Automated WebAssembly Function Purpose Identification

Alan Romano and Weihang Wang
University of Southern California, Los Angeles, USA
{ajromano,weihangw}@usc.edu

ABSTRACT

We construct WAspur, a tool to automatically identify the purposes of WebAssembly functions. Specifically, we construct and analyze an extensive and diverse collection of WebAssembly modules. We then construct semantics-aware intermediate representations (IR) of the functions. Finally, we encode the function IR for use in a machine learning classifier. To this end, the classifier can predict the similarity of given function against known named with an accuracy rate of 87.4%. We hope our tool will enable efficient inspection of optimized and minified WebAssembly modules.

1 INTRODUCTION

WebAssembly defines a portable and compact bytecode format to serve as a compilation target for other languages such as C, C++, and Rust. Unfortunately, the bytecode format of WebAssembly makes it *challenging to analyze and understand* the purpose of WebAssembly binaries (e.g., whether it is benign or malicious). In practice, many WebAssembly modules, including malicious modules, are delivered through third-party services where the source code is not available [11], making the problem critical. While previous work [4, 11] has looked at the purposes of WebAssembly programs, which is coarse-grained, there has been little work in understanding the functionality at the WebAssembly module level.

To this end, we develop WAspur, an automated classification tool to understand the intended functionality of individual WebAssembly functions within the applications. WAspur constructs abstractions on the semantic functionality of the module resilient to syntax differences, and these abstractions are used in a machine-learning classifier to identify what functionality the WebAssembly functions implement. We make the following contributions:

- We propose an intermediate representation (IR) to abstract underlying semantics of WebAssembly modules.
- We construct a dataset of diverse WebAssembly samples from real-world websites, Web browser extensions, and GitHub.
- We perform a comprehensive analysis of the collected WebAssembly samples, classifying 10 different purposes.
- We develop WAspur and it achieves an 87.4% accuracy rate.

2 DESIGN

We develop WAspur, an automated tool leveraging a semantics-resilient intermediate representation (IR) designed to capture the effects produced by WebAssembly instructions. WAspur classifies the functions in a WebAssembly module using two main components, as shown in Figure 1. (1) Abstraction Generator, collects the abstractions for all functions within the module to represent each function in our IR. (2) Classifier, uses the sequence of abstracted IR units as input into a neural network classifier. The Classifier is trained on the names of repeatedly found in WebAssembly modules and outputs the probability that an inspected function belongs to the group of functions found having similar names.

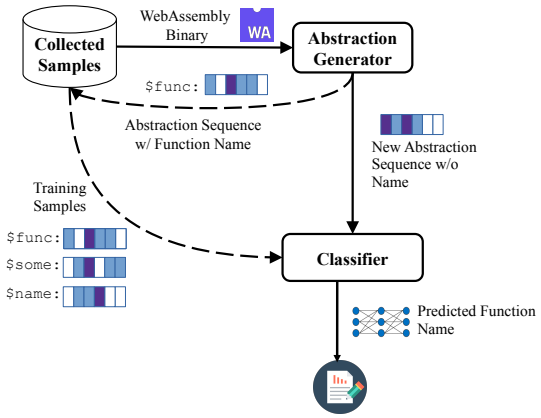

Figure 1: WAspur System Overview

Data Acquisition and Handling. We collected WebAssembly samples from four sources: (1) Alexa top 1 million websites (Oct. 2018~May 2020), (2) 17,682 top Chrome extensions sorted by installed users (all with more than 1,000 users, March 25~30, 2019), (3) 16,385 Firefox add-ons sorted by installed users (on July 30, 2019), and (4) Public Git Archive dataset [10] (on Oct. 3, 2019).

Classifier. Using the IR constructed from the program abstractions, the classifier determines the functionality of a function by predicting the name of a function with a similar abstraction trace.
– *Encoding Abstractions as Features:* It uses a neural network model to predict labels for the given abstraction sequence. Our input into the neural network is the sequence of abstractions produced in the interprocedural control-flow graph of the target function, e.g., "`set set for store if` ...". Such a string is embedded as a numeric vector with an integer representing one of the eight abstraction types we define. The vector requires predefined sequence length, so for abstraction traces longer than the length, we truncate it.
– *Training the Classifier:* We use non-minified names grouped together by their abstraction sequences as the labels for classification. The label strings are encoded using a multi-hot encoding scheme to map each label to an index of a numeric vector. The classifier outputs a vector whose floating-point values correspond to the probabilities that a certain label should apply to the sample. The classifier is trained and evaluated by splitting the dataset into a training set of 80%, a validation set of 10%, and a test set of 10%.
– *Neural Network Architecture:* We tune the hyperparameters of the neural network model by constructing a shallow neural network with two hidden layers of 1024 and 512 units, respectively. Each layer uses the *ReLU* activation function [1]. The abstraction sequence string is embedded in an embedding layer as a numeric vector truncated to at most 250 integers. The output layer consists of 189 units that use the *softmax* activation function. We use Adam gradient descent [7] and configure the network to use 30 iterations for training.

