# Concolic Testing of JavaScript using Sparkplug

Zhe Li<sup>1</sup>, Fei Xie<sup>2,\*</sup> <sup>12</sup>Portland State University, Portland, OR, USA zl3@pdx.edu, xie@pdx.edu

Abstract— JavaScript has become the most popular programming language for not only web front-end development but also a wide range of server-side applications. Many of such applications handle sensitive information such as financial transactions and private conversions. Errors in such applications not only affect user experiences, but also endanger the safety, security, and privacy of users. While the reliability of JavaScript code will be of more importance, testing techniques for the language remain insufficient, compared to other languages. In-situ concolic testing of JS scripts is a framework that enables concolic testing of JS scripts in their native environments and is able to automatically generate test cases. However, its Qemu based execution tracing engine is slow in capturing traces, which is also not portable and involves two translation stages, which is a complex and errorprone process. In this paper, our approach proposed to deploy a new execution tracer leveraging V8's Sparkplug baseline compiler to improve the tracing process and a new assembly to LLVM IR using remill libraries. We evaluated its effectiveness and efficiency by comparing the coverage, bug detection, and time consumption with the in-situ approach on the same test set, which are 160 Node.js libraries that heavily utilize the String type and its operations. The results show our approach achieves similar statement coverage on these libraries within no more than 10% difference on average and is able to detect all bugs that are detected by the in-situ method and more, which only use a fraction of the time needed by the in-situ approach.

#### 1. INTRODUCTION

Since its emergence as a scripting language for dynamic web elements, JavaScript (JS) has experienced a surge in popularity and has evolved into a versatile and extensively utilized application programming language. The Node.js runtime, leveraging Chrome's V8 JS engine as its foundation, empowers developers to create a diverse range of server-side and clientside browser-less applications using pure JavaScript [1]. An entire ecosystem of Node.js libraries has been cultivated, accessible via the Node Package Manager (NPM), and extensively employed in the development of applications [2]. As JavaScript continues to gain significance in the web, mobile, and cloud infrastructure of modern systems, the repercussions of bugs and security vulnerabilities in JS scripts become increasingly severe [3]. JS scripts, whether browser or Node.js based, are often perceived by many developers as significant security vulnerabilities. Common security concerns associated with browser-based JS scripts include cross-site scripting (XSS) and SQL injection (SQLi) [4], etc. Errors and failures in JS scripts executed on Node.js can result in server crashes or compromises. Among the most prevalent security issues in Node.js are NPM phishing and denial of service (DoS) attacks targeting regular expressions. NPM provides developers with the capability to develop and distribute JS libraries for reuse, yet this flexibility introduces notable security risks [5]. Developers face a pressing demand to construct comprehensive test suites capable of early bug and security vulnerability detection. However, manually crafting such suites has become a costly and time-consuming bottleneck in software development [6]. Symbolic execution is a powerful technique for automating the generation of test cases and identifying bugs in real-world software. It entails executing a program using symbolic values, monitoring program path conditions via symbolic expressions, and producing test cases to explore these paths by solving symbolic path conditions [7]. Concolic testing is a hybrid verification technique designed to address the challenge of path explosion often encountered in symbolic execution [8]. Concolic testing employs symbolic execution to traverse only the branches along a concrete execution path determined by a concrete input of the program being tested. This approach effectively reduces the explored path space, mitigating the issue of path explosion [9]. Traditional symbolic or concolic execution engines primarily focus on analyzing code written in languages like C/C++ or those that compile to low-level intermediate representations (LLVM) [10] or binary code, e.g., KLEE [11], BitBlaze [12], S2E [13], DART [14], CUTE [15], SAGE [16], and CRETE [17].

