WasmFuzzer: A Fuzzer for WebAssembly Virtual Machines

Bo Jiang, Zichao Li, Yuhe Huang
State Key Laboratory of Software Development Environment
School of Computer Science and Engineering
Beihang University
Beijing, China
{jiangbo, lizichao, yhhuang}@buaa.edu.cn

Zhenyu Zhang
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing, China
zhangzy@ios.ac.cn

W.K. Chan
Department of Computer Science
City University of Hong Kong
Hong Kong
wkchan@cityu.edu.hk

***Abstract*—WebAssembly is a fast, safe, and portable low-level language suitable for diverse application scenarios. And The WebAssembly virtual machines are widely used by Web browsers or Blockchain platforms as execution engine. When there is a bug in the implementation of the Wasm virtual machine, the execution of WebAssembly may lead to errors or vulnerability in the application. Due to the grammar checks by WASM VMs, fuzzing at the binary level is ineffective to expose the bugs because most inputs cannot reach the deep logic within the WASM VM. In this work, we propose WasmFuzzer, a bytecode level fuzzing tool for WASM VMs. WasmFuzzer proposes to generate initial seeds for Fuzzing at the Wasm bytecode level and it also designs a systematic set of mutation operators for Wasm bytecode. Furthermore, WasmFuzzer proposes an adaptive mutation strategy to search for the best mutation operators for different fuzzing targets. Our evaluation on 3 real-life Wasm VMs shows that WasmFuzzer can significantly outperform AFL in terms of both code coverage and unique crash.**

Keywords-fuzzing; WebAssembly; Virtual Machine

I. INTRODUCTION

In order to improve the performance of Web applications, a number of companies and organizations have designed and implemented a new low-level language that can be executed across platforms, called WebAssembly [1].

WebAssembly was born in Web technology, and many browsers, including Chrome, have provided compatibility, allowing WebAssembly code files to be embedded in Web pages. WebAssembly modules will be able to call into and out of the JavaScript context and access browser functionality through the same Web APIs accessible from JavaScript. WebAssembly has some great features. First, it is a refined target language, with a significantly shorter code length than both scripting languages and many compiled native codes. As a result, it has a small footprint for deployment. Secondly, the instruction set of WebAssembly is designed to correspond directly to CPU instructions as much as possible. In some experiments, it runs more than 20 times faster than JavaScript and is more suitable for implementing more complex applications. Furthermore, since WebAssembly is a back-end language supported by LLVM [2], many source codes that support LLVM toolchains, including C [3], C++ [4], Rust [5], can be compiled into WebAssembly code, which allows much software originally implemented in traditional languages to generate WebAssembly code with the same functionality after adaptation, not only reducing the difficulty of program migration, but also allowing WebAssembly code to reuse existing library code. Because of these advantages, WebAssembly is now not only used as a technology for Web applications, but also integrated in blockchain platforms [6].

The WebAssembly code is executed within WebAssembly virtual machine [7]. The existing Wasm virtual machine implementations include WAVM [8], Wasmtime [9], Wasmer [10], etc. Virtual machines are the infrastructure that executes WebAssembly and should be implemented correctly, efficiently, and robustly. However, if there are errors in the implementation of the virtual machine, the execution of WebAssembly may lead to wrong results, or the program may exit abnormally. Some of these bugs can even lead to security vulnerabilities. For example, there are 7 CVEs reported for the Wasm VM called WAVM [8] in 2018. To avoid these situations, we can adopt fuzzing techniques [11] to identify errors in virtual machine implementations.

