WebAssembly Classification Based on Static Analysis

Akshay Kokane
Computer Science Department
University of Georgia
Athens, USA
Akshay.Kokane@uga.edu

Dr. Kyu Lee
Computer Science Department
University of Georgia
Athens, USA
kyuhlee@uga.edu

***Abstract*—WebAssembly (WASM) is the new player in the world wide web. WebAssembly is supported by all major browsers (Google Chrome, Mozilla Firefox, Safari and IE). The WebAssembly has a fascinating speed and ease of developing the software with traditional languages support like C, C++ and Rust. In early stage of WebAssembly, it became popular among web cryptocurrency miners. Other than cryptocurrency, it is used in web games, libraries and utility software. In this research, I proposed the method to statistically analyzing the WebAssembly code and applying machine learning technique for classifying the WebAssembly modules in four different categories. The statistical analysis was performed at entire WebAssembly module level and at function level.**

I. Introduction

WebAssembly (a.ka. WASM) is the binary instruction format that allows C/C++/Rust code to be executed in the web browser. WebAssembly describes its high-level features of being efficient and fast, safe, open and debuggable, etc. Two of the important features are its fast and its reliable. The WASM apps are said to be as fast as native apps. This feature took much consideration from the researches and developers. Most of the earlier applications were built on C/C++. It was never thought of these applications will one day get a chance to run in the browser. WASM is trying to bring revolution by offering state-of-art native software run on the browser. Major browsers, like Google Chrome, Mozilla Firefox, Safari and IE now supports WebAssembly. Due to its features, it came to the great attention of the web cryptocurrency miner in early phase. It was having a desirable features for web miners, who uses the user's CPU for mining cryptocurrency. Web Mining became a monetizing source for the website owner. They began to mine cryptocurrency by using the CPU power of the users in return for the free content on the website. Mining without consent of user is consider as attack known as cryptojacking. In my past work[5], I used the modified V8 engine[1] for performing taint analysis on the web assembly code. I tried to match the pattern of the web mining website with the randomly selected thousand websites which provide free content. It was found that 60-70 % of these websites do cryptocurrency mining and out of which, 40-50% website uses WebAssembly. Other than cryptocurrency mining, WebAssembly is used for high graphics apps like web games, utlity software, etc. In Unity 2018.2, WebAssembly replaced asm.js as default linker target.

WebAssembly is a binary code format which far more compact than JavaScript. But note that WebAssembly is not the replacement for JavaScript, but they both are meant to work together. JavaScript invokes the WebAssembly functions. So if you want to invoke a function writtern in WebAssembly then you will need JavaScript.

In this paper, the method proposed is to statistically analyzed the WebAssembly code to understand its behavior and to develop a model for classification.

II. WebAssembly

A WebAssembly binary is a sequence of operation codes. WebAssembly (abbreviated WASM) is a binary instruction format for a stack-based virtual machine.[3] The virtual machine then can be embedded in the other programming language like JavaScript. WebAssembly is carefully isolated from rest of the programs.

In the following sections, I would like to explain some of the WebAssembly terms in more details:

- WebAssembly Module : A WebAssembly Module object contains state- less WebAssembly code that can be efficiently shared with Workers, cached in IndexedDB and instantiated multiple times.[4] Module is considered as the fundamental unit of code in WebAssembly. Every WebAssembly code is enclosed in the ¡module¿. So if you open . WASM file in human-readable format (S-expression), you will find, the code is always wrapped inside the ¡module¿ tag.
- WebAssembly Stack Machine: WebAssembly is, in its simplest form, is a stack machine. The more example of languages that are stack machine based are Forth, RPL, PostScript, BibTeX etc. In the stack machine, all of its arithmetic operations works by popping values off the stack. Once the operation is performed on the operands, the result obtained, is again pushed back to stack.
- Linear Memory: WebAssembly uses linear memory. A linear memory is a contiguous, byte addressable range of memory spanning from offset 0 and extending up to varying memory size. This memory can be allocated either by JavaScript via the WebAssembly.Memory method or by WebAssembly. Linear memory can be either imported or defined inside the modules. Linear memory is disjoint

from code space, the execution stack, and the engine's data structures; therefore compiled programs cannot corrupt their execution environment, jump to arbitrary locations, or perform other undefined behavior.

