7-3-2024

Concolic Testing for Scripting Languages

Zhe Li
Portland State University

Concolic Testing for Scripting Languages

by

Zhe Li

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Portland State University
2024

Abstract

Scripting languages, such as JavaScript and Lua, are becoming more and more popular. They are typically easy to learn and use, making them accessible to a wide range of developers, even those with limited programming experience. Lua, for instance, is a lightweight, efficient, and versatile scripting language. It is designed to be easy to integrate into other systems and is often used as an embedded scripting language in larger applications such as NMap, which is a network scanning tool.

As another example, web front-end development with JavaScript (JS) is a popular choice for developers due to its ability to add interactivity to websites. JavaScript has also evolved into a versatile and popular programming language for not only front-end development but a wide range of server-side and client-side applications. With such popularity, there is a great demand for thorough testing of scripting language applications.

We propose to develop a holistic framework for applying concolic testing to applications in scripting languages. Concolic testing synergistically integrates concrete and symbolic execution for test generation, which alleviates the path explosion problem often encountered in symbolic execution by only exploring symbolically along a concrete execution path. Under this framework, scripts are executed in their native execution engines concretely instead of the modeled test environments, these executions are efficiently traced in OS-level virtual machines (VMs) and analyzed in a customized manner within symbolic engines, and new test inputs generated by symbolic engines are fed back into the concrete execution in their native environment to drive new iterations of test generation. As a result, test cases generated reflect realistic usage to the full extent.

First, we present an approach for applying concolic execution on attacking scripts in NMap for Lua language in order to automatically generate lightweight fake versions of the targeted services so that they can fool or slow down the attacking scripts. The behavior of the attacking scripts was captured within their native execution environments for symbolic analysis later. By doing so in an automated and scalable manner, this approach can enable rapid deployment of custom honeyfarms leveraging the results of concolic execution to trick an attacker's script into returning a result chosen by the honeyfarms, making the script unreliable for the use by the attackers.

Second, for JavaScript, we present an 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 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 applications 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.

Third, we present a novel approach to concolic testing of front-end JavaScript web applications based on in-situ concolic testing. This approach leverages widely used JavaScript testing frameworks such as *Jest* and *Puppeteer* and conducts concolic execution on JavaScript functions in web applications for unit testing. The seamless integration of concolic testing with these testing frameworks allows injection of symbolic variables within the native execution context of a JavaScript web function and the precise capture of concrete execution traces of the function under test. Such concise execution traces greatly improve effectiveness and efficiency of the subsequent symbolic analysis for test generation. We have implemented our approach on both *Jest* and *Puppeteer*. The application of our *Jest* implementation on *Metamask*, one of the most popular Crypto wallets, has uncovered 3 bugs and 1 test suite improvement, whose bug reports have been accepted by *Metamask* developers on GitHub. We also applied our *Puppeteer* implementation to 21 Github projects and detected 4 bugs.

At last, we improved how execution traces of scripts are captured concretely and analyzed symbolically. These traces were captured through OS-level VMs and were at the binary level before, which is time-consuming and complex. Symbolic analysis of binary traces was also less efficient than analyzing higher-level traces. We have developed 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 execution tracing and trace translation 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 approach, which only uses a fraction of the time needed by the in-situ approach.

Dedication

To Dad, Mom and Kellen

Acknowledgments

First of all, I would like to express my sincere gratitude to my advisor, Professor Fei Xie. His unwavering encouragement and support have provided me with a conducive environment to concentrate on my research and successfully complete this dissertation. Fei consistently challenged me to cultivate independence as a researcher, foster collaboration, and excel as a mentor. He also facilitated numerous opportunities for me to glean insights from others. His passion for addressing real-world challenges and constructing practical systems has profoundly influenced my doctoral research and will undoubtedly shape my future career.

I extend my heartfelt appreciation to Professor Fang Song, Professor Suresh Singh, and Professor Jingke Li for graciously serving on my dissertation committee. Their invaluable feedback on the dissertation draft and the thought-provoking questions they posed during my defense was instrumental in refining my work.