There are two major challenges faced with Wasm VM fuzzing. First, the Wasm VM often performs Wasm code validation before execution, which makes it hard to generate effective input to reach the deep logic within the VM. Although AFL, the mainstream fuzzing test software, can be used to test WebAssembly virtual machines written in C/C++, the test cases they generate are all binary data without considering Wasm bytecode grammar, which is hard to pass through the code validations commonly performed by the Wasm VMs. To solve this problem, we propose a Wasm bytecode level fuzzing framework that can both generate and mutate Wasm modules to test Wasm VMs. In particular, our proposed mutation operators can systematically mutate a Wasm module at different granularity. Second, there are many different implementations of

DOI reference number: 10.18293/SEKE2022-165

WASM VM and they have different code structures and bug patterns. A fixed mutation strategy is hard to accommodate the differences among those Wasm VMs to achieve the best fuzzing effectiveness. To solve this problem, we propose an adaptive mutation strategy that can dynamically update the probabilities of different mutation operators for a testing target.

The contributions of this work are as follows:

First, we propose a Wasm bytecode level fuzzing framework called WasmFuzzer for Wasm VMs, the tool can generate and mutate Wasm bytecode modules to reach the deep logic within the Wasm VMs.

Second, we propose an adaptive mutation strategy that can dynamically update the probabilities of different mutation operators. In this way, we can optimize the mutation operator configurations for a testing target.

Finally, we have systematically performed fuzzing on 3 real-life Wasm VMs with WasmFuzzer. Our evaluation results show that WasmFuzzer is more effective than AFL in terms of both code coverage and bug detection. And WasmFuzzer has detected 235 unique crashes within WAVM, WAMR, and EOS-VM.

The following sections are organized as follows. In section II, we will present the background knowledge on Wasm. In section III, we will discuss the design of WasmFuzzer in detail. In section IV, we present our fuzzing experiment with WasmFuzzer and AFL on 3 popular Wasm VM implementations and discuss the experiment results. Finally, we present related works and conclusion in section V and VI.

II. Background

In this section, we present background information on Wasm bytecode. In general, Wasm is a binary instruction format for a stack-based virtual machine. It is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications.

Wasm provides only four basic number types. These are integers and IEEE 754-2019 numbers, each in 32 and 64 bit width [1]. The computational model of WebAssembly is based on a stack machine. The instructions of Wasm fall into two main categories: simple instructions performing basic operations on data and control instructions altering control flow. The instructions are in turn organized into separate functions. A table in Wasm stores an array of untyped function references, which a program can call indirectly through a dynamic index into a table. WebAssembly adopts a linear memory structure, which is a contiguous, mutable array of raw bytes. A program can load and store values from/to a linear memory at any byte address. Finally, a WebAssembly binary takes the form of a module that contains definitions for functions, tables, linear memories, and global variables. In addition to definitions, modules can define initialization data for their memories or tables.

A. The Workflow of WasmFuzzer

The workflow of WasmFuzzer is shown in Figure 1. , which follows the general workflow of coverage-guided grey-box fuzzing. At first, WasmFuzzer will generate a set of Wasm files as seed inputs. Then it will enqueue these Wasm files and start the Wasm VM under fuzzing. Within the fuzzing loop, it will dequeue the first Wasm module and execute it against the Wasm VM. After execution, if the execution of the Wasm module leads to any new code coverage or new crashes, the module is considered a good candidate for mutation. And the WasmFuzzer will perform mutation on it to generate new Wasm modules, which are then enqueued for further fuzzing. Note that WasmFuzzer proposes several different mutation strategies to perform mutation. Then WasmFuzzer will further check the condition to stop the fuzzing process. If the fuzzing has reached the predefined time limit, the fuzzing will halt. Otherwise, it will continue the fuzzing loop the dequeue the next Wasm module for execution.

III. The Design of WasmFuzzer

A. The Generation of Wasm Bytecode

The input to the Wasm VM is the Wasm bytecode. To extensively fuzz the WebAssembly VM, WasmFuzzer proposes to generate valid Wasm bytecode for execution and mutation. Compared with binary input and mutation, the bytecode level inputs have a higher chance to reach deeper logic of the Wasm VM.

