Automated WebAssembly Function Purpose Identification With Semantics-Aware Analysis

Alan Romano
ajromano@usc.edu
University of Southern California

Weihang Wang
weihangw@usc.edu
University of Southern California

ABSTRACT

WebAssembly is a recent web standard built for better performance in web applications. The standard defines a binary code format to use as a compilation target for a variety of languages, such as C, C++, and Rust. The standard also defines a text representation for readability, although, WebAssembly modules are difficult to interpret by human readers, regardless of their experience level. This makes it difficult to understand and maintain any existing WebAssembly code. As a result, third-party WebAssembly modules need to be implicitly trusted by developers as verifying the functionality themselves may not be feasible.

To this end, we construct WASPur, a tool to automatically identify the purposes of WebAssembly functions. To build this tool, we first construct an extensive collection of WebAssembly samples that represent the state of WebAssembly. Second, we analyze the dataset and identify the diverse use cases of the collected WebAssembly modules. We leverage the dataset of WebAssembly modules to construct semantics-aware intermediate representations (IR) of the functions in the modules. We encode the function IR for use in a machine learning classifier, and we find that this classifier can predict the similarity of a given function against known named functions with an accuracy rate of 88.07%. We hope our tool will enable inspection of optimized and minified WebAssembly modules that remove function names and most other semantic identifiers.

ACM Reference Format:
Alan Romano and Weihang Wang. 2023. Automated WebAssembly Function Purpose Identification With Semantics-Aware Analysis. In *Proceedings of the ACM Web Conference 2023 (WWW '23), May 1–5, 2023, Austin, TX, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3543507.3583235

1 INTRODUCTION

WebAssembly (abbreviated Wasm) is the newest web standard to arrive. Since appearing in 2015 [6], 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 to improve performance over JavaScript in use cases such as barcode reading [35], pattern matching, and TensorFlow.js machine learning applications [44]. Currently, all major browsers support WebAssembly [29].

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 them at near-native speeds. Rather than being written directly, compilers such as Emscripten [4] and Wasmbingden [39] generate WebAssembly bytecode. WebAssembly also defines a text format meant to make debugging easier. The text format provides a readable representation of the module's internal structure, including types, memory limits, and function definitions.

Although readable, the text format still has a steep learning curve compared with high-level languages. There are two characteristics of WebAssembly that make it challenging for human readers to interpret. First, WebAssembly has only four numeric data types, `i32`, `i64`, `f32`, and `f64`, making the instruction sequences of several applications, such as string manipulation and cryptographic hashing, similar. Second, its stack machine design makes deriving the value of a variable at a given location difficult. The stack must be traced from a given location to identify the computed value at a specific code location. These two factors contribute to the difficulty of understanding WebAssembly code. Source maps can be used to find the corresponding functionality in a high-level source language. However, many WebAssembly modules, including malicious modules, are delivered through third-party services where the source code is not available [32]. For such cases, end users need to verify a WebAssembly module's actual functionality manually. Previous work [14, 32] has looked at the purposes of WebAssembly samples. However, there has been little work to help developers understand the functionality implemented by a WebAssembly module.

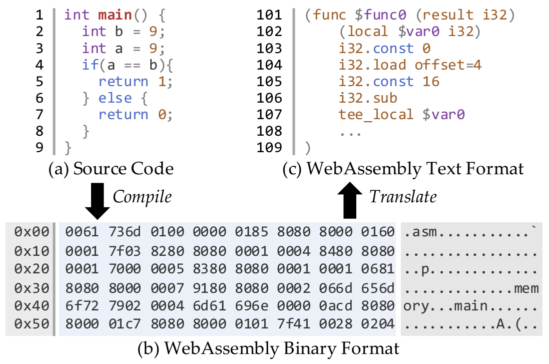
To this end, we develop an automated classification tool, WASPur, to help developers understand the intended functionality of individual WebAssembly functions within the applications. WASPur constructs abstractions on the semantic functionality of the module that are resilient to syntactic differences, and these abstractions are used in a machine-learning classifier to identify what functionality the WebAssembly functions implement.

Specifically, our work makes the following contributions:

- We propose an intermediate representation (IR) to abstract underlying semantics of WebAssembly applications that enables syntax-resilient analysis.
- We construct a dataset of diverse WebAssembly samples by crawling real-world websites, Firefox add-ons, Chrome extensions, and GitHub repositories.
- We perform a comprehensive analysis of the collected WebAssembly samples. We identify the purposes of these samples and classify them into 12 categories.
- We develop an automated classification tool, WASPur, that can accurately label a given WebAssembly function with an appropriate function name according to its functionality with an 88.07% accuracy rate.

WWW '23, May 1–5, 2023, Austin, TX, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9416-1/23/04.
https://doi.org/10.1145/3543507.3583235

2 BACKGROUND



```
int main() {
  int b = 9;
  int a = 9;
  if(a == b){
    return 1;
  } else {
    return 0;
  }
}
```

(a) Source Code

```
(func $func0 (result i32)
    (local $var0 i32)
    i32.const 0
    i32.load offset=4
    i32.const 16
    i32.sub
    tee_local $var0
    ...
)
```

(c) WebAssembly Text Format

```
0x00 | 0061 736d 0100 0000 0185 8080 8000 0160 .asm...........`
0x10 | 0001 7f03 8280 8080 0001 0004 8480 8080 ................
0x20 | 0001 7000 0005 8380 8080 0001 0001 0681 ..p.............
0x30 | 8080 8000 0007 9180 8080 0002 066d 656d ..............mem
0x40 | 6f72 7902 0004 6d61 696e 0000 0acd 8080 ory...main......
0x50 | 8000 01c7 8080 8000 0101 7f41 0028 0204 ...........A.(..
```

(b) WebAssembly Binary Format

Figure 1: WebAssembly Code Sample

WebAssembly defines a compact bytecode format designed to be efficiently transmitted across network calls. The standard also defines a text representation of the bytecode to allow developers to debug the code. Fig. 1 shows a WebAssembly module in the bytecode and the text formats. The C++ code snippet shown in Fig. 1(a) contains a `main` function that assigns two variables and then compares their values. This code compiles to the WebAssembly binary shown in Fig. 1(b). The binary format is how a WebAssembly module is delivered to and compiled by browsers. The WebAssembly binary can be translated to its text format shown in Fig. 1(c), and it shows examples of WebAssembly instructions, such as `i32.sub` and `i32.load`.