In-situ concolic testing of JS scripts is a novel framework that enables concolic testing of JS scripts in their native environments and can automatically generate test cases that achieve comparable, if not better, code coverage than manually crafted unit test suites for Node.js libraries and discovered previously unknown bugs in these libraries [18]. Most approaches of concolic testing on JavaScript typically take JS scripts out of their native execution environments and analyze them in artificial test harnesses. For example, the Kudzu engine addresses the problem of client-side code injection vulnerabilities for JavaScript [19]. It involves modifying the JS interpreter to build a new symbolic execution engine, which requires significant effort in implementation and maintenance. Such JS-specific symbolic engines have not demonstrated the effectiveness and efficiency that warrants wide adoption [20]. In-situ concolic testing for JavaScript using JavaScript's native execution environments becomes its biggest strength. However, it has several limitations [18]. It utilized the tracing engine of CRETE, which leverages the interpreted mode of Qemu, a dynamic translator [21], to capture the execution trace of JS scripts and uses KLEE as the backend symbolic execution engine. The concrete execution trace is converted from a piece of code to the host instruction set, and the instruction set is then translated to qemu-ir by the tiny code generator (TCG) of *Qemu* dynamic translation backend. This process hinders the efficiency of the tracing process greatly since the in-situ approach uses the interpreted mode with TCG to enable tracing. The execution tracer of CRETE takes 3 minutes to trace a JS function with 12 lines of code on average, which is inefficient. The execution traces are then translated from qemu-ir to LLVM IR by an offline translator based on  $S^2E$ . This workflow involves two stages of translation for the execution traces, which gives more chances for introducing errors and mistakes.

To improve the efficiency of the execution tracer, reduce the number of translation stages, and conduct concolic testing in their native environments like the in-situ approach at the same time, our approach proposed to deploy a new execution tracer leveraging V8's Sparkplug baseline compiler to improve the tracing process and a new assembly to LLVM IR using *remill* libraries in this paper. We evaluated its effectiveness and efficiency by comparing the coverage, bug detection, and time consumption with the in-situ approach on the same test set, which are 160 Node.js libraries that heavily utilize the String type and its operations. The results show our approach achieves similar statement coverage on these libraries within no more than 10% difference on average and is able to detect all bugs that are found by the in-situ method, which only uses a fraction of the time needed by the in-situ approach.

2. BACKGROUND

## 1. Sparkplug

Sparkplug is a non-optimizing JavaScript compiler of V8 [22]. It is engineered for swift compilation. Its speed is remarkable, enabling us to compile at our convenience, thereby facilitating a more aggressive tiering up to Sparkplug code [22]. There are a couple of techniques employed by the Sparkplug compiler to achieve its impressive speed. Firstly, it utilizes a shortcut; the functions it compiles are already processed into bytecode by a prior stage, which handles complex tasks such as variable resolution and parsing arrow functions. Sparkplug bypasses these intricate processes by compiling JavaScript from bytecode rather than directly from source code. Secondly, Sparkplug adopts a unique approach by skipping the generation of an intermediate representation (IR), a step typical in most compilers. Instead, it directly translates bytecode into machine code in a single linear pass using bytecode handlers [22], aligning the emitted code with the execution flow of the bytecode. We will discuss the bytecode handler in detail in Section 4. This feature guarantees that the emitted machine code execution trace we used for concolic analysis represents the execution flow of the source code. Remarkably, the entire Sparkplug compiler operates within a switch statement nested within a for loop, efficiently dispatching to predetermined bytecode handlers, the machine code generation functions based on the bytecode encountered. The absence of an IR restricts



Figure 1: Sparkplug Restricted optimization feature

optimization opportunities to localized peephole optimizations as shown in Figure 1, we heavily utilize this feature of Sparkplug.

