DFAFuzz: Fuzzing for Embedded JavaScript Virtual Machines with Type-Directed DFA

Haiwei Lai Baojian Hua*
School of Software Engineering, University of Science and Technology of China, China
Suzhou Institute for Advanced Research, University of Science and Technology of China, China
sa23225261@mail.ustc.edu.cn bjhua@ustc.edu.cn

***Abstract*—JavaScript is rapidly being deployed in security-critical embedded domains, including IoT devices, edge computing, and smart automotive applications. Embedded JavaScript virtual machines (VMs) are critical in powering such deployments, which should be secure and trustworthy. Fuzzing is an effective approach in detecting deep bugs in these VMs by generating diverse VM bytecode programs. However, existing JavaScript fuzzers are still limited in generating diverse and valid bytecode that finds deep bugs, because they fail to track the VM operand-stack state, which leads to invalid programs and missed bugs.**

In this paper, we present a novel fuzzing approach called DFAFuzz, to detect deep bugs in embedded JavaScript VMs. Our key idea is to use a type-directed deterministic finite automaton (DFA) to track instruction type information, guiding the generation of new type-correct JavaScript bytecode programs from existing fuzzing seeds. First, our approach employs type reconstruction to track variable type changes during bytecode generation. Second, our approach utilizes type transitions with the aid of DFA to guide bytecode generation to produce valid bytecode. We implement a software prototype DFAFuzzer and conduct extensive experiments to evaluate its effectiveness on JerryScript and QuickJS. Our results show that DFAFuzzer improves the bytecode validity ratio by 7.6% and 2.8% for JerryScript and QuickJS over AFL++, respectively. Furthermore, DFAFuzzer detects 111 bugs that are missed by state-of-the-art fuzzers, AFL++ and Fuzzilli.

***Index Terms*—Fuzzing, program security, JavaScript bytecode**

I. Introduction

In recent years, JavaScript [1] is rapidly being deployed in security-critical embedded domains including IoT devices, edge computing, and smart automotive applications, showing promising potential. Specifically, such new deployments of JavaScript are fundamentally enabled by embedded JavaScript virtual machines (VMs) executing bytecode programs that efficiently utilize limited computing resources on embedded computing platforms to enforce secure and efficient execution. However, bugs are inevitable in these embedded JavaScript VMs, as they are complex software that provide sophisticated functionalities including binary loading, parsing, execution, just-in-time compiling, garbage collection, among others. Any bugs in these VMs will lead to incorrect execution results or even corrupt the computing environment, defeating the VMs' security guarantees. For example, JerryScript, a widely used embedded JavaScript VM, contains a buffer overflow vulnerability CVE 2023-36109 [2]. Adversaries can exploit this CVE by crafting malicious input data to overwrite memory, leading to arbitrary code execution to gain full control of the system. Consequently, developing novel approaches to detect security vulnerabilities in embedded JavaScript VMs is both urgent and critical.

Recognizing such urgency and criticality, researchers have proposed to use fuzzing [3] [4] to detect vulnerabilities [5] [6]. Specifically, to fuzz an embedded JavaScript VM, a fuzzer randomly generates new bytecode programs and feeds them into the target VM to identify bugs based on potential abnormal behaviors including crashes and hangs. Furthermore, to enhance fuzzing efficacy, state-of-the-art fuzzing frameworks incorporate dynamic feedback mechanisms [7] [8] to collect runtime information from the target VM, including but not limited to code coverage metrics and execution path traces, to determine whether novel program states or previously untriggered edge cases have been exercised. Specifically, when execution traces reveal previously unexecuted code paths, these paths are preserved as seed inputs for subsequent mutations. Consequently, this approach significantly increases the probability of triggering latent vulnerabilities through semantically valid yet edge-case-pushing input sequences. Therefore, generating diverse and valid bytecode programs is critical, as overly simplistic bytecode programs fail to exercise subtle edge-case vulnerabilities, while semantically invalid bytecode causes premature termination of VM execution, severely compromising fuzzing efficiency and state space exploration.

Although existing fuzzing approaches show promise in detecting bugs in embedded JavaScript VMs, they remain limited in generating diverse and valid bytecode to uncover deep bugs. First, existing studies to fuzz JavaScript VMs predominantly adopt source-based approaches to generate JavaScript source programs based on predefined syntax rules [6] [8], and then generate bytecode by leveraging JavaScript compilers. Consequently, test cases generated by these approaches lack diversity and often fail to cover all potential execution paths and edge-case scenarios [9] [10]. Second, while direct bytecode-level mutations [5] [7] can produce more varied bytecode sequences, they often generate syntactically or semantically incorrect bytecode programs, due to the lack of precise runtime state information [10]. For instance, while AFL++ [7], a state-of-the-art fuzzer, can mutate binary programs by employing

* The corresponding author.

random binary mutation strategies including bit flips, byte flips, and arithmetic mutations, it frequently generates invalid bytecode that violates either stack operation rules or type constraints due to overlooking the operand stack states and variable type transitions during the mutation process [11].

In this paper, we propose a novel fuzzing approach called DFAFuzz, which automatically and effectively fuzzes embedded JavaScript VMs to detect deep bugs by generating diverse and valid JavaScript bytecode programs. Our key insight for generating valid bytecode programs is to leverage both operand stack consistency [12] and type consistency [13] during instruction generation to guarantee semantic correctness of generated programs. First, operand stack consistency requires each bytecode instruction to preserve proper stack state during execution. A key technical challenge to track the state is that complex control flow structures (*e.g.*, conditional branches and loops) often lead to divergent stack states across different execution paths. To address this challenge, we adopt a flow-sensitive approach from static program analysis [14] to precisely track and manage stack states along all potential control flows. Second, type consistency demands that the operands of each instruction match their expected types. Otherwise, any type errors will lead to semantically invalid instructions. A key challenge in tracking and checking types is that JavaScript, unlike statically typed languages such as Java, is a dynamically typed language, making static type checking difficult. To address this challenge, we leverage type inference to reconstruct accurate type information [15].

With these key insights, our approach encompasses two key components. First, we design a type-directed deterministic finite automaton (DFA) to track operand stack state changes and variable type information. We then deploy the type-directed DFA on the control flow graph (CFG) of the bytecode programs to guide the generation of valid instructions. Second, we design a type reconstruction to infer the latest variable type information during instruction generation, and utilize that information to guide subsequent instruction generation.

