WASim: Understanding WebAssembly Applications through Classification

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 a new programming language built for better performance in web applications. It defines a binary code format and a text representation for the code. At first glance, WebAssembly files are not easily understandable to human readers, regardless of the experience level. As a result, distributed third-party WebAssembly modules need to be implicitly trusted by developers as verifying the functionality requires significant effort. To this end, we develop an automated classification tool WASim for identifying the purpose of WebAssembly programs by analyzing features at the module-level. It assigns purpose labels to a module in order to assist developers in understanding the binary module. The code for WASim is available at https://github.com/WASimilarity/WASim and a video demo is available at https://youtu.be/usfYFIeTy0U.

ACM Reference Format:
Alan Romano and Weihang Wang. 2020. WASim: Understanding WebAssembly Applications through Classification. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3324884.3415293

1 INTRODUCTION

WebAssembly (abbreviated Wasm) is the newest web standard to arrive. Since appearing in 2015 [2], WebAssembly has created huge buzz in the front-end world. Prominent tech companies, such as eBay, Google, and Norton, adopt the technology in user-facing projects for use cases improving performance over JavaScript such as barcode reading [12], pattern matching, and TensorFlow.js machine learning applications [16]. Currently, WebAssembly is supported by all major browsers [9].

The language defines a portable and compact bytecode format to serve as a compilation target for other languages such as C, C++, and Rust. This allows porting native programs to the web as modules and executing at near-native speeds. Rather than being written directly, WebAssembly bytecode is generated using compilers such as Emscripten [1] or Wasm-bingden [14]. WebAssembly also defines a text format meant to ease understanding for debugging. The text format provides a readable representation of the module's internal structure, including type, memory, and function definitions.

Fig. 1 shows a WebAssembly module in the bytecode and the text formats. The C++ code snippet shown in Fig. 1(a) is the source code the WebAssembly module is compiled from. The `main` function assigns two variables and then compares their values. This C++ code is compiled to the WebAssembly binary as shown in Fig. 1(b). The binary format is how a WebAssembly module is delivered to and compiled by browsers. To ease debugging, the WebAssembly binary can be translated to its text format (Fig. 1(c)), and it shows examples of WebAssembly instructions, such as `i32.sub` and `i32.load`.

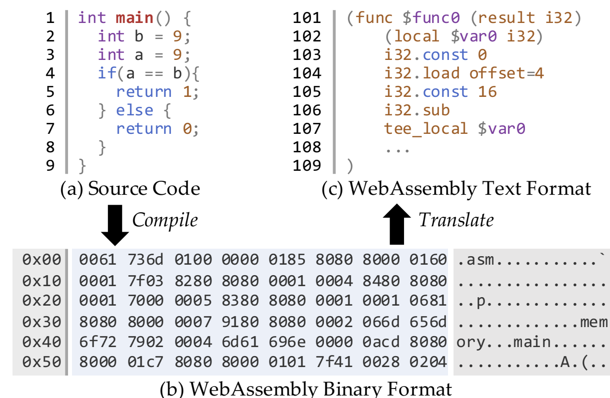

Figure 1: WebAssembly Code Sample

Although readable, the textual format still has a steep learning curve as the assembly-like language is more challenging to understand compared with high-level languages. The language only defines four numeric types, `i32`, `i64`, `f32`, and `f64`, making the text formats of several applications such as string manipulation and cryptographic hashing similar. This makes it difficult to understand a module's functionality from its code. Source maps can be used to find the corresponding functionality in a high-level source language. However, many malicious cryptominers using WebAssembly modules are delivered through third-party services where the source code is not available [10]. For such cases, developers and end users must either verify a WebAssembly module's actual functionality manually or trust its purpose blindly. Additionally, previous work [10] has looked at the purposes of WebAssembly samples, no tools are made to automatically label samples.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASE '20, September 21–25, 2020, Virtual Event, Australia

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3415293

To this end, we develop an automated classification tool, WASim, to help developers and users understand the purposes of WebAssembly applications. WASim analyzes program features at the module-level to use in a machine-learning classifier and identifies WebAssembly modules as applications such as games, cryptographic libraries, grammar utilities, and others.

