2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-6654-5278-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/SANER56733.2023.00031

In-Situ Concolic Testing of JavaScript

1st Zhe Li
Portland State University
Portland, Oregon, USA
zl3@pdx.edu

2nd Fei Xie
Portland State University
Portland, Oregon, USA
xie@pdx.edu

***Abstract*—JavaScript (JS) has evolved into a versatile and popular programming language for not only the web, but also a wide range of server-side and client-side applications. Effective, efficient, and easy-to-use testing techniques for JS scripts are in great demand. In this paper, we introduce a holistic approach to applying concolic testing to JS scripts in-situ, i.e., JS scripts are executed in their native environments as part of concolic execution and test cases generated are directly replayed in these environments. We have implemented this approach in the context of Node.js, a JS runtime built on top of Chrome's V8 JS engine, and evaluated its effectiveness and efficiency through application to 180 Node.js libraries with heavy use of string operations. For 85% of these libraries, it achieved statement coverage ranging between 75% and 100%, a close match in coverage with the hand-crafted unit test suites accompanying their NPM releases. Our approach detected numerous exceptions in these libraries. We analyzed the exception reports for 12 representative libraries and found 6 bugs in these libraries, 4 of which are previously undetected. The bug reports and patches that we filed for these bugs have been accepted by the library developers on GitHub.**

I. Introduction

Since its inception as a scripting language for dynamic web elements, JavaScript (JS) has seen its popularity balloon and has become a versatile and widely used application programming language. The Node.js runtime [1], which is built upon Chrome's V8 JS engine [2], allows developers to build various server-side and client-side browser-less applications in pure JavaScript. A whole ecosystem of Node.js libraries is developed, available through the Node Package Manager (NPM) [3], and widely used in application building. NPM is considered the largest package manager based on the number of packages it manages [4]. This number is still growing at an average rate of 996 more packages per day in the past year [5].

Many developers consider JS scripts (either browser or Node.js based) a major security vulnerability because of its growing popularity in today's systems [6]. Common security issues of browser-based JS scripts include cross-site scripting (XSS) [7], SQL injection (SQLi) [8], etc. Errors and failures in JS scripts running on Node.js can lead to server crashes or compromises. The most common Node.js security issues include NPM phishing [9] and regular expressions denial of service (DoS) [10]. NPM allows developers to create and upload JS libraries for reuse purposes. This flexibility enables developers to build applications very easily by leveraging libraries already implemented by others. However, this extensive cross-dependencies among JS libraries further exacerbate the security threats [11]. Studies also show on average 6.8% of the code from a Node.js application is the original code and 93.2% of the code is from other JS libraries [4]. And only 45.2% of those JS libraries have test suites provided [12]. Thus, there is a great need for developers to craft high-coverage test suites that detect bugs and security vulnerabilities early. However, handcrafting such test suites has become costly endeavours and bottlenecks for software development [13].

A powerful technique for automatically generating test cases and finding bugs in real-world software is symbolic execution, which executes a program with symbolic values, accumulates program path conditions as symbolic expressions, and generates test cases exploring these paths by solving symbolic path conditions [14]. Concolic testing is a hybrid verification technique that alleviates path explosion that often bogs down symbolic execution [15]. Concolic testing utilizes symbolic execution to only explore the branches along a concrete execution path of the program under test, therefore, narrowing down the search space for path exploration [16]. Traditional symbolic or concolic execution engines mostly target C/C++, low-level intermediate representation (LLVM) [17] or binary code, e.g., KLEE [18], BitBlaze [19], S2E [20], DART [21], CUTE [22], SAGE [23], and CRETE [24].

Although early applications of symbolic execution for testing JS scripts have shown some promise, they never reach the same scale and effectiveness as those for C/C++ applications. Generally speaking, JS scripts are not statically compiled, but are interpreted by an interpreter. A simple JS statement can encapsulate complex operations that, in lower-level languages, would be implemented in tens, if not hundreds, lines of codes [25]. This complexity makes naïve applications of traditional symbolic execution engine to JavaScript intractable and can easily lead to path explosion. Consequently, efforts in applying symbolic execution to JavaScript have been focused on building JS-specific symbolic engines which 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 [26]. It involves modifying the JS interpreter to build a new symbolic execution engine, which requires significant efforts in implementation and maintenance. Such JS-specific symbolic engines have not demonstrated the effectiveness and efficiency that warrants wide adoption [27].

In this paper, we introduce a new approach to applying concolic testing to JS scripts in-situ, i.e., JS scripts are executed in their native environments as part of concolic execution and test

2640-7574/23/$31.00 ©2023 IEEE
DOI 10.1109/SANER56733.2023.00031

cases generated are directly replayed in these environments. We have implemented this approach in the context of Node.js and its V8 JS engine. As a JS script is executed on Node.js, its binary-level execution trace is captured and later analyzed through symbolic execution for test generation. This brings the power of binary-level concolic testing to JavaScript. We have evaluated the effectiveness and efficiency of this approach through application to 180 Node.js libraries with heavy use of the string operations. For 85% of the libraries, it achieved the statement coverage between 75% and 100% and for 61% of the libraries, it achieves the statement coverage between 85% and 100%, which is a close match in coverage with the hand-crafted unit test suites by the developers in their NPM distributions. This shows our approach can help reduce the efforts needed for developing unit test suites. Our approach has detected many exceptions in these libraries. We analyzed the exception reports for 12 representative libraries, and found 6 clear-cut bugs, 4 of which are previously undetected. The bug reports and patches that we filed for these bugs have been accepted by the library developers on GitHub. This shows that our approach can detect bugs missed by handcrafted test suites.

II. Background

A. Concolic Execution

Symbolic execution exercises a program under test with symbolic inputs, which can potentially lead to path explosion, i.e., too many feasible program paths to be explored efficiently. One effective technique to cope with path explosion is concolic execution, which integrates concrete and symbolic execution. It uses symbolic execution to only explore the branches along a concrete execution path of the program under test, therefore, narrowing down the path exploration space.

To enable concolic execution of JS scripts, we build on CRETE, a binary-level concolic testing framework [24]. CRETE features an open and highly extensible architecture allowing easy integration of concrete execution front-ends and symbolic execution engine back-ends.

