JASLOAD: Dynamically Analyzing JavaScript Bytecode via A Load-Time Instrumentation Approach

Hao Jiang, and Baojian Hua*
School of Software Engineering, University of Science and Technology of China
Suzhou Institute for Advanced Research, University of Science and Technology of China
jh7@mail.ustc.edu.cn, bjhua@ustc.edu.cn
*Corresponding author

Abstract—JavaScript is increasingly being deployed as binaries in security-critical embedded domains, such as IoT devices, edge computing, and intelligent vehicle platforms. This widespread adoption highlights the importance of dynamic analysis to ensure the security of JavaScript applications, particularly at the bytecode level. However, existing dynamic analysis techniques often rely on static instrumentation, which significantly increases of the executable size. This, in turn, leads to higher memory consumption and performance degradation—issues that are especially problematic in resource-constrained environments.

In this paper, we present the first dynamic analysis approach for JavaScript bytecode that leverages load-time instrumentation to address this issue. We begin by designing a custom intermediate representation (IR) for JavaScript bytecode constructed at load time. We then develop a set of low-level hooks that are triggered at key points in the program execution flow. In addition, we introduce a set of flexible APIs to support customized instrumentation and dynamic analyses. We implement a software prototype, JASLOAD, and conduct extensive evaluations. Evaluation results demonstrate that our approach significantly enhances the efficiency and effectiveness of dynamic analysis on resource-constrained devices. By combining JASLOAD's bytecode loading/unloading with adaptive instrumentation, we reduce runtime overhead by up to 70.53% and decrease code size expansion under full instrumentation from 603.68% to 144.25%, compared to prior JavaScript bytecode analysis methods.

Keywords–JavaScript Bytecode; Load-ime Instrumentation; Dynamic Analysis

1. INTRODUCTION

JavaScript, a foundational language for modern web development, is increasingly being deployed in security-critical and resource-constrained environments, such as Internet of Things (IoT) devices [1], wearable technologies [2], and edge computing systems. These embedded deployments introduce unique challenges and risks for JavaScript, including strict performance constraints and emerging security vulnerabilities, all under the imperative of optimized resource management. For example, the recent VPNFilter malware campaign compromised over 500,000 devices across 54 countries—including routers and network-attached storage—through sophisticated attacks [3]. The inherent dynamism and runtime limitations of JavaScript in such environments underscore the urgent need for comprehensive dynamic analysis to ensure robustness and security. This urgency highlights the pressing demand for advanced analytical frameworks capable of delivering real-time behavioral insights into JavaScript execution under constrained resource conditions.

Dynamic analysis has demonstrated significant potential for analyzing JavaScript programs. Early tools such as Chrome DevTools [4] focus primarily on browser environments, leveraging sophisticated instrumentation techniques for runtime inspection. Subsequent advancements, including Jalangi [5] and Jalangi2 [6], introduced static instrumentation methods that insert analysis code at compile time to enable detailed execution monitoring. Unfortunately, despite their effectiveness, these approaches rely heavily on static instrumentation, which substantially increases the size of the resulting executables. This overhead poses serious implementation challenges, particularly in resource-constrained embedded environments. For example, JerryScript [7], a widely used JavaScript engine for embedded systems, imposes strict hardware constraints—such as 64KB of RAM and 200KB of flash memory—that significantly exacerbate the limitations of static instrumentation techniques.

In this paper, recognizing the critical need for effective dynamic analysis of JavaScript bytecode, we propose a load-time instrumentation approach to overcome the limitations of static instrumentation. Unlike static methods, our approach avoids modifying the original program code, thereby preserving memory efficiency and optimizing runtime performance. Crucially, our load-time instrumentation enables the analysis of programs as they are loaded—including those dynamically loaded after initialization—while maintaining close alignment with the runtime environments for improved instrumentation precision. Additionally, our approach supports adaptive instrumentation by integrating runtime context, allowing for dynamic updates to configuration and analytical logic during execution. Although load-time instrumentation has been successfully adopted in Java ecosystems—evidenced by frameworks such as Javassist [8], Byteman [9], and AspectJ [10]—no comparable solution currently exists for JavaScript bytecode implementations.

We implement our load-time instrumentation-based dynamic analysis approach in a software prototype, JASLOAD, de-

TABLE I: Overview of existing dynamic analysis approaches and JASLOAD.

	Pin [11]	Valgrind [12]	DiSL [13]	RoadRunner [14]	NVBit [15]	Jalangi [5]	Jalangi2 [6]	Wasabi [16]	**JASLOAD**
Platform	x86-64	x86-64	JVM	JVM	CUDA	JavaScript/ SpiderMonkey	JavaScript/ V8 [39]	WebAssembly	JavaScript
Level	binaries	binaries	bytecode	bytecode	assembly	sources	sources	binaries	binaries
Analysis Language	C/C++	C	Java	Java	C++	JavaScript	JavaScript	JavaScript	JavaScript
Technique	instrumentation + callbacks/hooks	low-level instrumentation	aspect-oriented	event stream	callbacks/ hooks	callbacks/ hooks	callbacks/ hooks	callbacks/ hooks	callbacks/ hooks

signed to analyze JavaScript bytecode in resource-constrained environments. The prototype comprises three core components, each addressing distinct technical challenges. First, we design LITEBYTE, a specialized intermediate representation (IR) tailored for JavaScript bytecode at load-time. LITEBYTE provides a structured abstraction that facilitates streamlined analysis and manipulation. It resolves instruction set discrepancies by offering a holistic operational IR that fully supports dynamic analysis. Second, we implement hierarchical classification scheme for compact bytecode instructions, organizing them into three levels. This is combined with strategically embedded low-level execution hooks at critical execution points. Together, these techniques enable precise analysis targeting, fine-grained runtime state monitoring, and controlled performance overhead—key for deployment in embedded systems. Third, we develop a flexible API suite for dynamically customizable instrumentation policies and analysis workflows. These APIs are supported by a dedicated data structure that maintains instruction address consistently during instrumentation— effectively addressing critical challenges of target address modification in jump instructions.