#### 2. Interpreter Stack Frame Mirroring

V8 JavaScript engine supports two modes for executing a JS script, namely interpreted mode and optimized just-intime compilation mode. The interpreted mode is where the JS bytecode [23] translated from the JS script is interpreted by its interpreter, Ignition [24], which is the foundation of in-situ concolic execution for JavaScript [?]. The optimized just-intime compilation mode is where the bytecode is compiled by the V8 engine into optimized machine code using its just-intime compiler, Turbofan [25], and then executed on the target machine. Sparkplug as a baseline JavaScript compiler can restrict JS script from being optimized to mitigate complexity for later concolic execution. Furthermore, Sparkplug mirrors the execution of Ignition for JavaScript. Sparkplug intentionally aligns its stack frame layout with that of Ignition, ensuring that when Ignition stores a value in a register, Sparkplug does the same. This design choice simplifies Sparkplug compilation by allowing it to mirror the behavior of Ignition without the need for complex mappings between interpreter registers and Sparkplug's state. Therefore, it also allow us to improve the efficiency for the in-situ approach and keep its effectiveness at the same time. Sparkplug primarily consists of bytecode handler calls, which are short sequences of machine code embedded within the binary, along with control flow. Ignition and Sparkplug share significant portions of the bytecode handlers. In essence, Sparkplug serves as a serialization of Ignition execution, invoking the same built-ins and maintaining identical stack frames. This feature allows us to trace JS bytecode execution in its corresponding machine code like the execution in Ignition. Futhermore, Sparkplug effectively precompiles certain unavoidable interpreter overheads, such as

operand decoding and dispatching to the next bytecode. This streamlined strategy contributes to Sparkplug's efficiency and performance. Therefore, Sparkplug can generate machine code that contains the same control flow as JS script, which can be used for code translation from machine code (assembly code) to LLVM.

## 3. Remill

McSema is an executable lifter that specializes in converting executable binaries from their machine code into LLVM. This process enables the translation of low-level binary instructions into a higher-level intermediate representation. Within McSema [26], the instruction translation functionality is powered by the Remill library. Unlike some other tools, Remill exclusively handles machine code translation into LLVM [27]. The versatility of *Remill* extends to both static and dynamic binary translation scenarios. Notably, it has been employed in symbolic execution workflows alongside tools like KLEE [11]. KLEE, which performs symbolic execution, typically operates on the LLVM IR generated from source code using the LLVM toolchain [10]. By utilizing Remill to machine code into the LLVM IR, previously inaccessible targets become available for analysis with KLEE, thus expanding the range of symbolic execution capabilities.

Remill delegates the implementation of memory accesses and specific types of control flow to the consumers of the generated LLVM. This deferral is facilitated through Remill intrinsics, which are special functions representing various actions within the translated program. For instance, the \_\_\_remill\_read\_memory intrinsic function symbolizes the act of reading 8 bits of memory. By leveraging these intrinsics, downstream tools can differentiate between LLVM load and store instructions and access to the modeled program's memory. Moreover, downstream tools have the flexibility to implement memory intrinsics using LLVM's native memory access instructions. This approach allows us to create a seamless integration of *Remill* generated LLVM into existing LLVM-based workflows while providing the necessary flexibility for custom memory access implementations tailored to specific analysis requirements. We utilized this feature to adapt the output to LLVM-based symbolic analysis tools.

## 3. Approach

## 1. Overview of goals

Our approach aims to make improvements in efficiency for the in-situ approach mainly in generating execution traces and execution trace translation. Our approach strives to apply concolic testing on JS scripts in their native environment to generate effective test data for unit testing of these scripts. The workflow of concolic execution on JS scripts contains the following steps. As shown in Figure 2, the concrete execution step in the leftmost box of concolic testing is conducted in the native execution environment for JS scripts, where the trace of this concrete execution is captured using the JS execution tracer. The trace is then analyzed in the symbolic execution step in the rightmost box of concolic testing to generate test cases automatically.

- Execution trace capture. Concrete execution traces of JS scripts are captured with a JS execution tracer, which is the interpretation of JavaScript bytecode. The concrete execution traces are in the form of assembly code, which represents the interpretation of JS bytecode execution.
- **Translation.** In this step, our approach uses a translator to translate assembly code generated by the JS execution tracer into LLVM bitcode.
- **Symbolic Analysis.** The execution trace represented by LLVM is fed into a symbolic execution engine to generate test cases.