WebAssembly modules have a clear structure defined. Each module is composed of 10 sections that each describe different components of the module:

(1) *Types* - This section defines all function types used within the module, including the parameter and result data types.
(2) *Functions* - This section defines all WebAssembly functions by their type, the local variables used, and the function body comprised of a sequence of WebAssembly instructions.
(3) *Tables* - This section defines the function tables used as the targets of indirect function call, i.e., using *call_indirect*.
(4) *Memory* - This section defines the properties of the linear memory sections of the module.
(5) *Globals* - This section lists the global variables that are accessible across all functions in the module.
(6) *Elements* - This section lists the function indices that will be used to initialize a specified function table.
(7) *Data* - This section lists the byte sequences that will be used to initialize the specified linear memory sections.
(8) *Start* - This section defines whether any one function is called once the module initializes.
(9) *Imports* - This section declares the functions imported from JavaScript and will be called within a WebAssembly function.
(10) *Exports* - This section specifies which WebAssembly functions are exported to the host JavaScript context so that they can be invoked there.

Our classification approach focuses on the *Functions* section, as this section contains most the procedural functionality implemented through the WebAssembly instructions.

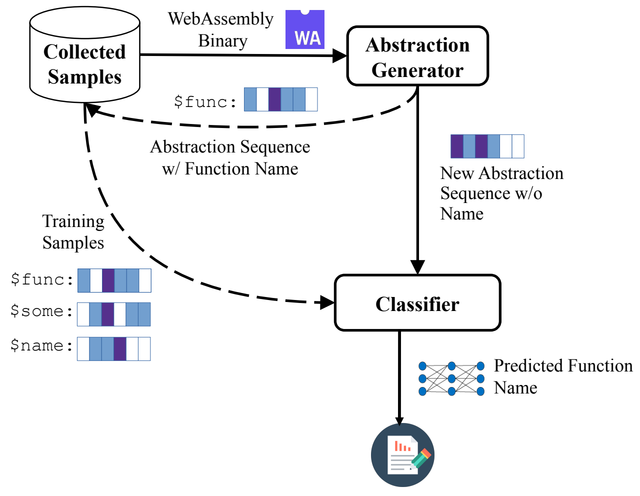


Figure 2: WASPur System Overview

3 SYSTEM DESIGN

To help developers understand the functionality implemented by WebAssembly modules, we develop WASPur, an automated tool that leverages a semantics-aware 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 2. The first component, the Abstraction Generator, collects the abstractions for all functions within the module to represent each function in our IR (Section 3.1). The second component, the Classifier, uses the sequence of abstracted IR units as input into a neural network classifier (Section 3.3). The Classifier is trained on the names of functions repeatedly found in WebAssembly modules and outputs the probability that an inspected function belongs to the group of similar named functions.

3.1 Abstraction Generator

The goal of our approach is to generate a high-level intermediate representation (IR) that can recover semantic meaning from the low-level WebAssembly bytecode. To produce the high-level IR, WASPur is constructed on a core set of abstraction rules:

Definition 1 (Abstraction rule)**.** An abstraction rule is a tuple (S, a, Def) where:

- S represents one or more stack operations that simulate the effects of a WebAssembly instruction on the virtual stack,
- a is a transformation function that maps a WebAssembly instruction to a C-like abstraction, and
- Def is the definition set of all alive variables at the current code location.

We present abstractions rules that abstract WebAssembly bytecode into five groups:

(1) **Numeric Instructions:** Perform numeric computations on stack values.
(2) **Parametric Instructions:** Manipulate virtual stack without additional computations.

Table 1: Abstraction Rules.

Instruction	Stack Simulation[1,2,3]	Definition Set Update	Abstraction
	Numeric Instructions		
i32.const c	push(c)		
i64.mul	pop()→e_1, pop()→e_2, push($e_1 \times e_2$)		
f32.eq	pop()→e_1, pop()→e_2, push($e_2 = e_1$)		
f64.max	pop()→e_1, pop()→e_2, push($max\{e_1,e_2\}$)		
i32.eqz	pop()→e push($e = 0$)		
	Parametric Instructions		
drop	pop()→e		
	Variable Instructions		
get_local v get_global v	push(v)		
set_local v set_global v	pop()→e	Def(v) = Def(v) ∪ $\{e^l\}$	$v = e$;
tee_local	pop()→e, push(e)	Def(v) = Def(v) ∪ $\{e^l\}$	$v = e$;
	Memory Instructions		
i32.load	pop()→e, push(R32(e))		
i32.store	pop()→e_1, pop()→e_2	Def(e_2) = Def(e_2) ∪ $\{\text{addr}(e_1)^l\}$	W32(e_2,e_1);
i32.store8	pop()→e_1, pop()→e_2	Def(e_2) = Def(e_2) ∪ $\{\text{addr}(e_1)^l\}$	W8(e_2,e_1);
	Control Instructions		
loop g:			g:
if I	pop()→e		if (e) {
br g			goto g;
br_if g	pop()→e		if (e) goto g;
block B			B: {
end			}
call f	paraNum(f)→n, for($i = n; i > 0; i$--) pop()→e_i		$f(e_1,e_2,...,e_n)$;
call_indirect	pop()→e, paraNum(e)→n, for($i = n; i > 0; i$--) pop()→e_i		$f(e_1,e_2,...,e_n)$;

1. retLen() gives the number of return values (either 1 or 0) of current function.
2. paraNum(f) returns the number of return values of function f.
3. func(e) returns the function name by checking the function table with index e.