Compared to existing approaches (Table I), JASLOAD offers three technical advantages: 1) Precision and accuracy: JASLOAD operates directly at the JavaScript bytecode level, enabling fine-grained behavior capture during the program loading phase. This surpasses the granularity of source-level approaches such as Jalangi2 [6]) and bytecode-level systems like RoadRunner [14]), particularly in resource-constrained environments. 2) Flexibility and dynamism: JASLOAD employs a callback- and hook-based instrumentation architecture that facilitates detailed runtime monitoring and dynamic context collection. This enables adaptive decision-making at load time, unlike static tools such as DiSL [13] and Valgrind [12]), which lack runtime adaptability. 3) Resource optimization: JASLOAD implements selective code injection that reduces redundancy and minimize both memory footprint and performance overhead. This makes it more suitable for resource-limited devices compared to traditional static instrumentation frameworks such as Pin [11] and NVBit [15]).

We evaluate JASLOAD across five dimensions—usability, efficiency, code size, runtime overhead, and effectiveness—using micro-benchmarks and real-world JerryScript applications. Our results demonstrate that JASLOAD achieves millisecond-scale binary processing while fully preserving program behavior. In terms of code size, adaptive instrumentation generally incurs less than 10% increase, though some edge cases, such as full monitoring of jerry/string-iterators, result in a maximum growth of 959.37%. Runtime overhead varies based on the type and intensity of instrumentation: it ranges from 1.5x for lightweight hooks (e.g., `create_obj`, `unary`, and `pop`) to 6.5x for more intensive hooks (e.g., the `call`). Furthermore, we introduce two key optimizations that significantly improve efficiency under constrained conditions: 1) an adaptive hook mechanism: by prioritizing critical operations (e.g., `assign_obj_prop`, `call_pre`), we reduce JerryScript runtime costs by up to 70.53%; and 2) selective code injection: this optimization bring micro-benchmark code inflation down to 144.25%, compared to 603.68% observed with conventional static instrumentation approaches—maintaining analysis fidelity while significantly reducing resource demands.

To summarize, our work makes the following contributions:

- We propose the first dynamic analysis approach for JavaScript binaries that leverages a load-time instrumentation, addressing critical limitations of existing static techniques.
- We design and implement a software prototype JASLOAD, to validate the practicality and effectiveness of our approach.
- We conduct extensive experiments using micro-benchmarks and real-world JavaScript applications to demonstrate the effectiveness, performance, and adaptability of JASLOAD in resource-constrained environments.

The rest of this paper is organized as follows: Section 2 presents the background for this study. Sections 3 and 4 present our approach. Section 5 presents evaluation results. Section 6 discusses limitations and directions for future work. Section 7 discusses the related work, and Section 8 concludes.

2. BACKGROUND

To be self-contained, in this section, we present the background of JavaScript bytecode, embedded JavaScript virtual machine, load-time instrumentation, and dynamic program analysis.

JavaScript bytecode. JavaScript bytecode is a compact, efficient binary representation compiled from JavaScript sources. For instance, Fig. 1 illustrates both the JavaScript source code (left) and the corresponding JavaScript bytecode (middle) for a Fibonacci function. Compared to source code, JavaScript bytecode offers enhanced performance [17], improved security [18], and cross-platform development capabilities [19]. These advantages make it especially suitable for resource-constrained environments like embedded devices.

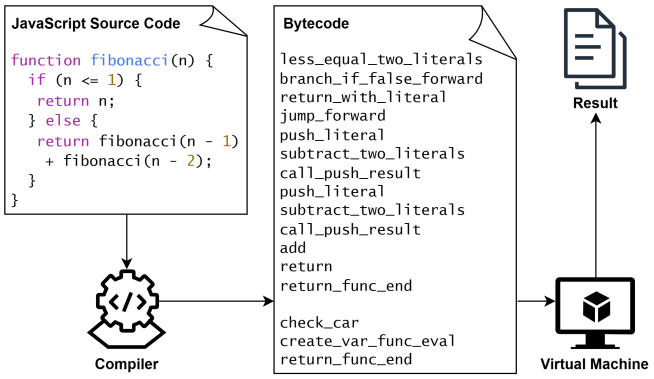

Figure 1: The JavaScript source is first compiled to JavaScript bytecode, and then executed by JavaScript virtual machines. For better understanding, we present the JavaScript bytecode in assembly form rather than binary form.

Embedded JavaScript virtual machines. Embedded JavaScript VMs are specialized execution engines designed to run JavaScript bytecode efficiently in resource-limited environments such as IoT [20]. Real-world embedded JavaScript VMs (e.g., JerryScript [7], Duktape [21] [22], QuickJS [23], and MuJS [24]) employ advanced techniques like JIT compilation and memory compaction to boost speed and minimize resource consumption. As embedded/IoT devices proliferate, ensuring the security of these VMs without sacrificing performance has become critically urgent.

Load-time instrumentation. Load-time instrumentation [15] dynamically inserts code into programs during loading, enabling real-time monitoring, modification, and behavioral enhancement. It is valuable for dynamic analysis, performance tuning, and security checks for embedded systems by offering several key benefits: preserving original program integrity while injecting monitoring code; enabling real-time optimization based on device state; and providing non-intrusive functionality to diverse applications without major redevelopment.