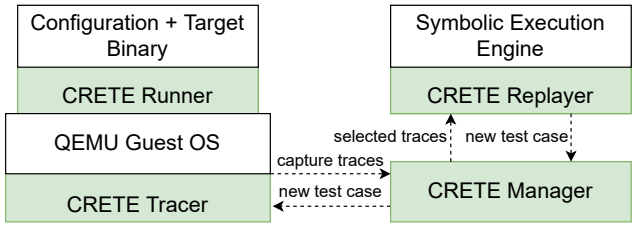


Fig. 1: Architecture of CRETE

As shown in Figure 1, CRETE uses a configuration file to mark symbolic and concrete inputs in the CRETE runner. As the target program is concretely executed in a modified QEMU virtual machine [28], the CRETE tracer, a QEMU extension, captures concrete execution traces. These traces are in the form of LLVM bytecode augmented to indicate the execution paths induced by the concrete inputs [17]. If a path contains a symbolic variable marked in the configuration file, CRETE feeds the captured trace of the path to its symbolic execution engine (in this case KLEE [18]), to run it symbolically via CRETE replayer. CRETE extends KLEE to avoid forking unnecessary states and generates test cases only for feasible branches confined by concrete traces. This results in fewer paths exercised symbolically. CRETE uses a Dynamic Taint Analysis (DTA) algorithm to implement selective tracing [29]. It only captures the execution traces relevant to the marked symbolic values using DTA. CRETE uses tainted memories to represent memories relevant to the variables initially marked as symbolic. For example, if variable "a" is marked as symbolic, when there is an assignment operation involving "a", such as "b=a", the memory slot that "b" possesses is also marked as symbolic. So CRETE will capture any execution trace involving memory slots of "a" and "b". CRETE provides two helper interface functions: `crete_make_symbolic` and `crete_start_tracing` to allow users to mark symbolic variables and initiate tracing of concrete execution. We leverage these interface functions to implement our approach.

B. Node.js Runtime and V8 JS Engine

1) Node.js Runtime: Node.js is an open-source, cross-platform JS runtime environment. It builds around the V8 JS engine and enables high-performance execution of JavaScript. Node.js provides a broad set of asynchronous I/O primitives to the application, which enables it to run unblocked. Node.js allows extensions to its functionalities through `addon` libraries. Such libraries are typically written in C/C++ and can be loaded into Node.js as ordinary Node.js modules using `require()` statements in JavaScript.

2) V8 JS Engine: V8 is Google's high-performance JS and WebAssembly engine [2]. V8 can run standalone or can be embedded in C++ applications such as Node.js and Chrome. As shown in Figure 2, V8 supports two modes for executing a JS script: (1) *interpreted mode* where the JS bytecode [30]

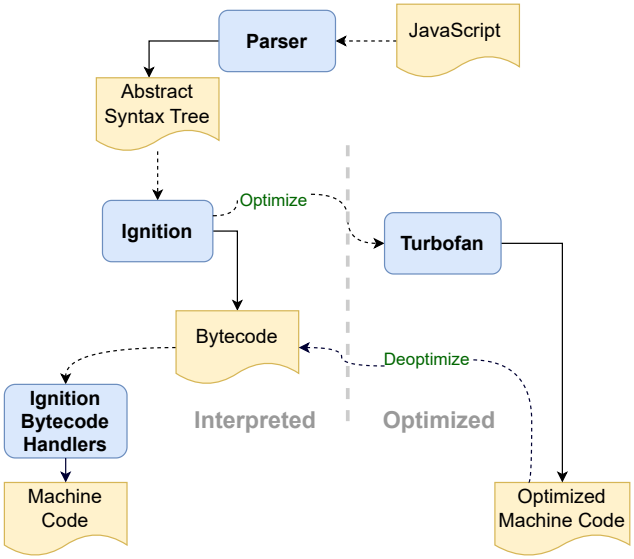


Fig. 2: How V8 runs a JS Script: Interpreted vs Optimized
translated from the JS script is interpreted by its interpreter,

Ignition [31]; (2) *optimized just-in-time compilation mode* where the bytecode is compiled by V8 engine into optimized machine code using its just-in-time compiler, Turbofan [32], and then executed on the target machine. As Ignition interprets a bytecode statement, it invokes the corresponding bytecode handler for this statement that is pre-compiled to the machine code of the target host. If a piece of bytecode is being interpreted repeatedly, the Ignition interpreter may decide that it deserves further optimization. It sends this piece of bytecode and its runtime information from the prior interpretation to Turbofan. Turbofan will then analyze the bytecode and its runtime information to generate further optimized machine code that is then executed in place of the bytecode.

Builtin functions in V8 are intrinsic functions that handle common operations without the need to invoke the optimizing compiler. They are designed to provide internal functionality, or to implement the functions of builtin objects in JavaScript such as `String.Prototype` and `String.Map`. In V8, these builtin functions are implemented in CodeStubAssembler (CSA). CSA provides efficient low-level functionality that is very close to the assembly language, but also offers an extensive library of higher-level functionality. For example, CSA as part of V8's builtins can load data from a specified address, and it can modify the internal data of JavaScript objects [33]. Ignition's bytecode handlers are also implemented in CSA. A key advantage of CSA is that it makes V8's builtin functions platform-independent and those builtin functions are compiled into the binaries for a target platform by V8's unified code generation as shown in Figure 3. CSA allows us to create

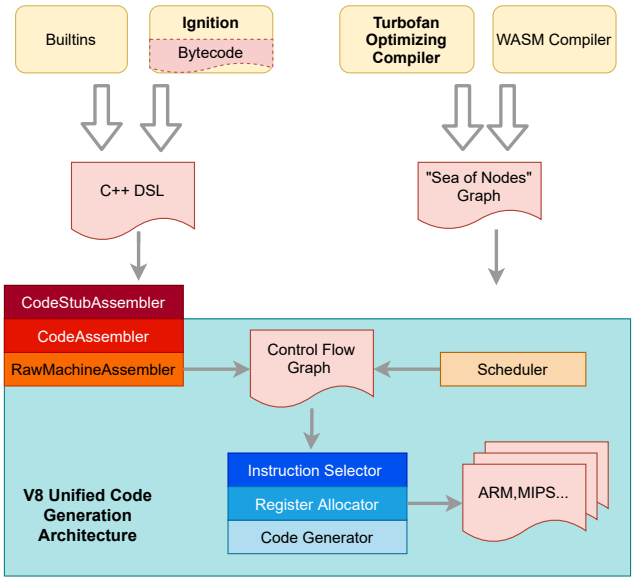


Fig. 3: V8's Unified Code Generation

new V8 builtin functions to extend V8's functionality [32]. We leverage this feature to integrate concolic execution into the V8 engine.

III. Our Approach

A. Overview

JavaScript, as one of the most popular scripting languages for both client side and server-side applications, is often deeply embedded in its execution platform, e.g., web browsers and Node.js runtime. Although taking a JS script out of its native environment and analyzing it in an artificial test environment through modeling would make the analysis more tractable [34], the analysis often becomes less accurate. Test cases generated are not able to fully reflect realistic use cases and can only represent part of the use cases that are accurately modeled [35], and bugs detected may also be false positives [36]. Thus, it is strongly desirable to analyze a JS script in its native environment under its normal usages.

Our approach conducts concolic testing on JS scripts in-situ, as illustrated in Figure 4. The concrete execution step of concolic testing as indicated by the dashed box on top is conducted in the native execution environment for JS scripts, where the trace of this concrete execution is captured. The trace is then analyzed in the symbolic execution step of concolic testing to generate test cases and these test cases are then fed back into the native concrete execution to drive further test case generation.