(3) **Control Instructions:** Change the control flow of the program using values from the stack.
(4) **Variable Instructions:** Assign and fetch the local variables.
(5) **Memory Instructions:** Assign and fetch memory values.

Table 1 shows a subset of WebAssembly instructions and how we abstract each group of instructions.

3.1.1 *Numeric Instructions.*

WebAssembly execution is based on a stack machine architecture, so our abstraction models WebAssembly instructions based on their effects on the stack. Modeling the stack allows our abstractions to capture the data and control flows of the program. The first group of abstractions models the instructions that perform numeric computations on stack values. For example, the instruction "`i32.const` c" pushes a 32-bit constant c onto the stack. The prefix `i32` indicates that the data type of c is a 32-bit integer. Similarly, a 64-bit integer is prefixed by `i64`, and a 32-bit floating point number is prefixed by `f32`. We capture these instructions by applying them to symbolic values representing the parameters, variables, and loaded values.

3.1.2 *Parametric Instructions.*

This group includes two WebAssembly instructions, "`drop`" and "`select`". These instructions can drop a value from the stack without performing other computations. We model these instructions through their effects on the virtual stack.

3.1.3 *Control Instructions.*

WebAssembly supports control flow constructs such as `if`, `loop`, and `block`. Conditional statements such as "`if` I" are abstracted to "`if` (e)", where the condition e is popped from the stack. The abstractions defined within the scope of the `if` block are stored in a list of inner abstractions.

We model the "`block`" and "`loop`" instructions using "`block`" and "`for`" abstractions, respectively. Both abstractions define a label, e.g., "`loop` g:", that encapsulates a block of code. The '`br` g" and "`br_if` g" instructions control whether the control flow will go to the beginning of the labeled block. To model this behavior, we also store any abstractions defined with the blocks into the list of inner abstractions. The "`for`" abstraction additionally contains the condition that controls whether the loop terminates.

"`call`" instructions make direct calls with explicit function names, whereas "`call_indirect`" instructions make indirect calls using an index to the function table. We model both instructions with corresponding "`call`" and "`call_indirect`" abstractions.

3.1.4 *Variable Instructions.*

Variable instructions can either load values from local or global variables or assign values to them. To model the instructions loading variables such as "`get_global`", we use symbolic values to represent them on the stack. To model the instructions that assign values to variables, such as "`set_local`", we use the "`set`" abstraction to record information on the targeted variable and assigned value. We also record the history of variable assignments in the variable definition set. For example, to handle the operation "`set_local` v", our definition set would be updated to reflect that the local variable v currently contains the value e.

3.1.5 *Memory Instructions.*

Similar to the variable instructions, there are memory instructions that can load (e.g., "`i64.load`") or store (e.g., "`i32.store`") values in the linear memory. We model the loading instructions using symbolic values to represent the value referred to at a specific memory index. We model the memory store instructions by constructing a "`store`" abstraction that tracks the targeted memory location and value. We also record the history of these memory stores in the variable definition set. For example, "`i32.store8`" would be abstracted to "`W8(`e_2`,` e_1`)`", where e_2 is the source address and e_1 is a destination address. The definition set is updated to reflect that the memory index e_2 contains the value e_1.

We observe that WebAssembly applications typically use *consecutive* memory copy operations, so we merge consecutive "`store`" abstractions into a single abstraction based on two rules. Sequential writes to consecutive memory buffers are simplified to a "`memcpy`" abstraction by inferring the starting address, destination address, and memory length to copy from the "`store`" abstractions. The semantics of writing to consecutive memory buffers can also be realized by using a loop to write each byte and similarly inferring the addresses and memory size to copy.

3.2 Applying Abstractions

To apply our abstractions, we first build an intraprocedural control flow graph (iCFG) for each function. This graph contains the abstraction sequence constructed by traversing the instruction sequence of a single function. A small set of transformations are applied to condense abstractions, such as combining consecutive repeated operations into loops.

After the iCFG of each function is built, we then construct an interprocedural CFG (ICFG) by linking individual iCFGs on "`call`" abstractions. A separate ICFG is built for each function, with the desired function being used as the starting point for graph traversals. We limit the depth to two levels of calls to prevent cycles caused by recursive functions. For "`call_indirect`" abstractions, we link all functions matching the declared function type.

3.3 Classifier

Using the IR constructed from the program abstractions, the classifier determines the functionality of a function by predicting the function name of a function with a similar abstraction trace. We describe the details of how the abstractions are encoded and how the classifier is trained in the following sections.

3.3.1 Encoding Abstractions as Features. The classifier uses a neural network model to predict labels for the given abstraction sequence. The input needs to be encoded into a numeric representation to be fed into the model. Our input into the neural network is the sequence of abstractions produced when traversing the interprocedural control-flow graph (ICFG) of the targeted function. The sequence is then treated as a string of the abstraction types, e.g., "`set set for store if` ..." This string is embedded as a numeric vector with an integer representing one of the eight abstraction types we define. The vector requires a predefined sequence length, so abstraction traces longer than this length are truncated.

3.3.2 Training the Classifier. The classifier is trained on the generated function abstraction sequences of the collected WebAssembly files. To train and evaluate the classifier, the WebAssembly functions with non-minified names are grouped together by their abstraction sequences. We use these function names 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%.

3.3.3 Neural Network Architecture. The neural network underlying our classifier takes in the abstraction sequence as input. An embedding layer encodes the abstraction sequence string as a numeric vector of at most 250 integers. Each hidden layer uses the fully connected *Dense* layer type provided by TensorFlow [1]. The output layer consists of 189 units that use the *SoftMax* activation function. The units in this layer correspond to the indices of the label values predicted by the network. The network is configured to use the cross-entropy loss between true and predicted labels as the loss function, and it uses the Adam gradient descent method [19] as the optimization algorithm. We configure the network to use 30 iterations when training the model.