To demonstrate the effectiveness of our approach, we implement a prototype of dubbed DFAFuzzer and evaluate it on two mainstream embedded JavaScript VMs JerryScript [16] and QuickJS [17]. Our evaluation shows that DFAFuzzer uncovers 50 and 198 bugs in JerryScript and QuickJS, respectively. Among all the 248 bugs DFAFuzzer uncovered, 111 (44.75%) are missed by state-of-the-art fuzzers of AFL++ [7] and Fuzzilli [8]. Furthermore, owing to DFAFuzzer's effectiveness in generating diverse and valid JavaScript bytecode programs, DFAFuzzer achieves 7.6% and 2.8% higher validity ratios than AFL++ in generating bytecode for JerryScript and QuickJS, respectively, which also result in a coverage of 62.95% for JerryScript and 35.32% for QuickJS. Additionally, we conduct an ablation study to analyze the contributions of different components of DFAFuzzer (*i.e.*, type-directed DFA, and type reconstruction) to the whole fuzzing effectiveness. Finally, we leverage a qualitative approach to conduct a manual investigation into the bugs DFAFuzzer uncovered to reveal the practical security implications. To ensure responsible disclosure, all vulnerabilities uncovered by DFAFuzzer have been reported to the maintainers of JerryScript and QuickJS. At the time of writing, none of the reports have received responses so far. We will continue to follow up and update their status in future work.

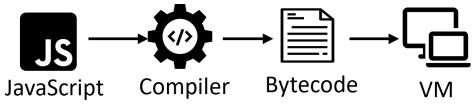

Fig. 1: The compilation and loading of bytecode.

To summarize, our work makes the following contributions:

- We present a novel fuzzing approach called DFAFuzz, which utilizes a type-directed DFA to generate diverse and valid JavaScript bytecode programs for fuzzing.
- We design and implement a software prototype DFAFuzzer to validate our approach.
- We conduct extensive experiments to evaluate our approach. And our results show that DFAFuzzer outperforms the state-of-the-art fuzzers with respect to valid program generation, code coverage, and bug detection.
- We make our approach, software prototype, datasets, and evaluation results publicly available in the interest of open science at: `https://doi.org/10.5281/zenodo.15224284`.

The rest of this paper is organized as follows. Section II presents the background for this study. Section III presents the motivation. Sections IV and V present the design and implementation of DFAFuzz. Section VI presents evaluation results. Section VII discusses limitations and directions for future work. Section VIII discusses the related work, and Section IX concludes.

II. Background

For completeness, in this section, we present the background of embedded JavaScript VM (§ II-A), and fuzzing (§ II-B).

A. Embedded JavaScript VM

In security-critical embedded domains such as IoT, edge computing, and smart automotive systems, embedded JavaScript VMs are increasingly used. To optimize limited resources while ensuring secure, efficient execution, these VMs [16] [17] [18] employ a bytecode-based model. As shown in Fig. 1, JavaScript source code is precompiled into compact bytecode, allowing rapid loading and execution across diverse embedded platforms.

Unlike VMs such as V8 and SpiderMonkey, which are primarily designed for browser environments and execute applications by deploying JavaScript source code, embedded JavaScript VMs emphasize bytecode execution for cross-platform deployment and efficient runtime performance. However, this execution model has not yet undergone systematic security testing. The complexity of stack-based execution, instruction dispatch, and type handling makes many subtle

```
1   case ADD: {
2       assert(stack_top_p - stack_base_p >= 2);
3       value_t l_val = stack_top_p[-2];
4       value_t r_val = stack_top_p[-1];
5       stack_top_p -= 2;
6       ......
7       if (are_values_int_numbers(l_val, r_val)) {
8           int_value_t l_int =
9               get_int_from_value(l_val);
10          int_value_t r_int =
11              get_int_from_value(r_val);
12          *stack_top_p++ =
13              make_int32_value(l_int + r_int);
14          ......
15      } else {
16          //Throw type error
17          ......
18      }
19  }
```

Fig. 2: Add Instruction Implementation in JerryScript.

runtime behaviors hard to test, leaving potential vulnerabilities undetected and limiting the secure deployment of these VMs.

For instance, CVE-2023-36109 [2] is a buffer overflow in JerryScript that allows crafted bytecode to achieve arbitrary code execution, potentially compromising the entire system. Similarly, CVE-2020-24187 [19] lets local attackers trigger runtime errors via malicious inputs, causing denial-of-service. These cases demonstrate that embedded JavaScript VMs remain vulnerable to complex or malicious bytecode, highlighting the need for systematic, in-depth testing.

B. Fuzzing

Fuzzing automatically generates numerous random or mutated inputs to test programs and find security vulnerabilities [3] [4]. When applied to embedded JavaScript VMs, the objective is to produce malformed or malicious bytecode that triggers crashes, abnormal behavior, or other exploitable faults.

To explore program state space more effectively, modern approaches use coverage-guided fuzzing, which steers input generation by dynamically monitoring executed code paths and adapting tests to maximize path coverage, thereby increasing the chance of revealing defects and vulnerabilities.

Edge-coverage fuzzers like AFL++ [7] mutate bytecode without guaranteeing semantic correctness, favoring broad path exploration that is effective at exposing edge-case failures. By contrast, Fuzzilli [8] generates syntactically valid JavaScript source code and compiles it to bytecode, producing more semantically coherent inputs that improve depth of testing and fuzzing efficiency.

III. Motivation and Challenges

In this section, we present our motivation (§ III-A) and the technical challenges (§ III-B) for this study.

A. Motivation

In JavaScript bytecode fuzzing, generating diverse and valid bytecode is essential to explore the VM's state space and uncover vulnerabilities. Valid bytecode must strictly satisfy stack and type consistency rules, as violations render it invalid.

1) *Stack inconsistency* requires that operations within each basic block remain aligned along any execution path, and

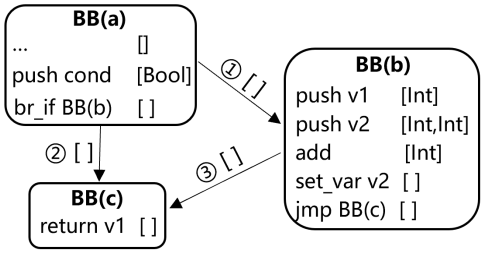

Fig. 3: Valid bytecode with operand stack state and type consistency.

that stack states match at control-flow merge points despite differing branches.