I am also very grateful to my shepherds, Dr. Bo Chen, Dr. Zhenkun Yang, Dr. Li Lei, and Dr. Kai Cong. Their mentorship played an important role in initiating my Ph.D. research journey. Bo shared important and useful experiences and advice about how to progress through obstacles. Zhenkun generously volunteered in many discussions and provided invaluable insights into progressing my research. Li consistently offered constructive advice that encouraged me to think creatively. Beyond their professional roles, they have been inspirations for me towards life and careers.

I extend my gratitude to my co-author, Wu-chang Feng, for his collaboration and contributions to our work. Additionally, I would like to acknowledge and appreciate the support of my colleagues at PSU: Huan Wu, Li Shi, Yanzhao Wang, Bo Chen, Haifeng Gu, and Bin Lin, whose camaraderie and assistance enriched my Ph.D. experience. They are also great friends to me. A special thanks goes to Rebecca Sexton-Lee, Ella Barrett, Kristine-Anne Sarreal, and all the staff from the Department of Computer Science at PSU for their diligent handling of administrative matters throughout my Ph.D. journey. Their efforts ensured a smooth and efficient academic environment, allowing me to focus on my research endeavors.

Finally, I'd like to thank our dog Sedona. Throughout late nights of studying and writing papers, she was always by my side, offering quiet companionship. Most importantly, I would like to thank my parents for their unconditional love, support, and trust. They gave me the absolute courage to try, fail, and accomplish anything in life. Particularly, many thanks to Kellen McInerny, my husband, for his everlasting love, care and tolerance. He came into my life and went through most of the ups and downs of my Ph.D. study, and was always there for me when things didn't go well, which I would never expect. I am very happy that we did it together.

Table of Contents

List of Tables

List of Figures

1 Introduction

1.1 Overview

Scripting languages are becoming more frequently used in programs and applications because they allow easy and rapid programming at a higher level of abstraction and can be easily embedded into other frameworks. Embedding a script is a very powerful way of configuration because applications can execute any script as an argument that users customize. They are often used for rapid prototyping or for creating proof-of-concept applications. Their ease of use and flexibility make them ideal for quickly testing out ideas. Therefore, approaches to testing that are comprehensive and automated become more crucial. The goal of this dissertation is to improve the reliability of applications using scripting languages by strengthening popular applications of widely used scripting languages individually and by improving state-of-art techniques.

1.1.1 Symbolic Execution for Emerging Scripting Languages at Binary Level

Scripting languages have gained a lot of popularity in recent years, in part because of their flexibility and ease of use. One of the benefits of using these languages is that developers can create their own functionality scripts and share them with others through library managers such as NSE libraries (Nmap Scripting Engine Library) and npm (Node Package Manager). These package managers provide a central repository for developers to publish and distribute their code, making it easy for others to find and use their scripts. This has led to a vast ecosystem of third-party libraries and modules that can be used to extend the functionality of these languages and accelerate the development process. However, it also comes with some security risks. Because anyone can publish code to these package managers, there is a risk that malicious code could be introduced, either intentionally or unintentionally. For example, one security issue that has been identified in Node.js in the past is related to the way it handled HTTP headers. Insufficient testing of the HTTP parser in older versions of Node.js led to the discovery of a vulnerability that could allow an attacker to execute arbitrary code on a target system by sending a specially crafted HTTP request. Therefore, thorough and automated testing of scripts written in such languages becomes important.

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 [60]. Concolic testing is a hybrid verification technique that alleviates path explosion that often bogs down symbolic execution [63]. 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 [88]. Traditional symbolic or concolic execution engines mostly target C/C++, low-level intermediate representation (LLVM) [66] or binary code, e.g., KLEE [47], BitBlaze [91], S2E [50], DART [54], CUTE [90], SAGE [55], and CRETE [49].

However, scripting languages are never statically compiled as low-level languages like C/C++. They are interpreted by an interpreter. Statements of interpreted languages can wrap complex operations that, in lower-level languages, would be implemented through libraries. Due to such complexity, simply applying traditional concolic execution on scripting languages can easily cause path explosions. Consequently, testing embedded scripts still need to be done manually.