The classification performance of our model depends on using appropriate values for the hyperparameters. We tune the hyperparameters of the neural network model to identify a suitable configuration for predicting labels from the given abstraction sequence input. To identify the optimal number of hidden layer units, activation function, and number of layers for our model, we construct our neural network using different values of hyperparameters to identify the highest accuracy value that our classifier can attain.

4 DATA COLLECTION AND HANDLING

We collect WebAssembly samples to build the training and evaluation datasets for our neural network model. We describe our process for collecting a diverse set of WebAssembly binary samples from various sources in the following section. We also describe these samples in detail and the use cases that they implement.

4.1 Data Acquisition

We collect WebAssembly samples from four sources: (1) Alexa top 1 million websites, (2) 17,682 top Chrome extensions sorted by installed users, (3) 16,385 popular Firefox add-ons sorted by installed users, and (4) 112 million GitHub repositories.

4.1.1 Alexa Top 1 Million Websites. We crawled the Alexa top 1 million websites from October 2018 to May 2020. For each website, we visited the homepage and all first-level subpages. We decided to limit the crawling to first-level subpages rather than all subpages because a full scan would require hours for complex websites that include thousands of subpages. To download WebAssembly binaries running on a page, we modified the Chromium browser version 77 with the "`-dump-wasm-module`" flag enabled to dump any WebAssembly module the browser decodes [50].

4.1.2 Chrome Extensions. We get WebAssembly samples from Chrome extensions by running all extensions with more than 1,000 users through the modified Chromium browser. We crawled the Chrome extensions from March 25 to March 30, 2019. It took one day to download all the Chrome extensions and four days to assess each extension. This resulted in a total of 17,682 Chrome extensions.

4.1.3 Firefox Add-ons. Samples from Firefox add-ons were obtained by crawling the official Firefox Add-ons website to download the `.xpi` add-on archives. The `.xpi` archives were scanned for the files ending with "`.wasm`". We crawled the Firefox add-ons on July 30, 2019. It took one day to download and scan all the add-ons. In total, 16,385 Firefox add-ons were analyzed.

4.1.4 GitHub Repositories. We obtained WebAssembly samples from the Public Git Archive dataset [27] using the `pga` command line tool [45]. We specified the "`--lang`" filter to obtain the repositories using WebAssembly as the language filter. We then scanned these repositories using the GitHub REST API [15] to find and download all WebAssembly binary files (`.wasm`) [15]. This process took one day and was performed on October 3, 2019.

4.2 Statistics of Collected Data

We collect two different datasets: (1) Dataset for WebAssembly Binaries and (2) Dataset for WebAssembly GitHub Repositories.

Table 2: Statistics of Collected WebAssembly Binaries.

Source	# of Samples	# of Samples Using Wasm	# of Wasm
Websites	1,000,000	3,154	4,520
Chrome Ext.	17,862	55	90
Firefox Add-ons	16,385	30	43
GitHub	112,663,634	435	2,116
	# of Wasm in Total (Distinct)	3,397	6,769 (1,829)

4.2.1 *WebAssembly Binaries.*

Table 2 shows the number of collected WebAssembly binaries for each of the sources crawled. It shows the number of crawled apps/projects from each source, the number of apps/projects that used WebAssembly, and the number of WebAssembly programs identified from each source. In total, we identified 6,769 WebAssembly samples from all sources. These applications/projects can use multiple WebAssembly files to implement complex use cases, thus there were more WebAssembly samples identified than the number of apps/projects. Of the 6,769 WebAssembly modules collected, 1,829 of the modules are unique.

Table 3: GitHub Repository Purposes

Category	Subcategory (# of Repos)	Total
WebAssembly Project		
Application	Game (25), Graphics Library (19), Cryptography (10), Numeric Processing (6), Visualization (4), Blockchain Wallet (3), Machine Learning (3), Sound Processing (3), Compression (2), VR (1), Others (21)	97
Dev. Tools	Development Tools (31)	31
WebAssembly Examples	WebAssembly Examples (98), Benchmark (10), Exploit POC (3)	111
	Total	**239**
Projects for WebAssembly Support		
WebAssembly Runtime	Browser Engine (8), WebAssembly Runtime (41), WebAssembly Interpreter (13), WebAssembly VM (6)	68
Blockchain	Blockchain Platform (36)	36
Frameworks	Application Framework (29)	29
Compiler	WebAssembly Compiler (26)	26
Toolkit	WebAssembly Toolkit (24)	24
Documentation	Documentation (13)	13
	Total	**196**

4.2.2 *GitHub Repositories.*

Among the 112M GitHub repositories, we identify 435 repositories relevant to WebAssembly. Table 3 shows the results of our manual inspection including the purposes of the repositories. These WebAssembly GitHub repositories include 239 repositories that are built with WebAssembly (97 applications, 31 development tools, and 111 WebAssembly examples) and 196 repositories that provide WebAssembly support (68 WebAssembly runtimes, 36 blockchain projects, 29 frameworks, 26 compilers, 24 toolkits, and 13 documentation repositories). Additionally, we identified finer gained subcategories within the groups, such as 25 games and 3 machine learning applications within the Applications group and 8 browser engines and 6 WebAssembly virtual machines within the WebAssembly Runtime group.

4.3 Module-Level Categorization

Identifying the category (i.e., the intended purpose) of WebAssembly programs found in the wild is crucial for understanding the landscape. To discover the intended purposes of the samples, we manually inspect and label the files by 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 4, we categorize the samples into 12 distinct categories. We find a variety of these categories across all the sources of WebAssembly samples that we investigated. For each category, we present the counts of modules found in each source location, statistics on the sizes of the text files, and statistics on the sizes of the binary files in Table 5.