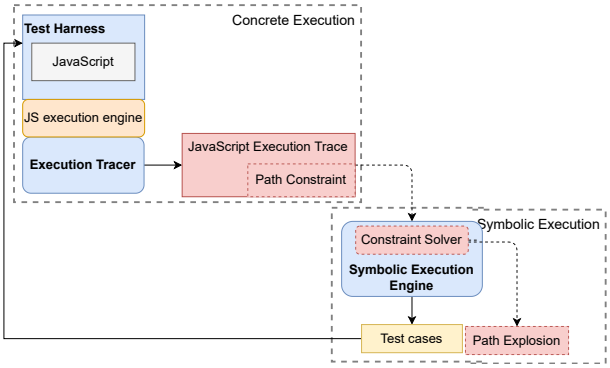


Fig. 4: Workflow for Concolic Testing of JavaScript

Central to our approach is the quality of the captured concrete execution traces of JS scripts in terms of correctness and precision. If the traces captured are incorrect, the test cases generated in symbolic execution will often be misguided, thus not effective. On the other hand, if the traces captured are not concise, they are often unnecessarily complex and lead to path explosion in symbolic execution, thus not efficient. Therefore, while developing our approach, we focus on how to capture the concrete execution trace of a JS script under test from its native execution environment so that the captured trace is both correct and concise. To obtain such traces, we must address two major challenges as follows:

- *Sheer complexities of native execution environments*The embedding environments for JS scripts, web browsers or Node.js, are often quite complex, not only the runtimes themselves, but with their numerous extensions available.
- *JS scripts are heavily optimized.* Both client-side and server-side JS scripts are often optimized just-in-time to

achieve the best performance. Such optimizations tend to obfuscate the execution flows of these JS scripts [37].

Due to the popularity of the Node.js runtime and its embedded V8 JS engine, we address the above challenges in this context. The solutions are readily generalized to other JS runtimes and engines. We have explored two methods for tracing the concrete execution of JS scripts running in the Node.js runtime as follows:

- *Shallow Integration of Tracing in Node.js.* Tracing of the concrete execution of a JS script is invoked within Node.js, but outside V8. The V8 engine is treated as a black box.
- *Deep Integration of Tracing in V8.* Tracing of the concrete execution is invoked inside V8; therefore, irrelevant parts of Node.js are not traced.

B. Shallow Integration of Tracing in Node.js

As shown in Figure 5, in order to use concolic execution to test a JS script, we need to extract the execution trace of this script as it is running on Node.js with an execution tracer, and then feed the execution trace to a symbolic execution engine to generate test cases. `Addons` in Node.js are dynamically-linked

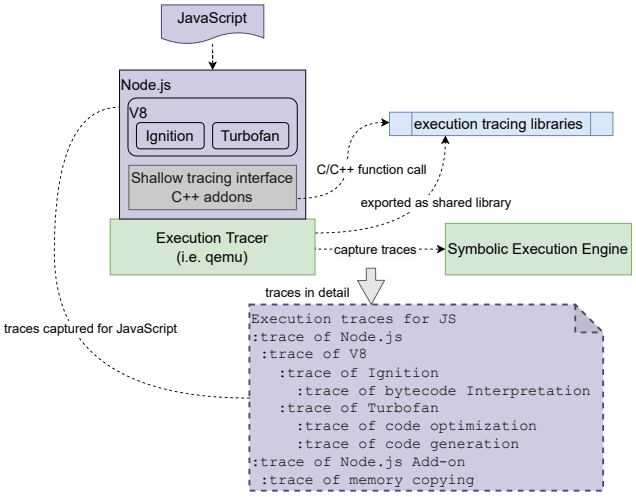


Fig. 5: Shallow Integration of Tracing in Node.js

shared libraries written in C++. This `addons` feature offers an interface between the JavaScript and C/C++ libraries. A library of execution tracers for concolic testing can be made available to the JS script as Node.js modules by leveraging the `addons` feature. Such a library needs to support two general functions: `make_symbolic` and `start_tracing` respectively. The `make_symbolic` function allows us to mark the variables as symbolic in the execution. The `start_tracing` function allows us to take control of the underlying execution tracer so that we can start tracing for symbolic execution when necessary. We use this library to initiate concolic execution for a JS script under test, which is typically done in the test harness to avoid modifications to the JS script itself. This initiation involves setting symbolic variables and informing the execution tracer of when to trace.

As shallow integration of tracing is invoked in Node.js which builds around V8, it has the disadvantage of capturing overly complicated execution traces. The execution tracer, e.g., the CRETE tracer in QEMU, treats Node.js as a whole binary program and captures all of its traces once tracing starts. Furthermore, V8 includes a JavaScript interpreter (Ignition) and a JavaScript just-in-time compiler (Turbofan). Hence, when JavaScript runs on top of Node.js, the execution tracer will capture the execution traces of the entire Node.js, which includes not only traces of the JS script under test, but also traces of Ignition, Turbofan, other parts of V8 and Node.js. The resulting trace is often massive and contains unnecessary execution trace segments. After feeding it to the symbolic execution engine, the engine essentially analyzes the JS script under test and all parts of Node.js and V8 that are involved. This may cause path explosion for symbolic execution. However, such integration of tracing for concolic execution using Node.js `addons` has the advantage of simplicity, i.e., requiring no modification to Node.js and particularly the V8 engine. It is our baseline tracing method to enabling concolic execution for JavaScript.

C. Deep Integration of Tracing in V8

The part of an execution trace that is of the highest relevancy to test case generation using symbolic execution is the binary code that is directly corresponding to the bytecode of JS script under test. Therefore, the best place to trace such binary code is inside the V8 engine. As shown in Figure 6, for deep integration of tracing, we move the interface for interacting with the execution tracer from the Node.js using `C++ addons` into the V8 engine using CSA runtime builtin functions. This interface allows us to only capture the execution traces representing the interpretation of JS bytecode instead of the execution traces of the entire Node.js captured by shallow integration in Figure 5.

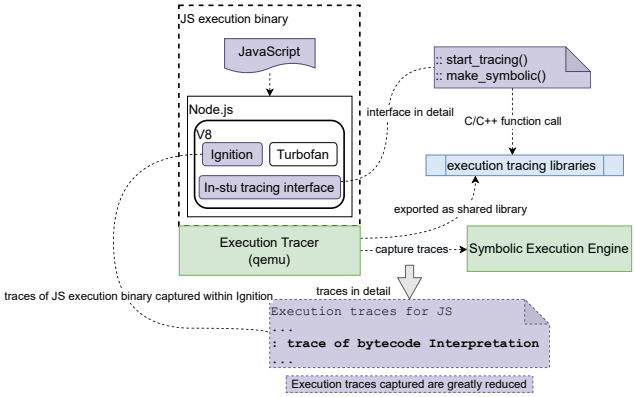


Fig. 6: Deep Integration of Tracing in V8

JS bytecode interpretation happens in V8's Ignition interpreter. As shown in Figure 7, for each JS statement in bytecode, there is a corresponding bytecode handler in Ignition for its interpretation [38]. Ignition bytecode handlers are compiled at V8 build time and embedded into the binary. Interpretation of JS bytecode means that the bytecode handlers themselves are executed. Hence, in order to get an execution

trace that closely represents JS bytecode, we defer tracing till the interpretation of JS bytecode starts, using deep integration of tracing. More specifically, this deep tracing interface captures traces of the execution of Ignition bytecode handlers during interpretation, which closely matches JS bytecode. This way we also avoid capturing the execution traces of the code generation and the optimization in Turbofan. This process of deep integration of tracing is illustrated in the green dashed box of Figure 7. On the contrary, the shallow integration of tracing with Node.js `addons` will capture the whole execution traces for every component as shown in Figure 7. Thus,