Dynamic program analysis. Dynamic program analysis [25] examines program behavior and performance during execution. By collecting runtime data, it enables bug detection [26] [27] [28], security analysis, and performance profiling [29]. Unlike static analysis [30] [31], it effectively detects runtime errors, performance issues, and unexpected behaviors—particularly for complex, dynamic structures like JavaScript binary programs.

3. Approach

In this section, we present our approach in designing JasLoad. We first describe our design goals (§ 3.1), followed by an overview of its workflow (§ 3.2). We then discuss bytecode loader/unloader (§ 3.3), language model (§ 3.4), bytecode instrumentation (§ 3.5), and dynamic program analysis (§ 3.6).

3.1 Design Goals

JasLoad is designed with three main goals: high accuracy/completeness, excellent flexibility/adaptability, and optimal compactness. First, it must accurately and comprehensively monitor application runtime behavior, capturing detailed traces and critical events. Second, it should be highly flexible to support various instrumentation and analysis tasks, allowing customization of instrumentation points and monitoring code. Third, it must minimize bytecode size increase to meet the strict storage limits of embedded systems, preserving the lightweight nature of the application.

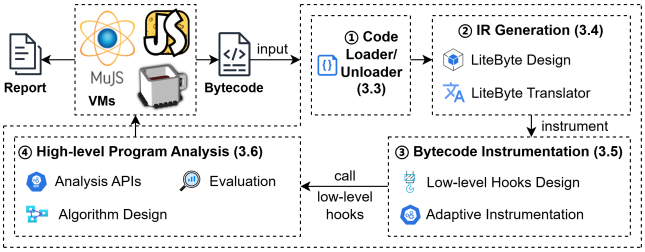

Figure 2: An overview of JasLoad's workflow.

3.2 Overview

Guided by these goals, we present the overall workflow of JasLoad in Fig. 2, comprising four core phases. First, the code loader/unloader (①) dynamically loads JavaScript bytecode while computing stack/register needs for the instrumented version. Next, IR generation (②) creates LiteByte, a uniform intermediate representation abstracting JavaScript bytecode, including its translation and optimization. Then, the bytecode instrumentation (③) embeds hierarchical low-level hooks into LiteByte. These hooks invoke adaptive instrumentation and analysis algorithms for fine-grained behavior tracking without disrupting the original environment. Finally, high-level program analysis (④) executes JavaScript-based analysis APIs at runtime. Leveraging the instrumented hooks, these APIs capture low-level behaviors, generate execution reports, and produce results—all without altering the original program's semantics.

3.3 Bytecode Loader/Unloader

The bytecode loader/unloader profiles the host JavaScript VM's runtime hardware capabilities, including CPU cores, thread concurrency, stack/heap memory, and storage capacity, among others. Using real-time system metadata, the dynamic adaptation module evaluates whether full-scale bytecode instrumentation can be deployed without exceeding resource thresholds.

If hardware limitations prevent full instrumentation (e.g., insufficient IR memory or thread contention risks), the module activates adaptive instrumentation. This selectively instruments critical bytecode segments, including hot functions and memory-sensitive operations, while bypassing non-essential regions, minimizing runtime overhead and analysis latency. Instrumentation granularity is heuristically balanced for code coverage, performance impact, and hardware capabilities.

Strict memory alignment and ABI compliance are enforced during code injection to preserve original bytecode integrity.

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

Figure 3: Representative abstract syntax of LITEBYTE.

The module significantly reduces instrumentation costs on resource-constrained devices (e.g., IoT edge nodes) while maintaining actionable insights for debugging/profiling/security. Inspired by NVBit's dynamic code management [15], this design extends adaptability to heterogeneous JavaScript environments, enabling hardware-tailored lightweight instrumentation.

3.4 IR Generation

LITEBYTE design. Defining a formal syntax is critical for establishing a robust intermediate representation. We design LITEBYTE, a universal intermediate language defined via a context-free grammar in Fig. 3, aligning with embedded JavaScript VM bytecode while enabling high-level analysis. A LITEBYTE *module* contains *functions* with signatures $type^{*}_{val} \rightarrow type^{*}_{val}$ (supporting multiple parameters/returns). Its instruction set includes operations (unary/binary, memory, stack), control flow constructs, and function calls. The language abstracts key JavaScript bytecode features: stack-based execution (e.g., `add` pops operands, pushes results) [32] and structured control flow.

LITEBYTE translator. The LITEBYTE translator processes JavaScript bytecode binaries to construct the intermediate representation. Given the direct alignment between LITEBYTE's syntax in Fig. 3 and conventional bytecode, the conversion largely involves one-to-one instruction translation. It extracts auxiliary data including constant tables and debug symbols to ensure comprehensive, semantically equivalent IR generation, enabling subsequent analysis and optimization.

3.5 Bytecode Instrumentation

The bytecode instrumentation module processes LITEBYTE IR as input, to dynamically inject instrumentation code via strategies orchestrated by the bytecode loader/unloader. Guided by real-time hardware constraints and execution context, it selectively instruments critical segments (e.g., performance-sensitive operations) while bypassing non-essential regions to minimize overhead. Using strict memory alignment and ABI-compliant injection, it preserves original bytecode structures while embedding hierarchical low-level hooks. The output is an instrumented AST augmented with monitoring logic, generating runtime telemetry (execution traces, resource metrics, behavioral signatures) for debugging/optimization/security analysis. This maintains deterministic execution semantics in resource-constrained JavaScript environments.

TABLE II: Hierarchy of low-level hooks.