Function names in the modules usually carry informative descriptions of the module's use case. For a source file in a GitHub repository, the file's use in the context of the project is used to identify the category features. For browser extensions, we looked at the extension's description page and the WebAssembly and JavaScript files bundled in the extension archive to identify what the extension's purpose was and what role the WebAssembly module played. In total, we produce more than 204,619 signatures from the import, export, and internal function names of the modules. Table 6 summarizes these signatures.

Table 4: 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.

4.4 Function-Level Categorization

Understanding the purposes of the individual WebAssembly functions comprising the module can also help developers understand the whole module as well. WebAssembly is a compiled language that usually undergoes compiler optimizations. In many cases, these optimizations minify the function names of the WebAssembly functions defined in the module, as well as in the accompanying JavaScript code. We observe 923 modules of 1,829 total modules use minified function names, removing a key piece of information from developers seeking to understand the module functionality.

To identify the intended functionalities of WebAssembly modules, we leverage the presence of function names in the collected modules. These function names can indicate the presence of C, C++, Rust, etc... common utilities, such as *malloc* and *strcmp*. Other function names indicate that the functions implement application-specific behavior, such as the names *AutoThresholdImage* and *IsPDF*.

Table 5: 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.

Table 6: # 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 with examples presented on their right-side columns. If there is no features present on a particular column, there is no example.

We create labels from the names of functions that appear in at least two unique modules. We also condense similar function names, e.g., *$malloc*, *$_malloc*, and *$memory.allocate*, into a single label representing the group, e.g., *malloc*. We obtain 189 different function categories through this process. Function names appearing across multiple module categories indicate that these functions implement common utilities.

5 EVALUATION

We train and evaluate WASPur on the dataset of 1,829 unique WebAssembly files were collected from the Alexa top 1 million websites, Chrome extensions, Firefox add-ons, and GitHub repositories. To use these samples, we extract every function and build the abstraction IR for each function. We then encode the abstraction sequence as a numeric vector to use in the neural network model. To identify the neural network hyperparameters that provide the best predictive power, we measure the classification metrics of our model using different numbers of hidden units within a layer, different activation functions, and different numbers of layers.

5.1 Function Name Dataset

To evaluate WASPur, we use this dataset of WebAssembly modules to build a dataset of individual WebAssembly functions. For each function, we record its name, abstraction traversal sequence, and its parent module. We then use the function name to provide a label for the function, according to our description in Section 4.4. Our function dataset consists of 11,524,686 functions extracted from the 1,829 unique WebAssembly files. Of these functions, 151,662 have function names while 11,373,024 have no function names because of optimization or minification steps. We use the 151,662 labeled functions to train our classifier.

5.2 Classifier Setup

WASPur is built on Node.js [8] and Python [9]. 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 [48] and TensorFlow [1] libraries. The classifier models are trained and evaluated on a laptop with an Intel Core i7 CPU@2.1GHz and 64GB of RAM.

5.3 Classification Results

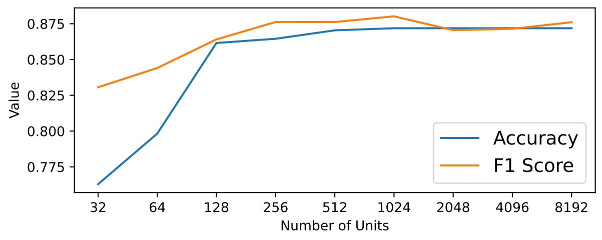


Figure 3: Accuracy and F1 Score using Different Numbers of Hidden Units in a single layer with a Linear Activation Function

5.3.1 Testing Different Numbers of Hidden Units per Layer. In a neural network, each layer consists of hidden units that each have associated weights, biases, and input values from previous layers. Each hidden unit uses the activation function to emit a value for the units in the next layer to process. Adding more hidden units allows for the learned weights to model underlying relationships in the data more closely, but it increases the risk of over-fitting to the training data and generalizing to new data poorly. To find an optimal number of hidden units to use, we evaluate different numbers of hidden units on a single layer. For the numbers of hidden units, we evaluate values ranging from 32 to 8,192. Figure 3 presents the accuracy and F1 scores of the neural network with different numbers of hidden units. We find that using 1,024 hidden units gives the highest accuracy of 87.19% and F1 score of 0.88, so we use this number of hidden units in the following experiments.

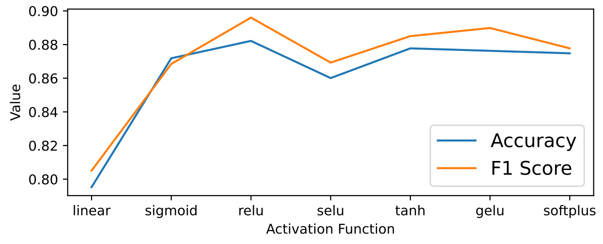


Figure 4: Accuracy and F1 Score using Different Activation Functions in a single layer with 1024 units.

5.3.2 Testing Different Activation Functions. The activation function determines which units within the hidden layers emit an output value to use in the next hidden layer. We test different activation functions provided by the TensorFlow library to find the function that gives the best classification performance. Figure 4 shows the accuracy and F1 scores of the model when using various activation functions in a single layer of 1,024 hidden units. We find that the ReLU activation function performs best with an accuracy of 88.22% and an F1 score of 0.90.

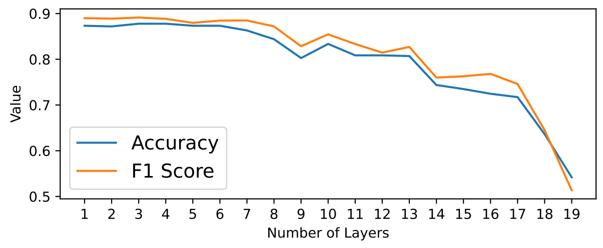


Figure 5: Accuracy and F1 Score over Different Number of Layers with the ReLU Activation Function and 1024 units per layer

5.3.3 Testing Different Numbers of Layers. When constructing neural networks, an increased number of hidden layers can help the network learn relationships that are not linearly separable. However, a large number of layers increases the computational cost of training and can lead to over-fitting the training data. We evaluate our neural network with different numbers of layers ranging from 1 to 20. Figure 5 shows the accuracy and F1 scores of the model when using different numbers of hidden layers, each with 1,024 hidden units and each using the ReLU activation function. We find that the accuracy and F1 scores of the models with 1 to 4 hidden layers show very little change in performance. After 5 hidden layers, both the accuracy and F1 scores begin to decline. We find that 3 hidden layers provides the highest accuracy of 87.78% and F1 score of 0.89.