2 WASIM

WASim identifies the purposes of WebAssembly programs by analyzing code features. As shown in Fig. 2, WASim has three major components: *the Wasm Collector*, *the Feature Extractor*, and *the WebAssembly Classifier*. Given a URL, the Wasm Collector scans the web page for WebAssembly and downloads the WebAssembly binary files executed on the page (if any). Next, the Feature Extractor converts the binary files into WebAssembly text-formatted files, and then scans the text files to extract the desired features. Finally, the WebAssembly Classifier takes in the extracted features and uses them to classify the WebAssembly binary into one of eleven predefined classes. The final output of WASim is a report that lists identified program features and reports the identified purpose.

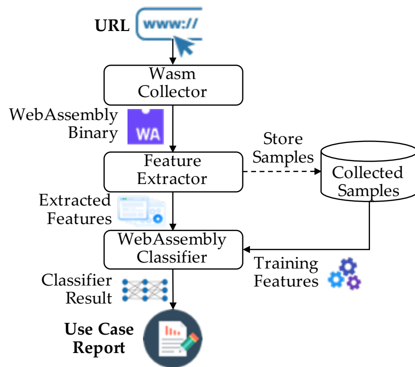

Figure 2: WASim Overview

2.1 Wasm Collector

The Wasm Collector is responsible for *collecting the WebAssembly binary files used on the desired web page*. It works as follows. Given a URL, a Chromium instance is launched, navigated to the URL, and allowed to run for 15 seconds. When the browser loads a WebAssembly module, the bytes are out to a file.

To collect WebAssembly binaries, we compile the Chromium browser from the source with the "`–dump-wasm-module`" flag enabled. This flag dumps any WebAssembly module the browser decodes (largest observed is 45.62MB) to a file. Unlike network request capturing or WebAssembly API instrumentation, the Chromium flag method ensures that any WebAssembly instance run on the page will be saved to a file with negligible runtime overhead.

2.2 Feature Extractor

The Feature Extractor first *converts the WebAssembly binary file to its text format*, and then *extracts desired features from the text file*.

2.2.1 Standard-Conforming Binary Converter. The WebAssembly binary format can be converted to the human readable text format by using toolkits such as WABT [7] and Binaryen [6]. However, since Chromium automatically re-compiles `asm.js` code into WebAssembly binaries that use special asm.js-only opcodes [5], some files are unreadable by WebAssembly binary-to-text tools.

Table 1: Asm.js Opcode Mapping Examples

Instruction Type	Count	Opcode Mapping	Mnemonic Mapping
Integer Division	4	0xd3 ↦ 0x6d	I32AsmjsDivS ↦ i32.div_s
Floating Point Math	11	0xa3 x y ↦ (0x44 x, 0x94) × y	F64Pow(x, y) ↦ (f64.const x, f64.mul) * y
Memory Load	7	0xd7 ↦ 0x2c	I32AsmjsLoadMem8S ↦ i32.load8_s
Memory Store	5	0xde ↦ 0x3a	I32AsmjsStoreMem8 ↦ i32.store8
Type Conversions	4	0xe3 ↦ 0xa8	I32AsmjsSConvertF32 ↦ i32.trunc_f32_s

Table 2: Program Features Used in Classification

Feature Type	Feature
File-Related	Binary File Size Text File Size Expansion Factor
Program Complexity	Total Lines of Code Minimum Lines of Code in Functions Maximum Lines of Code in Functions Average Lines of Code in Functions Number of Functions
WebAssembly Specific	Is an Asm.js Module Number of Types Number of Data Sections Number of Import Functions Number of Export Functions
Function Signature	Names of Import Functions Names of Export Functions

To support these files, we define 31 mapping rules between the `asm.js` opcodes and standard WebAssembly equivalents to safely remove the asm.js opcodes Examples of the `asm.js` instructions and their mappings are shown in Table 1. For example, an instruction `0xd7` (`I32AsmjsLoadMem8S`) is converted into an `0x2c` (`i32.Load8_s`) instruction.

2.2.2 Extracting Features from WebAssembly Text Files. As shown in Table 2, the features of interest include: *file-related*, *program complexity*, *WebAssembly-specific*, and *function signature features*.

These features were carefully chosen based on manual inspection of the text files. File-related features such as the expansion factor between the binary and text formats are simple to obtain and can easily help in broadly demarcating purposes. For example, 0.6KB binary module is unlikely to be a Game or Cryptominer module as these require more functionality than can be fit into 0.6KB. Program complexity features, such as the number of lines of code in functions, also provide useful information. Because of the terse syntax of WebAssembly, important operations can only be reduced so much in terms of lines of code before the functionality can no longer be implemented. Features specific to WebAssembly, such as the numbers of data sections and table entries, can signal probable purpose categories. For example, game modules were found to contain a large number of types and data sections. Finally, the two text features, import/export function names, are useful when deciding between two close choices.

