, the following commands can be run in the project folder:

```
$ npm install
$ npm run build
$ npm run prod
```

`npm install` installs the tool's dependencies. `npm run build` generates the server pages. The web server is started with the command, `npm run prod`, and can be accessed at the address printed after the command. Settings, such as the server port, can be modified in the `config.json` file in the project root.

4.3 Performance

WasmView performs dynamic analysis through instrumentation and carries a runtime overhead. For the messaging app *cyph.app*, the instrumentation and graph construction added an additional 2.774 seconds of overhead compared with the original time of the scan. For the more complex game *diep.io*, the overhead was 5.998 seconds.



(a) Scanning URL (b) Interactive Browser (c) Logged Calls (d) Resulting Call Graph

Figure 2: WasmView screenshots. (a) shows the first step starting the scan, (b) shows the next step where the browser opens to interact with the app, (c) shows the collected stack traces during a scan, and (d) shows the visual call graph made for the app.

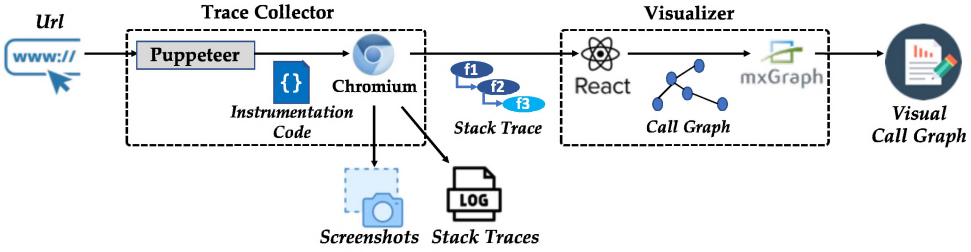


Figure 3: WasmView overview.

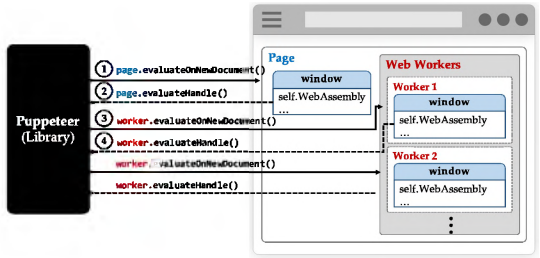


Figure 4: Collect stack trace with Puppeteer.

```
var originalInstantiate = WebAssembly.instantiate;
WebAssembly.instantiate = (buffer, importObject) => {
  var frames = new Error().stack;
  recordInstantiateCall(frames);
  var newImport = instrumentImportCalls(importObject);
  return originalInstantiate(buffer, newImport)
  .then( (instance) => {
    var newInstance = instrumentExportCalls(instance);
    return newInstance;
  });
};
```

Figure 5: Instrumentation code to collect stack trace.

5 CASE STUDY

In the past two years, many compelling use cases for WebAssembly have emerged due its performance advantage and capability of porting native code to the web. For example, WebAssembly has been used to implement games (e.g., *diep.io* [16]), barcode reader applications (e.g., Dynamsoft [4], eBay [12]), and image analysis browser extensions (e.g., Firefox add-on CardSpotter [14]), cryptographic libraries (e.g., supersphincs [2] and Sodium [3]), and online messaging apps (e.g., *cyph.app* [1]).

The visual call graphs and the screenshots effectively facilitate the applications' development. In this section, we use two examples to show how WasmView helps visual testing for the cases that (1) the WebAssembly application is interactive, and (2) the WebAssembly application has API updates.

5.1 Testing for Interactive Applications

diep.io [16] is an online multiplayer game built using the Unity game engine [17]. It features a WebSocket connection to handle multiplayer communication. Fig. 6 shows simplified JavaScript and WebAssembly interactions for *diep.io*:

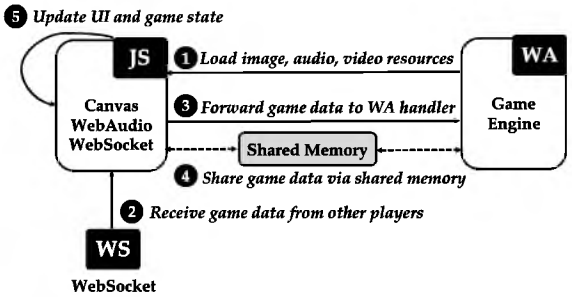


Figure 6: The workflow of the game app *diep.io*.

(1) *diep.io* invokes an exported WebAssembly function to initialize the game (by loading image, audio and video resources).

(2) The page then opens a WebSocket and registers the WebSocket's `onMessage` method to listen to other players' actions.
(3) Since WebAssembly cannot directly interact with the WebSocket APIs, messages received through the WebSocket are forwarded to another exported WebAssembly function.
(4) Internally, the WebAssembly game engine computes a lot of math equations. The results of these computations are shared with JavaScript through a global `DataView` object [10].
(5) Using the shared memory, a JavaScript function in a loop to periodically redraws the screen and updates the game state.

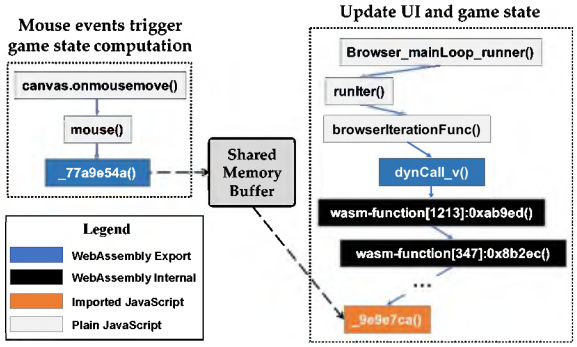


Figure 7: Update UI and game state for the `.mousemove()` event.