First level	Second level	Third level
start	start	start
assign	create	create_var, create_obj
	assign_var	assign_var_ss, assign_var_si, assign_var_ii
	assign_obj_prop	assign_obj_prop_si, assign_obj_prop_sss, assign_obj_prop_ssi, assign_obj_prop_sii
unary	unary	unary_s, unary_i
	unary_lvalue	unary_lvalue_i, unary_lvalue_ss
binary	binary	binary_ss, binary_si, binary_ii
stack	push	push_i, push_ii, push_iii, push_iv, push_v, push_vi
	push_obj_prop	push_obj_prop_i, push_obj_prop_ss, push_obj_prop_si, push_obj_prop_ii
	pop	pop_s, pop_si
call	call	call_pre, call_post
control flow	branch	br_if, br_equal
	begin	func_begin, block_begin
	end	func_end, block_end
return	return	return_s, return_i, return_0

Low-level hooks design. To avoid impractical per-instruction hooks, we utilize a hierarchical design categorizing JavaScript bytecode into *8* major groups and *42* subcategories (Table II). This three-tiered hierarchy groups instructions by operational semantics and parameter sources. For example, the binary category includes subcategories like `binary_ss` (stack-source operands), and `assign_obj_prop_ssi` denotes an object property assignment using two stack-based operands and one index-based operand. Hook names explicitly indicate operand sources (*s*: stack, *i*: index, *v*: value) and quantities. This approach groups semantically similar instructions, enables shared hook functions, reduces code redundancy, and minimizes instrumentation overhead while maintaining granularity and system maintainability.

Adaptive instrumentation. We dynamically inject analysis hooks into JavaScript bytecode to track instructions. As shown in Table III, each instrumentation rule maps original bytecode (e.g., `new`, `assign_var`) to instrumented counterparts via low-level hooks. Implemented in JavaScript and compiled to bytecode, these hooks capture behaviors like object creation or arithmetic operations. For example, a `binary` hook saves

TABLE III: Instrumentation rules of JavaScript bytecode instructions.

Categories of hooks	Original instructions	Instrumented instructions	Descriptions
create_obj	*instruction_create_obj*	*instruction_create_obj*	original instruction (e.g., `new`)
		`pop reg_n`	save obj on stack to register
		`push reg_n`	restore obj to stack
		`push hook_func`	push hook name onto stack
		`push reg_n`	push obj required by the hook onto stack
		`call hook_func`	instrumentation hook
assign_var	`push value`	`push value`	push assignment value onto stack
		`pop reg_n`	save assignment value to register
		`push reg_n`	restore assignment value to stack
		`push hook_func`	push hook name onto stack
		`push index, reg_n`	push index of var and assignment value required onto stack
		`call hook_func`	instrumentation hook
	instruction_assign_var	*instruction_assign_var*	original instruction (e.g., `assign_var index`)
assign_obj_prop	`push value`	`push value`	push assignment value onto stack
		`pop reg_n`	save assignment value to register
		`push reg_n`	restore assignment value to stack
		`push hook_func`	push hook name onto stack
		`push this, index, reg_n`	push obj, index of prop and assignment value onto stack
		`call hook_func`	instrumentation hook
	instruction_assign_obj_prop	*instruction_assign_obj_prop*	original instruction (e.g., `assign_this_prop index`)
binary	`push left, right`	`push left, right`	push operands for binary operation onto stack
		`pop reg_n, reg_{n + 1}`	save operands on stack to register
		`push hook_func`	push hook name onto stack
		`push operation`	push operation code required by the hook onto stack
		`push reg_{n + 1}, reg_n`	push operands required by the hook onto stack
		`push reg_{n + 1}, reg_n`	restore operands to stack
	instruction_binary	*instruction_binary*	original instruction
		`pop reg_n`	save result of binary operation on stack to register
		`push reg_n`	push result required by the hook onto stack
		`call hook_func`	instrumentation hook
		`push reg_n`	restore result to stack
push	*instruction_push*	*instruction_push*	original instruction (e.g., `push value`)
		`push hook_func`	push hook name onto stack
		instruction_push	push value required by the hook onto stack
		`call hook_func`	instrumentation hook
call_pre	`push callee_func`	`push callee_func`	push callee function onto stack
	`push args`	`push args`	push arguments required by callee function onto stack
		`pop reg_n...`	
		`pop reg_{n + args_num}`	save parameters on stack to register
		`push reg_{n + args_num}...`	
		`push reg_n`	restore stack
		`push hook_func`	push hook name onto stack
		`push reg_{n + args_num}...`	
		`push reg_n`	push the parameters required by the hook onto the stack
		`call hook_func`	instrumentation hook
	instruction_call	*instruction_call*	original instruction
br_if	`push condition`	`push condition`	push condition onto stack
		`push hook_func`	push hook name onto stack
		`pop reg_n`	save condition on stack to register
		`push reg_n`	push condition required by the hook onto stack
		`push true_branch`	push true branch id onto stack
		`push false_branch`	push false branch id onto stack
		`call_push_result hook_func`	call hook and push condition from hook
	instruction_branch_if	*instruction_branch_if*	original instruction
block_begin		`push hook_func`	push hook name onto stack
		`push block_id`	push block id required by the hook onto stack
		`call hook_func`	instrumentation hook
	instruction_block_start	*instruction_block_start*	original instruction
return	`push return_val`	`push return_val`	push return value onto stack
		`pop reg_n`	save return value on stack to register
		`push reg_n`	restore return value to stack
		`push hook_func`	push hook name onto stack
		`push func_id`	push function id required by the hook onto stack
		`push reg_n`	push return value required by the hook onto stack
		`call hook_func`	instrumentation hook
	instruction_return	*instruction_return*	original instruction

operands, pushes metadata, executes the original instruction, records results, and invokes the hook—ensuring detailed logging without compromising integrity.