There have been tools to automate these testing tasks for scripting languages using symbolic execution. For Lua language, CHEF is an execution engine for standalone Lua interpreters. However, most of the use cases of a Lua interpreter are to be embedded in a more comprehensive environment such as Nmap. Test cases generated under such comprehensive environments are not able to reflect real use cases. For JavaScript, most symbolic execution methods 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 [68]. It modifies Rhino JS engine for symbolic execution [58]. 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 [89]. COSETTE is another symbolic execution engine for JavaScript using an intermediate representation, namely JSIL, translated from JavaScript [83]. 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 [71]. Kudzu targeted AJAX applications by implementing a dynamic symbolic interpreter that takes a simplified intermediate language for JavaScript [85]. To the best of our knowledge, no symbolic execution framework for scripting languages has directly utilized existing powerful binary-level concolic execution engines in their native execution environment [69].

1.1.2 Concolic Execution of NSE Scripts for Automated Honeyfarm Generation

Lua, in particular, is intended to be embedded into C applications and provides developers with a mature C API to integrate with. As a result, it has been extensively used with C in many practical applications such as Apache2 (web server), OpenResty3 (application server), and Awesome4 (window manager for X). It is also used within NMap [72], a popular open-source utility for network discovery and security auditing that is prevalently used by security practitioners and adversaries alike. NMap as a network tool allows attackers to efficiently perform reconnaissance. Many offensive tools are built using modular frameworks that support extensibility via scripts, allowing developers to continuously update the capabilities of the tool. Such updates are often published immediately after new vulnerabilities are disclosed, allowing anyone (both good and bad) to locate and exploit vulnerable systems. For example, within a month of the Eternal Blue release [43], updates to attack tools allowed adversaries to leverage the flaw with devastating effects before systems could be patched. In an even more severe case, on the day the Apache Struts vulnerability involved in the Equifax breach was disclosed, identification and attacking scripts were published for it, thus allowing adversaries to instantly scan for vulnerable systems and exploit them soon after [45]. Honeypots, honeynets, tarpits, and other deceptive techniques can be used to slow attackers down. However, such approaches have difficulty keeping up with the sheer number of vulnerabilities being discovered and attacking scripts that are being released. Therefore, an automated way of generating such honeypots is strongly desired for defending attacking scripts.

Applying concolic execution on attacking scripts in NMap in order to automatically generate lightweight fake versions of the vulnerable services that can fool the scripts will address such issues.

1.1.3 Effective Bug-finding for NPM libraries of Back-end JavaScript

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 [15], which is built upon Chrome's V8 JS engine [21], 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) [16], and widely used in application building. NPM is considered the largest package manager based on the number of packages it manages [62]. This number is still growing at an average rate of 996 more packages per day in the past year [32].

Many developers consider JS scripts (either browser or Node.js based) a major security vulnerability because of its growing popularity in today's systems [53]. 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 [99] and regular expressions denial of service (DoS) [52]. 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-dependences among JS libraries further exacerbate security threats [96]. Studies also show on average 6.8% of the code from a Node.js application is original code and 93.2% of the code is from other JS libraries [62]. And only 45.2% of those JS libraries have test suites provided [42]. 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 endeavors and bottlenecks for software development [76].

Although early applications of symbolic execution for testing JS scripts have shown some promise in automatically generating such test suites, 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 [46]. 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 [85]. 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 effectiveness and efficiency that warrant wide adoption [94]. Therefore, we propose an approach of in-situ concolic testing of JavaScriptto find bugs in NPM libraries effectively.

1.1.4 Automated Bug Detection for Front-end JavaScript Application

JavaScript as a frond-end programming language is used by 95.1% of websites [40]. Many such websites handle sensitive information such as financial transactions and private conversions. Errors in these websites not only affect user experiences, but also endanger the safety, security, and privacy of users. Therefore, these websites, particularly their dynamic functions that are often implemented in JS, must be thoroughly tested to detect software bugs. There have been many testing frameworks for JS applications, such as *Jest* and *Puppeteer*. These frameworks provide a systematic way to test JS applications and reduce the tedious testing setup, particularly for unit testing. However, although these testing frameworks simplify the execution of testing, they do not provide test data for web applications. Such test data still needs to be provided manually by the application developers, which is often very time-consuming and laborious. And achieving high code and functional coverage on web applications with high-quality test data still remains a challenge [76].