## 2. Improvement

In-situ concolic testing offers the capability of tracing inside the V8 JS engine to capture the execution trace that closely matches the JS bytecode interpretation [18], [23]. The conciseness of an execution trace determines the efficiency and effectiveness of later symbolic analysis and test case generation. Therefore, we intend to preserve such traits and achieve improvement of execution efficiency at the same time. Our approach improves in-situ concolic testing in 2 aspects. In Figure 2, the in-situ approach is represented by the diagram in the red box and our approach in the green box. Firstly, compared to the in-situ approach of concolic testing for scripting languages, our approach frees the execution tracer from dependence on an emulator, which is normally slow. The concrete execution is obtained by the execution tracer, which leverages V8's Sparkplug engine instead of CRETE execution tracer based on *qemu* in the in-situ approach. This speeds up the execution trace capture process since qemu is based on an emulator. At the same time, it preserves the character that the execution trace capture happens in the native execution environment for JS script because we leverage the native Sparkplug baseline engine as the execution tracer.

## a) Why we choose Sparkplug?

Sparkplug disables the Turbofan path naturally. It compiles from bytecodes that Ignition emits as shown in Figure 3. JS bytecode preserves all necessary control flow JS source code has. Therefore, execution traces captured by Sparkplug have a one-to-one correspondence to the JS source code. The execution tracer based on Sparkplug directly traces the bytecodes translated from JS source code inside of V8. Furthermore, as mentioned in Section 2-B, Sparkplug mirrors Ignition's execution for JavaScript, Sparkplug and Ignition have almost identical stack frame [22]. This simplifies the design by removing the deep tracing control interface used in the insitu approach shown in the red box by actually tracing within Sparkplug. To retrieve the most concise execution trace for JS script, our approach only extracts bytecodes that contribute to the control flow of JS script execution with Instruction Extraction component, which removes the stack verificationrelated bytecodes in the generated execution trace without influencing the verification workflow of Sparkplug.



Figure 2: Workflows of In-situ Concolic Testing Based on Sparkplug and CRETE



Figure 3: Workflow of Execution Tracer between In-situ Approach and Our Approach



Figure 4: Workflow of the Translator

Secondly, the in-situ approach uses an offline translator to translate qemu-ir to LLVM IR. *Qemu*, the emulator first translates assembly code to the intermediate presentation of qemu-ir and then uses an offline translator to translate qemu-ir to LLVM IR. LLVM is a widely used intermediate presentation for symbolic analysis. Our approach simplifies this process by directly translating the captured execution traces from assembly code to LLVM IR shown in the middle box in Figure 2. In this process, we introduce a helper component in the translator. This helper component aims to make the translated execution trace amenable to symbolic analysis tools by providing the main entry point and marking

symbolic variables as shown in Figure 4. As a result, the output of the translator produces a complete concrete execution trace for later symbolic execution engine to generate test cases.

#### 4. IMPLEMENTATIONS

In this section, we demonstrate the feasibility of our approach by implementing its complete workflow with an execution tracer based on V8's Sparkplug, a translator leveraging Remill, and a symbolic execution engine using KLEE [11].

