JasFree: Grammar-free Program Analysis for JavaScript Bytecode

Hao Jiang Haiwei Lai Si Wu Baojian Hua*
School of Software Engineering, Suzhou Institute for Advanced Research
University of Science and Technology of China
{jh7, sa23225261, wusi98}@mail.ustc.edu.cn bjhua@ustc.edu.cn

***Abstract*—JavaScript is rapidly being deployed as binaries in security-critical embedded domains, including IoT devices, edge computing, and smart automotive applications. Ensuring the security of JavaScript binaries in these domains necessitates comprehensive binary code analysis. However, despite the urgent need, a universal approach to analyzing JavaScript binaries is lacking due to the bytecode heterogeneity across the various JavaScript virtual machines.**

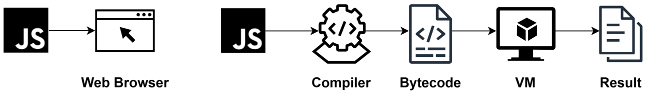
In this paper, to fill this gap, we present the first grammar-free, universal program analysis approach tailored for JavaScript binaries. We first design a syntax-independent intermediate representation called JasByte to encode diverse JavaScript binaries. We then develop a universal translator equipped with a set of APIs to transform JavaScript binaries into JasByte. We design a suit of program analysis algorithms for error detection, debugging, and fuzzing, to identify bugs in JavaScript VMs. We design and implement a software prototype JasFree and conduct extensive evaluations. Our results show that JasFree effectively enables construction of diverse static and dynamic analysis by reducing the overhead from 660.38% to 290.84%, outperforming the state-of-the-art tool Jalangi2. Moreover, JasFree facilitates effective mutation of JasByte, resulting in the detection of 25 new vulnerabilities, all of which were missed by existing methods.

***Index Terms*—JavaScript Bytecode, Grammar-Free, Program Analysis**

I. Introduction

Originally designed for the Web, JavaScript [1] is rapidly being deployed in security-critical embedded domains, including the Internet of Things (IoT) [2], edge computing [3], and smart automotive applications [4]. This growing deployment has fundamentally altered the mechanisms by which JavaScript programs are compiled and executed. Specifically, as shown in Fig. 1, in the Web domain, a JavaScript program is distributed in source form through the network and then is executed by a target browser (e.g., JavaScriptCore [5], V8 [6], SpiderMonkey [7], or ChakraCore [8]). However, in the embedded domain, a JavaScript program is first compiled by a compiler into standalone JavaScript bytecode, which is then loaded and executed by an underlying embedded VM (e.g., JerryScript [9], QuickJS [10], Duktape [11], or MuJS [12]).

Unfortunately, while such new deployments significantly benefit embedded domains through JavaScript's secure guarantees and rapid prototyping, they introduce two new attack vectors for adversaries. First, vulnerabilities in JavaScript bytecode programs are susceptible to exploitation. For example, the exploitation of the CVE-2023-20198 [13] vulnerability in 2023 caused a massive cyber-attack on more than ten thousand Cisco devices [14]. Second, the underlying embedded JavaScript VMs may contain exploitable vulnerabilities due to their substantial code size and complex logic. For example, the CVE-2020-13991 vulnerability [15] in the JerryScript VM allows for control-flow hijack by manipulating registers. Therefore, these new attack vectors highlight the urgency of developing innovative approaches to study security of JavaScript bytecode and the underlying embedded VMs.



(a) JS source is executed by a target browser.

(b) JS source is first compiled by a compiler into bytecode, which is then loaded and executed by an embedded JS VM.

Fig. 1: The deployment of JavaScript in security-critical embedded domains has altered the JS's execution mechanism. (a) In the Web domain, the JavaScript source code is distributed in sources through the network, then is executed by a target browser. (b) In the embedded domain, the JavaScript sources is first compiled by a compiler into standalone bytecode, which is then loaded into an embedded VM for execution.

Despite this urgency, few studies have been conducted in this area. On one hand, while many approaches for JavaScript program analysis have been proposed (e.g., ESLint [16], Esprima [17], Google's Closure Compiler [18], and Jalangi2 [19]), they are ineffective for analyzing JavaScript binaries, as these approaches focus on JavaScript source programs. Instead, vulnerabilities in JavaScript bytecode may not manifest at source-level, because they can be introduced by buggy JavaScript compilers [20] or manually crafted and injected directly at binary-level by adversaries. On the other hand, although many studies have aimed to detect bugs in JavaScript VMs [21] [22], they only consider Web-oriented engines and do not consider embedded JavaScript VMs. Furthermore, adapting existing studies to investigate embedded JavaScript VMs remains challenging due to the significant differences in

[*] The corresponding author.

VM designs and implementation logic.

We argue that a fundamental challenge in developing new approaches to analyze JavaScript bytecode is addressing the issue of *grammar heterogeneity*. Specifically, JavaScript bytecode from different VMs have heterogeneous grammar for bytecode definition due to the lacking of a standardized specification. Hence, even if we can develop an analysis algorithm targeting a specific JavaScript VM's bytecode, it remains challenging to adapt that algorithm to another VM's bytecode, given the significant difference in bytecode design and also the rapidly growing number of VMs. Furthermore, analyzing and modifying existing JavaScript VMs is difficult due to their substantial code size and complex implemtation logic. For example, the Espruino VM [23] has reached 1,065,247 lines of code and is still rapidly growing. Consequently, there is a pressing need for a JavaScript bytecode analysis approach that is free from grammar dependencies tied to specific JavaScript embedded VMs.

In this paper, we present the first grammar-free approach, JASFREE, for JavaScript bytecode analysis. Our approach encompasses three components. First, we propose a universal intermediate representation (IR) called JASBYTE to uniformly represent diverse JavaScript bytecode grammars used by existing JavaScript embedded VMs. Specifically, JASBYTE employs a stack-based bytecode instruction design due to its technical advantages of regularity and compactness, which are particularly valuable for embedded JavaScript VMs where binary code size and execution speed are primary concerns. Second, we develop a programming interface that includes a set of application programming interfaces (APIs). By leveraging these APIs, end-users can create JavaScript bytecode translator to transform diverse JavaScript bytecode into the JASBYTE representation. Third, we introduce a set of program analysis algorithms to analyze JASBYTE, which includes static analyses such as control-flow, data-flow, and call graph analysis [24] [25] [26], and dynamic analyses such as profiling. Furthermore, we develop a group of syntax-directed mutation strategies to mutate bytecode programs, enabling effective syntactic-based fuzzing of embedded JavaScript VMs.

We implement a software prototype of JASFREE and conduct extensive experiments to evaluate its usability, efficiency, and fuzzing capabilities on both micro- and real-world benchmarks. The experimental results indicate that JASFREE efficiently perform dynamic analysis, introducing a code size overhead of 290.84% on average, which is significantly lower than Jalangi2's overhead of 660.38%. During the fuzzing of two embedded VMs using JASFREE's grammar-free mutation approach, JASFREE successfully detected 25 new vulnerabilities across three categories, all of which were missed by existing tools.

To summarize, our work makes the following contributions:

- We propose the first grammar-free approach that comprises new IR and APIs design, program analysis, and mutation strategies, to analyze heterogeneous JavaScript bytecode.
- We design and implement a software prototype JASFREE to validate our approach.
- We conduct extensive experiments to evaluate our approach. And experimental results show that JASFREE can efficiently perform program analysis, and the use of JASBYTE for fuzzing can effectively uncover new vulnerabilities in embedded JavaScript VMs.
- 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.13859256`.

The rest of this paper is organized as follows. Section II presents the background for this study. Section III presents our motivation. Sections IV and V present our approach. 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

To be self-contained, in this section, we present the background knowledge of JavaScript bytecode and embedded JavaScript VMs (§ II-A), program analysis (§ II-B), and fuzzing (§ II-C).