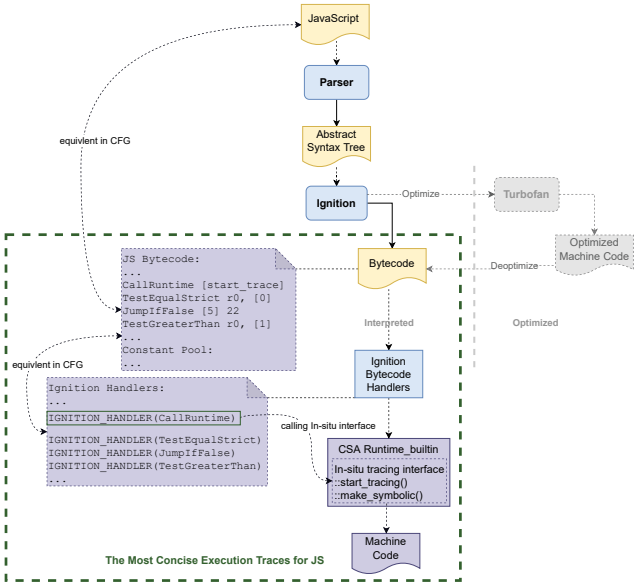


Fig. 7: How Deep Integration of Tracing Captures the Most Concise Execution Traces

our deep tracing interface embedded in V8 can reduce the problem of path explosion when applying symbolic execution on JavaScript by having a precise execution trace that closely matches the JS bytecode.

IV. Implementation

A. Overview

In our implementation, we use CRETE as our concolic execution engine. CRETE provides two interface functions for accessing its execution tracer: `crete_start_tracing` and `crete_make_symbolic`. Through these functions, developers can gain control over when to start tracing and what to capture through the execution tracer. In order to trace the JS library under test, we expose CRETE's tracing control interfaces to the JS script. Our implementation of shallow tracing is to achieve this through Node.js `addons`. We implement a new `addon` library in C++, which is later loaded into the Node.js runtime during JS script execution. This `addon` library wraps around the CRETE's tracing control interfaces and provides them to the JS script running on Node.js. This implementation requires no modification on Node.js, but only introducing a new `addon` library for tracing control. The JS script under test can invoke the tracing control library as it invokes any other Node.js modules. Our shallow tracing implementation contains 527 lines of C++. This implementation treats the V8 JS engine as a whole; thus, in addition to traces of the JS script, it may also capture extensive traces from the V8 engine.

Our implementation of the deep tracing is to integrate the tracing control interface into the V8 JS engine to gain more precise control over tracing. We achieve the implementation by extending V8 builtin functions to integrate the tracing control interface for symbolic execution in V8. V8 builtin functions allow developers to extend the internal functionalities of the V8 engine. These builtin functions are implemented in V8's `CodeStubAssembler` and provide accesses to CRETE's tracing interface. They are compiled into binary by V8's unified code generation and integrated into the Ignition interpreter. The JS script under test can then invoke CRETE's tracing interface through these builtin functions. This deep tracing implementation provides better control for tracing the JS script by only tracing the bytecode handlers within V8 which are corresponding to the bytecode of the JS script, but not other parts of V8. V8's mechanism of builtin functions allows precise accesses to the bytecode handlers. Our deep tracing implementation contains 2041 lines of C++, 463 lines of JavaScript and 178 lines of bash.

Also note that everything in JavaScript is represented as an object. As we make inputs to the JS script symbolic, we must make sure that the objects that we set symbolic remains valid objects during symbolic execution.

B. Shallow Tracing Interface as C++ Addons

Figure 8 illustrates our implementation of the shallow tracing interface as a Node.js `addon` library, which supports two tracing control functions: `start_tracing` and `make_symbolic`. Node.js provides a standard way of implementing an `addon` library in C++. The `addon` library can be loaded as a Node.js module using `require()` statements in the JS script. The two tracing control interface functions

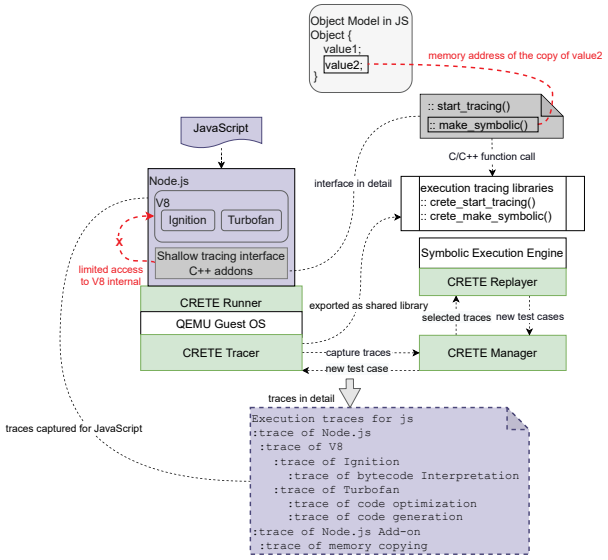


Fig. 8: Implementation of Shallow Tracing using Addons

are first exported from CRETE execution tracer and can later be invoked from the JS script to mark symbolic variables and initiate tracing through the `addon` library. This is done in the test harness of the JS script under test so that the JS script itself is not modified. Although the `addons` library, as part of Node.js, offer a bridge between JavaScript and C/C++ libraries, it has the following drawbacks in tracing for concolic execution:

- Separate address spaces: As shown in Figure 9, the `addon` library has a different address space from V8 while V8 allocates JS variables within its own address space as storage cells [39]. Therefore, when a JS script invokes the `addon` library in Node.js, it involves memory translations in between. Due to the fact that CRETE uses *Dynamic Taint Analysis*, which will capture relevant traces of memory translations related to symbolic variables, symbolic execution may get lost among memory address translations between the `addon` library and V8.