It is strongly desirable to apply concolic testing to front-end JS web applications to generate high-quality test data automatically, so manual efforts can be reduced and test coverage can be improved. However, front-end JS applications pose major challenges to concolic testing. These applications typically execute in the contexts of web browsers, which tends to be complex, and they are usually event-driven, user-interactive, and string-intensive [79]. There are few symbolic analysis frameworks for JS web applications. Oblique injects symbolic JS library into the page's HTML. When a user loads the page, it conducts a symbolic page load to explore the possible behaviors of a web browser and a web server during the page load process. It generates a list of pre-fetch `url` for client-side to speed up page load [61]. It is an extension of the ExpoSE concolic engine. Therefore, we propose an approach to concolic testing of front-end JS web applications to address this problem.

1.1.5 Improvement for the Execution Tracing and Trace Translation

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 [70]. 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 [85]. 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 [94]. In-situ concolic testing for JavaScript using JavaScript's native execution environments becomes its biggest strength. However, it has several limitations [70]. It utilized the tracing engine of CRETE, which leverages the interpreted mode of *Qemu*, a dynamic translator [44], 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.

1.2 Solution Overview

1.2.1 Customized Concolic Execution at Binary Level of Scripting Language

A script conceptually executes both on the high level—the level of the script language and the low level—the level of the host language. In general, an application that consists of C as the host language and the scripting language has a three-layer structure. Figure 1.1 illustrates this structure. The base layer includes the host program of the application in C. The top layer consists of embedded scripts in scripting languages used by the user. Users can customize the application as they wish without recompiling the entire program by providing various scripts. The glue layer, which is also written in a low-level language like C, has often built-in interpreters and glues the gap between C and the scripting language.

We provide a framework of applying concolic execution in the embedded scripts making use of the glue layer naturally existing in the scripting Language so that we can not only bridge scripting languages to low-level languages together but also allow concolic testing to happen in their native execution environment. Conceptually, we introduce three important interfaces to involve symbolic execution: `start_analysis()`, `mark_symbolic()`, `end_analysis()`. These interfaces will give us control over symbolic execution. Modifying the glue layer to include the interfaces can transfer concolic execution from the binary level to the script level. Due to the complexity of scripting languages, being able to start symbolic execution from the script level will generate a massive execution trace which leads to path explosion. Because one line of scripting code can contain thousands of lines of c code implementation and also scripts can run in collaboration to complete one task. Hence, we introduce customized concolic execution to only target the script under test. We have `start_analysis()` and `end_analysis()` to allow us to customize symbolic analysis against the execution traces. As a result, when scripts are running in collaboration (library scripts can be invoked within the target scripts), we can easily avoid analyzing unwanted scripts symbolically and only analyze the scripts under test symbolically by calling the interfaces. Different scripting languages will require various strategies for implementing our framework since each scripting language has a different execution engine. In the coming Sections, we will go into further depth about each strategy.

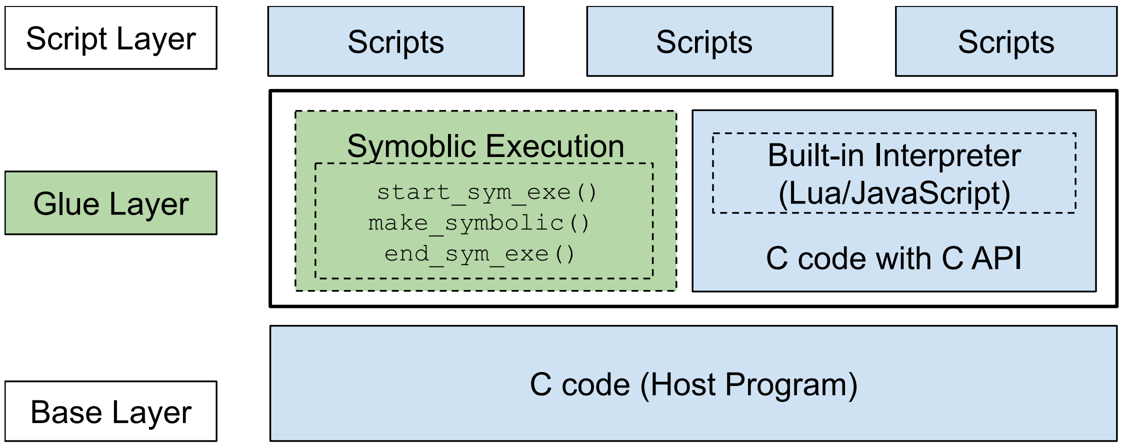


Figure 1.1: Structure for Applications with Embedded Scripts and Symbolic Execution Interface