A. *JavaScript Bytecode and Embedded Virtual Machines*

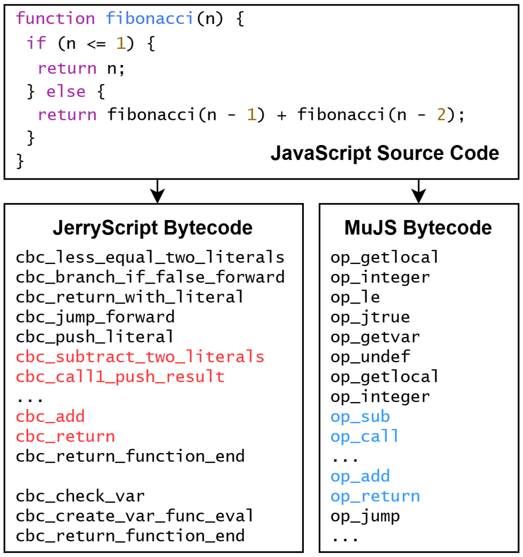
JavaScript bytecode is an abstract instruction format for distribution and execution. Specifically, JavaScript source code is compiled by JavaScript compilers into JavaScript bytecode, which is then loaded and executed by the underlying embedded JavaScript VMs (see Fig. 1b).

JavaScript bytecode offers several technical benefits making it well-suited for embedded domains. First, JavaScript bytecode enables more rapid program parsing and loading, thereby enhancing runtime efficiency [27]. Second, JavaScript bytecode supports effective code obfuscation, preventing reverse engineering and safeguarding intellectual property [28]. Finally, JavaScript bytecode is stack-based thus making the binary program compact, which minimizes the resource requirements and adds significant value in resource-constrained environments.

JavaScript bytecode is executed by the underlying embedded virtual machines, that are designed with a focus on low memory footprint, rapid startup times, and high portability [29], tailored for embedded environments. Consequently, numerous embedded JavaScript VMs are developed and deployed, including JerryScript [30], QuickJS [10], Duktape [31], and MuJS [12]. In the coming decade, a desire to deploy JavaScript programs in more embedded domains such as such as edge computing and IoT devices [32] will make JavaScript VMs more attractive and promising.

B. *Program Analysis*

Program analysis is an crucial aspect of software engineering, encompassing both static and dynamic techniques to analyze program behaviors and properties. Static program analysis [33] [34] examines software behavior without execution



```
function fibonacci(n) {
 if (n <= 1) {
  return n;
 } else {
  return fibonacci(n - 1) + fibonacci(n - 2);
 }
}
```

```
cbc_less_equal_two_literals
cbc_branch_if_false_forward
cbc_return_with_literal
cbc_jump_forward
cbc_push_literal
cbc_subtract_two_literals
cbc_call1_push_result
...
cbc_add
cbc_return
cbc_return_function_end

cbc_check_var
cbc_create_var_func_eval
cbc_return_function_end
```

```
op_getlocal
op_integer
op_le
op_jtrue
op_getvar
op_undef
op_getlocal
op_integer
op_sub
op_call
...
op_add
op_return
op_jump
...
```

Fig. 2: The JerryScript and MuJS VMs generate different bytecode for the same Fibonacci function. For better understanding, we present the bytecode's assembly form instead of its binary form. Specifically, the two bytecode sequences generated for the same recursive call `fibonacci(n - 1) + fibonacci(n - 2)` are colored with red (left) and blue (right).

[35]. It constructs internal data structures encoding programs and then analyzes static information by leveraging analysis algorithms. Furthermore, static analysis is indispenible in identifying security vulnerabilities, including buffer overflows [36], SQL injections [37], and XSS attacks [38].

Dynamic program analysis [39] analyzes program behavior and performance during execution. Specifically, it first collects runtime information through instrumentation, sampling, or special hardware support, and then uses that information to effectively analyze program behaviors. Due to its capability for more precise analysis, dynamic program analysis is widely employed in security assessment and vulnerability detection [40] [41] [42].

C. *Fuzzing*

Fuzzing is an automated software testing method that injects invalid, malformed, or unexpected inputs into a system to reveal software defects and vulnerabilities [43] [44] [45] [46]. A key factor for effective fuzzing is generating complex and valid random inputs to exercise as many execution paths as possible, hence increasing program *coverage* [47]. To achieve this, the *mutation* approach [48] has been introduced to guide the generation of random inputs. In this approach, a set of initial valid inputs, referred to as *seeds*, are mutated to generate new inputs for testing. These new inputs are added to the seed set if their execution covers new program paths that have not yet been explored.

TABLE I: The substantial code sizes of embedded JavaScript VMs and program analysis frameworks make it difficult to adapt program analysis from one VM to another one.

Existing Research	Implementing Language	Version	LoC (Total)
JerryScript [30]	C	v2.4.0	204,856
MuJS [12]	C	1.3.4	18,501
Espruino [23]	C	2V23	1,065,247
Jalangi [49]	JavaScript	v0.11	462,664
Jalangi2 [19]	JavaScript	v0.2.5	498,257

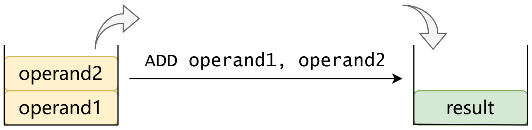


Fig. 3: The high-level abstract instruction `ADD operand1, operand2` establish Stack operation of `add` instruction in JavaScript VM.

III. Motivation and Challenges

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

A. *Motivation*

The reliance on bytecode grammar imposes significant limitations on the program analysis and mutations of JavaScript bytecode. Specifically, all program analysis algorithms and mutation strategies depend on the syntax of the bytecode for their respective JavaScript VMs. At the instruction level, even instructions that perform the same functionality can have different names and opcodes across various VMs. For example, as shown in Fig. 2, JerryScript [30] and MuJS [12] generate markedly different instruction sequences for the same JavaScript function calculating Fibonacci numbers. While these syntactic differences may seem minor at instruction level, they become magnified at the program level, complicating program analysis and mutation efforts.

Even more concerning, as depicted in Table I, existing embedded JavaScript VMs and program analysis frameworks for JavaScript have substantial code sizes, making the adaptation of exisiting analysis algorithms challenging. For example, the JerryScript VM comprises more than 204K lines of C, while the Jalangi2 analysis framework consists of more than 498K lines of JavaScript. Consequently, even if we successfully adapt Jalangi2 from JerryScript to another VM (e.g., Espruino), the task remains labor-intensive and error-prone.

Additionally, the embedded nature of JavaScript VMs constrains the size of code that can run on them, precluding the inclusion of program analysis functionalities. Therefore, there is an urgent need to develop grammar-free program analyses and mutations for embedded JavaScript VMs' bytecode, to overcome these syntactic variations.

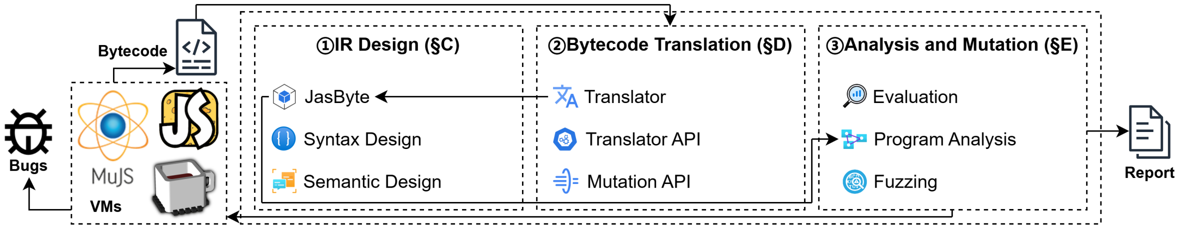


Fig. 4: An overview of JasFree's workflow.

B. Technical Challenges

The central challenge in developing a grammar-free program analysis and mutation approach for analyzing bytecode from diverse embedded JavaScript VMs is **designing a uniform program representation that maintains both syntactic and semantic accuracy**. To address this challenge, our key idea is to leverage the semantic information inherent in existing embedded JavaScript bytecode, designing a universal, grammar-free program intermediate representation (IR), on which program analyses and mutations can be performed.