- WebAssembly s-expression format : This format is known as human-readable format of the webassembly. WebAssembly is represented in different formats as show in Figure [1]. The human-readable format was more useful for my work.

C program code	Binary	Text representation
	20 00	get_local 0
	42 00	i64.const 0
	51	i64.eq
int factorial(int n) {	04 7e	if i64
if (n == 0)	42 01	i64.const 1
return 1	05	else
else	20 00	get_local 1
return n * factorial(n-1)	20 00	get_local 1
}	42 01	i64.const 1
	7d	i64.sub
	10 00	call 0
	7e	i64.mul
	0b	end

Fig. 1. WebAssembly Formats.[6]

- WebAssembly Instructions: Like other programming languages, WebAssembly categorizes instructions into Numeric, Parametric, Variable, Memory, Control and Expression instructions.

III. RECENT WORK

In [1], the taint analysis tool called TaintAssembly is been developed by modifying the V8 JavaScript engine. They incorporated the tool in Chromium build. In my earlier work[5], I leverage the TaintAssembly tool for In-browser cryptocurrency detection. Wasbi[2] is the dynamic analysis framework for WebAssembly. It works by statically instrumenting the instruction in WebAssembly binaries(.wasm). The Marius Musch's paper [6] gave a good insight about the number of websites using WebAssembly. They collected the ample of WebAssembly samples from the internet by crawling 1M Top Alexa Domains. Further classification of the WebAssembly codes shows that WebAssembly was used mostly in Game development and second is cryptocurrency mining. The similar research was done in [7], where they collected the Wasm samples and tried to focus on the cryptocurrency mining WebAssembly codes. The result showed that CoinHive was the major Web cryptocurrency miner which used WebAssembly.

IV. METHOD LOGY

My research is focused on developing the method of statically analyzing the WebAssembly codes. Further, based on the analysis classifying the code into different categories. The in depth, explanation of my each step involved in this method is given below:

A. Static Analysis of WebAssembly

As mentioned earlier, WebAssembly can be represented in different forms. One way to analyze it, is to convert to binary to C code using open source WebAssembly Binary Toolkit(WABT)[8]. My initial approach was to convert WebAssembly code to C and run the available static analyzer. But WebAssembly code while compiling from C to WASM, is packed with many WebAssembly system modules like Linear Memory, Function pointers etc. So, analyzing C code had lot of complexity. To overcome this complexity, I decided to go for S-expressions based representation of WebAssembly codes. The S-expression is well-structured and human-readable format of WebAssembly. Taking advantage of the feature of S-expression and combining the WebAssembly semantics and Instruction set, I analyzed the WebAssembly code. I tried to categorized the WebAssembly entire module and selected the most frequently used instructions(like add, and, sub. mul etc). To get better understanding of the control flow, I tried to analyze the code at more granular level. I divided the code function-wise. Then, in each function I categorized each S-expression into categorized like Memory ,Numeric , Variable , Control and Parametric Instructions, imports, table, data etc. Based on the mentioned method, I analyzed the few wasm files of CryptoNight currency, HushCrypto currency, rocket game, games developed using Unity and few simple WebAssembly codes. It was found that cryptocurrency codes shows similar instructions distributions while games shows similar structure in both the case. Further by using [6] open source tool, I generate call flow graph for the above four WebAssembly Code. Figure[2] represents the call flow graph for CryptoNight and Figure[3], is the call flow graph for simple game. Further, I analyzed the CFG by following the path of the the exported functions. Exported functions are the functions which can be invoked from the JavaScript. So considering this fact, I analyze the path starting from invocation functions. Based on the path, I categorized the instructions of each functions in the path.