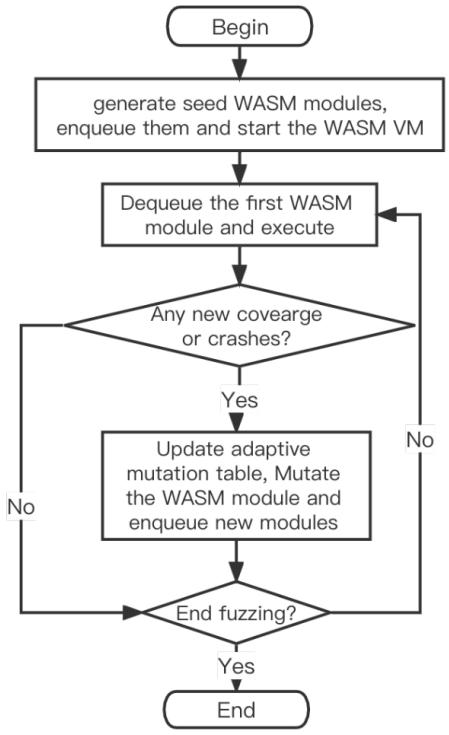

Figure 1. The Workflow of WasmFuzzer

According to the characteristics of the instruction, there are two main approaches to generating parameters: selecting parameters from the module and generating parameters from the domain of data type. Selecting a parameter from a module is used when the parameter of the instruction depends on the

internal state of the module. For example, the **global.set** instruction is to set a global variable at the top of the stack, and its parameter is the id of the global variable. Therefore, WasmFuzzer obtains the ids of all global variables from the global array in the module and selects one of them as the parameter of the instruction. Generating parameters from the domain of data type is used when the parameter is of certain data type. In such case, WasmFuzzer randomly returns a value within the domain of the data type.

WasmFuzzer extends the WebAssembly Binary Toolkit (WABT) to help generate different kinds of instructions. To be specific, it uses the internal functions of the WABT to generate different kinds of opcode, which are combined with the corresponding parameters to build different instructions. Finally, the instructions are further assembled into functions and modules as seed inputs.

B. Mutation Operator for Wasm Bytecode

Modules are the basic unit of deployment for WebAssembly. With an existing module, you can mutate it to generate new modules for fuzzing. To support feedback-directed fuzzing, we have systematically designed a set of mutation operators for Wasm modules.

1) Mutation operations

Mutation operations are divided into 2 types: mutation operations on instructions and other mutation functions. The mutation operations currently supported by WasmFuzzer are shown in TABLE I. .

TABLE I. LIST OF WASMFUZZER MUTATION OPERATIONS

Classification	***Mutation Operator***	***Description***
Mutation operations on Instructions	insertInstruction	Insert an instruction
	eraseInstruction	Delete an instruction
	moveInstruction	Move an instruction
	addFunction	Add an empty function
	eraseFunction	Delete a function
	swapFunction	Swap the positions of two functions
Other mutation operations	addGlobal	Add a global variable
	eraseGlobal	Delete a global variable
	swapGlobal	Swap the positions of two global variables
	addExport	Add an export entry
	eraseExport	Delete an export entry
	swapExport	Swap the positions of two export entries
	addType	Add a type
	addMemory	Add a block of storage space
	setStart	Set the start function
	eraseStart	Delete start function

The mutation operations on instructions are performed at the instruction level or at the function level. They randomly insert, delete, or change the instructions or functions to perform the mutation. The other mutation operations aim at changing the global variables, the export entries, the memory, or the start functions. To ensure the mutated WebAssembly code can pass through the validation [12] process of Wasm VM, we control the probability of different mutation operators such that the newly generated Wasm modules have a higher chance to be valid.

2) Adaptive Random Mutation Strategy

WasmFuzzer proposes an adaptive random mutation strategy to perform mutation. During the mutation step, each mutation operator has a probability to be selected. In general, our mutation strategy will reward the mutation operators leading to new code coverage or crash by dynamically increasing their probabilities. In this way, those more "promising" mutation operators have a higher chance to be selected.