Table 1: # of Category Features and Examples For Each Category.

Category	Export Functions (# of Features and Examples)		Import Functions (# of Features and Examples)		Internal Functions (# of Features and Examples)		File Source (# of Features and Examples)	
C1. Auxiliary Library	193	matches	261	create_element_Document	2,284	GetStrOffset	45	Https Everywhere
C2. Compression	9	lz4BlockEncode	0		32	decompressFunc	19	gorhill/uBlock
C3. Cryptography	1,273	pbkdf2_generate_block	54	BlockHash	16	_sphincsjs_public_key_bytes	953	shanlusun/blockchain
C4. Cryptominer	90	_cryptonight_hash_variant_1	0		52	_cryptonight_destroy	769	bitcoin.co.ua
C5. Game	4,170	Runtime_Animation	2,319	mousedown_callback	60,025	_SparseTextureGLES	168	juegosfriv3.com
C6. Text Processing	48	DjiSpellcheckerWasmMain	0		945	expand_rootword	64	Co:Writer Universal
C7. Image Processing	451	_build_gaussian_coefs	66	draco_receive_decoded_mesh	2,653	decode_RGB565	52	BabylonJS/Website
C8. JavaScript Carrier	1	data	0		0		11	wpost.co
C9. Numeric Processing	40	div_s	3	env.logit	65	sqrt	545	Moonlet Wallet
C10. Support Test Stub	4	test	1	SomeOtherFunction	0		1,273	codeburst.io
C11. Standalone Apps	22,621	BarcodeReader	6,198	pthread	15,069	dlmalloc	61	01alchemist/TurboScript
C12. Unit Test	3,683	good	376	wasi_unstable.fd_renumber	77,581	testFunctionPi	76	xtuc/webassemblyjs

Note: Columns in gray present# of features, where examples are presented on their right-side columns. If there is no features present on a particular column, there is no example.

3 PRELIMINARY RESULTS

Module-Level Categorization. We manually inspect and label the files by mostly relying on four types of information obtained from the WebAssembly binaries: *import function names*, *export function names*, *internal function names*, and *file source*. As shown in Table 2, we categorize the samples into 12 different categories. For each category, we present the counts of modules found in each source location, statistics on the sizes of the WebAssembly binaries in text and binary formats in Table 3 in Appendix. In total, we produce more than 204,619 signatures from the import, export, and internal function names, as summarized in Table 1.

Classifier. WAspur is built on Node.js [2] and Python. The system contains two components. The Abstraction Generator component is implemented as a Node.js application, while the Classifier component is implement using Python. The classifier uses a neural network model constructed using the Keras and TensorFlow.

Table 2: WebAssembly Use Case Categories.

Category	Description
Compression	Performs data compression operations.
Cryptography	Performs cryptographic operations (e.g., hashing).
Game	Implements stand-alone online games.
Text Processing	Performs text or word processing.
Image Processing	Analyzes or edits images.
Numeric Processing	Provides commonly used mathematical or numeric functions.
Support Test Stub	Probes environment for WebAssembly support.
Standalone Apps	Independent standalone programs.
Auxiliary Library	Provides commonly used data structures or utility functions.
Cryptominer	Performs cryptocurrency-mining operations.
Code Carrier	Stores JavaScript/CSS/HTML payloads.
Unit Test	Ensures conformance to language specification.

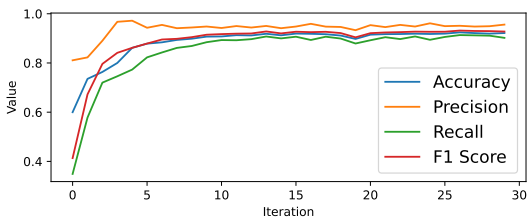

Figure 2: Training Metrics over Iterations

Figure 2 presents the metrics obtained by the neural network while training it over 30 iterations. After training the neural network, evaluating the network on a test set shows that the model can obtain a final accuracy of 87.4%. The precision and recall of the model are 0.9 and 0.86 respectively (F1 score is 0.88).