When a developer interacts with the game using mouse and keyboard movements, WasmView outputs visual call graphs for JavaScript and WebAssembly functions which assists in testing complex interactive applications. Fig. 7 shows the traces of steps (4) and (5) for a `.mousemove()` event (other steps are omitted for simplicity). In the figure, the left side shows the function calls when the developer moves the mouse in the game canvas. The JavaScript event handler invokes the exported WebAssembly function `_77a9e54a()` to update the game state stored in a memory buffer shared between WebAssembly and JavaScript. The right-side call graphs show the traces to update the UI and game state. The JavaScript function `Browser_mainLoop_runner()` is invoked periodically to eventually call the JavaScript import `_9e9e7ca()` to update the UI from the global game state in the shared buffer.

To enable offline analysis and reproducing defects, WasmView saves trace logs and screenshots of the interactive applications when scanned. These can be used to construct visual call graphs for moments matching the corresponding application screenshots.

5.2 Refactoring for API Updates

WasmView can support code maintenance when refactoring for API updates. Refactoring JavaScript may not lead to syntactic errors but can still cause runtime errors. This means that JavaScript functions calling a WebAssembly function that has undergone a type change may encounter runtime errors that cannot be easily noticed by the developer. Refactoring to resolve code dependencies is especially important for web applications using third-party libraries. In a concurrent study on over 3,000 WebAssembly applications, we found that over half (53.7%) of the observed WebAssembly applications are from third-party services.

As an example, we will look at the encrypted messaging app, *cyph.app*. In order to perform the encryption, it relies on several cryptographic libraries that mix JavaScript and WebAssembly code. Using WasmView, visual call graphs can be generated to see how the multiple libraries interact. The call graph generated from this app (omitted in the paper) shows that the WebAssembly export function, `m()`, is called by three JavaScript functions from separate libraries, `_mceliecejs_decrypted_bytes()`, `_ntrujs_encrypt()`, and `_sidhjs_decrypt()`. Manual inspection shows that function `m()` performs bit operations on a key. Consider the case where `m()` is refactored, changing its parameters. Callers of `m()` must also be updated to accommodate the change. The visual call graphs made by WasmView assists in finding the functions that require changes.

6 CONCLUSION AND FUTURE WORK

We developed WasmView, which features visual call graphs capturing WebAssembly-JavaScript interactions, recorded stack traces, and screenshots of interactive WebAssembly applications. We demonstrate two case studies detailing how WasmView could be used to assist development, debugging, and code refactoring for interactive applications.

In the future, we plan to conduct a user experience study on the tool. We plan on approaching web developers working on WebAssembly projects and compare a group of developers using our tool with a group that relies only on browser tools on criteria such as time to locate defects, time to resolve defects, and time to refactor the code. In addition, we plan to include calls from other WebAPIs.

Acknowledgments. We thank the anonymous reviewers for their constructive comments. This work was partially supported by a Mozilla Research Award.

REFERENCES

[1] Cyph. 2019. Cyph - Encrypted Messenger. https://www.cyph.com/
[2] Cyph. 2019. cyph/supersphincs. https://github.com/cyph/supersphincs
[3] Frank Denis. 2019. jedisct1/libsodium. https://github.com/jedisct1/libsodium
[4] Dynamsoft. 2019. JavaScript Barcode Reader Demo (WebAssembly). https://demo.dynamsoft.com/dbr_wasm/barcode_reader_javascript.html
[5] Node.js Foundation. 2019. Node.js. https://nodejs.org/en/
[6] Google. 2019. GoogleChrome/puppeteer. https://github.com/GoogleChrome/puppeteer
[7] WebAssembly Community Group. 2019. webassembly/binaryen. https://github.com/WebAssembly/binaryen
[8] JGraph. 2019. mxGraph 4.0.4. https://jgraph.github.io/mxgraph/
[9] Judy McConnell. 2019. WebAssembly support now shipping in all major browsers - The Mozilla Blog. https://blog.mozilla.org/blog/2017/11/13/webassembly-in-browsers/
[10] Mozilla. 2019. DataView - JavaScript | MDN. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/DataView
[11] Mozilla. 2019. Strict mode - JavaScript | MDN. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode
[12] Senthil Padmanabhan and Pranav Jha. 2019. WebAssembly at eBay: A Real-World Use Case. https://tech.ebayinc.com/engineering/webassembly-at-ebay-a-real-world-use-case/
[13] React. 2019. React - a javascript library for building user interfaces. https://reactjs.org/
[14] Relgin. 2019. CardSpotter (Magic Card Spotter). https://addons.mozilla.org/en-US/firefox/addon/cardspotter-magic-card-spotter/
[15] Andreas Rossberg. 2019. WebAssembly Core Specification. https://www.w3.org/TR/wasm-core-1/
[16] Diep.io team. 2019. diep.io. https://diep.io/
[17] Marco Trivellato. 2019. WebAssembly Load Times and Performance - Unity Blog. https://blogs.unity3d.com/2018/09/17/webassembly-load-times-and-performance/
[18] ZEIT. 2019. Next.js - The React Framework. https://nextjs.org/