To realize this, WasmFuzzer defines an adaptive mutation table, which is an array of function pointers of length 256. These function pointers may point to different mutation operators. The first 16 positions of this array are read-only areas and they correspond to the 16 mutation operators. In this way, WasmFuzzer ensures each mutation operator at least has a chance of 1/256 to be selected for performing mutation, which is not affected by the adaptive strategy.

TABLE II. ALGORITHM UPDATING ADAPTIVE MUTATION TABLE

Input	table: adaptive mutation table,
	func: pointer to the current mutation operator
Output	updated adaptive mutation table
1	#define NEW_PATH_REWARD 3
2	#define CRASH_REWARD 6
3	int increase = 0;
4	if (new paths found)
5	increase += NEW_PATH_REWARD;
6	if (new crash triggered)
7	increase += CRASH_REWARD;
8	for (int i = 0; i < increase; ++i) {
9	int num = randomBetween(16, 255);
10	table[num] = func;
11	}
12	return table;

The positions starting from 16 to the 255 can be both read and written, which is used for dynamically changing the selection probability of the mutation operators. At first, all positions in the table are initialized to various mutation operations with equal probability. The algorithm to update the adaptive mutation table is shown in Table II. During fuzzing, if the Wasm module obtained from a mutation operator called *M* leads to new code coverage or crash, WasmFuzzer will increase the selection probability of the mutation operator *M* (lines 3 to 7). Then, WasmFuzzer will generate a random number between 16 and 255 as index into the mutation table, and overwrite the position in the table corresponding to the index with the pointer of *M* (line 9 to line 10). In this way, the probability of those more effective mutation operators for a fuzzing target will increase gradually while those ineffective mutation operators for a target will decrease gradually. When testing multiple Wasm VMs, the

adaptive mutation strategy can automatically change the probability of each mutation operation to find the best mutation probability for each Wasm VM.

C. Test Oracle and Bug Report Generation

When the software under testing crashes or aborts during fuzzing, the system will send out signals such as SIGSEGV or SIGABT. WasmFuzer will capture these signals to report errors. Furthermore, WasmFuzzer also utilizes the AddressSanitizer [13] to detect memory-related software bugs such as use-after-free, buffer overflow, stack overflow, memory leaks, etc.

When WasmFuzzer has detected an error, it will generate bug reports to facilitate further debugging. The bug reports include two sections: the Wasm bytecode triggering a unique crash, and the Wasm bytecode triggering a unique hang. By "Unique", it means the execution of these Wasm bytecode leads to unique code path. Furthermore, we also measure the code coverage achieved during fuzzing as another metric.

IV. Evaluation

In this section, we evaluate WasmFuzzer by fuzzing 3 large-scale Wasm VMs.

A. Research Question

Based on the implementation of WasmFuzzer, this chapter mainly focuses on its test capability and test efficiency. Various performance metrics of WasmFuzzer and AFL were compared, including code coverage, number of unique crashes that could be found, and type of software problem, through comparative experiments under the same conditions.

B. Experiment Design

In our experiment, we compare WasmFuzzer with AFL to evaluate its fuzzing effectiveness.

1) Subjects

We have selected 3 real-life Wasm VM implementations to evaluate WasmFuzzer. These 3 Wasm VMs (WAVM, WAMR, and EOS VM) are written in C/C++, which is friendly for instrumentation and collecting code coverage. WAVM [8] is a popular WebAssembly virtual machine designed for non-browser applications. WebAssembly Micro Runtime [14] (WAMR for short) is a small WebAssembly virtual machine frequently used in embedded systems.

EOS-VM [15] is a WebAssembly virtual machine designed for blockchain applications. Since the command line interface provided by EOS-VM only supports the call of exported functions without parameters. To perform fuzzing, we modified the interface of EOS-VM to call the exported functions with parameters.

2) Experimental Setup

Our experiments were performed using a desktop with Intel(R) Core (TM) i7-6700 CPU @ 3.40 GHz and 16GB of memory. The operating system is Ubuntu 20.04 LTS. The version number of the AFL tool for comparison is 2.51b.