Specifically, although bytecode for the same operation varies significantly across different JavaScript VMs, these variations share a common foundation in stack machine-based semantics, reflecting the code compactness requirement of the underlying embedded environment. For example, while JerryScript uses the bytecode instructions `cbc_add` and MuJS employs `op_add` to encode the addition operation + (see Fig. 2), both instructions exhibit the same operational semantics: they pop two operands, `operand1` and `operand2`, from the top of the stack, execute the `add` operation, and then push the result back onto the top of the stack, as shown in Fig. 3.

Based on this insight, we first propose a universal grammar-free IR, JasByte, to represent heterogeneous JavaScript bytecode (§ IV-C). This representation features formal syntax, rigorous semantics, and a set of APIs to encode bytecode. We then develop a comprehensive suite of program analysis algorithms and mutation strategies, all facilitated by JasByte (§ IV-E). Evaluation results (§ VI) demonstrated that our approach effectively addresses the challenge identified, providing a grammar-free program analysis and mutation, making an important step towards defining a unified program analysis framework for the embedded JavaScript ecosystem.

IV. Approach

In this section, we present our approach in designing JasFree. We first describe our design goals (§ IV-A), followed by an overview of its workflow (§ IV-B). We then discuss the grammar-free language model (§ IV-C), universal translator (§ IV-D), and program analysis and fuzzing mutations JasFree enabled (§ IV-E).

A. Design Goals

We design JasFree with three main goals: 1) comprehensiveness, 2) full automation, and 3) easy extension. First, JasFree aims for the comprehensiveness to support a large spectrum of program analysis and mutations. Second, JasFree should be fully automated, minimizing manual intervention and operations. Third, JasFree should be extensible to support a variation of bytecode and to adapt to future VM implementations.

$$\begin{array}{lcl}
type_{val} & ::= & \texttt{undefined} \mid \texttt{null} \mid \texttt{bigint} \\
 & \mid & \texttt{number} \mid \texttt{string} \\
type_{func} & ::= & type^{*}_{val} \rightarrow type^{*}_{val} \\
type & ::= & type_{val} \mid type_{func} \\
unary & ::= & \texttt{plus} \mid \texttt{not} \mid \texttt{negate} \mid \ldots \\
binary & ::= & \texttt{add} \mid \texttt{mul} \mid \texttt{shl} \mid \ldots \\
load/store & ::= & type_{val}.\texttt{load} \mid type_{val}.\texttt{store} \\
call & ::= & \texttt{call}\ function \\
instr & ::= & unary \mid binary \mid load/store \\
 & \mid & local\ op. \mid global\ op. \mid call \\
 & \mid & \texttt{nop} \mid \texttt{push} \mid \texttt{pop} \mid \texttt{jump}\ a \\
 & \mid & \texttt{loop} \mid \texttt{end} \mid \texttt{br}\ a \mid \texttt{br\_if}\ a \\
 & \mid & \texttt{assign} \mid \texttt{create} \mid \texttt{set} \mid \texttt{return} \\
 & \mid & \texttt{select} \mid \texttt{memory\_grow} \\
 & \mid & type_{val}.\texttt{const}\ c \mid \ldots \\
function & ::= & type_{func}\ x\{instr^{*}\} \\
module & ::= & function^{*}
\end{array}$$

Fig. 5: Representative abstract syntax of JasByte, defined by a context-free grammar.

B. Overview

With these design goals, we present an overview of JasFree's workflow in Fig. 4, consisting of three primary components. First, in the IR design (①), we design an intermediate representation dubbed JasByte, to represent heterogeneous bytecode from various embedded JavaScript VMs. Second, in the bytecode translation (②), we propose a versatile translator that converts bytecode from different JavaScript VMs into our IR. Finally, in the analysis and mutation (③), we develop a suite of algorithms for static and dynamic program analysis, along with a set of mutation strategies for fuzzing.

C. Intermediate Representation Design

Syntax Design. Designing a formal syntax is a first step towards defining an intermediate representation. To this end,

$$
\begin{array}{rcl}
(c)\ unary & \rightarrow & unary(c) \\
(c_1)\ (c_2)\ binary & \rightarrow & c \\
\texttt{nop} & \rightarrow & e \\
v_1\ v_2\ (0)\ \texttt{select} & \rightarrow & v_2 \\
v_1\ v_2\ (k+1)\ \texttt{select} & \rightarrow & v_1 \\
(0)\ (\texttt{br\_if}\ j) & \rightarrow & e \\
(k+1)\ (\texttt{br\_if}\ j) & \rightarrow & \texttt{br}\ j \\
s\ arg_1\ ...\ arg_j\ ;\ call\ j & \rightarrow & call\ s_{func}(arg_1, ..., arg_j) \\
v_1^j\ v\ v_2^k\ ;\ \texttt{get\_local}\ j & \rightarrow & v \\
v_1^j\ v\ v_2^k\ ;\ v'\ (\texttt{set\_local}\ j) & \rightarrow & v_1^j\ v'\ v_2^k\ ;\ e
\end{array}
$$

Fig. 6: Representative semantic rules for the operational semantics of JASBYTE instructions. The left column represents the current operand stack state with an instruction to be evaluated, while the right column gives the instruction's evaluation result.

we design an abstract universal intermediate language called JASBYTE utilizing a context-free grammar, as illustrated in Fig. 5. Our IR design closely follows existing stack machine bytecode deployed in current embedded JavaScript VMs, while also being abstract and high-level enough to facilitate the manipulation and development of effective analysis algorithms.

Specifically, a module *module* in JASBYTE consists of a list of functions $function$, where each $function$'s body comprises a series of instructions *instr*. A $function$ takes multiple parameters and returning results, represented by its type $type^*_{val} \rightarrow type^*_{val}$ in which the symbol * denotes a Kleene closure.

An instruction *instr* encompasses binary/unary operations, memory load/stores, structured control flows, and function call/returns. To keep the IR concise, some bytecodes are encoded by the same underlying IR instruction. The instruction design in our IR reflects important characteristics of bytecode from the embedded JavaScript VMs: first, our IR design follows the embedded JavaScript VM's stack-based execution model, which is in turn similar to traditional stack machines such as the JVM [50]. Operand values and results are consistently located at the top of the operand stack. Second, our IR design supports structured control flows, offering commonly used structured statements and control structures, such as conditions (e.g., `if-else`), loops (e.g., `while` and `for`), and jumps (e.g., `break` and `continue`). This design decision significantly facilitates the development of analysis algorithms.

Semantic design. To define the mathematically rigorous semantics of our IR JASBYTE, we adopt the approach of operational semantics, in which semantics specification are given through the state transitions of the underlying abstract virtual machines. For brevity, we present representative semantic specification rules of JASBYTE in Fig. 6. The left column represents the current operand stack state with an instruction to be evaluated, while the right column presents the instruction's evaluation result. For example, the rule $(c_1)\,(c_2)\ binary \rightarrow c$ specifies that if the constants residing at the top of the operand stack are c_1 and c_2, then the execution of binary instruction $binary$ will give a constant c where $c = c_1\ binary\ c_2$. Similarly, for the branch instruction `br_if`, if the top of operand stack is 0, the flow falls through to the next instruction e; otherwise, if the top of operand stack is $k+1 \neq 0$, the flow transfers to the specified jump target j. The semantics rules for other instructions are similar and therefore require no further explanation.

```
// new
jasbyte_ins *op_new_array(jasbyte_func *func);
jasbyte_ins *op_new_object(jasbyte_func *func);
// stack
jasbyte_ins *op_dup(jasbyte_func *func);
jasbyte_ins *op_rot2(jasbyte_func *func);
jasbyte_ins *op_pop(jasbyte_func *func);
jasbyte_ins *op_push_true(jasbyte_func *func);
jasbyte_ins *op_push_false(jasbyte_func *func);
jasbyte_ins *op_push_this(jasbyte_func *func);
jasbyte_ins *op_push_int(jasbyte_func *func, int val);
jasbyte_ins *op_getlocal(jasbyte_func *func, int index);
jasbyte_ins *op_setlocal(jasbyte_func *func, int index);
jasbyte_ins *op_dellocal(jasbyte_func *func, int index);
// var
jasbyte_ins *op_hasvar(jasbyte_func *func, uint64_t
     raw_val, char *var_name);
jasbyte_ins *op_getvar(jasbyte_func *func, uint64_t
     raw_val, char *var_name);
jasbyte_ins *op_setvar(jasbyte_func *func, uint64_t
     raw_val, char *var_name);
jasbyte_ins *op_delvar(jasbyte_func *func, uint64_t
     raw_val, char *var_name);
// bop
jasbyte_ins *op_binary(jasbyte_func *func, jasbyte_op op);
// transfer
jasbyte_ins *op_jmp(jasbyte_func *func, jasbyte_op op,
     jasbyte_ins *target_ins);
jasbyte_ins *op_call(jasbyte_func *func, unsigned int
     arg_len);
jasbyte_ins *op_return(jasbyte_func *func);
```

Fig. 7: Representative translation APIs encapsulating JASBYTE internal design, which are classified into different categories of allocations, stack operations, variable manuplations, binary operations, and control transfers such as branch and function call/returns.