Interpreted JS Bytecode Array of Concrete Execution Trace	Captured execution trace	
6b f9 04 TestEqual r1, [4]		
[ VerifyFrame [ VerifyFrameSize 0x555e65804430 3f0 4989e2 REX.W movq r10,rsp 0x555e65804433 3f3 4983c240 REX.W addq r10,0x40 0x555e65804437 3f7 4c3bd5 REX.W cmpq r10,rbp 0x555e6580443a 3fa 740d jz 0x555e65804449 <+0x409>]		
Verify feedback vector 0x555e65804449 409 4c8b45d8 REX.W movq r8,[rbp-0x28] 0x555e6580444d 40d 41f6c001 testb r8,0x1 0x555e65804451 411 0f8424000000 jz 0x555e6580447b <+0x43b> 0x555e6580446a 42a 458b48ff movl r9,[r8-0x1] 0x555e6580446e 42e 4181f91d030000 cmpl r9,0x31d 0x555e65804475 435 0f840d000000 jz 0x555e65804488 <+0x448>	6b f9.04 TostFougl r1.[4]	
[ CallBuiltin 0x555e65804488 448 488b55c8 REX.W movq rdx,[rbp-0x38] 0x555e6580448c 44c bb04000000 movl rbx,0x4 ]	bb 19 04   Testequal (1, [4])     [ CallBuiltin   0x555e65804488   448   488b55c8   REX.W movq rdx,[rbp-0x38]     0x555e6580448c   44c   bb04000000   movl rbx,0x4 ]	
99 0f JumplfFalse [15]		
[ VerifyFrame [ VerifyFrameSize 0x555e65804496 456 4989e2 REX.W movq r10,rsp 0x555e65804499 459 4983c240 REX.W addq r10,0x40 0x555e6580449d 45d 4c3bd5 REX.W cmpq r10,rbp 0x555e658044a0 460 740d jz 0x555e658044af <+0x46f> ]		
Verify feedback vector 0x555e658044af 46f 4c8b45d8 REX.W movq r8,[rbp-0x28] 0x555e658044b3 473 41f6c001 testb r8,0x1 0x555e658044b7 477 0f8424000000 jz 0x555e658044e1 <+0x4a1> 0x555e658044d0 490 458b48ff movI r9,[r8-0x1] 0x555e658044d4 494 4181f91d030000 cmpl r9,0x31d 0x555e658044db 49b 0f840d000000 jz 0x555e658044ee <+0x4ae>]	99 Of JumplfFalse [15]	
[CallBuiltin 0x555e658044ee 4ae 3de10d0000 cmp rax,0xde1 0x555e658044f3 4b3 7505 jnz 0x555e658044fa <+0x4ba> 0x555e658044f5 4b5 e96a020000 jmp 0x555e65804764 <+0x724>]	[ CallBuiltin   0x555e658044ee 4ae 3de10d0000 cmp rax,0xde1   0x555e658044f3 4b3 7505 jnz 0x555e658044fa <+0x4ba>   0x555e658044f5 4b5 e96a020000 jmp 0x555e65804764 <+0x724>]	

Figure 5: How the execution tracer only extracts the execution traces that contribute to the main control flow of JS scripts

#### a) Modification on Bytecode Handlers of Sparkplug

To capture the most concise execution trace, we implemented the function extract\_function\_instr to filter out the stack verification-related compilation from Sparkplug and only extract the execution trace for bytecodes that contribute to the control flow of JS scripts. The left column of Figure 5 shows an example of an interpreted JS bytecode array of a concrete execution trace. Before interpreting each bytecode, Sparkplug verifies frame size and feedback vector. The execution tracer based on Sparkplug only removes the corresponding interpretation from the execution trace without changing Sparkplug's behavior. The green box indicates the bytecode extracted by the function and its correspondence assembly code generated by the bytecode handler of Sparkplug. The red box indicates the assembly instructions that are filtered out, which corresponds to stack frame verification. A special bytecode handler PIN\_SYMBOLIC is implemented to cache the symbolic value in the execution trace for later symbolic analysis.

#### b) Implementation on Remill translator

We utilized *remill* library to implement an assembly-to-LLVM translator. Figure 6 shows the important components we implemented for the translator. It first checks if an instruction is valid as in whether the memory is executable and readable. In this process, it identifies the symbolic memory we cached by the execution tracer based on Sparkplug. After the correctness check, the translator translates *remill* basic blocks to LLVM basic blocks. A helper component is added to create a main entry function to make the trace a self-contained LLVM module and mark symbolic memory for later symbolic analysis. The main function then calls into the basic blocks LLVM functions. At last, the resulting trace is readily consumable by the KLEE. We tested the execution tracer to ensure its correctness on 16 combinations of instructions such as math functions, basic arithmetic, for loop, if-else, etc.

During the symbolic execution stage, KLEE is modified to recognize the *remill* intrinsic function for log error and exception.



Figure 6: How the execution tracer only extracts the execution traces that contribute to the main control flow of JS scripts

The execution trace is fed into KLEE to generate test cases. Test cases are used as a seed for the next iteration of symbolic execution to generate a comprehensive set of test cases, which is done by execution harness scripts.