1.2.2 Concolic Execution of NMap Scripts for Automated Honeyfarm Generation

Nmap uses NSE scripts written in Lua to perform network scanning. One way to slow down such script scanning is to use fake networks and servers to either trick automated attacking tools into believing they are interacting with a real vulnerable system (such as with honeypots and honeynets) or to selectively terminate the operation of the script by denying access (such as with web application firewalls). Unfortunately, due to the massive code bases being used and the volume of vulnerabilities that are being discovered, it is difficult to keep such approaches up to date and to scale them to the number of vulnerabilities that are being disclosed. Thus, it is important that automated defenses keep up with this arms race and attempt to make some of the most common tasks an adversary relies upon more difficult and time-consuming. In particular, as reconnaissance and targeting are critical in an attack, slowing down or degrading this capability can provide defenders valuable breathing room in protecting their networks.

In this work, we introduce an approach for applying concolic execution on NSE scripts in NMap that are used for performing reconnaissance and scanning. The goal is to generate responses that can allow automated defenses to trick the script into an arbitrary state within itself. The approach is driven by the observation that most NMap scripts for scanning and identifying vulnerable hosts are well-structured and clean. By using concolic execution to generate responses that can fool such scripts into its various execution states, one can slow down an adversary enough to allow for vulnerabilities to be remediated. For example, returning a response that causes a script to identify a service as vulnerable could be used to set up potemkin honeyfarms [97], while returning an input that causes a script to identify a service as not vulnerable could be used at the network edge as an application firewall to stop reconnaissance. To support concolic execution of `NSE` scripts our implementation of integrating the framework focuses on extending C Modules to NSE libraries of NMap. We use CRETE as the concolic execution back-end engine and add C Modules providing C API to NMap to allow users to customize concolic execution for the target application. This includes allowing users to start concolic execution to introduce symbolic values and to stop concolic execution as needed from NSE scripts.

1.2.3 In-Situ Concolic Testing of JavaScript on Node.js Libraries

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 cases generated are directly replayed in these environments. We have implemented this approach in the context of Node.js and its V8 JS engine leveraging their intrinsic functions. 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 statement coverage between 75% and 100% and for 61% of the libraries, it achieves 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.

1.2.4 Concolic Testing of Front-end JavaScript

We introduce a novel approach to concolic testing of front-end JS web applications based on in-situ concolic testing. This approach leverages widely used JS testing frameworks such as *Jest* and *Puppeteer* and conducts concolic execution on JS web functions for unit testing [95]. These testing frameworks isolate the web function under test in the context of its embedding web page by mocking the environment and provide the test data that drives the function. This isolation of web function provides an ideal target for the application of concolic testing. We integrate concolic testing APIs into these testing frameworks. The seamless integration of concolic testing allows injection of symbolic variables within the native execution context of a JS web function and the precise capture of concrete execution traces of this function. As the testing framework executes the function under test with test data, parts or all of the test data can be made symbolic and the resulting execution traces of the function are captured for later symbolic analysis. Concise execution traces greatly improve the effectiveness and efficiency of the subsequent symbolic analysis for test generation. The new test data generated by the symbolic analysis is again fed back to the testing frameworks to drive further concolic testing. We have implemented our approach on *Jest* and *Puppeteer*. The application of our *Jest* implementation to *Metamask*, one of the most popular Crypto wallets, has uncovered 3 bugs and 1 test suite improvement, whose bug reports have been accepted by *Metamask* developers on Github. We have also applied our *Puppeteer* implementation to 21 Github projects and detected 4 bugs.

1.2.5 Concolic Testing of JavaScript using Sparkplug and Remill

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 and Related Works

2.1 Background

Concolic Testing Several approaches exist to ease the problems caused by path explosion, such as using heuristic path-finding to increase code coverage [73], reducing execution time by parallelizing independent paths [92] or simply merging similar paths [64]. However in general, one cannot completely avoid the problem, making exhaustive exploration unrealistic for most systems code.