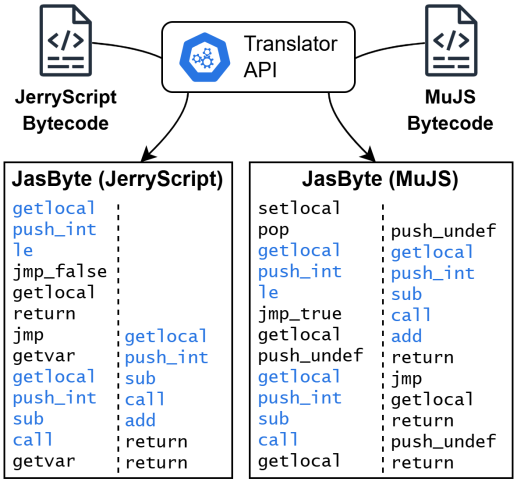
D. Bytecode Translation

Translation APIs. To utilize JASBYTE's capabilities, we first need to translate diverse bytecode into this uniform IR. A natural approach to achieve this is to develop dedicated compiler passes that convert bytecode into JASBYTE. However, a key challenge is that this method requires deep knowledge of JASBYTE internal design and may limit future extension due to external dependencies. To address this challenge, we have developed a set of APIs, as shown in Fig. 7, to encapsulate JASBYTE, following the design philosophy of implementation-interface separation.

These APIs are classified into different categories, according to their semantics. And our current categories include allocations, stack operations, variable manipulations, binary operations, and control transfers such as branches and function invocations. Specifically, we intentionally keep key data structures `jasbyte_func` for functions and `jasbyte_ins`

TABLE II: Representative instruction translation rules from bytecode for embedded JavaScript VMs into JASBYTE.

Categories	MuJS Bytecode	JerryScript Bytecode	JASBYTE Instruction
Stack operation instructions	op_pop	cbc_pop	pop
Unary operation instructions	op_typeof	cbc_typeof	typeof
Logical operation instructions	op_lognot	cbc_logical_not cbc_logical_not_literal	lognot
Arithmetic instructions	op_add	cbc_add cbc_add_right_literal cbc_add_two_literals	add
	op_mod	cbc_modulo cbc_modulo_right_literal cbc_modulo_two_literals	mod
Comparison instructions	op_gt	cbc_greater cbc_greater_right_literal cbc_greater_two_literals	gt
	op_le	cbc_less_equal cbc_less_equal_right_literal cbc_less_equal_two_literals	le
	op_stricteq	cbc_strict_equal cbc_strict_equal_right_literal cbc_strict_equal_two_literals	stricteq
Bitwise operation instructions	op_bitnot	cbc_bit_not cbc_bit_not_literal	bitnot
	op_bitand	cbc_bit_and cbc_bit_and_right_literal cbc_bit_and_two_literals	bitand
	op_bitor	cbc_bit_or cbc_bit_or_right_literal cbc_bit_or_two_literals	bitor
Stack operation instructions	op_jump	cbc_jump	jump
	op_return	cbc_return	return



```
getlocal   ¦
push_int   ¦
le         ¦
jmp_false  ¦
getlocal   ¦
return     ¦
jmp        ¦getlocal
getvar     ¦push_int
getlocal   ¦sub
push_int   ¦call
sub        ¦add
call       ¦return
getvar     ¦return
```

```
setlocal   ¦
pop        ¦push_undef
getlocal   ¦getlocal
push_int   ¦push_int
le         ¦sub
jmp_true   ¦call
getlocal   ¦add
push_undef ¦return
getlocal   ¦jmp
push_int   ¦getlocal
sub        ¦return
call       ¦push_undef
getlocal   ¦return
```

Fig. 8: Bytecode from JerryScript and MuJS are both translated into JASBYTE's uniform representation.

for instructions abstract, so that external implementations do not depend on internal representation. For example, to create an instruction pushing an integer onto the operation stack, one invokes the API `op_push_int` by passing an abstract `func` of the type `jasbyte_func`. We provide comprehensive documentation for these APIs as a reference for end developers.

Bytecode translations. With these APIs shown in Fig. 7, developers can translate diverse bytecode from embedded VMs into JASBYTE, using translation rules. We present representative translation rules for converting MuJS and JerryScript's bytecode into JASBYTE in Table II. Thanks to JASBYTE's clean interface design, most translation rules are syntax-directed, making them easy to understand and develop. For example, both the instruction `cbc_pop` from JerryScript and `op_pop` from MuJS are both directly translated into `pop` in JASBYTE. However, some JavaScript VMs have peculiarities in their bytecode designs; the translation rules generate cleaner JASBYTE by eliminating such peculiarities during translation, which considerably simplifies subsequent processing. For example, JerryScript has multiple instructions for addition, including `cbc_add`, `cbc_add_right_literals`, and `cbc_add_two_literal`. While these variations effectively represent instructions with integer literals thus saving code size by storing literals in constant tables instead of in bytecode streams, they do not significantly differ from the program analysis perspective. Consequently, the translation rules consolidate these variations into a single addition operation, add, in JASBYTE.

In Fig. 8, we present the generated JASBYTE for our running example illustrated in Fig. 2. The diverse bytecode from JerryScript and MuJS is translated into JASBYTE's uniform representation by invoking the translation APIs.

Mutation API. Our grammar-free intermediate representation, JASBYTE, facilitates mutation-based fuzzing for different embedded JavaScript VMs. In this approach, the fuzzer generates random IRs by mutating a set of input seeds. One key challenge in mutation-based fuzzing is effectively mutating the given seed to generate complex but valid output. To address this challenge, we propose a set of two APIs for mutation:

```
jasbyte_ins *mutate(jasbyte_ins *input);
bool validate(jasbyte_ins *ins);
```

the first mutate API randomly mutates an input JASBYTE instruction sequence to generate a new one. We employ standard strategies in our mutation API, including insertion, deletion, and substitution of relevant instructions. Furthermore, when employing coverage-guided fuzzing, generating valid instruction sequences is essential, as invalid instruction sequences often lead to early VM exit, thus reducing path coverage. To this end, we introduce an API validate to sanity-check the input instruction ins returned from mutations according to the semantics rules (see Fig. 6), significantly increasing coverage by exploring more potential paths.

E. Program Analysis and Mutation

Our grammar-free design of JASBYTE provides a solid foundation for developing expressive analysis algorithms and effective fuzzing mutation strategies. In the following sections, we present important design details for static and dynamic program analysis, as well as the fuzzing mutations enabled by JASFREE, to demonstrate its promising capabilities.

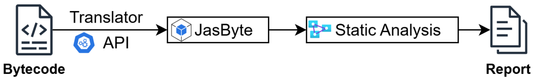


Fig. 9: The static analysis workflow of JASFREE.