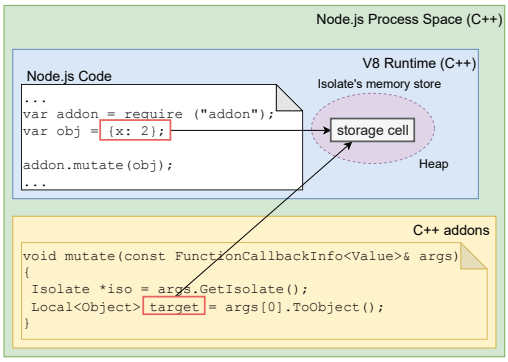


Fig. 9: Memory System for C++ Addons

- Limited V8 internal access: The `addon` library has limited access to V8 internals. Thus, when implementing `make_symbolic`, the `addon` library cannot access the runtime memory address on heap for a variable in the JS script, but a copy of its value. We can only get the memory address of this copy. As a result, the execution traces CRETE captured may contain irrelevant traces of underlying value copying during the execution of the JS script, thus, it is not a close match to the JS bytecode.
- Tracing inside Node.js but outside of V8: Through the `addon` library, tracing is initiated inside Node.js. CRETE tracer will treat V8 as a black box binary and trace its entire execution including the execution of Turbofan and other Node.js modules after the tracing starts. Such tracing captures the entire execution trace that contains the redundant execution traces indicated by line 5 to 9 listed below.

```
1  :trace of Node.js
2   :trace of V8
3     :trace of Ignition
4       :trace of bytecode Interpretation
5     :trace of Turbofan
6       :trace of code optimization
7       :trace of code generation
8   :trace of C++ addon
9    :trace of memory translation
```

The parts of the trace closely corresponding to the JS script are indicated by line 2 to 4.

C. Deep Tracing Interface as V8 Builtins

Figure 10 illustrates how we implement the deep tracing interface of `start_tracing` and `make_symbolic` as builtin functions, which reside inside the V8 engine and have access to the JS interpretation by Ignition. (We have explained the technical feasibility in Section II-B2, *V8 JS Engine*). V8

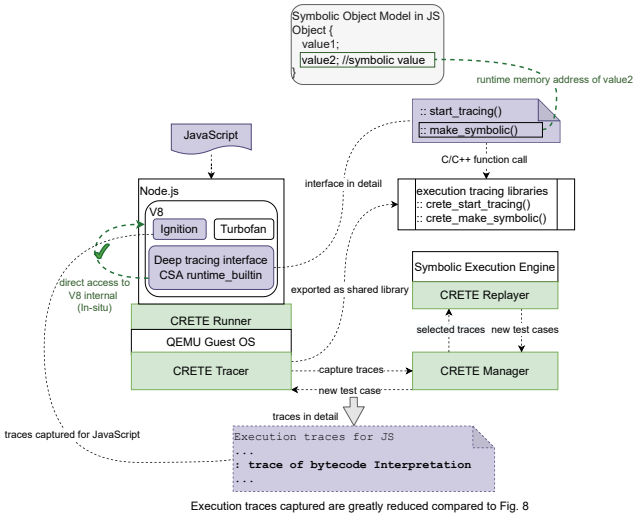


Fig. 10: Deep Tracing Interface in V8

allows developers to extend the set of builtin functions with new ones written in `CodeStubAssembler`. The new builtin functions are compiled into the binary of the target host by the V8's unified code generation and directly embedded into V8. Implementing the tracing interface as V8 builtin functions enables the control of CRETE execution tracer from within V8. Hence, we are able to defer tracing till JavaScript bytecode interpretation starts. This way we can keep the captured execution trace confined within the JavaScript interpretation. What's more, builtin functions have access to V8 internals and can be called from Ignition. Therefore, it is able to get the runtime address of an object or one of its fields. V8 runtime builtin functions can be called directly from JavaScript through a %-prefix with the flag `--allow-natives-syntax` as shown in line 3 and line 4 of Listing 1. The deep tracing interface allows precise tracing of the JS bytecode execution by tracing Ignition bytecode handlers. To avoid tracing of just-in-time code generation and optimization in Turbofan, we turn off Turban while tracing.

D. Symbolic JS Object for V8

In this sub-section, we explain how we make a JS Object symbolic for V8. V8 builtin functions allow us to access the runtime memory address of a JS object, which is allocated on heap when V8 creates a `HeapObject`. For safety reason, a `HeapObject` is pointed to by a pointer inside a handle in V8's C++ implementation [40]. As shown in Figure 11, a `String` object is a `HeapObject` that is allocated on

the heap during runtime. Since CRETE captures execution traces based on the memory addresses of the initial variables set as symbolic. We set the memory address that holds the actual value for the `String` allocated at runtime as symbolic. Therefore, the trace that CRETE captures is relevant to this `String` object. In V8's implementation, we are given the interfaces to use the `Handle` to access objects in JavaScript. Figure 11 shows how we get the memory address of the value in the `String` on heap using `Handle`. By setting symbolic inputs this way, we only set the memory address containing the actual value of an `Object` symbolic during symbolic execution to explore branches related to the value. It does not mark memory of other fields of the `Object` symbolic; otherwise, the object may be invalid. We mainly focus on JavaScript's `String` type because strings are popular inputs to JS scripts and making string variables symbolic leads to many valuable test cases.

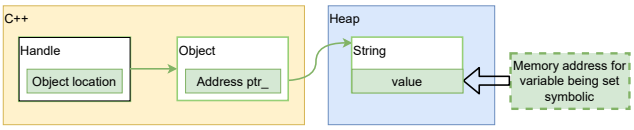


Fig. 11: V8 Object Memory Model

We encountered four cases when attempting to retrieve the memory address of the actual value of the `String` object for symbolic execution [41], they are listed as below:

- SeqOneByteString: The simplest form, containing a few header fields and then the string's bytes (which are not UTF-8 encoded and can only contain characters among the first 256 unicode code points).
- SeqTwoByteString: Similar form, but with two bytes for each character (using surrogate pairs to represent unicode characters that cannot be represented in two bytes).
- SlicedString: A substring of some other string, containing a pointer to the "parent" string and an offset and length.
- ConsString: The result of concatenating two strings (if over a certain size), containing pointers to both strings (which may themselves be any types of strings).

Listing 1 and Listing 2 show an example JS script and its bytecode during interpretation. CRETE only captures the trace related to the runtime memory address of the actual value of `str_var`, which is a `String` object in V8. The runtime address is `0x34ecf6d42849` as shown at line 26 of Listing 2. The actual value stored in this runtime address is loaded at line 5 of Listing 2 and this runtime address is later marked as symbolic at line 7. After `StartTracing` is called at line 8, CRETE captures the traces for all bytecode related to the symbolic runtime address, which are highlighted by the underscores in Listing 2, as the concrete execution trace. The captured trace also preserves all constraints corresponding to the JS script of Listing 1. Thus, the traces captured with our method are concise and accurate for symbolic execution.

```
1  var str_var = "init";
2
3  %MakeSymbolic(str_var);
4  %StartTracing();
5
6  if( str_var === "tests")
7      return "tests";
8
9  if( str_var > "tests1"){
10     return "tests1";
11 }else{
12     return "tests2"
13 }
```