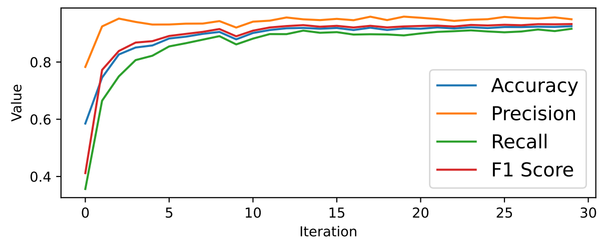


Figure 6: Training Metrics over 30 Iterations

5.3.4 Best-Performing Hyperparameters. Figure 6 presents the training metrics obtained by the neural network using the best-performing hyperparameters: 3 hidden layers, 1024 hidden units per layer, and the ReLU activation function. After training the neural network, evaluating the network on a test set shows that the model can obtain a final accuracy of 88.07%. This metric describes how often the model correctly classifies a sample with the correct label. The precision of the model, which describes how often the model correctly outputs a label whenever it predicts that particular label, is 0.91. The recall of the model, which describes how often the model correctly outputs a label against all the samples that have that label, is 0.87. The model achieves an F1 score of 0.89.

5.4 Performance and Memory Overhead

5.4.1 Training Time and Classification Time. 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. Specifically, the multi-layered neural network takes an average of 47.84 seconds to train and 0.17 seconds to predict labels for the test set.

5.4.2 Abstraction Generation Time and Space Overhead. For the abstraction generation step, the average file size that WASPur processed, among the WebAssembly modules with function names, is 326.37KB. The average memory overhead by WASPur at runtime is 252.01MB. While this memory usage value is large, we note that our prototype implementation can be further optimized to reduce memory usage, and we leave this task as future work. To scan a WebAssembly binary, WASPur constructs both intra- and inter-procedure control flow graphs for each function. The average time spent on the graph construction is 320.94 ms. We also observe that the graph construction time and the file size have a linear relationship, indicating that WASPur is scalable.

6 DISCUSSION

6.1 Threats to Validity

6.1.1 Internal Validity. Our study results are subject to possible errors in the manual inspection processes of labeling the WebAssembly samples and grouping the function names. These subjective steps can be biased due to our inference of the code's intention and the compilation practices in the lack of documentation. To reduce this threat, two authors analyzed the samples separately and discussed inconsistent results until they agreed on the labels.

6.1.2 External Validity. Our choice of when and where to search for WebAssembly samples may affect our results. While we have attempted to search for a large set of WebAssembly files from a variety of sources and during separate times, our findings may not be applied to other datasets. Moreover, we spend significant effort to manually analyze the samples. Specifically, we spent approximately 700 person-hours to analyze the 6,769 collected samples.

6.1.3 Construct Validity. The metrics and measurement procedures we used to assess the prevalence of WebAssembly, its use cases, and portability practices can construct validity threats. We may have missed other measures and metrics that would better or further support our conclusions. To mitigate this threat, we examined our data via several ways of measurement, including analyzing the group statistics of WebAssembly modules quantitatively and studying the individual use cases and indicative signatures qualitatively.

6.2 Future Work

6.2.1 Analyzing Beyond the Functions Section. Currently, the Classifier only uses the abstracted IR of the instructions defined within WebAssembly functions. It may be beneficial to include information from the remaining WebAssembly module sections as well. For example, including the function types within the abstracted traversal sequence may increase the predictive performance. The usage of imported functions defined in the *Imports* section may further increase the semantic information available to the classifier.

6.2.2 Including Symbolic Values. Our abstractions also contain symbolic values modeling function execution. These values capture the intermediate computations performed on the stack values. Leveraging information from the symbolic values, such as the data types involved, could be used to improve the performance.

7 RELATED WORK

WebAssembly Prevalence Study. Prior work aims to study the prevalence of WebAssembly by retrieving WebAssembly samples from web crawling [32]. Hilbig et al. perform an empirical study on the real-world usage of WebAssembly binaries [14].

WebAssembly-Based Cryptominer Studies. Prior works analyzing WebAssembly have focused on cryptojacking. Minesweeper detects WebAssembly-based cryptominers within the Alexa top 1 million [20]. Other works use JavaScript and WebAssembly characteristics of cryptominers to identify them [18, 31, 33]. MinerRay detects hybrid in-browser cryptominers by using an IR to trace semantics across JavaScript and WebAssembly [37].

WebAssembly Analysis Tools. Prior works develop tools to analyze WebAssembly security and execution. The security tools perform authoring [16] and taint tracking [10, 47]. Wasabi is a framework to dynamically analyze WebAssembly code via instrumentation [23]. WASim is a tool that automatically identifies the purpose of a WebAssembly module [36]. In comparison, WASPur identifies the functionalities of individual WebAssembly functions.

Natural Language Processing and Machine Learning on Binary Code. This work uses natural language processing (NLP) and machine learning techniques for program analysis. NLP techniques have been applied to binary and source code for several purposes, such as software categorization [7, 21] and code summarization [12, 22]. Other works have compared the used of traditional NLP methods with machine learning-based techniques for source code modeling [13].

Machine learning techniques have been applied on binary and source code to assist in program analysis [26, 28, 34, 38, 51], categorize software [24], identify vulnerabilities [17], and detect malicious executables [2, 3, 11, 40, 43].