To minimize bloat and overhead, we utilize adaptive instrumentation in the bytecode loader/unloader. The Loader/Unloader profiles runtime resources (memory, concurrency) to determine feasibility of full instrumentation. If thresholds are exceeded, it activates selective hooking: enabling critical hooks (e.g., `assign_var`, `binary`) while bypassing non-essential ones (e.g., `block_begin`). For memory-constrained IoT nodes, this might mean activating taint-tracking hooks but disabling `call_post` to avoid IR overhead. Memory alignment and ABI compliance are strictly enforced during injection.

A heuristic model further refines granularity by prioritizing "hot" functions or memory-sensitive operations, dynamically scaling with fluctuating resources. For instance, if heap memory depletes mid-execution, non-critical hooks (e.g., `push` instrumentation) can be unloaded to reallocate resources for essential analyses. By balancing granularity with hardware constraints, JASLOAD delivers efficient, context-aware analysis for resource-limited environments.

Handling address changes. Instrumentation alters branch target addresses. JASLOAD maintains a dynamic list of instrumented block headers, adjusting branch targets at runtime to redirect jumps to the first inserted instruction—preserving control flow correctness.

Instrumentation examples. Call hooks (`call_pre`/`call_post`) track function invocations: `call_pre` logs callee and arguments pre-execution; `call_post` captures return values post-execution. These enable call graph construction for security/performance analysis.

3.6 High-level Program Analysis

Analysis APIs. To enable customizable load-time instrumentation and dynamic analysis, JASLOAD provides user-level APIs that empower developers to interact with the framework, adapt its behavior, and extract actionable insights. These APIs are categorized into four functional groups. First, callback APIs allows users to register event-driven hooks (e.g., on program initialization, function calls, or code structure detection) to trigger custom logic during execution. Second, inspection APIs grant read access to internal program states, including variable values, abstract syntax tree (AST) nodes, and call graphs, enabling real-time structural and behavioral analysis. Third, instrumentation APIs support dynamic code injection, letting users insert custom snippets (e.g., data collectors, loggers) at specific bytecode locations during loading. Finally, control APIs manage framework behavior by toggling instrumentation on/off, configuring analysis parameters, or prioritizing critical instrumentation points.

Algorithm design example of call graph analysis. Algorithm 1 demonstrates dynamic JavaScript call graph construction using the low-level `call_pre` hook through four sequential phases. The initialization phase (lines 2-3) retrieves the caller's metadata from the JavaScript engine's global information using function ID F, ensuring accurate context binding in

Algorithm 1: Call graph analysis

Input : F: Caller function ID; N: Callee function name; A: Arguments list;
Output: call_graph: Set of caller-callee relationships

```
1 Procedure JasLoad.call_pre(F, N, A...)
2     global_info ← Execution context metadata;
3     callerName ← global_info.func[F].N;
4     if N = empty string then
5         N ← "native_function" ;          // Handle
            unnamed functions
6     edge ← "callerName → N";
7     call_graph.add(edge) ;  // Add edge to graph
8     log(F, N, A, callerName, N, A) ;      // Debug
```

dynamic execution environments. The normalization phase (lines 4-5) resolves JavaScript's anonymous function ambiguity by automatically assigning a "native_function" identifier for empty callee names. The graph construction core (lines 6-7) generates unique caller-callee edges via formatted string operations stored in a hash-based set structure, achieving $O(1)$ insertion complexity while preserving type information through array-based argument storage. Finally, the diagnostic phase (lines 8) implements dual-mode observability by producing structured logs containing raw inputs, resolved relationships, and argument snapshots. This enables both real-time debugging and post-hoc analysis without impacting core graph operations, demonstrating how low-level bytecode hooks bridge instrumentation granularity with actionable semantic insights for high-level program analysis.

Evaluation. Through the execution of dynamic analysis techniques including call graph analysis, JASLOAD systematically evaluates runtime program behavior across target datasets. As the result for the evaluation, JASLOAD outputs the results of the source program execution and generates a report for subsequent analysis.

4. IMPLEMENTATION

To validate our system design, we developed JASLOAD as a functional software prototype comprising 3,402 lines of code (2,637 lines of C and 765 lines of JavaScript, respectively). The open-source implementation is included in this paper's reproduction package. Below we elaborate on key implementation aspects.

Bytecode translator. Our bytecode translation framework leverages the JerryScript embedded VM [33] because: 1) its optimization for resource-constrained environments aligns with our target domain, and 2) its widespread adoption (7.2k GitHub stars [33]) ensures practical relevance. We implement the based translator in 977 lines of C code to convert JerryScript's snapshot binaries [34] into our custom LITEBYTE format. Notably, our dynamic analysis methodology remains engine-agnostic and applies to other stack-based VMs like QuickJS [23] and Hermes [35].

TABLE IV: The benchmarks used in the evaluation.

Program	Binary size before/after	Instrumentation time (ms)	Execution time before/after (ms)
test262 (avg.)	2,158/7,438	7.51	4.79/14.84
string-iterator	1,088/5,636	2.22	153.10/3,560.80
object-literal	19,254/50,770	5.52	8.67/35.88

Bytecode instrumentation and high-level program analysis. We implement the instrumentation layer with 42 low-level hooks across 1,498 lines of C code, utilizing C's memory management and pointers to precisely manipulate VM internals. For high-level analysis, we develop JavaScript-based algorithms (614 lines) including instruction mix profiling, branch coverage tracking, and call graph analysis. This dual-language architecture strategically combines C's low-level control for instrumentation with JavaScript's agility for semantic analysis, effectively analyzing JavaScript programs within their native execution context. We are continuing to expand this suite of dynamic analysis capabilities.