One fundamental idea to cope with these issues and to make symbolic execution feasible in practice is to mix concrete execution and symbolic execution together, also referred to as *concolic execution*, where the term concolic is a combination of the words "concrete" and "symbolic". For example, as Figure 2.1 shows, classic symbolic execution will explore all 5 paths in the figure.

Any feasible path relevant to the input value `x` is explored, once `x` is made symbolic with `make_symbolic`, which will lead to path explosion when testing complex programs. Path 4 is a concrete execution path of a target application driven down by a concrete initial value `x`. By forcing execution to take `br1` concretely before running the target application symbolically, a concolic approach would only execute `br3` and `br4` symbolically while avoiding `br2`, `br5`, and `br6`. Thus, concolic execution can reduce the possibility of path explosion, making it more suitable than symbolic execution for testing complex applications with an embedded interpreter.

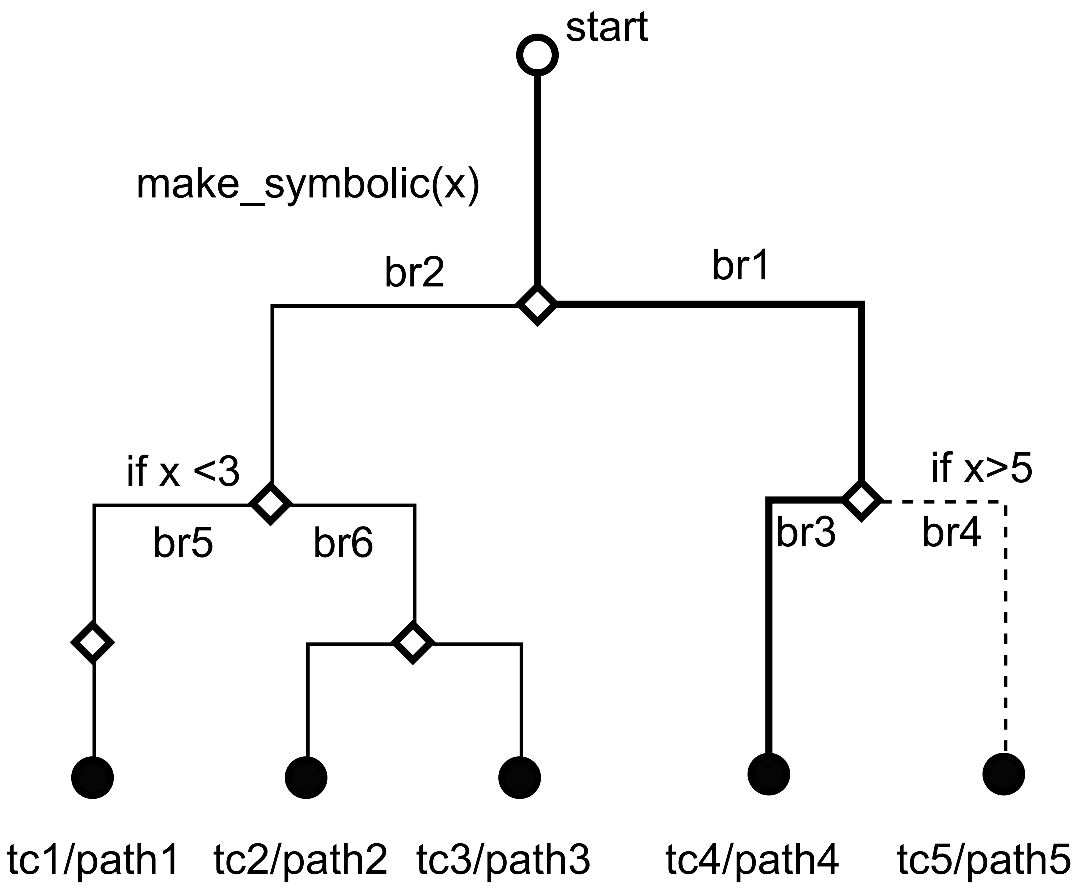


Figure 2.1: Symbolic execution covers all paths; Concrete execution covers path4; Concolic execution covers path4 and path5

CRETE To enable concolic execution of scripting languages, we build on CRETE, a binary-level concolic testing framework [49]. CRETE features an open and highly extensible architecture allowing easy integration of concrete execution front-ends and symbolic execution engine back-ends.