Static analysis. We present the static analysis workflow of JASFREE in Fig. 9. First, we translate VM bytecode into JASBYTE by invoking the translation APIs (see Fig. 7). We then develop dedicated static analysis algorithms on JASBYTE to generate the final analysis report for subsequent evaluation. To demonstrate JASFREE's capabilities, we implement key static analysis algorithms, including control-flow analysis, data-flow analysis, and call-graph analysis, which are widely used in analysis, optimization, and security. These static analysis algorithms are distributed in our open-source package for public use.

Dynamic analysis. We present the dynamic analysis workflow of JASFREE in Fig. 10. Our approach leverages the instrumentation method to implement the dynamic analysis.

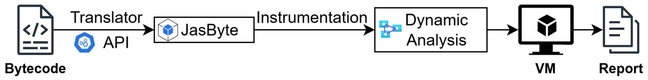


Fig. 10: The dynamic analysis workflow of JASFREE.

First, we instrument JASBYTE programs with relevant hooks to implement the desired functionalities. For example, to analyze dynamic call graphs, we instrument JASBYTE program by placing hooks at function entries and exits to trace the calls during execution. Second, we execute the instrumented binaries on JavaScript VMs. During program executions, the intrumented hooks will invoke the dynamic analysis libraries we implement in JavaScript, to generate analysis results.

To minimize the overhead introduced by instrumentation, such as code size increases, we employ an adaptive instrumentation approach, that allows for selective instrumentation based on analysis configurations. Our approach is feasible because most dynamic analyses require only a small subset of hooks, and different hooks are orthogonal to each other. For example, to dynamically analyze branch coverage, we adaptively hook branch-related instructions (see Fig. 5).

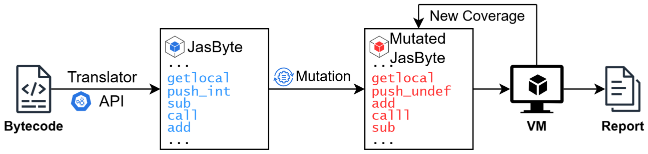
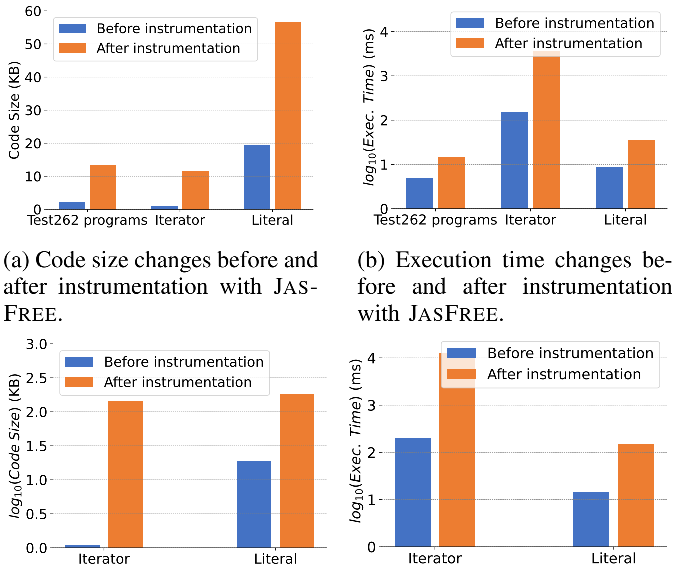


Fig. 11: The fuzzing mutation workflow of JASFREE.

Fuzzing mutations. We present the fuzzing mutation workflow of JASFREE in Fig. 11, to illustrate the effective mutations enabled by JASFREE. Our approach utilizes a combination of syntactic fuzzing [51] and greybox grammar fuzzing [52] methodologies. Specifically, we first generate a set of seeds that adhere to JASFREE's syntactic and semantic specifications. We then mutate these seeds to create new programs, based on various mutation strategies, including random alteration or substitution of instructions, arbitrary insertion of instructions, and indiscriminate removal of instructions. Next, we feed the newly generated programs to the target JavaScript VM, and add that program back to the seed set if that program's execution covers new paths that have not yet been explored. This iterative process continues until a threshold is reached (e.g., a timeout). A final report is generated, comprising unexpected behaviors such as VM crashes and suspensions, which often suggest potential flaws.

V. IMPLEMENTATION

To validate our approach, we designed and implemented a software prototype for JASFREE comprising a total of 5,169 lines of code (4,404 lines in C and 765 lines in JavaScript). The prototype is distributed as open-source software, and is included in our reproduction package for this paper. Next, we highlight some key implementation details.



(a) Code size changes before and after instrumentation with JasFree.

(b) Execution time changes before and after instrumentation with JasFree.

(c) Code size changes before and after instrumentation with Jalangi2.

(d) Execution time changes before and after instrumentation with Jalangi2.

Fig. 12: Comparison of code size and execution time between JasFree and Jalangi2. The Test262 programs serve as micro-benchmarks, while Iterator and Literal as macro-benchmarks.

Bytecode translation. To implement bytecode translation converting various bytecode into JasByte, we selected JerryScript [30] and MuJS [12] for this study, for several reasons. First, JerryScript is specifically designed for the resource-constrained embedded domain with 6.9k stars on GitHub [9], making it a representative embedded JavaScript VM. Second, MuJS is more compact consisting of only 18,501 lines of code, highlighting the necessity of studying lightweight JavaScript VMs. Our design of JasByte accurately models a stack-based machine, facilitating the translation from various bytecode format into JasByte. For example, we implemented the bytecode translation for JerryScript's bytecode (called snapshot) in just 977 lines of C code.

However, it is important to note that our approach to grammar-free program analysis and the bytecode translation proposed in this paper is independent of both JerryScript and MuJS, making it equally applicable to other JavaScript VMs.

Dynamic analysis. We implemented 42 low-level hooks in 1,498 lines of C code. Our choice of C as the implementation language for this component allowed us to manage low-level instrumentation details effectively, by leveraging C's low-level programming capabilities. We also implemented common program analysis algorithms, including dynamic call-graph analysis, profiling, and taint analysis, among others, using a total of 614 lines of JavaScript code. Our selection of JavaScript as the implementation language for this component facilitated not only agile development of analysis algorithms but also precise semantics reflection (i.e., analyzing JavaScript within JavaScript).

VI. Evaluation

In this section, we present the experiments we conducted to evaluate JasFree. Our evaluation is guided by the following research questions.

RQ1: Efficiency. As JasFree is designed to perform program analysis and mutations, is JasFree efficient and practical in executing these tasks?

RQ2: Coverage and bug detection. Since JasFree is designed to enable grammar-free analysis and mutation. Is JasFree's approach effective in covering more execution paths and detecting real-world bugs? How does it compare to existing methods?

RQ3: Security impact. What is the actual impact of the bugs detected by JasFree?

All the experiments are performed on a server with one 12 physical Intel i7 Core (20 hyper thread) CPU and 128 GB of RAM running Ubuntu 24.04 LTS.

A. Datasets

We created three datasets to conduct the evaluation: 1) a micro-benchmark of JavaScript programs from test262; and 2) a real-world benchmark comprising two large JavaScript programs distributed with JerryScript; and 3) two embedded JavaScript VMs, JerryScript and MuJS.

Micro-benchmarks. To create the micro-benchmark, we selected 20 programs from the widely used JavaScript benchmark suite test262 [53], comprising a total of 1,378 lines of non-empty, non-comment JavaScript code. We then parsed these JavaScript programs into bytecode, generating binaries averaging 2,158 bytes for each program.

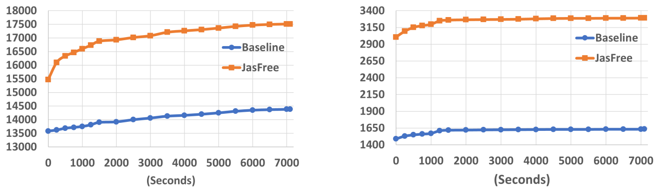
Real-world JavaScript programs. We select two real-world JavaScript programs, string-iterator.js and object-literal.js, from the official JerryScript distribution. Their binary sizes are 1.09 KB and 19.25 KB, respectively. Evaluating JasFree on these open-source projects demonstrates the effectiveness and performance on real-world JavaScript applications.