2) *Type inconsistency* requires that stack elements conform to the type constraints of each instruction.

Bytecode validity requires strict maintenance of both stack and type consistency. For example (Fig. 3), when flowing from basic block (a) to the merge point (c), the stack state must be identical after following paths ① and ③ or after following path ②; otherwise basic block (c) cannot correctly handle the divergent state. Similarly, in basic block (b) the push v2 instruction must leave two integers on the stack, or the subsequent add will trigger a type mismatch.

Existing fuzz testing tools often fail to meet stack or type constraints in the mutated bytecode due to the lack of precise state tracking. AFL++, which emphasizes control-flow and data-flow, uses random mutations that often break stack or type constraints in seeds [11]; Table 1 of WALTZZ reports that over 98% of AFL++'s generated WebAssembly tests are invalid due to ignored stack state [20]. This underscores the failure of mutation strategies that disregard stack invariants and motivates our work. Although type constraints may seem limiting, enforcing them reduces early rejection and lets the fuzzer explore deeper semantic behaviors.

B. Challenges

Stack inconsistency. In a stack-based VM [21], the operands of an instruction come from the operand stack. If the number of elements in the operand stack is insufficient, this stack inconsistency will cause the virtual machine to terminate prematurely. For instance, as shown in line 2 of Fig. 2, if the assertion that the add instruction requires at least two stack elements is violated, the VM will reject execution early. To solve this problem, after mutating the bytecode program, the fuzzer needs to ensure stack consistency through full-path stack state analysis. This process necessitates constructing precise control flow models, employing data flow analysis techniques to track cross-block stack height variations, and addressing path combination explosion problems caused by complex control flow patterns such as conditional branches and loop structures. The verification process that reconciles dynamic semantics with static analysis makes ensuring stack consistency in generated instruction sequences a significant technical challenge.

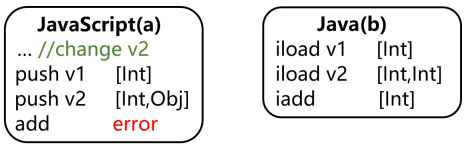

Fig. 4: Comparison between dynamically typed languages and statically typed languages.

Our solution. To address this challenge, we employ a flow-sensitive approach from static program analysis to precisely track and manage stack states along all potential control flows. To ensure stack consistency at control flow merge points, we impose constraints on the generation conditions of basic blocks within the CFG. Specifically, each basic block must produce instruction sequences that result in an empty stack state at their exit points, thereby preventing stack depth inconsistencies during control flow merges.

When generating instruction sequences within a basic block, we employ a top-down approach to track the stack state. This method utilizes a type-directed DFA to guide instruction generation. For example, in Fig. 3, to maintain stack consistency across the CFG, the exit stack states of basic blocks a, b, and c must be empty. While generating instructions for basic block b, upon generating the instruction push v2, the tracked stack state becomes [Int, Int]. Leveraging our constructed DFA, we select the add instruction based on the current stack state, subsequently updating the stack state to [Int].

Type inconsistency. JavaScript is a dynamically typed language, meaning that the type of a variable is determined at runtime rather than compile time. Therefore, unlike statically typed languages like Java, the type of a JavaScript variable can change during runtime, which means that type errors cannot be caught at compile time and only become apparent during execution. For example, as shown in Fig. 4, in JavaScript bytecode (a), the type of v2 can change to Object at runtime. If the type change is not detected and the push v2 instruction is generated, then during the execution of the add instruction, the VM will jump to line 15 in Fig. 2, throw a type error, and terminate early. In contrast, Java bytecode (b) can statically determine the type information of each variable, so it does not suffer from type inconsistency issues. This dynamic nature presents a challenge in maintaining type consistency during program execution, especially when generating or mutating bytecode.

Our solution. To address this challenge, we leverage type inference to reconstruct the accurate types. We track the types of all local variables and stack elements, and perform static analysis on the instruction sequences in the bytecode to ensure that when each instruction is generated, the required stack element types and operand types comply with its instruction constraints. For example, in the JavaScript bytecode (a) in Fig. 4, after generating v2, our analysis determines that the current stack element types are [Int, Obj]. Based on the instruction semantics, an appropriate instruction (such as pop) is then selected.

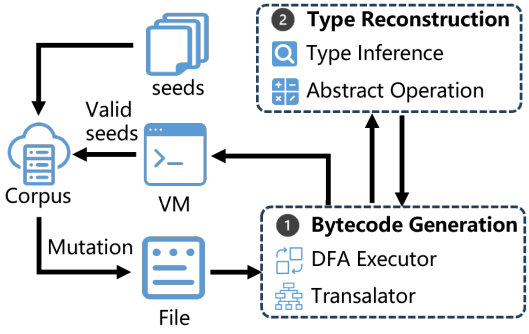

Fig. 5: The workflow of DFAFuzz (§ IV-A).

IV. Design

We introduce DFAFuzz, a type-directed DFA with type reconstruction that generates diverse, semantically valid bytecode to improve fuzzing of embedded JavaScript VMs. Its overall architecture is presented in IV-A, followed by detailed descriptions of bytecode generation (IV-B) and type reconstruction (IV-C).

A. Overview

Fig. 5 depicts DFAFuzz's workflow. After seed mutation, Bytecode Generation converts the resulting file into a CFG and uses a type-directed DFA to emit bytecode with consistent stack states and types. Type Reconstruction tracks variable type changes and supplies type information to Bytecode Generation.

B. Bytecode Generation

We incrementally generate programs while inferring their state, using a type-directed DFA to guide instruction generation within each basic block and maintain stack consistency. We then construct a CFG to merge basic blocks, resolving stack inconsistencies arising from control structures such as branches.

We first use the Translator to convert the file into a CFG. Instruction selection for each basic block is guided by Type Reconstruction's type information, the file content, and the DFA Executor, which extracts instructions satisfying current stack constraints. Generated instructions maintain type consistency and are fed back to Type Reconstruction, dynamically updating the types of stack elements and local variables.

DFA executor. To ensure consistent stack operations within a basic block, we construct a state transition graph based on stack depth. This graph, combined with the current stack state, guides the continuous selection of instructions, producing a valid and effective basic block.