Listing 1: A Simple Example of JavaScript and Calling Convention of In-Situ Tracing Interfaces

```
1  Parameter count 6
2  Frame size 8
3    0x40b8ec2c9a StackCheck
4    0x40b8ec2c9b LdaConstant [0]
5    0x40b8ec2c9d Star r0
6    0x40b8ec2c9f CallRuntime [MakeSymbolic],r0-r0
7    0x40b8ec2ca4 CallRuntime [StartTracing]
8    0x40b8ec2ca9 LdaConstant [1]
9    0x40b8ec2cab TestEqualStrict r0,[0]
10   0x40b8ec2cae JumpIfFalse [5](0x40b8ec2cb3)
11   0x40b8ec2cb0 LdaConstant [1]
12   0x40b8ec2cb2 Return
13   0x40b8ec2cb3 LdaConstant [2]
14   0x40b8ec2cb5 TestGreaterThan r0,[1]
15   0x40b8ec2cb8 JumpIfFalse [5](0x40b8ec2cbd)
16   0x40b8ec2cba LdaConstant [2]
17   0x40b8ec2cbc Return
18   0x40b8ec2cbd LdaConstant [3]
19   0x40b8ec2cbf Return
20   0x40b8ec2cc0 LdaUndefined
21   0x40b8ec2cc1 Return
22 Constant pool (size = 4)
23 - map: 0x01eccde023c1 <Map>
24 - length: 4
25         0: 0x34ecf6d42849 <String[4]: init>
26         1: 0x0040b8ec2949 <String[5]: tests>
27         2: 0x0040b8ec2969 <String[6]: tests1>
28         3: 0x0040b8ec2989 <String[6]: tests2>
```

Listing 2: Bytecode for JS Script in Listing 1

V. Evaluation

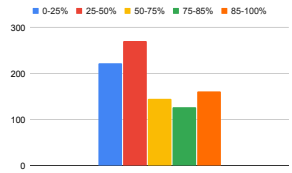
A. Overview

For our evaluation, we target Node.js libraries that are available on NPM. We install these libraries through NPM and their source code is also downloaded so we can access their unit test suites for comparison purposes. We apply our approach to in-situ concolic testing, both shallow tracing and deep tracing, on these libraries, and compare them in terms of performance. We have also evaluated the code coverage achieved by our automatically generated test cases with coverage achieved by hand-crafted unit test suites of these libraries as reference. This evaluation is carried out on an Ubuntu OS Version 18.04 with 4-core Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz and 16G memory.

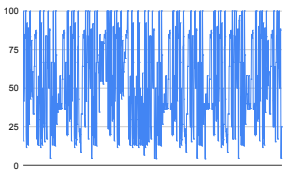
In order to apply our approach to these libraries, we built a test harness to systematically exercise all exported (public) methods in a given library with arguments whose type is `String`. The seed test cases are generated randomly within the test harness. We implemented an automation pipeline that

helps set up the concolic testing environment in CRETE for each Node.js library automatically. With the test harness and automation pipeline we can set up concolic testing for Node.js libraries conveniently and have applied our approach to 995 Node.js libraries which include approximately 9000 `JS` files. Our current study focuses on string-intensive libraries due to their popularity in Node.js applications. We randomly pull libraries from NPM. If the majority of a library's functions process strings, we select it. We set string-type parameters symbolic and non-string parameters to random concrete values in the test harness for each library. If a library contains no exported function with string-type parameters, we skip it.

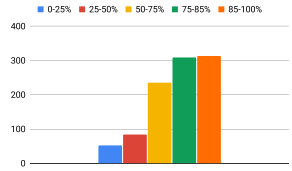
The overall statement coverage on all 995 Node.js libraries for shallow tracing and deep tracing is shown in Figure 12. Figure 12d and Figure 12b show that deep tracing via V8 builtin functions performs significantly better than shallow tracing via Node.js `addons` in terms of statement coverage. The darker shadow between 75% and 100% in Figure 12d indicates that more libraries achieved the coverage between 75% and 100% with deep tracing. Figure 12a and Figure 12c show the exact number of libraries in each coverage range.



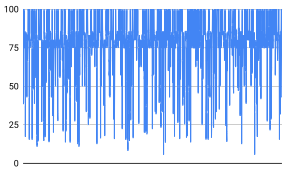
(a) Shallow Tracing: Number of libs in each coverage range



(b) Shallow Tracing: Statement coverage distribution



(c) Deep Tracing: Number of libs in each coverage range



(d) Deep Tracing: Statement coverage distribution

Fig. 12: Coverage on All 995 Node.js Libraries

Due to the sheer volume of libraries and JS files, we randomly select 180 libraries to conduct a deep-dive analysis of coverage achieved by shallow tracing and deep tracing methods respectively. Coverage for all JavaScript libraries are calculated using *istanbul*, a popular JS coverage tools used by V8 [42] and compatible with most JavaScript testing frameworks, e.g., Mocha [43] and Node-Tap [44]. 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

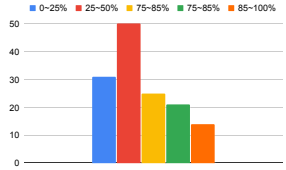
Metric	Range	Average
Line of Code	[93, 16910]	1687
Weekly Downloads	[3, 37491350]	9552965
Dependencies	[3, 18154]	282

TABLE I: Demographics for Libraries under Test

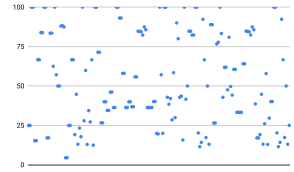
of the selected libraries. The LoC (lines of code) for a library under test is calculated with *github-loc* [45]. The number of weekly downloads of a library under test is calculated with *npm-stats-api* [46]. The number of dependencies is the number of dependent libraries that the library under test has. We calculated it with *dependent-counts* [47].

B. Results from Shallow Tracing Using Node.js Addons

For evaluation of concolic testing with shallow tracing of JavaScript libraries via the Node.js `addon` method, we wrap the 180 randomly selected libraries with our test harness, in which the shallow tracing is invoked through the tracing control interface made available via the Node.js `addon`. As shown in Figure 13a, the statement coverage achieved between 85% and 100% only accounts for 9.93% of the libraries under test, the coverage between 75% and 85% accounts for 14.89% of the libraries, the coverage between 50% and 75% accounts for 17.73% of the libraries, the coverage between 25% and 50% accounts for 35.46% of the libraries, and the coverage below 25% accounts for 21.99% of the libraries. We can see the overall performance of shallow tracing by looking at Figure 13b where most of the dots representing the coverage appear below the line of 75%. As we analyzed more libraries, the proportion of libraries that fall into a higher coverage range do not seem to improve, indicated by a mostly flat line in Figure 14, which shows the average coverage growth trends when the number of libraries grows. It can be observed from Figure 12a and Figure 13a that the overall coverage on 995 libraries closely resembles that of 180 representative libraries randomly selected.