Embedded JavaScript VMs. We selected two embedded JavaScript VMs, JerryScript v2.4.0 [30] and MuJS v1.3.4 [12], which are specifically designed for the resource-constrained embedded environments. These two VMs comprise 204,856 and 18,501 lines of C code, respectively.

B. RQ1: Efficiency

To evaluate the efficiency of JasFree in addressing **RQ1**, we apply JasFree to both micro- and macro-benchmarks. We first measure the changes of code size and execution time before and after applying JasFree, then compare the results with those from the state-of-the-art JavaScript analysis framework Jalangi2 [19]. Following prior study [22], we repeat our experiment 10 rounds for each test case, to calculate an average.

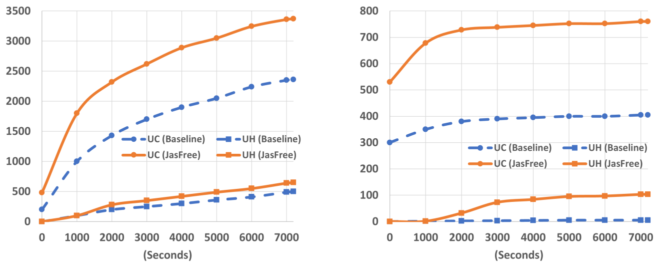
We present in Fig. 12 the changes of code size and execution time for micro-benchmarks and real-world programs when performing dynamic program analysis with all hooks instrumented. Our results show that JasFree introduces a code size increase between 163.68% and 418.01%, with an average of 290.84%, which is significantly lower than Jalangi2's average



(a) JASFREE's contribution to coverage is compared with the baseline (without the grammar-free mutation in JASFREE) for JerryScript.

(b) JASFREE's contribution to coverage is compared with the baseline (without the grammar-free mutation in JASFREE) for MuJS.

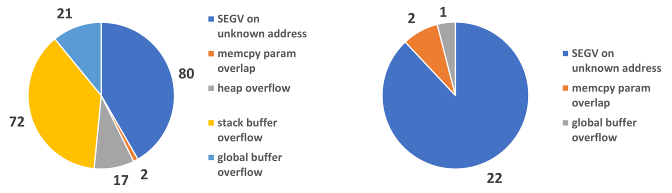
Fig. 13: Evaluating the contribution of JASFREE's grammar-free mutation to increased path coverage.



(a) JASFREE's contribution to bug detection compared with the baseline (without the grammar-free mutation in JASFREE) for JerryScript.

(b) JASFREE's contribution to bug detection compared with the baseline (without the grammar-free mutation in JASFREE) for MuJS.

Fig. 14: Evaluating the contribution of JASFREE's grammar-free mutation to bug detection. Bugs are classified as unique crashes (UC) and unique hangs (UH), according to how they manifest.



(a) Crashes detected by both JASFREE and the baseline.

(b) Crashes detected by JASFREE but missed by the baseline.

Fig. 15: Crash details and root causes.

code size increase of 660.38%. These results indicate that the code size increase introduced by JASFREE is acceptable compared with existing work.

JASFREE takes 7.51, 2.22, and 5.52 milliseconds, respectively, to process micro- and macro-benchmarks, introducing an average overhead of 3.1X, which is consistent with Jalangi2's overhead of approximately 3.2X. This experimental result demonstrates that JASFREE is efficient in practice for performing analysis.

TABLE III: The location of the bugs and the number of crashes triggered, detected by both JASFREE and the baseline.

Bug location	**#No.**	**Bug location**	**#No.**
vm.c: 1024	22	jerry-snapshot.c: 915	1
ecma-helpers.c: 212	8	ecma-function-object.c: 1100	1
ecma-helpers-collection.c: 93	4	ecma-literal-storage.c: 675	1
jerry-snapshot.c: 548	4	ecma-helpers-string.c: 152	1
lit-magic-strings.c: 130	3	jerry-snapshot.c: 673	1
ecma-literal-storage.c: 694	1		

TABLE IV: The location of the bugs and the number of crashes triggered, detected by JASFREE but missed by the baseline.

Bug location	**#No.**	**Bug location**	**#No.**
ecma-helpers-string.c: 152	5	vm.c: 433	1
vm.c: 1024	4	ecma-helpers.c: 1769	1
ecma-big-uint.c: 335	1	vm.c: 5292	1

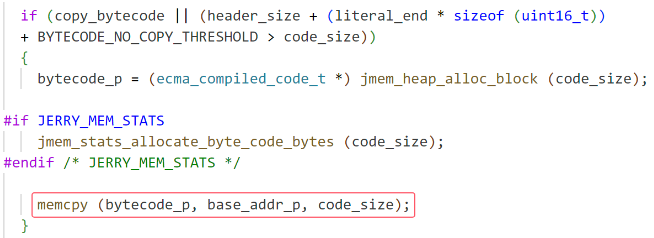
C. RQ2: Coverage and Bug Detection

To address **RQ2** and assess the efficacy of JASFREE in increasing path coverage and bug detection, we perform fuzzing test on the two VMs, JerryScript and MuJS, using mutation from JASFREE. Specifically, we first compile the two VMs with AddressSanitizer [54] to catch potential bugs, and then leverage AFL++ [55] to measure coverage. We set a timeout of 7,000 seconds, as we observe that the number of new paths explored tends to plateau after that duration.

We present experimental results demonstrating JASFREE's contribution to increased path coverage in Fig. 13. For the JerryScript VM (Fig. 13a), JASFREE successfully covers more than 17,500 distinct paths while the baseline (i.e., without the grammar-free mutation that JASFREE offers) only covers 14,480 distinct paths. As a result, JASFREE covers over 20.86% more distinct paths than the baseline. For the MuJS VM (Fig. 13b), JASFREE successfully covers more than 3,350 distinct paths whereas the baseline only covers 1,650 distinct paths, indicating an increase in path coverage of 103.03% compared to the baseline. These experimental results demonstrate that JASFREE's grammar-free mutation effectively increases path coverage.

We present the experimental results demonstrating JASFREE's contribution to bug detection in Fig. 14. Bugs are classified as unique crashed (UC) and unique hangs (UH), according to how they manifest. For the JerryScript VM (Fig. 14a), JASFREE successfully detects more than 4,450 bugs (3,450 UCs and 600 UHs), while the baseline detects only 2,900 bugs (2,400 UCs and 500 UHs). As a result, JASFREE detects over 1,550 bugs (a 53.45% increase) compared to the baseline. For the MuJS VM (Fig. 14b), JASFREE detects more than 850 bugs (750 UCs and 100 UHs) while the baseline detects only 410 bugs (400 UCs and 10 UHs). a result, JASFREE detects more than 440 bugs (a 107.31% increase) compared to the baseline. These experimental results demonstrate that JASFREE's grammar-free mutation effectively enhances bug detection.