As shown in Fig. 6, the nodes Si(i=0,1,2) represent stack depths of 0, 1, and 2, with edges indicating how executing associated instructions changes the stack. For the current state S2, possible transitions include S0 and S1, so valid instructions include *pop*, *add*, and *del_prop*. After filtering through the DFA Executor, a candidate instruction set is refined by Type Reconstruction to match type information. The Translator then

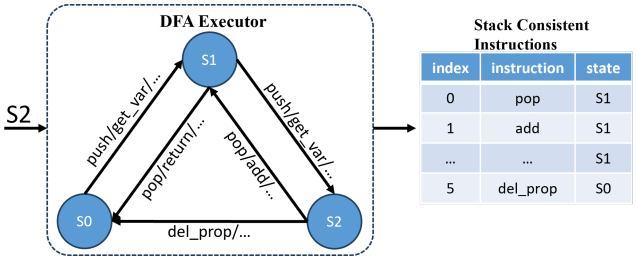

Fig. 6: Generate basic blocks that adhere to stack state consistency using the state transition graph.

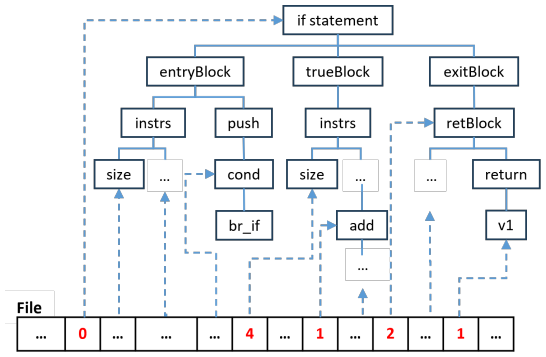

Fig. 7: Generating the bytecode in Fig. 3.

selects and indexes an instruction from this set, updating the basic block's current state.

Translator. To ensure stack consistency at control-flow merge points, we require each basic block in the CFG to start and end at stack depth zero (state S0). This invariant ensures that branching, loops, and other control structures can be merged without causing stack inconsistencies.

The Translator performs a preorder traversal of the input byte sequence, constructing basic block skeletons at control-flow boundaries and producing a complete CFG. For each block, instructions are deterministically selected based on the current context (e.g., stack depth, variable table) and valid instructions from the DFA Executor, ensuring type correctness. Each control-flow construct has a prologue and epilogue—for instance, a branch prologue pushes a Bool condition, and a loop prologue initializes a counter, while epilogues restore the stack to S0. If an operand type is missing, auxiliary loading instructions are inserted and the variable table updated.

Fig. 7 illustrates part of the generation process for the program in Fig. 3. After reading the file, the Translator uses the value 0 to identify an if-structured CFG. For basic block (b), the value 4 indicates four additional instructions. When generating the third instruction, the value 1 indexes the DFA Executor's valid set, selecting the add instruction. In the $exitBlock$, if or $loop$ structures can be generated recursively. The value 2 designates a $return$ block without any $jump$ instructions, producing basic block (c), and the final value 1 selects $v1$ from the variable table as the instruction operand.

C. Type Reconstruction

To ensure type-consistent bytecode, the Type Reconstruction module infers and maintains type information based on the current stack state and instruction requirements, preventing invalid or erroneous instructions.

Type inference. During bytecode generation, the Type Inference module ensures stack type consistency by validating whether each instruction's type requirements are met. Before inserting an instruction, the system checks the current stack state; for instance, as shown in Fig. 9, the del_prop instruction requires two operands of specific types. Instructions failing this check are discarded.

The Type Inference module not only validates each instruction but also updates and propagates type information after insertion, as certain operations (e.g., type conversions) may alter the types of stack elements. It ensures that the stack maintains type consistency before and after instruction execution, thereby preventing the generation of invalid bytecode. For operations involving object properties (e.g., `obj.x = 42`), the system performs lightweight tracking by recording inferred property types and reusing them during subsequent accesses to preserve semantic consistency. To reduce overhead, only top-level properties are tracked, while nested structures and prototype chains are omitted. In uncertain cases—such as dynamic property names or type conflicts across control-flow branches—the system conservatively infers the type as `Any`.

Abstract operations. We abstract program execution as a series of type transformations, simulating the behavior of bytecode instructions to propagate abstract values and infer type information. This abstraction enables systematic identification of potential type transformations within a program, facilitating efficient analysis of program behavior.

As each instruction affects the operand stack, it is essential to ensure that the stack data conforms to expected types and structure. To achieve this, we introduce an abstract representation of the stack state that maintains consistency between stack elements and operand types during instruction execution. Before and after executing each instruction, the system validates and updates the stack state according to type inference rules. As shown in Fig. 8, these rules define the type transformations for various instructions. For instance, the operation $binary : (Num)(Num) \rightarrow (Num)$ requires two numerical operands and produces a new numerical value at the stack top. If the stack types violate the requirements, the system immediately reports an error and aborts the test, ensuring the validity of generated bytecode. Furthermore, a recursive inference mechanism tracks and propagates type changes across execution paths, enabling systematic analysis of potential type transformations and maintaining type consistency throughout the program. This guarantees semantically valid bytecode and provides a reliable foundation for subsequent fuzzing.

V. Implementation

Based on our proposed DFAFuzz, we implement DFAFuzzer, a prototype for automatically detecting potential bugs in embedded JavaScript VMs by generating diverse,

$$
\begin{array}{rcl}
\text{nop}: & () & \rightarrow & () \\
\text{create\_var}\ (Any): & () & \rightarrow & () \\
\text{get\_var}\ (v): & () & \rightarrow & (T_v) \\
\text{create\_obj}: & () & \rightarrow & (Object) \\
\text{push}\ (T): & () & \rightarrow & (T) \\
\text{pop}: & (T) & \rightarrow & () \\
\text{return}: & (T) & \rightarrow & ()
\end{array}
\qquad
\begin{array}{rcl}
\text{set\_var}\ (v): & (T) & \rightarrow & () \\
\text{unary}: & (Num) & \rightarrow & (Num) \\
\text{binary}: & (Num)(Num) & \rightarrow & (Num) \\
\text{cmp}: & (Num)(Num) & \rightarrow & (Bool) \\
\text{get\_prop}: & (Object)(String) & \rightarrow & (T) \\
\text{del\_prop}: & (Object)(String) & \rightarrow & () \\
\text{set\_prop}: & (Object)(String)(T) & \rightarrow & ()
\end{array}
$$

Fig. 8: The changes to the stack after the instruction is executed, including the data types.