Deep learning techniques have also been used to analyze binaries and source code for improving code maintenance [5, 30, 54], modeling entire software repositories [52], inferring keywords for obfuscated executables [46], detecting malware [41, 42], and measuring binary code similarity [25, 49, 53].

8 CONCLUSION

In conclusion, we present WASPur, an automated classification tool that identifies the purpose of an individual WebAssembly function. The tool leverages a semantics-aware intermediate representation to identify the semantics of WebAssembly functions whose purposes are known. To train and evaluate the classifier, we construct a dataset of 6,769 WebAssembly samples, with 1,829 being unique, collected from real-world websites, Chrome extensions, Firefox add-ons, and GitHub repositories. We describe the use cases and file statistics found from these collected samples to gain insight into the dataset. We evaluate the classifier after training it on labeled functions and find that it achieves an accuracy of 88.07%. We hope our automated classification tool can help developers and end users understand the purposes of the functions within a minified WebAssembly module.

9 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. This work was partially supported by the US National Science Foundation under Grant No. 2047980. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Kevin Allix, Tegawendé Bissyandé, Quentin Jérome, Jacques Klein, Radu State, and Yves Le Traon. 2014. Empirical assessment of machine learning-based malware detectors for Android. *Empirical Software Engineering* 21 (12 2014), 1–29. https://doi.org/10.1007/s10664-014-9352-6

[3] M. A. Atici, S. Sagiroglu, and I. A. Dogru. 2016. Android malware analysis approach based on control flow graphs and machine learning algorithms. In *2016 4th International Symposium on Digital Forensic and Security (ISDFS)*. 26–31. https://doi.org/10.1109/ISDFS.2016.7473512

[4] Emscripten Contributors. 2020. Emscripten 1.39.4 documentation. https://emscripten.org/

[5] C. S. Corley, K. Damevski, and N. A. Kraft. 2015. Exploring the use of deep learning for feature location. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 556–560. https://doi.org/10.1109/ICSM.2015.7332513

[6] Brendan Eich. 2015. From ASM.JS to WebAssembly. https://brendaneich.com/2015/06/from-asm-js-to-webassembly/

[7] J. Escobar-Avila, M. Linares-Vásquez, and S. Haiduc. 2015. Unsupervised Software Categorization Using Bytecode. In *2015 IEEE 23rd International Conference on Program Comprehension*. 229–239. https://doi.org/10.1109/ICPC.2015.33

[8] Node.js Foundation. 2019. Node.js. https://nodejs.org/en/

[9] Python Software Foundation. 2023. Welcome to Python.Org. https://www.python.org/.

[10] William Fu, Raymond Lin, and Daniel Inge. 2018. TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly. *CoRR* abs/1802.01050 (2018). arXiv:1802.01050 http://arxiv.org/abs/1802.01050

[11] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2013. Structural Detection of Android Malware Using Embedded Call Graphs. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security* (Berlin, Germany) *(AISec '13)*. ACM, New York, NY, USA, 45–54. https://doi.org/10.1145/2517312.2517315

[12] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. 2010. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. In *2010 17th Working Conference on Reverse Engineering*. 35–44. https://doi.org/10.1109/WCRE.2010.13

[13] Vincent J. Hellendoorn and Premkumar Devanbu. 2017. Are Deep Neural Networks the Best Choice for Modeling Source Code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 763–773. https://doi.org/10.1145/3106237.3106290

[14] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In *Proceedings of the Web Conference 2021* (Ljubljana, Slovenia) *(WWW '21)*. Association for Computing Machinery, New York, NY, USA, 2696–2708. https://doi.org/10.1145/3442381.3450138

[15] GitHub Inc. 2019. GitHub API v3. https://developer.github.com/v3/

[16] 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 (ICCE)*. 1–2. https://doi.org/10.1109/ICCE.2018.8326230

[17] Matthieu Jimenez, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Yves Le Traon, and Mark Harman. 2019. The Importance of Accounting for Real-world Labelling when Predicting Software Vulnerabilities. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, New York, NY, USA, 695–705. https://doi.org/10.1145/3338906.3338941

[18] 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 *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19)*. ACM, New York, NY, USA, 840–852. https://doi.org/10.1145/3308558.3313665

[19] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. https://doi.org/10.48550/ARXIV.1412.6980

[20] 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 *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. ACM, New York, NY, USA, 1714–1730. https://doi.org/10.1145/3243734.3243858

[21] Alexander LeClair, Zachary Eberhart, and Collin McMillan. 2018. Adapting Neural Text Classification for Improved Software Categorization. *CoRR* abs/1806.01742 (2018). arXiv:1806.01742 http://arxiv.org/abs/1806.01742

[22] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A Neural Model for Generating Natural Language Summaries of Program Subroutines. *CoRR* abs/1902.01954 (2019). arXiv:1902.01954 http://arxiv.org/abs/1902.01954

[23] Daniel Lehmann and Michael Pradel. 2018. Wasabi: A Framework for Dynamically Analyzing WebAssembly. *CoRR* abs/1808.10652 (2018). arXiv:1808.10652 http://arxiv.org/abs/1808.10652

[24] Mario Linares-Vásquez, Collin McMillan, Denys Poshyvanyk, and Mark Grechanik. 2014. On using machine learning to automatically classify software applications into domain categories. *Empirical Software Engineering* 19, 3 (01 Jun 2014), 582–618. https://doi.org/10.1007/s10664-012-9230-z

[25] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. αDiff: Cross-Version Binary Code Similarity Detection with DNN. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. Association for Computing Machinery, New York, NY, USA, 667–678. https://doi.org/10.1145/3238147.3238199

[26] Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck. 2019. TypeMiner: Recovering Types in Binary Programs Using Machine Learning. In *Detection of Intrusions and Malware, and Vulnerability Assessment (Lecture Notes in Computer Science)*, Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren (Eds.). Springer International Publishing, Cham, 288–308. https://doi.org/10.1007/978-3-030-22038-9_14

[27] Vadim Markovtsev and Waren Long. 2018. Public Git Archive: A Big Code Dataset for All. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*. Association for Computing Machinery, New York, NY, USA, 34–37. https://doi.org/10.1145/3196398.3196464

[28] Luca Massarelli, Giuseppe A. Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. 2019. Investigating Graph Embedding Neural Networks with Unsupervised Features Extraction for Binary Analysis. https://doi.org/10.14722/bar.2019.23020

[29] 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/

[30] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: Learning to Repair Compilation Errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, New York, NY, USA, 925–936. https://doi.org/10.1145/3338906.3340455

[31] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. 2018. Web-based Cryptojacking in the Wild. *CoRR* abs/1808.09474 (2018). arXiv:1808.09474 http://arxiv.org/abs/1808.09474

[32] 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 *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 23–42.