Additionally, WASim constructs a control flow graph (CFG) of the WebAssembly text file by abstracting the instructions into high-level blocks of code used to abstract functions in the text file.

2.3 WebAssembly Classifier

The WebAssembly Classifier uses the extracted features to *determine the purpose category that the WebAssembly file belongs to.*

2.3.1 Preprocessing Features. Function names are embedded as numeric vectors by splitting them into separate words by camel-case and underscore separators and then using the lists of words to train a Doc2Vec [8] embedding. These vectors combined with the other numeric features to use as input for the classifier.

Initially, the CFG was also included into the feature set by embedding as a numeric vector using the graph2vec [11] graph representation. However, we found that it lowered the classifier metrics for the four classifiers tested. For example, inserting the CFG embedding into the neural network model with 8 layers and the 'tanh' activation function dropped the accuracy from 91.6% to 63.49%. It is likely that the control flow graph introduced more noise into the data than anticipated, so it was removed from the input features.

2.3.2 Training the Classifier. The classifier is trained on a data set of collected WebAssembly files. In order to train and evaluate the classifier, the files are manually inspected and tagged with an appropriate category label. Specifically, we manually inspected the import, export, and internal function names which usually carry informative descriptions of the module's purpose. When function names were obfuscated, we relied on external information such as the file's use in its source page.

The manual inspection results in eleven categories:

(1) **Auxiliary Utility:** gives data structures or utility functions.
(2) **Compression Utility:** performs data compression.
(3) **Cryptographic Utility:** performs cryptographic functions.
(4) **Cryptominer:** performs cryptocurrency-mining functions.
(5) **Game Application:** implements full online games.
(6) **Grammar Utility:** performs text or word processing.
(7) **Image Processing Utility:** analyzes or edits images.
(8) **JavaScript Carrier:** stores JavaScript payloads.
(9) **Numeric Utility:** gives mathematical or numeric libraries.
(10) **Support Checker:** checks if WebAssembly is supported.
(11) **Other Applications:** are full standalone programs.

Four types of classifier models, Naïve Bayes, random forest, SVM, and neural network, were evaluated to see which would predict best for use as the classifier in the full WASim system. The classifier is trained and evaluated with a 10-fold cross-validation scheme. These four models were selected because it was assumed that there would be a nonlinear relationship between the features and the label. We chose to include the Naïve Bayes model which favors linear models, random forest and support vector machine which perform well with both linear and nonlinear models, and neural networks since they favor modeling nonlinear models. Each classifier model is discussed in more detail in Sec. 4.

3 INSTALLATION AND USAGE

Installation. WASim is packaged as Docker containers, so Docker and Docker Compose are prerequisites. The `docker-compose.yml` file is needed to run the containers together, and is available on the WASim page, https://github.com/WASimilarity/WASim. Once downloaded, the system can be started by running the following command in the same directory as `docker-compose.yml`:

```
$ docker-compose up
```

This will pull the images from Docker Hub and start a web server on port `4000`. To use WASim without Docker, Python 3.7, Node.js, MySQL, and RabbitMQ need to be installed.

Usage. After starting the web server, WASim can be accessed by navigating a local browser to http://localhost:4000. A web page can be scanned for WebAssembly files using the "Scan" button, or a WebAssembly binary file can be uploaded using the "Upload" button. After performing an action, a results page will show the determined label and probability along with the extracted features.

4 EVALUATION

WASim is built on Node.js [3]. A modified version of Chromium browser version 77 controlled through the Puppeteer library [4] is used to locate the WebAssembly binary. The classifier models are trained and evaluated on a laptop with an Intel Core i7 CPU@2.8GHz and NVIDIA GeForce GTX 1050 GPU. The neural network model is constructed using the Keras library [17]. The naïve Bayes, random forest, and SVM models were obtained from the `scikit-learn` library [15]. The Doc2Vec model was obtained from the `gensim` library.

4.1 Dataset

In total, 734 unique WebAssembly files were collected by crawling the Alexa Top 1 Million websites (4,275 samples), Chrome and Firefox extensions (139 samples), and GitHub repositories (10 samples). The samples from websites and Chrome extensions were collected using the Wasm Collector described in Section 2.1. The samples from Firefox extensions and GitHub repositories were collected by scanning the extensions and repositories for `.wasm` files. The collected files serve various purposes, including cryptography, image processing, and mathematical computations. The binary files range from 0.01KB to 45.62MB with an average file size of 6.05MB. The 734 files were manually labeled to form the dataset.