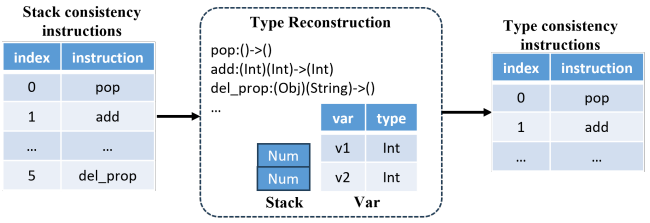

Fig. 9: Filtering instructions based on Abstract Operations and type information to ensure type consistency.

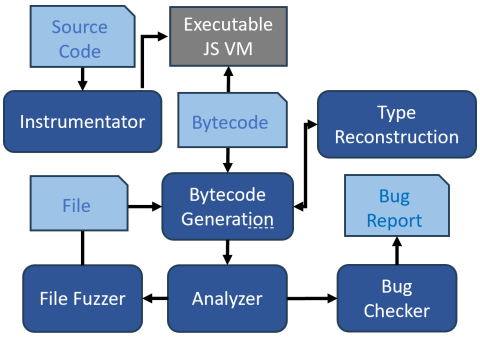

Fig. 10: Architecture of DFAFUZZER.

semantically valid bytecode. Fig. 10 shows DFAFUZZER's architecture, comprising six core modules: Instrumentator, Bytecode Generation, Type Reconstruction, Analyzer, File Fuzzer, and Checker. Together, they form a complete automated testing pipeline— from bytecode generation and type inference to fuzzing and vulnerability detection—enhancing both security coverage and bug-finding capabilities.

We employ AFL++'s code instrumentation to compile the target VM into an executable that accepts bytecode inputs. The Bytecode Generation module processes mutated inputs from the File Fuzzer, using type information from Type Reconstruction to produce semantically valid bytecode, which is executed by the VM to trigger runtime behaviors and collect execution data. Type Reconstruction ensures operand type consistency, improving bytecode correctness and execution success. The Analyzer examines runtime information to identify crashes or anomalies and guides seed selection for subsequent fuzzing. The File Fuzzer performs standard file-level fuzzing, forwarding mutations to Bytecode Generation, while the Checker verifies confirmed vulnerabilities via crash reports and exception logs, completing the detection loop.

VI. EVALUATION

In this section, we present the experiments we conduct to evaluate the effectiveness of DFAFUZZ. Our evaluation is guided by the following research questions.

RQ1: Can DFAFUZZER find bugs in real-world embedded JavaScript VMs by generating valid bytecode? (§ VI-B)

RQ2: Does DFAFUZZER outperform state-of-the-art JavaScript fuzzers? (§ VI-C)

RQ3: Which factors affect DFAFUZZER's effectiveness? (§ VI-D)

RQ4: How about the security impact of the bugs detected by DFAFUZZER? (§ VI-E)

A. Experimental Setup

Embedded JavaScript VMs. We evaluate our approach on the latest versions of JerryScript v3.0.0 [16] and QuickJS v2024.1.13 [17], two widely adopted JavaScript VMs optimized for resource-constrained embedded environments. These VMs have been extensively tested in prior work [22] [23] and are actively maintained (7.2K and 9K GitHub stars, respectively). Our method is not VM-specific and can be applied to other JavaScript VMs; the source code is available at `https://doi.org/10.5281/zenodo.15224284`.

Basic setup. All experiments are conducted on a server equipped with a 12-core Intel i7 CPU (20 threads) and 128 GB RAM running Ubuntu 24.04 LTS. We enable AddressSanitizer (ASAN) during compilation and execution of all target VMs to detect memory errors and security vulnerabilities. To prevent infinite loops potentially generated by fuzzers, each execution is limited to 500 ms. Each experiment is repeated five times, and the aggregated results are used to minimize the effects of randomness. Detailed configurations are described in the corresponding subsections.

B. RQ1: Bug Detection

To address RQ1, we evaluate the effectiveness of DFAFUZZ in detecting bugs in embedded JavaScript VMs through extensive fuzzing on JerryScript and QuickJS. Following prior studies [8] [24], we enable ASAN during VM compilation to capture memory safety issues and expose logical flaws, thereby improving bug detection. Each experiment is repeated five times to mitigate randomness, and runs for 12 hours, as the

TABLE I: Comparison with State-of-the-Art fuzzers.

	JerryScript				QuickJS			
	bugs	valid seeds	UIB_a	UIB_m	bugs	valid seeds	UIB_a	UIB_m
AFL++	16	2.9M/13.1M(22.1%)	80.71	71	163	1.5M/10.7M(14.0%)	37.13	26
Fuzzilli	6	0.4M/0.9M(44.4%)	14.26	11	0	0.1M/0.6M(16.7%)	16.31	12
DFAFuzzer	**50**	2.2M/7.4M(29.7%)	78.28	67	**198**	2.6M/15.5M(**16.8%**)	**39.66**	**31**

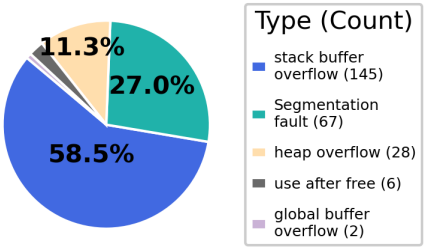

Fig. 11: Types of the bugs uncovered by DFAFuzzer.

number of discovered edge paths stabilizes beyond this period, indicating sufficient coverage for effective bug detection.

Fig. 11 presents the experimental results. During the experiments, DFAFuzzer reports a total of 248 bugs (50 in JerryScript and 198 in QuickJS), all of which are deduplicated based on stack traces to avoid inflating the count with duplicate bugs. To further ensure the validity of our findings, we manually inspect the crashing inputs, confirming that the majority trigger genuine memory safety violations or unexpected VM behavior. Among the 248 bugs, 41 are located in code regions that have not been previously exercised by AFL++ and Fuzzilli, indicating that DFAFuzzer successfully reaches new program paths and uncovers previously unknown bugs in those segments.

Furthermore, we categorize the 248 bugs discovered by DFAFuzzer into five types, based on how the target bug manifests: stack buffer overflow, segmentation fault, heap overflow, use after free, and global buffer overflow. Among these, stack buffer overflow is the most prevalent with 145 bugs (58.5%), followed by segmentation fault with 67 bugs (27.0%). While these categories are largely based on runtime error types captured by ASAN, a deeper semantic analysis reveals that many of these crashes are rooted in incorrect type handling and unexpected control flow transitions—highlighting the ability of DFAFuzzer to generate semantically valid yet error-triggering inputs.