5. Evaluation

In this section, we present the experiments we conducted to evaluate JasLoad. Our evaluation aims to answer the following research questions.

RQ1: Usability. How practical is it to conduct dynamic analyses by using JasLoad?

RQ2: Efficiency. How efficient is JasLoad when instrumenting JavaScript bytecode?

RQ3: Code size increase. How does the code size increase due to instrumentation?

RQ4: Runtime overhead. What is the runtime overhead introduced by the instrumentation?

RQ5: Effectiveness. Does the load-time instrumentation mechanism demonstrate advantages in improving program analysis efficiency and reducing resource consumption?

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.

5.1 Datasets

We create two datasets to conduct the evaluation: 1) a micro-benchmark from test262; and 2) a real-world benchmark consisting of two JavaScript programs distributed with JerryScript.

Micro-benchmarks. We create a micro-benchmark by selecting 20 programs from the test262 JavaScript benchmark suite [36], as listed in Table IV. These programs comprise a total of 1,378 lines of non-empty, non-comment JavaScript code. We parse them into binaries using JerryScript 2.4.0, yielding an average binary size of 2,158 bytes. Table IV details each program's binary size (before and after full instrumentation), full instrumentation time (in milliseconds), and execution time (averaged over 20 runs, before and after instrumentation).

Real-world JavaScript programs. We evaluate JasLoad using two real-world JavaScript programs, string-iterator.js and object-literal.js, sourced from the official JerryScript distribution. Their binary sizes are 1.09 KB and 19.25 KB, respectively. This demonstrates JasLoad's effectiveness and performance on practical JavaScript applications.

5.2 RQ1: Usability

To answer **RQ1**, we demonstrate JasLoad's usability in implementing dynamic analyses by evaluating the integration complexity of representative examples.

Call Graph Analysis maps function interactions to reveal dependencies. JasLoad automates call stack tracing, enabling detailed call graph generation with minimal effort, which is crucial for complex systems.

Null/Undefined Assignment Detection identifies improperly initialized variables/pointers. JasLoad's taint-tracking infrastructure enables precise detection of propagation paths with little added logic.

Dynamic Taint Analysis traces sensitive data flows for security (e.g., data leaks). JasLoad's built-in taint engine simplifies this, requiring only configuration of sources and sinks, scaling efficiently.

In summary, JasLoad significantly reduces the complexity of implementing diverse dynamic analyses. Its modular design, event hooks, and efficient instrumentation support sophisticated analyses from profiling to security. While focusing on the above, JasLoad also supports analyses like instruction mix, basic block profiling, and instruction/branch coverage, making it practical for many applications.

5.3 RQ2: Efficiency

To address **RQ2** and assess the efficiency of JasLoad, we measure its instrumentation time and runtime overhead across diverse benchmarks. Each test was executed 20 rounds, with results averaged for reliability.

As shown in Table IV, JasLoad demonstrates lightweight instrumentation. The test262 micro-benchmarks require an average instrumentation time of 7.51 ms, while real-world programs (string-iterator and object-literal) show even faster times of 2.22 ms and 5.52 ms, respectively. The binary size increased moderately post-instrumentation (e.g., from 2,158 to 7,438 KB for test262), reflecting the inherent trade-off between analysis granularity and overhead. Runtime performance varied across benchmarks. Notably, string-iterator execution time increased substantially from 153.10 ms to 3,560.80 ms, likely due to intensive dynamic checks. These results highlight JasLoad's efficiency in balancing instrumentation speed with practical usability, making it suitable for real-world dynamic analysis.

5.4 RQ3: Code Size Increase

To address **RQ3** on JasLoad's code size overhead, we compare original and instrumented JavaScript bytecode sizes. As Table IV shows, full instrumentation increased code size by 163.68% (jerry/object-literal) to 418.01% (jerry/string-iterator). While substantial, this remains acceptable for binary instrumentation frameworks and is significantly lower than Jalangi2's average 660.38% increase.

Fig. 4 illustrates the relative code size increase (normalized against original size) across test programs using different

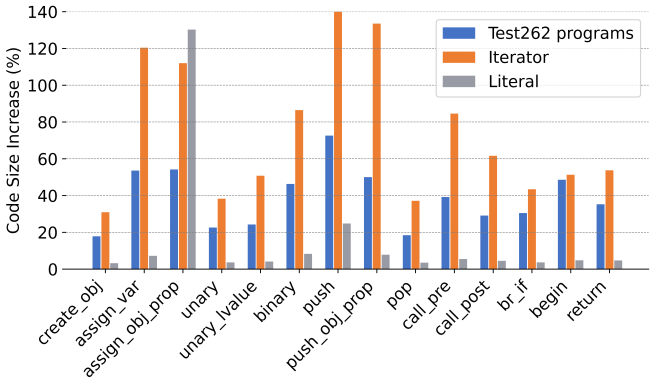

Figure 4: The code size increase, relative to the original size, when instrumenting the test programs with different low-level hooks.