Among all these bugs, some are more serious because they

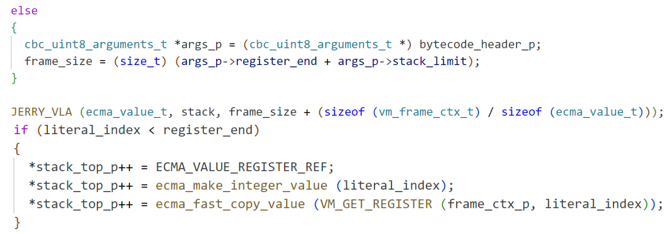


```
if (copy_bytecode || (header_size + (literal_end * sizeof (uint16_t))
+ BYTECODE_NO_COPY_THRESHOLD > code_size))
{
  bytecode_p = (ecma_compiled_code_t *) jmem_heap_alloc_block (code_size);

#if JERRY_MEM_STATS
  jmem_stats_allocate_byte_code_bytes (code_size);
#endif /* JERRY_MEM_STATS */

  memcpy (bytecode_p, base_addr_p, code_size);
}
```

Fig. 16: A heap overflow bug detected by JASFREE.



```
else
{
  cbc_uint8_arguments_t *args_p = (cbc_uint8_arguments_t *) bytecode_header_p;
  frame_size = (size_t) (args_p->register_end + args_p->stack_limit);
}

JERRY_VLA (ecma_value_t, stack, frame_size + (sizeof (vm_frame_ctx_t) / sizeof (ecma_value_t)));
 if (literal_index < register_end)
 {
   *stack_top_p++ = ECMA_VALUE_REGISTER_REF;
   *stack_top_p++ = ecma_make_integer_value (literal_index);
   *stack_top_p++ = ecma_fast_copy_value (VM_GET_REGISTER (frame_ctx_p, literal_index));
 }
```

Fig. 17: A stack buffer overflow bug detected by JASFREE.

lead to VM crashes. To further investigate serious bugs causing crashes, we conduct a manual inspection of the 192 crashes we detected. Our experimental results are presented in Fig. 15. We identified five root causes of the crashes (Fig. 15a): segmentation fault (SEGV), memory copy with overlapping addresses, heap overflows, stack buffer overflows, and global buffer overflows. Segmentation faults account for the majority, with 80 occurrences (41.67%), followed by 72 stack buffer overflows (37.5%). We further investigate JASFREE's contribution to crash detection, and present the crashes uniquely detected by JASFREE in Fig. 15b. JASFREE successfully detected 25 unique crashes across three categories, all missed by the baseline. These results demonstrate that JASFREE is highly effective in detecting unique crashes.

Furthermore, to understand the root causes of the crashes, we conducted a manual inspection of the relevant source code. The source code locations triggering these bugs are presented in Tables III and IV. We have reported these bugs to the developers, ensuring no ethical concerns arise.

D. RQ3: Security Impact

To demonstrate JASFREE's bug detection capabilities in more detail and assess its practical security impact, we present a detailed analysis of two bugs detected by JASFREE as case studies. The two bugs are classified as heap overflow and stack buffer overflows, respectively. A complete list of bugs JASFREE successfully detected is included in our reproduction package.

Heap overflow. We present in Fig. 16 a heap overflow bug detected by JASFREE from JerryScript. This code snippet copies code_size bytes from the source address base_addr_p into the destination address bytecode_p by invoking the C library function memcpy. Unfortunately, this code does not validate the code_size argument. Consequently, for large enough code_size, the function memcpy writes beyond the end of destination buffer bytecode_p, causing a heap overflow and overwriting the contiguous memory.

Worse yet, this code snippet does not validate the source address base_addr_p. Consequently, the function memcpy will access the memory beyond the end of base_addr_p for large enough code_size, potentially leading to information leakage if sensitive data resides in that memory.

Stack buffer overflow. We present in Fig. 17 a stack buffer overflow bug JASFREE detected. The variable frame_size is utilized to represent the size of the stack frame, which is constrained by the maximum stack size stack_limit. Additionally, the stack array stack is sized based on the sum of frame_size and the number of ecma_value_t elements in the vm_frame_ctx_t structure.

Unfortunately, if stack_limit is too small, it fails to allocate enough spaces to accommodate the required stack size. As a result, the pointer stack_top_p, which points to the top of the stack buffer exceeds the buffer's boundary, leading to stack overflows. Such overflows can hijack control flows by overwriting return addresses on the stack.

VII. DISCUSSION

In this section, we discuss the limitations of our approach in its current implementation and outline our plans to address them in future work.

Program analysis. The program analysis capabilities of JASFREE supports the instrumentation of all instructions in JASBYTE, facilitating advanced dynamic analysis with minimal effort. In the future, we plan to implement additional analyses based on JASFREE, including, but not limited to, memory access tracking, debugging information extraction, security vulnerability detection, and resource usage monitoring. These analyses will help optimize code performance, improve code quality, and enhance the overall security and reliability of embedded JavaScript applications.

More effective fuzzing. JASFREE supports syntax-directed mutation of JASBYTE, enabling effective greybox fuzzing of embedded JavaScript VMs. While our evaluation results demonstrate that our grammar-free approach significantly enhances fuzzing by increasing path coverage and bug detection, our focus in this work is not on fuzzing strategies but on how grammar-free mutation can benefit fuzzing. In the future, based on our experiences gained from the design of JASFREE, we plan to explore more effective fuzzing approach for embedded JavaScript VMs. In the short term, we plan to investigate a graph-guided fuzzing approach [56] in which the generation of new JASBYTE instructions is guided by a state graph tracking the depth of the operand stack. Using this approach, we will generate complex but valid JASBYTE instructions to effectively fuzz test the VMs.

Other vulnerabilities. In this work, we focus on the detection of security vulnerabilities, and our evaluation results demonstrate that our approach is effective in achieving this goal. While security vulnerabilities represent serious flaws in programs, there are other issues, such as functionality or correctness problems, that developers often struggle to address. In the future, we plan to extend JASFREE with capabilities to tackle these issues. Specifically, we plan to test embedded JavaScript VMs for correctness by leveraging oracle-based approaches [22] or differential testing methods [51].

VIII. Related Work

There has been significant research on JavaScript security and program analysis. However, the work in this paper represents a novel contribution to these fields.

JavaScript bytecode. JavaScript bytecode offers improved performance, enhanced security, and cross-platform development capabilities [57] to JavaScript applications. Additionally, bytecode can be easily obfuscated and encrypted, enhancing application security [58]. Several VMs have introduced their own bytecodes. V8 [6] is Google's high-performance JavaScript and WebAssembly engine, powering popular applications like Chrome, Node.js, and Android. SpiderMonkey [7], a JavaScript engine from Mozilla, serves as the core component of the Firefox web browser and other Mozilla products. JerryScript [9] is an embedded JavaScript engine, designed for resource-constrained devices and Internet of Things (IoT) applications. QuickJS [10] is a small, fast, and portable JavaScript engine that can be embedded into a variety of applications and systems.

However, a limitation of these VMs is that their bytecodes are heterogeneous, which complicates holistic JavaScript bytecode program analysis and mutations.

Program analysis of JavaScript. There are numerous program analysis tools for JavaScript that focus on different aspects, such as security [59], testing [60], type inconsistency [61], and race detection [62] [63]. Jalangi [49] is a dynamic symbolic execution and program instrumentation framework for JavaScript, while Jalangi2 [19] is an extension and reimplementation of the Jalangi framework, offering enhanced capabilities for dynamic analysis and instrumentation of JavaScript programs.

However, despite the strengths of tools like Jalangi and Jalangi2, they operate at source-level, requiring JavaScript source programs for instrumentation and overlooking VM bytecode. As a result, these tools cannot directly analyze JavaScript bytecode.

Fuzzing. There has been substantial research on fuzzing and the development of relevant tools. AFL [64] is a widely used fuzzing tool that operates by continuously generating and executing mutated input data, monitoring changes in program execution to explore new code paths. AFL++ [55], an enhanced version of AFL, offers additional features and and supports a variety of fuzzing types. OSS-Fuzz [65], supported by Google, is an open-source software fuzzing project offering an automated platform that integrates multiple fuzzing tools. Many fuzzers [22] [21] have been proposed to fuzz JavaScript engines such as V8.

However, a key distinction between these research efforts and this work is that in this work we propose a grammar-free approach to analyze and mutate JavaScript bytecode tailored for embedded JavaScript VMs. While our new IR, JASBYTE, facilitates more effective fuzzing mutations, leading to increased path coverage and bug detections, our approach is not limited to fuzzing. As our evaluation shows, it can be applied to broader areas, including static and dynamic program analysis.

IX. Conclusion

This paper presents JASFREE, the first grammar-free, universal program analysis approach for JavaScript bytecode. We begin by designing a intermediate representation JASBYTE to represent JavaScript bytecode. We then develop a universal translation framework equipped with a set of APIs to transform JavaScript bytecode into JASBYTE. Next, we design a suite of algorithms and mutations that enable effective program analysis and fuzzing. Our results show that JASFREE effectively facilitates the construction of diverse static and dynamic analysis and enhances the mutations, leading to the detection of 25 new vulnerabilities across three categories, outperforming state-of-the-art methods.