These evaluation results demonstrate that DFAFuzzer is effective in uncovering real and diverse bugs in practical JavaScript VMs. All bugs are responsibly disclosed to the maintainers, and we have not yet received any responses.

C. RQ2: Comparison with State-of-the-Art Fuzzers

To address RQ2, we evaluate the effectiveness of DFAFuzzer against state-of-the-art JavaScript fuzzers. We compare it with AFL++ [7], a widely used general-purpose fuzzer, and Fuzzilli [8], a JavaScript-specific fuzzer. Each fuzzer—DFAFuzzer, AFL++, and Fuzzilli—is executed on JerryScript and QuickJS for 12 hours under identical configurations to ensure fairness. We record the number of discovered bugs, edge coverage, and the ratio of valid seeds, which are then used for further analysis.

Coverage. As illustrated in Fig. 12, DFAFuzzer consistently outperforms the other fuzzers in terms of coverage. For instance, even when the initial seeds already provide a relatively high baseline coverage, DFAFuzzer still achieves approximately 8.01% higher final coverage than AFL++ on JerryScript. When the coverage contributed by the initial seeds is excluded, DFAFuzzer discovers, on average, 23% more coverage across these VMs compared to AFL++, demonstrating its superior capability for independent exploration. We attribute this advantage to the fact that AFL++ performs byte-level mutations without considering consistency checks on bytecode, which limits its ability to explore deeper execution paths.

In contrast, Fuzzilli relies on predefined syntax rules to generate inputs, ensuring that most test cases are valid, but its exploration space is significantly constrained by these predefined rules. The observation that DFAFuzzer consistently achieves higher coverage than Fuzzilli across different VMs further substantiates the effectiveness of DFAFuzzer in exploring diverse program states.

Bug detection capability. We evaluate the bug detection capability of DFAFuzzer against two SOTA fuzzers. As shown in Table I, DFAFuzzer uncovers 248 bugs across the two VMs, significantly outperforming AFL++ (179 bugs) and Fuzzilli (6 bugs). All reported bugs are manually deduplicated based on stack traces to ensure accuracy. The superior performance of DFAFuzzer is largely attributed to its higher coverage and type-directed generation strategy, which enables it to explore deeper and less-tested execution paths. Notably, DFAFuzzer discovers 111 bugs that are completely missed by both AFL++ and Fuzzilli, demonstrating its ability to reveal bugs. In contrast, Fuzzilli found only six bugs on these two VMs, mainly because its JavaScript syntax-based input generation method has been fully tested on these VMs, resulting in limited marginal benefits.

Seed validity. To verify whether DFAFuzz can generate a higher proportion of correct seeds, we measure the ratio of test cases that can execute successfully among all test cases generated during each fuzzing session. Table I presents the results. With the DFAFuzz approach, DFAFuzzer maintains stack consistency and type consistency in its generated bytecode. Compared to AFL++, DFAFuzzer achieves 7.6%

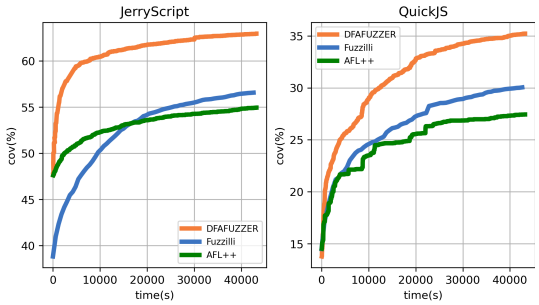

Fig. 12: Coverage comparison with state-of-the-Art fuzzers.

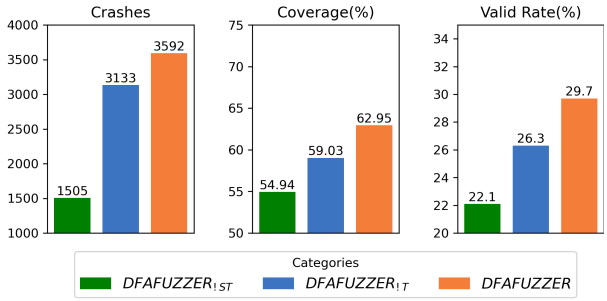

Fig. 13: Results of ablation study.

and 2.8% higher valid seed ratios on JerryScript and QuickJS, respectively. Note that Fuzzilli focuses on generating correct and diverse JavaScript source code, which is inherently easier than producing correct and diverse bytecode, resulting in its higher valid seed ratio.

Program diversity. To verify whether DFAFUZZ is capable of generating more diverse bytecode programs, we employ Unique Instruction Bigrams to evaluate the diversity of generated bytecode. Specifically, an instruction bigram refers to an ordered pair of adjacent bytecode instructions, and analyzing the variety of these combinations effectively captures inter-instruction patterns, thereby reflecting the structural complexity and diversity of bytecode.

For systematic diversity measurement, we introduce two metrics: UIB_a represents the average count of Unique Instruction Bigrams per bytecode sample, quantifying the breadth of generation capability; UIB_m denotes the median value across all samples, indicating the stability of generation results.

In bytecode diversity evaluation, fuzzers like AFL++ and DFAFUZZER that perform mutations directly at the bytecode level demonstrate significantly higher counts of Unique Instruction Bigrams compared to Fuzzilli. Particularly on JerryScript, the bytecode generated by DFAFUZZER achieves approximately 5.5 times higher UIB_a than Fuzzilli. This outcome aligns with intuitive expectations: when contrasted with QuickJS, JerryScript features a more extensive bytecode instruction set with finer granularity, consequently offering greater possibilities for instruction combinations.

D. RQ3: Ablation Study

To evaluate the practical contributions of both the DFA executor and type reconstruction techniques in our system, we conducted ablation experiments by selectively disabling these components and constructing three distinct variants. Specifically, the complete system, DFAFUZZER, implements all constraints. The DFAFUZZER$_{!T}$ variant specifically disables the type reconstruction technique, relaxing type verification during bytecode generation. The most limited variant, DFAFUZZER$_{!ST}$, disables both the DFA executor and type reconstruction techniques, entirely removing enforcement of operand stack and type consistency during generation.