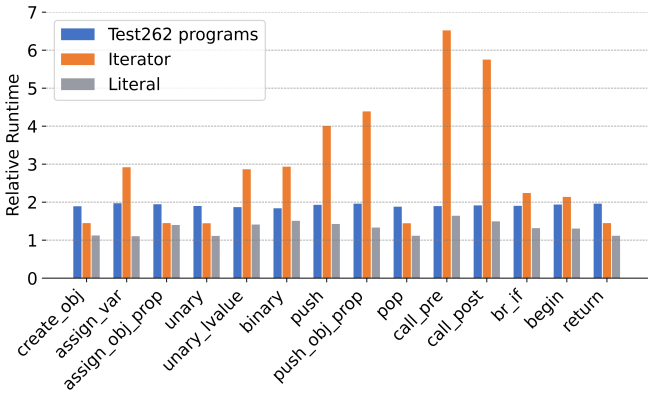

Figure 5: The instrumented programs' runtime relative to the uninstrumented runtime, measured per low-level hook, averaged over 20 runs. Binary sizes for the 20 test262 programs are presented as an average for clarity.

hooks. Over half of the hooks introduce minimal overhead. For example, in object-literal, hooks like create_obj, unary, unary_lvalue, and pop caused <2% increase; similarly, 20 test262 micro-benchmarks saw only ∼5% increase from these. However, hooks for frequent operations had more impact: call_pre/post (2.65%-58.82%), binary (5.60%-38.97%), and push (20.43%-72.61%).

Specific test cases showed higher increases for particular hooks. string-iterator's large increase stemmed from instrumenting assign_var, push, and call_pre hooks due to prevalent `init`/`call` instructions. object-literal had elevated overhead for assign_obj_prop from frequent `assign`/`set` operations, highlighting the link between size growth and instrumentable instruction density.

To reduce overhead, JASLOAD incorporates adaptive instrumentation, allowing selective hook activation based on analysis needs. This leverages hook independence and the fact that most analyses require only a small subset. Prioritizing necessary instrumentation helps balance code size with analysis precision, enhancing practicality.

5.5 RQ4: Runtime Overhead

To address **RQ4** and investigate JASLOAD's runtime overhead, we evaluate its impact on micro-benchmarks and real-world JavaScript programs, averaging results over 20 rounds. Table IV shows execution time comparisons before and after full instrumentation, revealing overheads ranging from 3.1x to 23.3x. This aligns with tools like Jalangi2 (average ∼3.2x), indicating acceptable overhead for practical use.

Fig. 5 details overhead per individual hook (y-axis: instrumented/original runtime ratio). For micro-benchmarks, each hook averaged 1.9x overhead. Real-world programs showed varied overhead depending on hook type and code complexity. Hooks like create_obj, unary, and pop incurred minimal overhead (<1.5x). Critical function-level hooks call and return showed modest overheads (up to 6.5x and 1.4x, respectively).

Higher overheads occurred with frequent operations: assign_var (1.1x-2.9x), unary_lvalue (1.4x-2.9x), binary (1.5x-2.9x), and push (1.4x-4.0x). Overhead variability stems from the number and complexity of instrumented instructions (e.g., extensive dynamic operations amplify call overhead). Adaptive instrumentation allows for balancing analysis needs with performance, as infrequent hooks contribute minimally. Averaged binary sizes of the 20 Test262 programs further contextualize the findings. JASLOAD's runtime overhead scales predictably with code size and hook complexity, ensuring adaptability without compromising usability.

5.6 RQ5: Effectiveness

To answer **RQ5** by investigating the effectiveness of the proposed load-time instrumentation approach, we systematically evaluate its performance impact using both micro-benchmarks (e.g., test262 suite) and real-world JavaScript applications. Each program was executed for 20 independent runs to mitigate environmental interference.

The evaluation results demonstrate that load-time instrumentation significantly optimizes analysis efficiency and resource consumption. For call graph analysis (a control-flow task), code size inflation decreases from a mean of 603.68% under traditional instrumentation to 43.05%, a 92.87% reduction. Runtime overhead decreases from 15.71x to 3.35x the baseline, a 78.67% improvement. For dataflow analyses like null/undefined assignment detection, code inflation reduces to 245.44% (a 59.34% reduction) and runtime overhead to 5.91x (a 62.38% reduction).

This optimization arises from the synergy between JASLOAD's bytecode loader/unloader and adaptive instrumentation strategy. Load-time instrumentation dynamically identifies critical instruction sequences during execution and selectively instruments only operations relevant to the target analysis (e.g., function calls, assignments). Combined with bytecode-level control, it avoids redundant instrumentation while preserving precision. A runtime feedback-driven strategy further optimizes hook activation thresholds, dynamically adjusting instrumentation density for high-frequency paths. Experimen-

tal results validate the generalization of this design in complex JavaScript scenarios, offering insights for engineering lightweight dynamic analysis tools.

6. Discussion

In this section, we discuss limitations of our work and outline directions for future research.

Further dynamic analysis. JasLoad's ability to instrument all LiteByte JavaScript instructions enables advanced dynamic analysis with minimal effort. On important future work is to leverage JasLoad to implement advanced and complex analyses like memory access tracing, debugging information extraction, security vulnerability detection, and resource usage monitoring. These analyses in turn will aid in optimizing performance, improving code quality, and enhancing application security and reliability.

Lightweight virtual machine optimization. Binary instrumentation can optimize JavaScript VMs, particularly the garbage collectors (GC). Instrumenting bytecode allows for collection of detailed GC runtime behavior and performance data, including trigger frequency, object types/sizes collected, and pause times. This information enables deeper insights for optimizing GC algorithms, improving JavaScript program performance, and resource utilization.

JavaScript universal dynamic analysis. Besides JerryScript studied in this paper, many lightweight JavaScript VMs, such as Hermes, QuickJS, Duktape, and IonMonkey, are widely used. While some VMs provide instrumentation capabilities similar to JasLoad, others lack them. Inspired by recent work [37], a universal binary-based dynamic analysis approach could be investigated. This approach would transcend individual VM boundaries, enabling seamless implementation of dynamic analyses regardless of specific VM constraints.