## 5. EVALUATIONS

For evaluations, we targeted 160 Node.js libraries used in the in-situ approach to show the effectiveness and efficiencies after improvement. For effectiveness, we calculated the average time used for executing all libraries between the two methods. For efficiency, we evaluated the code coverage achieved by two methods. This evaluation is carried out on a Ubuntu OS Version 18.04 with 4-core Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz and 16G memory.

To compare the two methods with these libraries, we built a test harness to systematically exercise all exported (public) methods in a given library with arguments whose type is String. The seed test cases are generated randomly within the test harness. We implemented an automation pipeline that helps set up the concolic testing environment for each Node.js library automatically. Coverage for all libraries is calculated using *istanbul*, a popular JS coverage tool used by V8 [28] and compatible with most JavaScript testing frameworks, e.g., Mocha [29] and Node-Tap [30]. Coverage may vary slightly due to the randomness of the seed test case generation. By default, the coverage that we show in this evaluation is statement coverage. Table I shows the demographics of the

Metric	Range	Average
Line of Code	[93, 16910]	1687
Weekly Downloads	[3, 37491350]	9552965
Dependencies	[3, 18154]	282

TABLE I: Demographics for Libraries under Test

selected libraries. The LoC (lines of code) for a library under test is calculated with *github-loc* [31]. The number of weekly

downloads of a library under test is calculated with *npm-stats-api* [32]. The number of dependencies is the number of dependent libraries that the library under test has. We calculated it with *dependent-counts* [33].

# 1. Coverage Analysis

Figure 7 shows the comparison of statement coverage achieved between our approach and the in-situ approach. The red line presented the statement coverage of the in-situ approach and the blue line indicates the statement coverage of our approach of improvement. We can see that they represent a similar trend of achieving statement coverage over 160 Node.js libraries under test. Figure 8 indicates the distribution of statement coverage between the two approaches, where the red dots represent the result of the in-situ approach and the blue dots indicate that of our approach. We can see major dots of both colors fall above the line of coverage of 75%. Only 9 libraries achieved a coverage below 25% and the reason is that it is a function with multiple arguments of String type, which can be made symbolic. Our test harness did not catch all of the arguments and only managed to set one of them as symbolic input. Therefore, it only explored the branches that are related to that one argument we set as symbolic input within the test harness. Among the libraries achieved below the coverage of 75%, the red dots appear more times than the blue dots, which indicates our approach achieved higher coverage on average.







Figure 8: Coverage Distribution Comparison between our approach and In-situ approach

functions	Bugs	Known
formatNumber	No boundary check for empty string	No
encodeDate	No NULL check for function argument	No
regexExec	Unhandled input syntax error	No
isVAT	Mishandled country code	No
chalkClass	Deprecated constructor invoked	Yes
stringify	Incorrect parsing of separators	Yes

TABLE II: Bugs detected in functions

We also compared our approach with an existing tool, ExpoSE [34], by testing the same set of libraries as shown in Figure ??, on which ExpoSE has been applied. Our method and the in-situ approach achieved similar higher coverage consistently. This comparison only partially reflects our method's ability to achieve higher coverage since ExpoSE mainly targets solving regular expression problems for its symbolic execution engine JALANGI.



Figure 9: Statement Coverage Comparison among our approach, In-situ approach and ExpoSE

## 2. Bug Detection Efficiency

As the test cases generated by our approach are replayed on the libraries under test, our method detected all the bugs that the in-situ method found. At the same time, our method only uses a fraction of the time that the in-situ approach needs. Typically, the in-situ approach takes about 3 to 5 minutes to complete an iteration of test case generation and it only needs about 5 seconds to complete an iteration with our approach. Exceptions are thrown during execution. There are two types of exceptions: unhandled and handled. The unhandled exceptions tend to indicate potential valuable bugs. The handled exceptions often indicate that the developers are aware of these exceptions, but want to deal with them later. Such exceptions are also valuable to both the developers and potential hackers, albeit less valuable than unhandled ones.