3) Instrumentation Procedure

To instrument the Wasm VMs for code coverage collection, we use Gcc compiler with code coverage profiling options enabled. To detect memory-related bugs, we also enabled the address sanitizer during compilation.

4) The experimental process

For each WebAssembly VM, we performed 8 hours of fuzzing using both WasmFuzzer and AFL. Then we use the afl-cov tool to analyze the code coverage achieved by each tool. We also manually analyzed the test cases leading to the crash or hang in the VMs to confirm the bug detected.

C. Results and Analysis

In this section, we present and compare the results of WasmFuzzer and AFL in terms of code coverage and unique crashes.

1) Code coverage

The code coverage results for the 3 Wasm VMS are shown in TABLE III. For WAVM, the code coverage of WasmFuzzer is 25.7% while the code coverage for AFL is 23.6%. For WAMR, the code coverage of WasmFuzzer is 25.9% while the code coverage for AFL is 22.7%. For EOS-VM, the code coverage of WasmFuzzer is 84.7% while the code coverage for AFL is 59.3%. We can see that WasmFuzzer consistently performs better than AFL in terms of code coverage at line level.

TABLE III. CODE COVERAGE RESULTS

Subjects	***Code Coverage***	
/	***WasmFuzzer***	***AFL***
WAVM	25.7%	23.6%
WAMR	25.9%	22.7%
EOS-VM	84.7%	59.3%

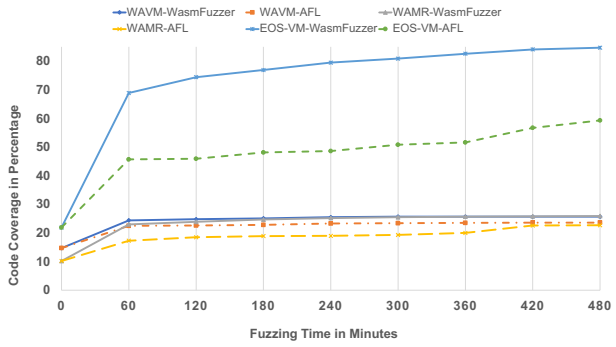

Figure 2. Code Coverage over Time

As shown in Figure 2, we also present the code coverage with respect to fuzzing time for WasmFuzzer and AFL on all 3 Wasm VMs. In this way, we want to understand the code coverage results of WasmFuzzer during the fuzzing process. We can see that for each subject VM, WasmFuzzer consistently performs better than AFL in terms of code coverage over time. And the advantage is more significant on EOS-VM than the other 2

Wasm VMs. Therefore, for different fuzzing time limit, WasmFuzzer can outperform AFL in terms of code coverage.

Based on the results above, we can conclude that WasmFuzzer can indeed achieve more code coverage than AFL if given the same fuzzing time.

2) Unique crashes

The main goal of fuzzing is to find bugs in the system. Therefore, we further present and compare the unique crashes detected by WasmFuzzer and AFL. The number of unique crashes for WasmFuzzer and AFL are shown in TABLE IV. We can see that WasmFuzzer consistently outperforms AFL on all three WebAssembly VMs. For WAVM, the difference between WasmFuzzer and AFL is small. But on WAMR and EOS-VM, the advantage of WasmFuzzer is significant.

In particular, for EOS-VM, the AFL fails to detect any error after 8 hours of fuzzing. We double-checked the code of EOS-VM, and we find that it performs strict code validation checks before executing the Wasm code. Most of the inputs generated by AFL are rejected during the code validation phase. As a result, AFL cannot detect the bugs hidden in the VM execution program logic. In contrast, WasmFuzzer can build and mutate valid Wasm modules, which makes it easier to test the execution logic of EOS-VM.