We systematically evaluated these variants on the target VMs with respect to crash discovery, edge-path coverage, and valid seed generation. From a fuzzing perspective, crash count serves as a strong indicator of potential vulnerabilities, as it reflects how effectively the generated inputs can perturb program behavior without requiring manual analysis.

Fig. 13 presents the results quantitatively to demonstrate how each technique affects overall fuzzing efficacy. First, after disabling type consistency and stack consistency validations, DFAFUZZER$_{!ST}$ and DFAFUZZER$_{!T}$ exhibited 8.01% and 3.92% reduction in edge coverage and 2,087 and 459 fewer discovered crashes compared to the complete DFAFUZZER variant. This substantial discrepancy underscores the critical importance of maintaining type and stack consistency for effective fuzzing.

Second, the DFAFUZZER variant demonstrated a 4.2% and 7.6% improvement in bytecode generation validity compared to DFAFUZZER$_{!T}$ and DFAFUZZER$_{!ST}$, respectively. This enhancement primarily stems from the rigorous type consistency and stack consistency guarantees provided by DFA executor and type reconstruction. In contrast, when these technologies were disabled in DFAFUZZER$_{!T}$ and DFAFUZZER$_{!ST}$, the generated bytecode lacked essential type and stack state validation, resulting in a higher probability of semantically invalid programs.

These results conclusively demonstrate that the technologies of ensuring bytecode semantic correctness in DFAFUZZER, dramatically enhance DFAFUZZER's capability to explore deeper execution paths and uncover potential bugs by generating more valid seeds.

E. RQ4: Security Impact

To further demonstrate DFAFUZZER's bug detection capability and assess its practical security impact, we conduct two case studies on bugs discovered by DFAFUZZER, classified as a global buffer overflow and a heap overflow, respectively.

Global buffer overflow. We present in Fig. 14 a global buffer overflow bug detected by DFAFUZZER from JerryScript. This code reads the branch offset `branch_offset` of the instruction and then updates the current program counter `byte_code_p` to point to the target address. Next, the VM will fetch and execute the instruction with respect to the new target address. Unfortunately, the program counter may point to an out-of-bounds instruction address because the jump address is not validated, leading to a global buffer overflow.

```
if (boolean_value) {
  byte_code_p = byte_code_start_p + branch_offset;
  if (opcode_flags & VM_OC_LOGICAL_BRANCH_FLAG) {
    /* "Push" the value back to the stack. */
    ++stack_top_p;
    continue;
  }
}
const uint8_t *byte_code_start_p = byte_code_p;
uint8_t opcode = *byte_code_p++;
uint32_t opcode_data = opcode;
```

Fig. 14: A global buffer overflow bug detected by DFAFUZZER.

This bug could lead to critical security vulnerabilities. For example, an adversary may craft malicious jump addresses to hijack program control flow, enabling remote code execution (RCE) through redirected instruction pointers. As another example, invalid jumps also risk triggering segmentation faults that crash the program, leading to denial-of-service (DoS).

```
copy_size = JERRY_MIN (string_size, buffer_size);
if (copy_size < string_size) {
  .....
}
// Copy the string to the buffer
memcpy (buffer_p, chars_p, copy_size);
```

Fig. 15: A heap overflow bug detected by DFAFUZZER.

Heap overflow. Fig. 15 demonstrates a heap overflow bug uncovered by DFAFUZZER. This code snippet copies `copy_size` bytes from the source address `chars_p` into the destination address `buffer_p` by invoking a notorious C library function `memcpy` that lacks proper pointer range checking. Unfortunately, this code does not validate the `copy_size` argument. Consequently, for large enough `copy_size`, the function `memcpy` writes beyond the end of destination buffer `buffer_p`, triggering a heap overflow to overwrite the contiguous memory.

Worse yet, this code snippet does not validate the source address `chars_p` also. Consequently, the function `memcpy` might access the memory beyond the end of `chars_p` for large enough `copy_size`, potentially leading to information leakage of sensitive data residing in that memory.

VII. DISCUSSION

In this section, we discuss the limitations of DFAFUZZ and our plans for addressing these limitations in future work.

More precise bytecode semantics. DFAFUZZ adopts a type-directed DFA approach to generate diverse and valid bytecode for detecting deep bugs in embedded JavaScript VMs. Although it improves bytecode validity, some type errors remain (see Table I), mainly due to imprecise tracking of object types in memory, which affects type inference during generation.

To mitigate this, we plan to integrate advanced static analysis techniques to guide memory type-state transitions and enhance semantic accuracy. Prior work has shown that static analysis can effectively detect memory errors and improve code quality [25]. By leveraging such techniques, DFAFUZZ can achieve more precise memory modeling and type tracking, further improving the correctness of generated bytecode.

Support for other embedded JavaScript VMs. While DFAFUZZ is designed to be general, evaluations so far focus on JerryScript and QuickJS. Extending it to other bytecode-based VMs, such as Duktape [18], faces challenges due to bytecode heterogeneity.

We plan to leverage the IR from JASFree [26] to unify abstraction across VMs, preserving type and stack consistency. This enables semantically sound fuzzing on diverse bytecode formats. Furthermore, combining this with differential testing [27] could uncover not only security bugs but also functional discrepancies across VMs.

Future work on exploitation paths and mitigation. While DFAFUZZ reveals crash-type vulnerabilities in embedded JavaScript VMs, for high-severity issues (e.g., global buffer overflows) we will perform deeper exploit-path and mitigation analyses. We will emulate stack-smashing and return-oriented programming to evaluate the feasibility of code injection and control-flow hijacking, and adopt a DisARM-style validation-instrumentation workflow [28] to automatically insert compile-time boundary checks and integrity verifications to mitigate injection-based and reuse-based buffer-overflow attacks.

VIII. RELATED WORK

Fuzzing for JavaScript VMs. Early JavaScript fuzzers, such as jsfunfuzz [29], generated syntactically valid programs based on predefined rules and vulnerability patterns to pass strict syntax checks in JavaScript VMs. Later, Superion [6] enhanced coverage by adopting syntax-aware mutations, while Godefroid et al. [30] used symbolic execution [31] and constraint solving to ensure syntactic correctness.

To explore deeper VM states, researchers then aimed to generate semantically valid test cases [9] [32]. DIE [9] preserved key semantics during mutation; Skyfire [33] learned probabilistic models from real-world samples; CodeAlchemist [34] recombined code snippets via data flow analysis; and Fuzzilli [8] introduced an intermediate language, FuzzIL, to maintain semantic correctness throughout generation and mutation.