Table **??** shows a summary of the bugs that we discovered. The bug from *benchmarkify* is a missing boundary check for empty string. It causes the formatNumber function to return a NULL object. When another function is later invoked on this NULL Object, it throws a TypeError exception. In the encodeDate function of *msgpack5*, a parameter, dt, is used directly without checking for NULL value. In *is-regex*, an input syntax error is not handled in the regexExec function. In *validator*, a particular country code is not handled and leads to the execution of a catch block in the isVAT function. In *chalk*, a deprecated constructor is used in an else branch in the chalkClass function, causing an unhandled exception. In *stringify*, incorrect parsing of separators in the stringify function causes an unhandled exception.

# 6. RELATED WORK

Our approach is closely aligned with prior research on execution tracing within native execution environments and enhancing trace translation in symbolic execution for JavaScript. Typically, the focus is on JavaScript scripts, including those running in browsers and browser-less runtimes like Node.js. Many symbolic execution techniques for JavaScript involve the development of application-specific symbolic execution engines or substantial modifications to JavaScript execution engines to facilitate symbolic execution. These methods often rely on intermediate representations during trace translation. For instance, SymJS is an example of a framework designed for testing client-side JavaScript scripts using symbolic execution [35]. It modifies Rhino JS engine for symbolic execution [36]. For browser-less JavaScript, JALANGI is a framework for writing heavy-weight dynamic analysis, which can be enabled on JavaScript as a symbolic execution engine [37]. COSETTE is another symbolic execution engine for JavaScript using an intermediate representation, namely JSIL, translated from JavaScript [38]. ExpoSE applies symbolic execution on standalone JavaScript and uses JALANGI as its symbolic execution engine. ExpoSE's contribution is in addressing the limitation that JALANGI does not readily support regular expressions for JavaScript [34]. Kudzu targeted AJAX applications by implementing a dynamic symbolic interpreter that takes a simplified intermediate language for JavaScript [19]. To the best of our knowledge, no symbolic execution framework for JavaScript has directly utilized JavaScript's native execution environments for execution tracing [39].

#### 7. CONCLUSIONS

In this paper, our approach introduced improvements to the insitu concolic testing of JavaScript. We have deployed a new execution tracer leveraging V8's Sparkplug baseline compiler to improve the tracing process and a new assembly to LLVM IR using *remill* libraries. It improves the efficiency and effectiveness of the infrastructure of the in-situ concolic testing for JavaScript while keeping the native execution environments for JS scripts under test. We evaluated its effectiveness and efficiency by comparing the coverage, bug detection, and time consumption with the in-situ approach on the same test set, which are 160 Node.js libraries that heavily utilize the String type and its operations. The results show our approach achieve similar statement coverage on these libraries within no more than 10% difference on average and is able to detect all bugs that are detected by the in-situ method, which only use a fraction of the time needed by the in-situ approach.

## REFERENCES

- [1] "Node.js," https://nodejs.org/en/, 2021.
- [2] "Npm," https://www.npmjs.com/, 2021.
- [3] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 181–191. [Online]. Available: https://doi.org/10.1145/3196398.3196401
- [4] S. Lekies, B. Stock, and M. Johns, "25 million flows later: Large-scale detection of dom-based xss," in *Proceedings of the 2013 ACM SIGSAC Conference* on Computer and Communications Security, ser. CCS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 1193–1204. [Online]. Available: https://doi.org/10.1145/2508859.2516703
- [5] N. van Ginkel, W. De Groef, F. Massacci, and F. Piessens, "A server-side javascript security architecture for secure integration of third-party libraries," *Security and Communication Networks*, vol. 2019, 2019.
- [6] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Jseft: Automated javascript unit test generation," in 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST). IEEE, 2015, pp. 1–10.
- [7] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.
- [8] S. Krishnamoorthy, M. S. Hsiao, and L. Lingappan, "Tackling the path explosion problem in symbolic execution-driven test generation for programs," in 2010 19th IEEE Asian Test Symposium. IEEE, 2010, pp. 59– 64.
- [9] K. Sen, "Concolic testing," in Proceedings of the twentysecond IEEE/ACM international conference on Automated software engineering, 2007, pp. 571–572.
- [10] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis transformation," in *International Symposium on Code Generation and Optimization*, 2004. CGO 2004., 2004, pp. 75–86.
- [11] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests

for complex systems programs." in OSDI, vol. 8, 2008, pp. 209–224.