References

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

[2] F. L. Oliveira and J. C. Mattos, "State-of-the-art javascript language for internet of things," in *Anais Estendidos do IX Simpósio Brasileiro de Engenharia de Sistemas Computacionais*. SBC, 2019, pp. 149–154.

[3] K. Cao, Y. Liu, G. Meng, and Q. Sun, "An overview on edge computing research," *IEEE access*, vol. 8, pp. 85 714–85 728, 2020.

[4] L. Sciullo, L. Gigli, F. Montori, A. Trotta, and M. Di Felice, "A survey on the web of things," *IEEE Access*, vol. 10, pp. 47 570–47 596, 2022.

[5] "Javascriptcore — apple developer documentation," https://developer.apple.com/documentation/javascriptcore.

[6] "V8 javascript engine," https://v8.dev/.

[7] "Spidermonkey javascript/webassembly engine," https://spidermonkey.dev/.

[8] "Chakracore," https://github.com/chakra-core/ChakraCore.

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

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

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

[12] "Mujs," https://github.com/ccxvii/mujs.

[13] "Cve-2023-20198," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-20198.

[14] "Cve-2023-20198 detail - nvd," https://nvd.nist.gov/vuln/detail/CVE-2023-20198.

[15] "Cve-2020-13991 detail - nvd," https://nvd.nist.gov/vuln/detail/CVE-2020-13991.

[16] "Find and fix problems in your javascript code - eslint - pluggable javascript linter," https://eslint.org/.

[17] A. Hidayat, "Esprima: Ecmascript parsing infrastructure for multipurpose analysis," 2017.

[18] M. Bolin, *Closure: The definitive guide: Google tools to add power to your JavaScript*. " O'Reilly Media, Inc.", 2010.

[19] "Jalangi2," https://github.com/Samsung/jalangi2.

[20] B. Zeng, "Static analysis on binary code," Tech. rep. Tech-report, 2012 (cit. on pp. 17, 22, 23), Tech. Rep., 2012.

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

[22] J. Wang, Z. Zhang, S. Liu, X. Du, and J. Chen, "{FuzzJIT}:{Oracle-Enhanced} fuzzing for {JavaScript} engine {JIT} compiler," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1865–1882.

[23] G. Williams, "Espruino," 2012.

[24] L. Xu, F. Sun, and Z. Su, "Constructing precise control flow graphs from binaries," *University of California, Davis, Tech. Rep*, pp. 14–23, 2009.

[25] U. Khedker, A. Sanyal, and B. Sathe, *Data flow analysis: theory and practice*. CRC Press, 2017.

[26] M. Gorbovitski, Y. A. Liu, S. D. Stoller, T. Rothamel, and T. K. Tekle, "Alias analysis for optimization of dynamic languages," in *Proceedings of the 6th Symposium on Dynamic Languages*, 2010, pp. 27–42.

[27] L. Gong, M. Pradel, and K. Sen, "Jitprof: Pinpointing jit-unfriendly javascript code," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 357–368.

[28] X. Ren, M. Ho, J. Ming, Y. Lei, and L. Li, "Unleashing the hidden power of compiler optimization on binary code difference: an empirical study," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 142–157.

[29] D. Sin and D. Shin, "Performance and resource analysis on the javascript runtime for iot devices," in *Computational Science and Its Applications–ICCSA 2016: 16th International Conference, Beijing, China, July 4-7, 2016, Proceedings, Part I 16*. Springer, 2016, pp. 602–609.

[30] E. Gavrin, S.-J. Lee, R. Ayrapetyan, and A. Shitov, "Ultra lightweight javascript engine for internet of things," in *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, 2015, pp. 19–20.

[31] M. Kim, H.-J. Jeong, and S.-M. Moon, "Small footprint javascript engine," *Components and Services for IoT Platforms: Paving the Way for IoT Standards*, pp. 103–116, 2017.

[32] K. Grunert, "Overview of javascript engines for resource-constrained microcontrollers," in *2020 5th International Conference on Smart and Sustainable Technologies (SpliTech)*. IEEE, 2020, pp. 1–7.

[33] A. Møller and M. I. Schwartzbach, "Static program analysis," *Notes. Feb*, 2012.

[34] P. Thomson, "Static analysis: An introduction: The fundamental challenge of software engineering is one of complexity." *Queue*, vol. 19, no. 4, pp. 29–41, 2021.

[35] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis*, 2003, pp. 24–27.

[36] T. Ye, L. Zhang, L. Wang, and X. Li, "An empirical study on detecting and fixing buffer overflow bugs," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 91–101.

[37] M. Nasereddin, A. ALKhamaiseh, M. Qasaimeh, and R. Al-Qassas, "A systematic review of detection and prevention techniques of sql injection attacks," *Information Security Journal: A Global Perspective*, vol. 32, no. 4, pp. 252–265, 2023.

[38] G. E. Rodríguez, J. G. Torres, P. Flores, and D. E. Benavides, "Cross-site scripting (xss) attacks and mitigation: A survey," *Computer Networks*, vol. 166, p. 106960, 2020.

[39] T. Ball, "The concept of dynamic analysis," *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6, pp. 216–234, 1999.

[40] T. M. Chilimbi and V. Ganapathy, "Heapmd: Identifying heap-based bugs using anomaly detection," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: Association for Computing Machinery, Oct. 2006, pp. 219–228.

[41] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," in *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: Association for Computing Machinery, Mar. 2010, pp. 167–178.

[42] J. Park, B. Choi, and S. Jang, "Dynamic analysis method for concurrency bugs in multi-process/multi-thread environments," *International Journal of Parallel Programming*, vol. 48, no. 6, pp. 1032–1060, Dec. 2020.

[43] 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.

[44] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

[45] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: a survey for roadmap," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, pp. 1–36, 2022.

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

[47] S. Yan, C. Wu, H. Li, W. Shao, and C. Jia, "Pathafl: Path-coverage assisted fuzzing," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020, pp. 598–609.

[48] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "The fuzzing book," 2019.

[49] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for javascript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 488–498.

[50] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java virtual machine specification*. Addison-wesley, 2013.

[51] X. Liu, X. Li, R. Prajapati, and D. Wu, "Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 1044–1051.

[52] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1980–1997, 2019.

[53] E. TC39, "Test262 test suite," 2017.

[54] "Asan: address sanitizer," zer. https://github.com/google/sanitizers/ wiki/AddressSanitizer.

[55] 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

[56] J. Fu, J. Liang, Z. Wu, M. Wang, and Y. Jiang, "Griffin: Grammar-free dbms fuzzing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[57] J. Oh, J.-w. Kwon, H. Park, and S.-M. Moon, "Migration of web applications with seamless execution," in *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2015, pp. 173–185.

[58] A. Radovici, R. Cristian, and R. ŞERBAN, "A survey of iot security threats and solutions," in *2018 17th RoEduNet conference: networking in education and research (RoEduNet)*. IEEE, 2018, pp. 1–5.

[59] D. Yu, A. Chander, N. Islam, and I. Serikov, "Javascript instrumentation for browser security," *ACM SIGPLAN Notices*, vol. 42, no. 1, pp. 237–249, 2007.

[60] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip, "A framework for automated testing of javascript web applications," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 571–580.

[61] M. Pradel, P. Schuh, and K. Sen, "Typedevil: Dynamic type inconsistency analysis for javascript," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 314–324.

[62] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby, "Race detection for web applications," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 251–262, 2012.

[63] E. Mutlu, S. Tasiran, and B. Livshits, "Detecting javascript races that matter," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 381–392.

[64] "American fuzzy lop," https://github.com/google/AFL, May 2023.

[65] A. Arya, O. Chang, J. Metzman, K. Serebryany, and D. Liu, "OSS-Fuzz." [Online]. Available: https://github.com/google/oss-fuzz