B. Data

The next step was to collect the data . I collected the samples from three sources. First , was by crawling the Alexa Top 1M domains and took the WASM dump. I followed same technique as mentioned in [6] but just modified little to log website URLs, so that I can classify the WASM dumps. And second source is I collected the samples from the earlier research[6] and [7]. The main difference I found between data I collected and the data they got, is the number of cryptocurrency mining dumps. I found that after collapse of CoinHive, the number of mining websites significantly reduced. The [7] data samples was collected during the time, when CoinHive and other web miners was dominant on the Internet. I got sufficient samples of Web miner from [7]. From earlier research[6], they collected 1050 Wasm Modules from top 1M domains. Out of which only 150 where unique samples. According my analysis, out of 150, 10 samples where having only different function names.

Third source was from know websites (www.wasmRocks.com, github.com,

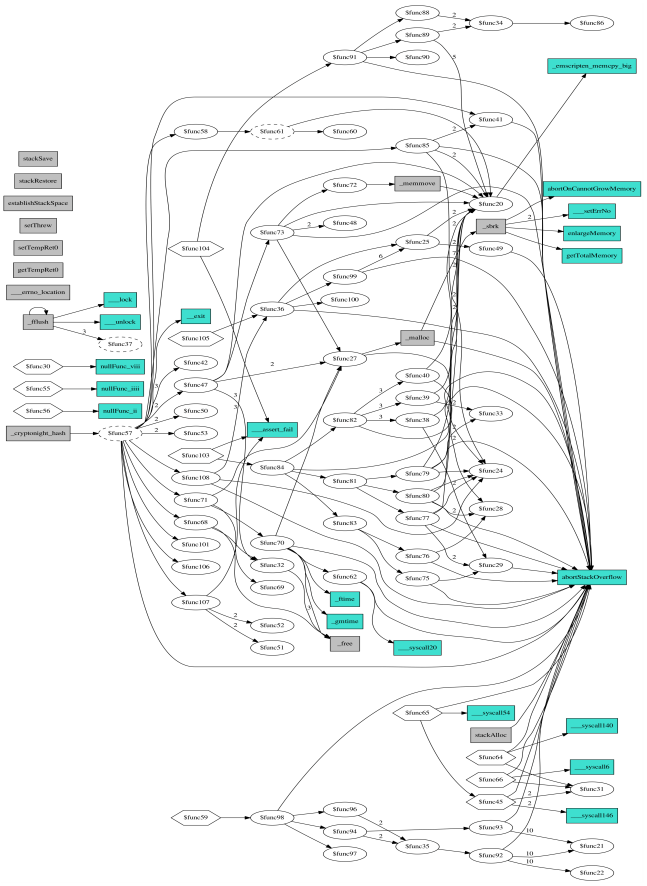

Fig. 2. Call Flow Graph of CryptoNight.

TABLE I
MODULES IN EACH CATEGORIES

Category	# WASM Module
Cryptocurrency	55
Game	43
Util/Library	37
Other/Unknown/Test	18
Total	153

www.webassemblygames.com/) . I manually visited all the know sources and took the WASM dump by using wasm dump flag available in the Chromium debug mode.

Further, I manually categorized the samples into four categories by manually analysis the code and categorizing based on the function names. To be more sure about categorization, I did the deep web crawling for collected WebAssembly codes and verifying its category by visiting the websites. Finally I removed the duplicates and got the final data. Table[1], gives more information about the data

C. Classification Phase

Based on the static analysis and collected samples, I applied different machine learning models. I performed mainly two things : (a) Binary Classification (Cryptocurrency Vs. Benign) (b) Multi-class Classification (Cryptocurrency, Games, Util/Lib, Other/Unknown)

- **Classification of cryptocurrency Vs. benign WebAssembly modules**: I analyzed entire WebAssembly modules irrespective of different blocks(like function, data, memory, etc.) . After analysing entire module, I selected the most frequently occurred instructions. I prepared the dataset with the data of cryptocurrency mining WASM modules and remaining other. I collected 55 cruptocurrency samples and 98 benin samples. I applied Support Vector Machine and Neural Network techniques of binary classification on the dataset.
- **Classification of cryptocurrency Vs. benign WebAssembly Block-level**: To take the analysis on granular level, I analyzed at function or block level. The main difference between module level and block level was, there was many common function between samples. For example, the games developed using Unity , shared some of common functions. So on this dataset, I first removed the common functions and then applied the binary class classification methods. Before removing duplicates functions, total of 3301 cryptocurrency samples and 794890 benign samples was part of dataset. I applied the multi-class machine learning techniques KNeighborsClassifier and Neural Network.