TABLE IV. NUMBER OF UNIQUE CRASHES

Subjects	***Unique Crash***	
/	***WasmFuzzer***	***AFL***
WAVM	56	55
WAMR	97	77
EOS-VM	82	0

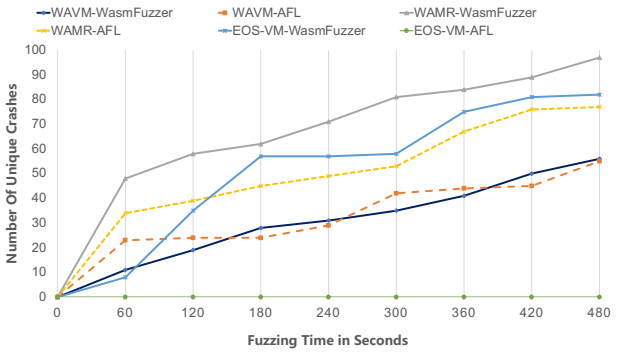

Figure 3. Unique Crashes over Time

The number of unique crashes over time for WasmFuzzer and AFL on the 3 Wasm VMs are shown in Figure 3. For WAMR and EOS-VM, WasmFuzzer consistently detected much more crashes than AFL over time. However, for WAVM, WasmFuzzer and AFL found almost the same number of unique crashes over time. A closer analysis on the crashes shows that WasmFuzzer and AFL can indeed detect different unique crashes. Therefore, when there are abundant resources during fuzzing, it is desirable to adopt both tools to perform fuzzing so they can complement each other. However, when the testing resource is limited, WasmFuzzer is preferred than AFL.

Based on the results above, we can conclude that WasmFuzzer can perform as good as or better than AFL in terms of unique crashes.

V. RELATED WORK

Park et al. designed a new test case mutation technique called aspect-preserving, and implemented a JavaScript fuzzing tool called DIE [16]. They believe that there are certain patterns in test cases that can trigger vulnerabilities. For the JavaScript language, the combination of some code structures and variable types is more likely to trigger vulnerabilities in the JavaScript execution engine. Therefore, DIE tends to retain these combinations when performing mutation.

Fuzzing tools can also be combined with neural network models to generate inputs that can trigger vulnerabilities more easily. Lee et al. developed a fuzzing tool based on neural network language model for JavaScript engines named Montage [17]. They train the model with the abstract syntax subtree converted from the JavaScript abstract syntax tree. In this way, the model can generate valid JavaScript code. With this approach, their tool has detected previously undiscovered software bugs in the JavaScript execution engine under fuzzing.

Zhong et al. designed and implemented a fuzzing tool called Squirrel for relational databases [18], whose input data is structured query language. Since the structured query language needs to meet certain grammatical rules, the proportion of input that can be executed by the database when directly mutating binary data is small. Therefore, they designed an intermediate representation capable of generating structured query language code and performed type-based mutation on the intermediate representation. In this way, the proportion of input that can be executed by the database is significantly increased.

Fuzzing tools are also effective to find functional implementation bugs in the software implementation. For example, Chen et al. implemented a fuzzing tool for differentially testing Java virtual machines [19]. The main idea is to use the same input to execute multiple Java virtual machines and compare the running results among them. If there is any difference in their results, one of these Java virtual machines must contain a bug. Engineers can further perform analysis and debugging based on the fuzzing results to find the position of the software error.

Ventuzelo proposes to use mainstream fuzzing tools to test WebAssembly virtual machines, and they integrated a fuzzing tool called WARF [20]. WARF can fuzz WebAssembly virtual machines and test them with binary data. WARF has found several bugs in the WebAssembly virtual machine implementation. WARF is implemented in Rust language and integrates three mainstream fuzzing tools, AFL++ [21], Honggfuzz [22] and libFuzzer [23].