Performance and Memory Overhead. We find that the neural network within the Classifier can be trained in a short amount of time, and classification is near-instant for the abstraction sequence provided to the classifiers: an average of 47.84 seconds to train and 0.17 seconds to predict labels for the test set.

4 RELATED WORK

[4, 11] aims to study the prevalence and real-world usage of WebAssembly. In particular, prior works [6, 8, 12] analyzing WebAssembly have focused on cryptojacking. There are a few tools [3, 5] made to analyze WebAssembly security and execution. Wasabi [9] is a framework to perform dynamic analysis on WebAssembly code via instrumentation.

5 FUTURE WORK

Currently, the Classifier only uses the abstracted IR of the instructions defined within WebAssembly functions. It may be beneficial to include other additional information from the remaining WebAssembly module sections as well. For example, the predictive performance may be increased by including the function types within the abstracted traversal sequence.

REFERENCES

[1] Abien Fred Agarap. 2018. Deep Learning using Rectified Linear Units (ReLU).
[2] Node.js Foundation. 2019. Node.js. https://nodejs.org/en/
[3] William Fu, Raymond Lin, and Daniel Inge. 2018. TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly. (2018). arXiv:1802.01050
[4] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In *WWW'21*.
[5] H. Jeong, J. Jeong, S. Park, and K. Kim. 2018. WATT : A novel web-based toolkit to generate WebAssembly-based libraries and applications. In *2018 IEEE International Conference on Consumer Electronics*. 1–2.
[6] Amin Kharraz, Zane Ma, Paul Murley, Charles Lever, Joshua Mason, Andrew Miller, Nikita Borisov, Manos Antonakakis, and Michael Bailey. 2019. Outguard: Detecting In-Browser Covert Cryptocurrency Mining in the Wild. In *WWW'19*.
[7] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. https://doi.org/10.48550/ARXIV.1412.6980
[8] Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna. 2018. MineSweeper: An In-depth Look into Drive-by Cryptocurrency Mining and Its Defense. In *ACM CCS'18*. 1714–1730.
[9] Daniel Lehmann and Michael Pradel. 2018. Wasabi: A Framework for Dynamically Analyzing WebAssembly. (2018). arXiv:1808.10652
[10] Vadim Markovtsev and Waren Long. 2018. Public Git Archive: a Big Code dataset for all. *CoRR* abs/1803.10144 (2018). arXiv:1803.10144
[11] 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. In *DIMVA'19*. Springer, 23–42.
[12] Alan Romano, Yunhui Zheng, and Weihang Wang. 2020. MinerRay: Semantics-Aware Analysis for Ever-Evolving Cryptojacking Detection. In *ASE'20*. 12 pages.

Table 3: Categories of the WebAssembly Binary Samples.

Category	# of Websites	# of Chrome Extensions	# of Firefox Add-ons	# of GitHub Repos	LOC (Wat)			Wasm File Size (KB)		
					min	max	avg	min	max	avg
Compression	1	6	10	5	581	284,910	19,090.91	1.20	875.48	64.02
Cryptography	1,348	51	2	48	73	184,318	5,671.93	0.22	433.70	10.36
Numeric Processing	516	2	0	40	8	18,518	358.43	0.04	2,076.63	4.47
Game	279	16	0	22	413	13,190,961	6,447,261.13	0.89	34,957.01	15,638.83
Text Processing	65	2	0	0	511	1,071,478	19,154.64	1.06	2,512.93	47.91
Image Processing	32	0	1	33	67	1,073,067	98,587.79	0.17	2,985.78	240.55
Standalone Apps	15	4	8	76	6	2,590,432	346,139.82	0.05	7,483.21	937.00
Cryptominer	900	0	0	3	20,770	81,124	44,789.89	43.82	163.56	96.39
JavaScript Carrier	99	0	0	0	9	9	9	0.25	0.60	0.42
Auxiliary Library	7	9	22	86	7	485,928	27,773.30	0.05	1,577.53	104.77
Support Test Stub	1,258	0	0	40	1	31	5.82	0.01	0.36	0.03
Unit Test	0	0	0	1,763	1	1,705,804	9,746.07	0.01	5,478.25	30.49

Note: Cells in gray indicate the categories have the top four (or five for the websites) applications.