(a) Number of libraries in each coverage range



(b) Statement coverage distribution

Fig. 13: Coverage Achieved by Shallow Tracing

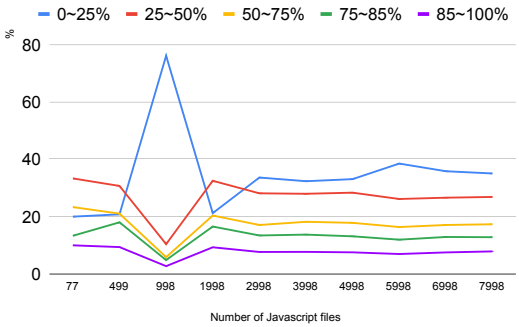
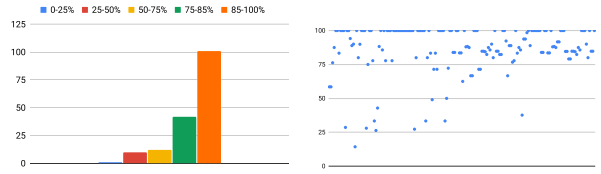


Fig. 14: Coverage Growth Trend with Shallow Tracing

C. Results from Deep Tracing with V8 Builtins

To evaluate the method of deep tracing with V8 builtins, we apply it to the same set of 180 Node.js libraries. For each



(a) Number of Libraries in Each Coverage Range (b) Statement Coverage Distribution

Fig. 15: Coverage Achieved by Deep Tracing

library, in its test harness, we invoke deep tracing through the tracing control interface made available via the V8 builtins. We can see an overview of the deep tracing method's performance in Figure 15b. Most of the dots indicating the coverage appear above the line of 75%. Only one library 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.

As shown in Figure 15a, it is clear that the deep tracing method is able to achieve the coverage between 85% and 100% for most libraries indicated by the right most bar. This performance gain comes from the ability of being able to run symbolic analysis on a more precisely captured trace that closely corresponds to the JS bytecode, which has been explained in detail in Sections IV-C and IV-D. It can be observed from Figure 12c and Figure 15a that the overall coverage on 995 libraries closely resembles that of 180 representative libraries randomly selected.

D. Comparisons

1) Test Coverage Achieved by NPM Test Suites: A systematic investigation on test coverage of hand-craft test suites in NPM [48] is illustrated in Figure 16. The blue line (the lower line) represents statement coverage achieved by test suites found in the packages released in NPM registry where only 4.2% of the libraries in the evaluation set have statement coverage above 80%, 6.0% of the libraries have coverage above 20%, and 6.6% of the libraries contain tests with coverage barely above zero. This result shows that most libraries do not have unit tests at all in their releases in NPM. Only a small number of the libraries has high-quality unit tests. The green line (the upper line) represents the tests included in the latest commit of the master branch of the library repositories. We can see that the number of libraries in each coverage range has improved. However, those libraries that have coverage in the range of 80% to 100% are still inadequate. Our method can automatically achieve similar and even better coverage for JS library than the manually crafted test suites by its developers. It can significantly reduce the efforts in equipping these libraries with high-quality unit tests.

2) Performance Comparison between Shallow and Deep Tracing: For comparison, it can be observed from Figure 17a and Figure 17b that the number of libraries achieving code

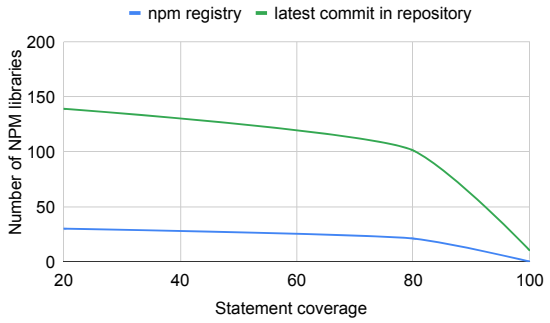
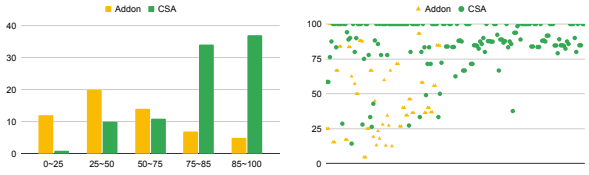


Fig. 16: Coverage by Hand-Crafted NPM Test Suites



(a) Number of Libraries in Each Coverage Range (b) Statement Coverage Distribution

Fig. 17: Coverage Comparison: Shallow vs. Deep Tracing

coverage above 85% using deep tracing is significantly higher than that of shallow tracing. And the number of libraries achieving code coverage between coverage 75% and 85% is also higher. This indicates that the deep tracing method has the ability to achieve higher coverage in JavaScript libraries at the cost of extending the V8 engine with new builtins.

3) Comparison with Related Work: We have compared our approach with an existing tool, ExpoSE [49]. ExpoSE has been evaluated on 4 JS libraries shown in Figure 18. We selected the same libraries for comparison. ExpoSE specifically targets solving regular expression problems for its symbolic execution engine JALANGI and detected a new bug in the "minimist" library. Our method of deep tracing via V8 builtin achieved better coverage consistently. This comparison partially reflects our method's ability in achieving higher coverage.

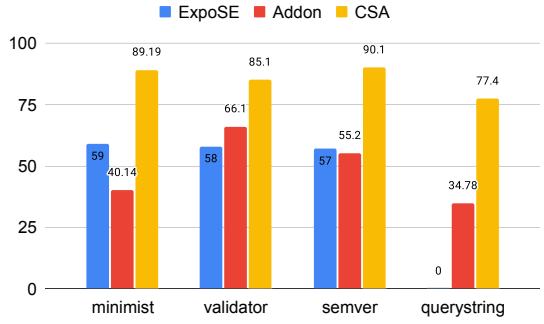


Fig. 18: Comparison with ExpoSE

E. Bugs and Exceptions

For the 180 libraries we selected for evaluation, on average, 4 exceptions are thrown per library on the generated tests. We had time to carefully analyze 12 libraries for their exceptions.

In total, 9 distinct exceptions are encountered for the 12 libraries. Among those exceptions, we identified 6 as clear-cut bugs: 2 are previously known bugs that have been fixed while 4 are previously unknown. After we filed these bugs on Github, they have been accepted and patched by their developers. The bugs we filed are all due to unhandled exceptions.

Node.js lib	Bugs	Known
benchmarkify	No boundary check for empty string	No
msgpack5	No NULL check for function args	No
is-regex	Unhandled input syntax error	No
validator	Mishandled country code	No
chalk	Deprecated constructor invoked	Yes
stringify	Incorrect parsing of separators	Yes

TABLE II: Bugs Detected in 12 NPM Libraries

Table II 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 it leads the execution to an error `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.

F. Discussions and limitations

The reason why our approach achieves the results above is that deep tracing via V8 builtin gets a most concise execution trace which is a close match to JavaScript bytecode. However, some bytecode might later become hot and is sent to TurboFan's optimizing compiler [37]. Under such circumstances, our approach becomes less effective due to the optimization conducted by Turbofan and will require new filters on tracing that are aware of the optimization.

Our implementation is based on CRETE which uses QEMU as its tracing platform [28]. This makes it less portable to browser-based JavaScript. We strive to lift this limitation. JavaScript execution in Node.js works in an event loop which includes a main thread and worker threads. CRETE captures concrete traces from a process, unless instructed otherwise, CRETE captures all binary code from the process, multi-threaded or not. Such a naïve application may cause path explosion in symbolic analysis. In our study, we targeted unit testing of Node.js libraries. Our test harness separated functions in a NPM library and ran each function individually. The libraries we used do not have async or callback functions so traces are restricted to one thread. Conceptually, our approach can run and test a multi-threaded JS program since CRETE captures traces from all threads within a process. However, additional algorithms are needed to handle multi-threaded executions to prevent path explosion, which is not the focus of this paper.