VI. Conclusion

WebAssembly is a fast, safe, and portable low-level language suitable for diverse application scenarios. And The WebAssembly virtual machines are widely supported by Web browsers for building Web applications. When there is a bug in the implementation of the Wasm virtual machine, the execution of WebAssembly may lead to errors in its supporting application. Due to the code validation performed by WASM VMs, fuzzing at the binary level is ineffective to expose the bugs because most inputs cannot reach the deep logic within the WASM VM. In this work, we propose WasmFuzzer, a bytecode level fuzzing tool for WASM VMs. WasmFuzzer proposes to generate initial seeds for Fuzzing at the Wasm bytecode level and it also proposes a systematic set of mutation operators for Wasm bytecode. Furthermore, WasmFuzzer proposes an adaptive mutation strategy to search for the best mutation operators for different fuzzing targets. Our evaluation on 3 real-life Wasm VMs shows that WasmFuzzer can significantly outperform AFL in terms of both code coverage and unique crash.

For future work, we plan to explore new seed generation scheme and fuzzing input scheduling scheme to improve the effectiveness of the fuzzing tool. We will also perform fuzzing on other popular Wasm VMs to further evaluate the effectiveness of WasmFuzzer.

Acknowledgments

This research is supported in part by the National Key R&D Program of China under Grant 2019YFB2102400, NSFC (project no. 61772056), the Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing, Innovative Technology Fund of HKSAR (project no. 9440226) and CityU MF_EXT (project no. 9678180). Zhenyu Zhang is the corresponding author.

References

[1] WebAssembly. https://webassembly.org/. Last access, 2022.

[2] The LLVM Compiler Infrastructure. https://llvm.org/. Last access, 2022.

[3] Ritchie D. M.. The Development of the C Language. *ACM Sigplan Notices 28.3*, 201-208, 1993.

[4] Stroustrup B.. The C++ programming language. Pearson Education India, India, 2000.

[5] Matsakis N. D.., Klock F. S.. The rust language. *ACM SIGAda Ada Letters 34.3*, 103-104, 2014.

[6] Wang S., Yuan Y., Wang X., et al. An overview of smart contract: architecture, applications, and future trends. *2018 IEEE Intelligent Vehicles Symposium (IV)*, 2018.

[7] Sauntry D. M., Gilbert M.. Generating a compiled language program for an interpretive runtime environment. US, US6327702 B1. 2001.

[8] WAVM. https://wavm.github.io/. Last access, 2022.

[9] Wasmtime. https://wasmtime.dev/. Last access, 2022.

[10] Wasmer. https://wasmer.io/. Last access, 2022.

[11] Ammann P., Offutt J.. Introduction to software testing. *UK: Cambridge University Press*, 2016.

[12] Validation — WebAssembly 1.1 (Draft 2021-11-18). https://webassembly.github.io/spec/core/valid/index.html. Last access, 2021.

[13] AddressSanitizer. https://github.com/google/sanitizers/wiki/AddressSanitizer. Last access, 2019.

[14] WebAssembly Micro Runtime. https://github.com/bytecodealliance/wasm-micro-runtime. Last access, 2021.

[15] EOS VM - A Low-Latency, High Performance and Extensible WebAssembly Engine. https://github.com/EOSIO/eos-vm. Last access, 2019.

[16] Park S., Xu W., Yun I., et al. Fuzzing JavaScript Engines with Aspect-preserving Mutation. *2020 IEEE Symposium on Security and Privacy (SP)*, 1629-1642, 2020.

[17] Lee S., Han H. S., Cha S. K., et al. Montage: A Neural Network Language Model-Guided JavaScript Fuzzer. *20th USENIX Security Symposium (USENIX Security 2020)*. 2020.

[18] Zhong R., Chen Y., Hu H., et al. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. https://arxiv.org/abs/2006.02398, 2020.

[19] Chen Y., Su T., Sun C., et al. Coverage-directed differential testing of JVM implementations. *In Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2016: 85-99.

[20] WARF - WebAssembly Runtimes Fuzzing project. https://github.com/pventuzelo/wasm_runtimes_fuzzing. Last access, 2022.

[21] The AFL++ fuzzing framework | AFLplusplus. https://aflplus.plus/. Last access, 2021.

[22] Honggfuzz | honggfuzz. https://honggfuzz.dev/. Last access, 2021.

[23] libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. Last access, 2022.