However, these tools mainly target JavaScript source code and overlook the underlying bytecode execution model used in embedded JavaScript VMs. In contrast, DFAFUZZ conducts fuzzing directly at the bytecode level, enhancing bytecode diversity and improving the effectiveness of fuzzing for embedded JavaScript VMs.

Coverage-guided Fuzzing. American Fuzzy Lop (AFL) [35] is a widely used fuzzing framework that employs compile-time instrumentation and genetic algorithms to discover inputs triggering new program states. By prioritizing seeds that expand code coverage, AFL effectively detects diverse vulnerabilities [36] and has inspired many follow-up works.

Subsequent studies improved AFL's coverage and efficiency. CollAFL [37] addressed path collisions through low-overhead instrumentation; VUzzer [38] integrated taint and static analysis to guide input generation; and AFLFAST [39] modeled fuzzing as a Markov process to optimize seed selection and energy allocation.

However, these approaches remain less effective for embedded JavaScript VMs, as they cannot capture bytecode states and thus struggle to ensure semantic correctness. In contrast, DFAFuzz tracks bytecode execution states during fuzzing, guaranteeing semantic validity and improving effectiveness.

IX. Conclusion

This paper proposes DFAFuzz, a bytecode-fuzzing approach for embedded JavaScript VMs. Leveraging a stack-based DFA enriched with type information, DFAFuzz generates semantically valid yet structurally diverse bytecode. A prototype and extensive experiments on multiple mainstream embedded JS VMs show superior bug-finding and coverage, uncovering 248 unique bugs — 111 of which were missed by state-of-the-art fuzzers. These results validate DFAFuzz's practical effectiveness for security testing.

References

[1] C. Severance, "Javascript: Designing a language in 10 days," *Computer*, vol. 45, no. 2, pp. 7–8, 2012.

[2] "Cve-2023-36109 detail," https://nvd.nist.gov/vuln/detail/CVE-2023-36109.

[3] J. Yun, F. Rustamov, J. Kim, and Y. Shin, "Fuzzing of embedded systems: A survey," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–33, 2022.

[4] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, pp. 1–13, 2018.

[5] C. Salls, C. Jindal, J. Corina, C. Kruegel, and G. Vigna, "{Token-Level} fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2795–2809.

[6] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 724–735.

[7] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[8] S. Groß, S. Koch, L. Bernhard, T. Holz, and M. Johns, "Fuzzilli: Fuzzing for javascript jit compiler vulnerabilities." in *NDSS*, 2023.

[9] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing javascript engines with aspect-preserving mutation," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1629–1642.

[10] H. Xu, Z. Jiang, Y. Wang, S. Fan, S. Xu, P. Xie, S. Fu, and M. Payer, "Fuzzing javascript engines with a graph-based ir," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 3734–3748.

[11] X. Liu, W. You, Y. Ye, Z. Zhang, J. Huang, and X. Zhang, "Fuzzinmem: Fuzzing programs via in-memory structures," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[12] J. Poial, "Validation of stack effects in java bytecode,"," in *Proc. of the Fifth Symposium on Programming Languages and Software Tools, June*, 1997, pp. 7–8.

[13] A. Gal, C. W. Probst, and M. Franz, "Java bytecode verification via static single assignment form," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 30, no. 4, pp. 1–21, 2008.

[14] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. springer, 2015.

[15] S. Chandra, C. S. Gordon, J.-B. Jeannin, C. Schlesinger, M. Sridharan, F. Tip, and Y. Choi, "Type inference for static compilation of javascript," *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 410–429, 2016.

[16] "Jerryscript: Javascript engine for the internet of things," https://github.com/jerryscript-project/jerryscript.

[17] F. Bellard, "Quickjs javascript engine," 2019.

[18] S. Vaarala, "Duktape embeddable javascript engine," *URL https://duktape. org*, 2020.

[19] "Cve-2020-24187 detail," https://nvd.nist.gov/vuln/detail/CVE-2020-24187.

[20] L. Zhang, B. Zhao, J. Xu, P. Liu, Q. Xie, Y. Tian, J. Chen, and S. Ji, "Waltzz: Webassembly runtime fuzzing with stack-invariant transformation."

[21] M. Schoeberl, "Design and implementation of an efficient stack machine," in *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2005, pp. 8–pp.

[22] "Fuzzing javascript engines with fuzzilli," https://blog.doyensec.com/2020/09/09/fuzzilli-jerryscript.html.

[23] "Libafl quickjs fuzzing," https://github.com/andreafioraldi/libafl_quickjs_fuzzing.

[24] Z.-M. Jiang, J.-J. Bai, and Z. Su, "{DynSQL}: Stateful fuzzing for database management systems with complex and valid {SQL} query generation," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4949–4965.

[25] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[26] H. Jiang, H. Lai, S. Wu, and B. Hua, "Jasfree: Grammar-free program analysis for javascript bytecode," in *2024 IEEE 23rd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2024, pp. 326–337.

[27] Y. Fu, M. Ren, F. Ma, X. Yang, H. Shi, S. Li, and X. Liao, "Evm-fuzz: Differential fuzz testing of ethereum virtual machine," *Journal of Software: Evolution and Process*, vol. 36, no. 4, p. e2556, 2024.

[28] J. Habibi, A. Panicker, A. Gupta, and E. Bertino, "Disarm: mitigating buffer overflow attacks on embedded devices," in *International Conference on Network and System Security*. Springer, 2015, pp. 112–129.

[29] W. Syndder and M. Shaver, "Building and breaking the browser," in *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, Aug 2007.

[30] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *PLDI*, 2008, pp. 206–215.

[31] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[32] J. Patra and M. Pradel, "Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data," *TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664*, 2016.

[33] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 579–594.

[34] H. Han, D. Oh, and S. K. Cha, "Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines." in *NDSS*, 2019.

[35] M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl/technical_details.txt, 2017, accessed: 2025-04-11.

[36] M. Böhme, C. Cadar, and A. Roychoudhury, "Fuzzing: Challenges and reflections," *IEEE Software*, vol. 38, no. 3, pp. 79–86, 2020.

[37] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 679–696.

[38] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *2017 Network and Distributed System Security (NDSS) Symposium:[Proceedings]*. Internet Society, 2017, pp. 1–14.

[39] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1032–1043.