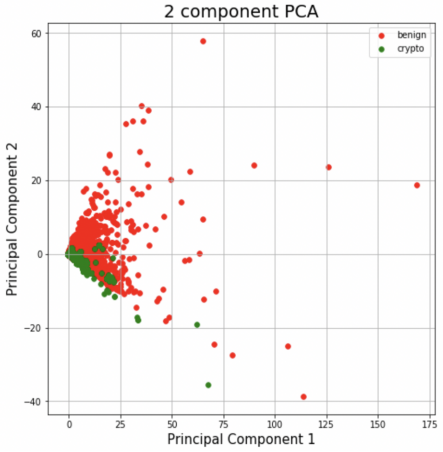

Fig. 3. Plot of 2 component PCA

- **Multi-class Classification of WebAssembly Modules**: As stated earlier, I categorized the WebAssembly samples into four categories. The classes was not as well separated when it came to Game and Crypto, but Util and Other/Test samples was well separated. The number of samples in each dataset can be seen from Table[1] . The machine learning techniques I applied considering the data are, Logistic Regression, SVM, Random Forest Classifier, MLPClassifier
- **Multi-class Classification of WebAssembly Block-wise**: The Table[2], shows information about the dataset. The machine learning techniques I applied on the the data

TABLE II
DATASET

Category	# WASM Functions
Cryptocurrency	3301
Game	765142
Util/Library	27756
Other/Unknown/Test	1992
Total	798,191

are, Logistic Regression, SVM, Random Forest Classifier, MLPClassifier.

V. RESULTS

- **Classification of cryptocurrency Vs. benign WebAssembly Modules**: This classification suffer from the less training dataset. Also in this classification, lot of redundant data was present as we didn't work on more granular level. The SVN showed 67% accuracy. The accuracy was expected due to fact, that classes seems to be not linearly separable. The SVN suffered from low recall. Neural Network gave good accuracy of 97 %. But the number of examples was not enough to firmly say about the Neural Network accuracy.
- **Classification of cryptocurrency Vs. benign WebAssembly Block-level**: The KNeighborsClassifier gave the accuracy of 99% and Neural Network gave accuracy around 98%. The both the classifier gave good precision and recall. The result obtained at block-level classification was more descent as compare to module-wise. The reason was the data was more good in quality and as well as quantity.
- **Multi-class Classification of WebAssembly Modules**:The accuracy of Logistic Regression, SVM, and MLPClassifier was almost same. It was in range of 50-60%. Random Forest Classifier gave accuracy 80 %. The multi-class classification in this case, also suffered from less number of data and also was not uniformly distributed. Fig[4] put forward the plot to compare the obtained predictive accuracy.
- **Multi-class Classification of WebAssembly Block-level**: The accuracy of Logistic Regression, SVM, and MLPClassifier, Random Forest was all above 90%. Random Forest Classifier gave highest accuracy 95 %. The dataset was sufficient to have the confidence on the result. Fig. 5 can help to compare the accuracy across different classifiers.

VI. DISCUSSION AND FUTURE NOTES

The analysis done on the WebAssembly samples was quite straight forward. Further the analysis on following path of Call Flow Graph, can help to give more significant result. The static analysis can be made more in sophisticates fashion and may be combining with the static analysis of C code with S-expression format will give us more insights.