7. Related Work

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

JavaScript bytecode. JavaScript bytecode enhances performance, security, and cross-platform compatibility [19]. Its obfuscation capabilities mitigate reverse engineering and unauthorized modification risks [38]. Major engines supporting it include V8 (Chrome, Node.js) [39], SpiderMonkey (Firefox) [40], JerryScript (IoT) [33], and QuickJS [23]. However, as the adoption of JavaScript bytecode accelerates across diverse computing paradigms, the need for advanced dynamic analysis tools capable of operating at the binary level becomes increasingly imperative.

Dynamic analysis. Significant progress exists in dynamic program analysis. General frameworks address challenges like concurrency bugs [41], taint tracking [42] [43], and performance optimization [29]. Domain-specific tools include DiSL (JVM) [13], RoadRunner (Java) [14], and DynaPyt (Python) [44]. JavaScript-specific tools target vulnerability detection [45], testing [46], type inconsistencies [47], and race conditions [48] [49], with Jalangi/Jalangi2 enabling source-level analysis [50] [6]. However, these face limitations like high overhead and source-based constraints, creating gaps in low-level analysis. JasLoad fills this gap, enabling fine-grained JavaScript bytecode analysis to advance the field and improve application quality, performance, and security.

Binary instrumentations. A gap exists in binary instrumentation infrastructure. While mature frameworks exist for native code (Pin [11], Valgrind [12], DynamoRIO [51]) and emerging ones for WebAssembly (Wasabi [16]), equivalent tooling is absent for JavaScript bytecode. However, this absence hinders critical applications such as fine-grained security analysis, VM-level optimization, and cross-engine testing, despite the ubiquity of JavaScript. JasLoad is thus expected to advance JavaScript binary analysis and optimization.

8. Conclusion

This paper presents JasLoad, the first dynamic analysis framework for JavaScript binaries. We design a novel intermediate representation, LiteByte, for representing and reasoning about JavaScript bytecode; implement low-level hooks to instrument the IR for analysis; and develop high-level program analyses to track low-level behaviors. Furthermore, we implement and open-source a JasLoad prototype. This research initiates the journey towards dynamic analysis of JavaScript binaries with JasLoad, enabling future advances in JavaScript performance, security, and reliability.

References

[1] 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.

[2] S. Seneviratne, Y. Hu, T. Nguyen, G. Lan, S. Khalifa, K. Thilakarathna, M. Hassan, and A. Seneviratne, "A survey of wearable devices and challenges," IEEE Communications Surveys & Tutorials, vol. 19, no. 4, pp. 2573–2620, 2017.

[3] W. Largent, "New vpnfilter malware targets at least 500k networking devices worldwide," https://blog. talosintelligence. com/ vpnfilter/, May 23 2018.

[4] "Devtools - chrome for developers," https:// developer.chrome.com/ docs/ devtools.

[5] "Jalangi," https://github.com/ SRA-SiliconValley/ jalangi.

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

[7] 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.

[8] S. Chiba, "Javassist—a reflection-based programming wizard for java," in Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java, vol. 174. Citeseer, 1998, p. 21.

[9] A. Dinn, "Byteman: A tool for bytecode manipulation and injection," http://byteman.jboss.org.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in ECOOP 2001—Object-Oriented Programming: 15th European Conference Budapest, Hungary, June 18–22, 2001 Proceedings 15. Springer, 2001, pp. 327–354.

[11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," Acm sigplan notices, vol. 40, no. 6, pp. 190–200, 2005.

[12] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," ACM Sigplan notices, vol. 42, no. 6, pp. 89–100, 2007.

[13] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi, "Disl: a domain-specific language for bytecode instrumentation," in Proceedings of the 11th annual international conference on Aspect-oriented Software Development, 2012, pp. 239–250.

[14] C. Flanagan and S. N. Freund, "The roadrunner dynamic analysis framework for concurrent programs," in Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, 2010, pp. 1–8.

[15] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "Nvbit: A dynamic binary instrumentation framework for nvidia gpus," in Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019, pp. 372–383.

[16] D. Lehmann and M. Pradel, "Wasabi: A framework for dynamically analyzing webassembly," in Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019, pp. 1045–1058.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] "Duktape," https://duktape.org/.

[22] 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.

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

[24] "Mujs," https://www.mujs.com/.

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

[26] 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.

[27] 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.

[28] 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.

[29] X. Yu, S. Han, D. Zhang, and T. Xie, "Comprehending performance from real-world execution traces: A device-driver case," in Proceedings of the 19th international conference on architectural support for programming languages and operating systems, 2014, pp. 193–206.

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

[31] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," IEEE software, vol. 25, no. 5, pp. 22–29, 2008.

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

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

[34] "High-level design," https://github.com/jerryscript-project/ jerryscript/ blob/ master/ docs/ 04.INTERNALS.md.

[35] "Using Hermes," https://reactnative.dev/docs/hermes.

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

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

[38] 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.

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

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

[41] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "Avio: detecting atomicity violations via access interleaving invariants," ACM SIGOPS Operating Systems Review, vol. 40, no. 5, pp. 37–48, 2006.

[42] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software." in NDSS, vol. 5. Citeseer, 2005, pp. 3–4.

[43] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in Proceedings of the 2007 international symposium on Software testing and analysis, 2007, pp. 196–206.

[44] A. Eghbali and M. Pradel, "Dynapyt: A dynamic analysis framework for python," in Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 760–771.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in International Symposium on Code Generation and Optimization, 2003. CGO 2003. IEEE, 2003, pp. 265–275.