4.2 Classifier Models

4.2.1 Naïve Bayes Classifier. Naïve Bayes classifiers work by determining conditional probabilities of the classes using Bayes Rule [13] with the features assumed to be conditionally independent. The `alpha` hyperparameter ensures non-zero probabilities and affects the generalizability of the underlying relationship on new data.

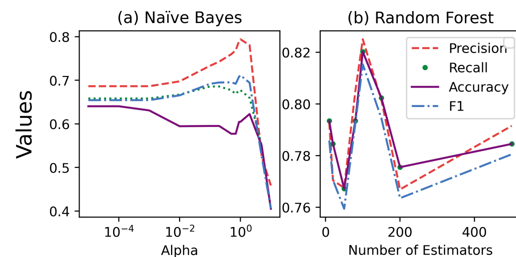

Figure 3: Naïve Bayes and Random Forest Classifier Metrics

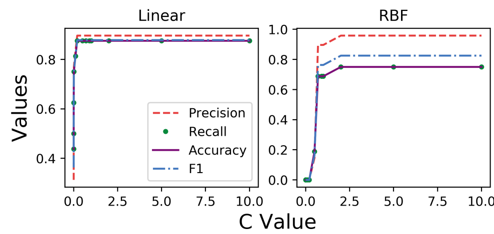

Figure 4: SVM Classifier Metrics

Different alpha values (0.00001, 0.0001, 0.001, 0.01, 0.1, 0.2, 0.5, 0.7, 0.9, 1, 2, 5, and 10) were tested. Fig. 3(a) shows the classification metrics of the Naïve Bayes classifier as the alpha values is increased. The best performing alpha value is 1 with an F1 score of 0.7115 and an accuracy of 64%. While this serves as a good baseline measurement, better predictive performance is desired.

4.2.2 Random Forest Classifier. Random forest classifiers use an ensemble of decision trees to decide the class that the input belongs to [18]. A decision tree uses a series of branching paths split by the possible values of the input features to reach a classification. In a random forest, uncorrelated decision trees are constructed and used to collectively predict the most likely classification.

Different numbers of estimators, 10, 20, 50, 80, 100, 150, 200, and 500, were tested to see which performed best. This parameter affects the number of decision trees used. Adding too many trees can lead to over-fitting and negatively impact the accuracy.

Fig. 3(b) shows the performance of the random forest classifier with different numbers of estimators. Increasing the number of estimators has a positive effect on the F1 score up to 100 estimators. After this, the increased values lead to decreasing F1 scores. The highest accuracy observed was 82% with an F1 score of 0.8153.

4.2.3 Support Vector Machine Classifier. Support vector machine (SVM) classifiers learn to distinguish between classes by learning a hyperplane that separates the data points into classes. The ideal hyperplane provides the maximum margin between data points from every class. Kernel functions transform the data points into different, higher-dimension feature spaces to enable searching for more hyperplanes. SVM uses a regularization parameter, C, as tolerance for misclassified points, with a lower C value leading to more misclassified data points.

The SVM classifier was evaluated with different C values (0.00001, 0.0001, 0.001, 0.01, 0.1, 0.2, 0.5, 0.7, 0.9, 1, 2, 5, and 10) and kernel types (linear and radial basis function). The kernel type affects the accuracy of current data while C affects performance for new data.

Fig. 4 shows the performance of the SVM classifier with different values. We found that the linear kernel provided better results between the two, achieving an accuracy of 87% and F1 score of 0.878. In both kernels, performance plateaus as C gets large.

4.2.4 Neural Network Classifier. Neural networks are constructed from multiple layers of neuron, with neurons of the next layers taking inputs from neurons in the previous layer, applying weights to the input values, summing the values, applying an activation function on the sum, and outputting a value. The weights of each neuron connection are learned during the model training. This classifier was constructed from a sequential network with hidden layers containing 1000 nodes each.

Table 3: Classifier Modeling Times (seconds)

Model	Training	Classification
Naïve Bayes	0.0263	0.00933
Random Forest	1.309	0.294
SVM	1.825	0.00327
Neural Network	12.055	0.0657

The neural network was evaluated with different numbers of layers and different activation functions. The depth of a neural network has a large impact on the accuracy of the model since deeper layers learn better intermediate information. The choice of activation function is also critical to building a useful model.