- [12] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *International Conference* on *Information Systems Security*. Springer, 2008, pp. 1– 25.
- [13] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *Acm Sigplan Notices*, vol. 46, no. 3, pp. 265– 278, 2011.
- [14] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005* ACM SIGPLAN conference on Programming language design and implementation, 2005, pp. 213–223.
- [15] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," ACM SIGSOFT Software Engineering Notes, vol. 30, no. 5, pp. 263–272, 2005.
- [16] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Communications of the ACM*, vol. 55, no. 3, pp. 40–44, 2012.
- [17] B. Chen, C. Havlicek, Z. Yang, K. Cong, R. Kannavara, and F. Xie, "Crete: A versatile binary-level concolic testing framework," in *Fundamental Approaches to Software Engineering*, A. Russo and A. Schürr, Eds. Cham: Springer International Publishing, 2018, pp. 281–298.
- [18] Z. Li and F. Xie, "In-situ concolic testing of javascript," In Proceedings of the 30th IEEE International Conference on Software Analysis, Evolution and Reengineering, 2023.
- [19] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in 2010 IEEE Symposium on Security and Privacy, 2010, pp. 513–528.
- [20] S. Süslü and C. Csallner, "Spejs: A symbolic partial evaluator for javascript," in *Proceedings of the 1st International Workshop on Advances in Mobile App Analysis*, ser. A-Mobile 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 7–12. [Online]. Available: https://doi.org/10.1145/ 3243218.3243220
- [21] F. Bellard, "Qemu, a fast and portable dynamic translator." in USENIX annual technical conference, FREENIX Track, vol. 41. Califor-nia, USA, 2005, p. 46.
- [22] "Sparkplug," https://v8.dev/blog/sparkplug, 2023.
- [23] "Understanding v8's bytecode," https://medium.com/ dailyjs/understanding-v8s-bytecode-317d46c94775, 2021.
- [24] "Firing up the ignition interpreter," https://v8.dev/blog/ ignition-interpreter, 2021.
- [25] "Turbofan: A new code generation architecture for v8," https://docs.google.com/presentation/d/1\_eLlVzcj94\_ G4r9j9d\_Lj5HRKFnq6jgpuPJtnmIBs88/htmlpresent, 2021.

- [26] "Sparkplug," https://github.com/lifting-bits/mcsema, 2023.
- [27] "Sparkplug," https://github.com/lifting-bits/remill, 2023.
- [28] "Istanbul," https://istanbul.js.org/, 2021.
- [29] "Mocha: simple, flexible, fun," https://mochajs.org/, 2021.
- [30] "Node-tap," https://node-tap.org/, 2021.
- [31] "github-loc," https://www.npmjs.com/package/ github-loc, Jun. 2021.
- [32] "npm-stats-api," https://www.npmjs.com/package/ npm-stats-api, Jun. 2021.
- [33] "dependent-counts," https://www.npmjs.com/package/ dependent-counts, Jun. 2021.
- [34] B. Loring, D. Mitchell, and J. Kinder, "Expose: practical symbolic execution of standalone javascript," in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, 2017, pp. 196–199.
- [35] G. Li, E. Andreasen, and I. Ghosh, "Symjs: automatic symbolic testing of javascript web applications," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 449–459.
- [36] X.-o. JIN, B.-y. ZHONG, and X. LI, "Research and implementation of interpreting javascript dynamic web page based on rhino engine [j]," *Computer Technology and Development*, vol. 2, no. 002, 2008.
- [37] 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.
- [38] J. F. Santos, P. Maksimović, T. Grohens, J. Dolby, and P. Gardner, "Symbolic execution for javascript," in *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, 2018, pp. 1–14.
- [39] Y.-F. Li, P. K. Das, and D. L. Dowe, "Two decades of web application testing—a survey of recent advances," *Information Systems*, vol. 43, pp. 20–54, 2014.