[33] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. 2019. Thieves in the Browser: Web-based Cryptojacking in the Wild. In *Proceedings of the 14th International Conference on Availability, Reliability and Security* (Canterbury, CA, United Kingdom) *(ARES '19)*. ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3339252.3339261

[34] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia. 2018. Detecting code smells using machine learning techniques: Are we there yet?. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, Los Alamitos, CA, USA, 612–621. https://doi.org/10.1109/SANER.2018.8330266

[35] 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/

[36] Alan Romano and Weihang Wang. 2020. WASim: Understanding WebAssembly Applications through Classification. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1321–1325. https://doi.org/10.1145/3324884.3415293

[37] Alan Romano, Yunhui Zheng, and Weihang Wang. 2020. MinerRay: Semantics-Aware Analysis for Ever-Evolving Cryptojacking Detection. *Proceedings of the 35th ACM/IEEE International Conference on Automated Software Engineering* (2020).

[38] Nathan E. Rosenblum, Xiaojin Zhu, B. Miller, and Karen Hunt. 2007. Machine Learning-Assisted Binary Code Analysis.

[39] rustwasm. 2020. wasm-bindgen. https://github.com/rustwasm/wasm-bindgen

[40] J. Sahs and L. Khan. 2012. A Machine Learning Approach to Android Malware Detection. In *2012 European Intelligence and Security Informatics Conference*. 141–147. https://doi.org/10.1109/EISIC.2012.34

[41] M. Santacroce, Daniel Koranek, David Kapp, Anca Ralescu, and R. Jha. 2018. Detecting Malicious Assembly with Deep Learning. In *NAECON 2018 - IEEE National Aerospace and Electronics Conference*. 82–85. https://doi.org/10.1109/NAECON.2018.8556657

[42] Joshua Saxe and Konstantin Berlin. 2015. Deep Neural Network Based Malware Detection Using Two Dimensional Binary Program Features. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. 11–20. https://doi.org/10.1109/MALWARE.2015.7413680

[43] M.G. Schultz, E. Eskin, F. Zadok, and S.J. Stolfo. 2001. Data Mining Methods for Detection of New Malicious Executables. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. 38–49. https://doi.org/10.1109/SECPRI.2001.924286

[44] Daniel Smilkov, Nikhil Thorat, Yannick Assogba, Ann Yuan, Nick Kreeger, Ping Yu, Kangyi Zhang, Shanqing Cai, Eric Nielsen, David Soergel, Stan Bileschi, Michael Terry, Charles Nicholson, Sandeep N. Gupta, Sarah Sirajuddin, D. Sculley,

Rajat Monga, Greg Corrado, Fernanda B. Viegas, and Martin Wattenberg. 2019. TensorFlow.js: Machine Learning for the Web and Beyond. Palo Alto, CA, USA. https://arxiv.org/abs/1901.05350

[45] sourced.tech Group. 2020. https://github.com/src-d/datasets/tree/master/PublicGitArchive/pga

[46] Fang-Hsiang Su, Jonathan Bell, Gail Kaiser, and Baishakhi Ray. 2018. Obfuscation Resilient Search Through Executable Classification. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Philadelphia, PA, USA) *(MAPL 2018)*. ACM, New York, NY, USA, 20–30. https://doi.org/10.1145/3211346.3211352

[47] Aron Szanto, Timothy Tamm, and Artidoro Pagnoni. 2018. Taint Tracking for WebAssembly. *CoRR* abs/1807.08349 (2018). arXiv:1807.08349 http://arxiv.org/abs/1807.08349

[48] Keras Team. 2019. Home - Keras Documentation. https://keras.io/

[49] Donghai Tian, Xiaoqi Jia, Rui Ma, Shuke Liu, Wenjing Liu, and Changzhen Hu. 2021. BinDeep: A Deep Learning Approach to Binary Code Similarity Detection. *Expert Systems with Applications* 168 (April 2021), 114348. https://doi.org/10.1016/j.eswa.2020.114348

[50] Mircea Trofin. 2018. Issue 2656103003: [wasm] flag for asm-wasm investigations - Code Review. https://codereview.chromium.org/2656103003/

[51] Shuai Wang and Dinghao Wu. 2017. In-Memory Fuzzing for Binary Code Similarity Analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 319–330. https://doi.org/10.1109/ASE.2017.8115645

[52] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. Toward Deep Learning Software Repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories* (Florence, Italy) *(MSR '15)*. IEEE Press, Piscataway, NJ, USA, 334–345. http://dl.acm.org/citation.cfm?id=2820518.2820559

[53] Shouguo Yang, Long Cheng, Yicheng Zeng, Zhe Lang, Hongsong Zhu, and Zhiqiang Shi. 2021. Asteria: Deep Learning-based AST-Encoding for Cross-platform Binary Code Similarity Detection. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 224–236. https://doi.org/10.1109/DSN48987.2021.00036

[54] Tianchi Zhou, Xiaobing Sun, Xin Xia, Bin Li, and Xiang Chen. 2019. Improving defect prediction with deep forest. *Information and Software Technology* 114 (2019), 204 – 216. https://doi.org/10.1016/j.infsof.2019.07.003