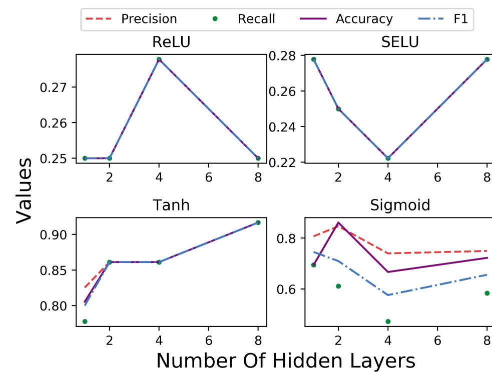

Figure 5: Neural Network Classifier Metrics

Fig. 5 shows the performance of the neural network classifier. Both the ReLU and SELU activation functions perform poorly. Meanwhile, the nonlinear tanh activation function with 8 hidden layers perform best with an accuracy of 91.6% and an F1 score of 0.916. This classifier gives the highest accuracy among those tested.

4.2.5 Training Time and Classification Time. Table 3 shows the average training and classification times for the four types of classifiers. All the classifiers can be trained in a short amount of time, and classification is near-instant for the input features provided to the classifiers. Naïve Bayes is the fastest to train while the neural network takes the longest to train since the multi-layered neural network is a resource-intensive model.

5 CONCLUSION AND FUTURE WORK

We developed an automated classification tool for understanding WebAssembly programs. The classifier leverages file and code features to predict the program purpose. By testing different models to find the optimal one, we achieved a predictive accuracy of 91.6%. Further work on the system could use heuristics to explore links that are most likely to contain WebAssembly as the current system only scans a single URL. In addition, we plan to generalize the classifier beyond the 11 identified categories by analyzing the programs at function level. Finer-grained analysis such as modeling the instructions used within functions would help discover new use cases and improve the classifier precision.

REFERENCES

[1] Emscripten Contributors. 2020. Emscripten 1.39.4 documentation. https://emscripten.org/
[2] Brendan Eich. 2019. From ASM.JS to WebAssembly. https://brendaneich.com/2015/06/from-asm-js-to-webassembly/
[3] Node.js Foundation. 2019. Node.js. https://nodejs.org/en/
[4] Google. 2019. puppeteer. https://github.com/GoogleChrome/puppeteer
[5] Google. 2019. v8/v8. https://github.com/v8/v8/blob/master/src/wasm/wasm-opcodes.cc
[6] WebAssembly Community Group. 2019. webassembly/binaryen. https://github.com/WebAssembly/binaryen
[7] WebAssembly Community Group. 2019. webassembly/wabt. https://github.com/WebAssembly/wabt
[8] Quoc V. Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. *CoRR* abs/1405.4053 (2014). arXiv:1405.4053 http://arxiv.org/abs/1405.4053
[9] Judy McConnell. 2019. WebAssembly support now shipping in all major browsers. https://blog.mozilla.org/blog/2017/11/13/webassembly-in-browsers/
[10] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. 2019. *New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild.* 23–42. https://doi.org/10.1007/978-3-030-22038-9_2
[11] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. 2017. graph2vec: Learning Distributed Representations of Graphs. *CoRR* abs/1707.05005 (2017). arXiv:1707.05005 http://arxiv.org/abs/1707.05005
[12] Senthil Padmanabhan and Pranav Jha. 2020. WebAssembly at eBay: A Real-World Use Case. https://tech.ebayinc.com/engineering/webassembly-at-ebay-a-real-world-use-case/
[13] Savan Patel. 2019. Chapter 1 : Supervised Learning and Naive Bayes Classification. https://medium.com/machine-learning-101/chapter-1-supervised-learning-and-naive-bayes-classification-part-1-theory-8b9e361897d5
[14] rustwasm. 2020. wasm-bindgen. https://github.com/rustwasm/wasm-bindgen
[15] scikit-learn developers. 2019. scikit-learn 0.21.2 documentation. https://scikit-learn.org/stable/supervised_learning.html#supervised-learning
[16] Daniel Smilkov, Nikhil Thorat, and Ann Yuan. 2020. Introducing the WebAssembly backend for TensorFlow.js. https://blog.tensorflow.org/2020/03/introducing-webassembly-backend-for-tensorflow-js.html
[17] Keras Team. 2019. Home - Keras Documentation. https://keras.io/
[18] Tony Yiu. 2019. Understanding Random Forest. https://towardsdatascience.com/understanding-random-forest-58381e0602d2