WebAssembly is quite the new and in developing phase. But the applications using WebAssembly, where mostly involved

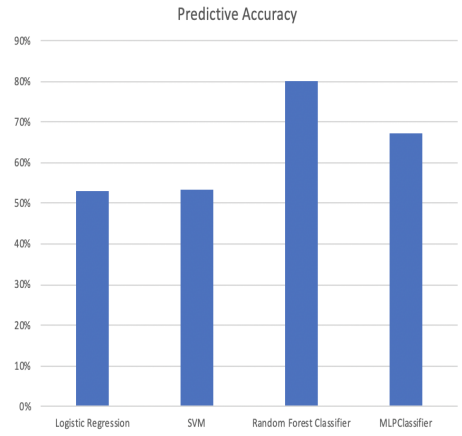

Fig. 4. Plot of predictive accuracy of multi-class classification module level

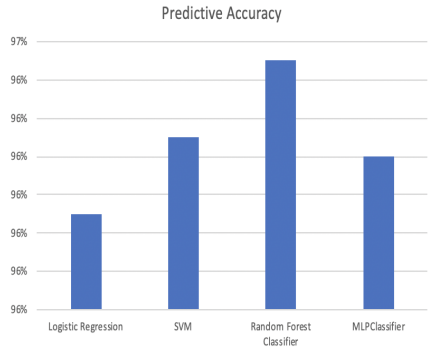

Fig. 5. Plot of predictive accuracy of multi-class classification block level

in some malicious intention. Due to the very reason, the WebAssembly may not got the popularity as it was expected. WebAssembly may gave power to the native software to shift to web platform. In future, the video streaming and web games can be at very advantage due to WebAssembly.

VII. CONCLUSION

My work provided the method of statistically analyzing WebAssembly code at module and block levels and applying binary and multi class classification machine learning techniques. The result obtained shows, block-wise classification provides better predictive accuracy as compared to module level. The method proposed was the foundation for statistically analyzing the WebAssembly code.

REFERENCES

[1] Fu, W., Lin, R. and Inge, D., 2020. Taintassembly: Taint-Based Information Flow Control Tracking For Webassembly. [online] arXiv.org. Available at: ¡https://arxiv.org/abs/1802.01050¿

[2] Lehmann, D. and Pradel, M., 2020. Wasabi: A Framework For Dynamically Analyzing Webassembly. [online] arXiv.org. Available at: ¡https://arxiv.org/abs/1808.10652¿.

[3] Webassembly.org. 2020. Webassembly. [online] Available at: ¡https://webassembly.org/¿
[4] McFadden, B., Lukasiewicz, T., Dileo, J. and Engler, J., 2020. Security Chasms Of WASM. [online] I.blackhat.com. Available at: ¡https://i.blackhat.com/us-18/Thu-August-9/us-18-Lukasiewicz-WebAssembly-A-New-World-of-Native-Exploits-On-The-Web-wp.pdf¿.
[5] GitHub. 2020. Akshaykokane/In-Browser-Cryptocurrency-Mining-Detection-Using-Taintassembly. [online] Available at: ¡https://github.com/akshaykokane/In-browser-Cryptocurrency-Mining-Detection-using-TaintAssembly¿ [Accessed 6 April 2020].
[6] Musch, M., Wressnegger, C., Johns, M. and Rieck, K., 2019, June. 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 (pp. 23-42). Springer, Cham.
[7] Konoth, R., Vineti, E., Moonsamy, V., Lindorfer, M., Kruegel, C., Bos, H. and Vigna, G., 2020. Minesweeper — Proceedings Of The 2018 ACM SIGSAC Conference On Computer And Communications Security. [online] Dl.acm.org. Available at: ¡https://dl.acm.org/doi/10.1145/3243734.3243858¿ [Accessed 6 April 2020].
[8] GitHub. 2020. Webassembly/Wabt. [online] Available at: ¡https://github.com/WebAssembly/wabt¿.
[9] Dl.acm.org. 2020. Bringing The Web Up To Speed With Webassembly — Proceedings Of The 38Th ACM SIGPLAN Conference On Programming Language Design And Implementation. [online] Available at: ¡https://dl.acm.org/doi/10.1145/3062341.3062363¿.