VI. Related Work

Our approach is closely related to work on symbolic execution for JavaScript. Commonly targeted JS scripts include the browser-based ones and those running on browser-less runtimes, e.g., Node.js. Most of symbolic execution methods for JavaScript required building application-specific symbolic execution engines or significantly modifying JavaScript execution engines to apply symbolic execution. As an example of symbolic execution targeting browser-based JavaScript, SymJS is a framework for testing client-side JS script [50]. It modifies Rhino JS engine for symbolic execution [51]. 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 [52]. COSETTE is another symbolic execution engine for JavaScript using an intermediate representation, namely JSIL, translated from JavaScript [53]. 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 [49]. Kudzu targeted AJAX applications by implementing a dynamic symbolic interpreter that takes a simplified intermediate language for JavaScript [26]. To the best of our knowledge, no symbolic execution framework for JavaScript has directly utilized existing powerful binary-level concolic execution engines [54].

Another related approach to testing JavaScript is fuzzing. There are a few fuzzers for JS, e.g., *jsfuzz* [55] and *js-fuzz* [56], which are largely based on the fuzzing logic of AFL (American fuzzy lop) [57] and re-implemented it for JavaScript. We view fuzzing and symbolic/concolic testing as complementing techniques: fuzzing for broader exploration of JS scripts while symbolic/concolic testing for deeper exploration.

VII. Conclusions and Future Work

In this paper, we have presented a novel approach to in-situ concolic testing of JS scripts. It enables concolic execution for JS scripts in their native environments and can automatically generate test cases that achieve comparable code coverage than manually crafted test suites for Node.js libraries and discovered previously unknown bugs.

We will further extend this approach to support a wider range of JS scripts, e.g., browser-based JS scripts. We will optimize the tracing mechanism, e.g., further reducing the complexities of binary-level traces captured for the JS script under test and subsequently reducing the overheads of symbolic execution and generating more effective test cases. In addition to optimizing the tracing mechanism, we aim to remove the dependency on the QEMU virtual machine.

Acknowledgments

This research received financial support in part from National Science Foundation (Grant #: 1908571).

REFERENCES

[1] "Node.js," https://nodejs.org/en/, 2021.
[2] "v8," https://v8.dev/, 2021.
[3] "Npm," https://www.npmjs.com/, 2021.
[4] I. Koishybayev and A. Kapravelos, "Mininode: Reducing the attack surface of node.js applications," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. San Sebastian: USENIX Association, Oct. 2020, pp. 121–134. [Online]. Available: https://www.usenix.org/conference/raid2020/presentation/koishybayev
[5] "Module counts," http://www.modulecounts.com/, 2022.
[6] 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
[7] 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
[8] L. K. Shar and H. B. K. Tan, "Predicting sql injection and cross site scripting vulnerabilities through mining input sanitization patterns," *Information and Software Technology*, vol. 55, no. 10, pp. 1767–1780, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584913000852
[9] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 995–1010. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman
[10] J. C. Davis, C. A. Coghlan, F. Servant, and D. Lee, "The impact of regular expression denial of service (redos) in practice: An empirical study at the ecosystem scale," ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 246–256. [Online]. Available: https://doi.org/10.1145/3236024.3236027
[11] 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.
[12] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why do developers use trivial packages? an empirical case study on npm," 08 2017, pp. 385–395.
[13] 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.
[14] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.
[15] 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.
[16] K. Sen, "Concolic testing," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007, pp. 571–572.
[17] 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.
[18] 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.
[19] 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.
[20] 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.
[21] 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.
[22] 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.
[23] 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.
[24] 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.
[25] S. Bucur, J. Kinder, and G. Candea, "Prototyping symbolic execution engines for interpreted languages," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, 2014, pp. 239–254.
[26] 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.
[27] 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
[28] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX annual technical conference, FREENIX Track*, vol. 41. Califor-nia, USA, 2005, p. 46.
[29] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 317–331.
[30] "Understanding v8's bytecode," https://medium.com/dailyjs/understanding-v8s-bytecode-317d46c94775, 2021.
[31] "Firing up the ignition interpreter," https://v8.dev/blog/ignition-interpreter, 2021.
[32] "Turbofan: A new code generation architecture for v8," https://docs.google.com/presentation/d/1_eLlVzcj94_G4r9j9d_Lj5HRKFnq6jgpuPJtnmIBs88/htmlpresent, 2021.
[33] "Codestubassembler builtins," https://v8.dev/docs/csa-builtins, 2021.
[34] S. Sapra, M. Minea, S. Chaki, A. Gurfinkel, and E. Clarke, "Finding errors in python programs using dynamic symbolic execution," vol. 8254, 11 2013, pp. 283–289.
[35] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler, "The scalable commutativity rule: Designing scalable software for multicore processors," *ACM Trans. Comput. Syst.*, vol. 32, no. 4, jan 2015. [Online]. Available: https://doi.org/10.1145/2699681
[36] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE way to test openflow applications," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, Apr. 2012, pp. 127–140. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/canini
[37] M. Selakovic and M. Pradel, "Performance issues and optimizations in javascript: an empirical study," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 61–72.
[38] "Ignition: V8 interpreter," https://docs.google.com/document/d/11T2CRex9hXxoJwbYqVQ32yIPMh0uouUZLdyrtmMoL44/mobilebasic, 2016.
[39] "How (not) to access v8 memory from a node.js c++ addon's worker thread," https://nodeaddons.com/how-not-to-access-node-js-from-c-worker-threads, 2021.
[40] "V8's object model using well-defined c++," https://docs.google.com/document/d/1_w49sakC1XM1OptjTurBDqO86NE16FH8LwbeUAtrbCo/edit, 2019.
[41] "V8 stringobject," https://v8docs.nodesource.com/node-0.8/d9/d38/classv8_1_1_string_object.html, 2016.
[42] "Istanbul," https://istanbul.js.org/, 2021.
[43] "Mocha: simple, flexible, fun," https://mochajs.org/, 2021.
[44] "Node-tap," https://node-tap.org/, 2021.
[45] "github-loc," https://www.npmjs.com/package/github-loc, Jun. 2021.
[46] "npm-stats-api," https://www.npmjs.com/package/npm-stats-api, Jun. 2021.
[47] "dependent-counts," https://www.npmjs.com/package/dependent-counts, Jun. 2021.

[48] H. Sun, A. Rosà, D. Bonetta, and W. Binder, "Automatically assessing and extending code coverage for npm packages," *arXiv preprint arXiv:2105.06838*, 2021.
[49] 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.
[50] 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.
[51] 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.
[52] 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.
[53] 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.
[54] 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.
[55] "Jsfuzz: coverage-guided fuzz testing for javascript," https://github.com/fuzzitdev/jsfuzz, Jan. 2022.
[56] "js-fuzz," https://github.com/connor4312/js-fuzz, Jan. 2022.
[57] "Afl: American fuzzy lop," http://lcamtuf.coredump.cx/afl/, Jan. 2022.