As shown in Figure 2.2, CRETE uses a configuration file to mark symbolic and concrete inputs in the CRETE runner. As the target program is concretely executed in a

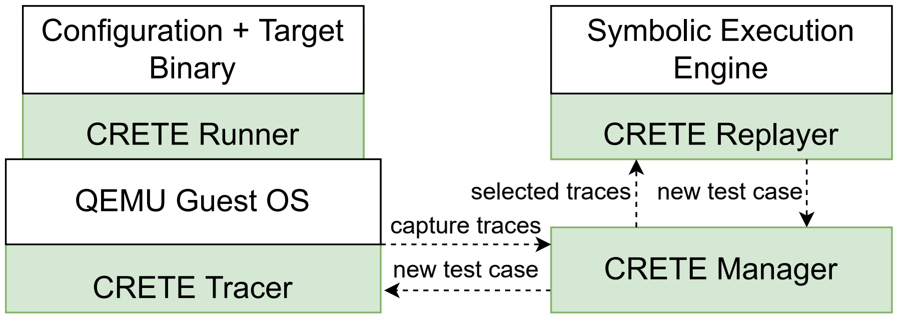


Figure 2.2: Architecture of CRETE

modified QEMU virtual machine [44], 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 [66]. 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 [47]), 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 [86]. 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 CRETE and its interface functions to realize customized concolic execution proposed.

2.2 Related Works

2.2.1 Binary Level Concolic Execution Engines

Most existing symbolic and concolic execution engines target low-level code representations. For example, symbolic execution engines such as KLEE [47], BitBlaze [91] and S2E [50] as well as concolic execution engines such as DART [54], CUTE [90] and SAGE [55] work with either machine code or LLVM intermediate representation code [66] that has been statically compiled. The scripts that we are dealing with, are however, interpreted, not statically compiled. There have also been efforts in building symbolic engines targeting script languages. However, such implementation requires a significant amount of work for every single language and constant maintenance if the target language is updated.

2.2.2 Symbolic Execution for Scripting Languages

NICE [48] for Python and Kudzu [85] for Javascript are early efforts to directly implement symbolic execution engines for dynamically interpreted scripts in high-level languages. Existing symbolic execution engines that can support Lua only target standalone interpreters such as CHEF [46] while `NSE` scripts are interpreted by an interpreter embedded in NMap.

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 [68]. It modifies Rhino JS engine for symbolic execution [58]. 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 [89]. COSETTE is another symbolic execution engine for JavaScript using an intermediate representation, namely JSIL, translated from JavaScript [83]. 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 [71]. Kudzu targeted AJAX applications by implementing a dynamic symbolic interpreter that takes a simplified intermediate language for JavaScript [85].

2.2.3 Fuzzing Testing

Another alternative approach to automatically generate test inputs to applications is fuzzing, which uses code coverage as feedback to test generation. Therefore, the effectiveness of test cases generated by fuzzing greatly depends on the accuracy of code coverage feedback. Without the knowledge of the source code, it's difficult for fuzzing to penetrate into nested branches. While fuzzing is random test case generation guided by code coverage, symbolic execution is guided by code trace. Therefore, symbolic execution and fuzzing may have similar outcomes regards to code coverage when source codes have less nested branches. Symbolic execution can also deal with source codes with more complicated logic more efficiently.

AFL (American Fuzzy Lop) is one of the most popular fuzzing tools that has been used widely in the industry [22]. Becasue it is fast, simple and efficient. While testing the target application, AFL runs in loops. It uses coverage as a guide to mutate the seed in order to generate more new test cases and then filters test cases that improve code coverage into a queue for the next iteration. Internally, AFL checks if that input makes the program reach new code path or increases the coverage (either completely new blocks, or different sequence of blocks). If this is the case, the input is marked as 'interesting' and will be selected into the seed pool and mutated later or remixed with other random or interesting inputs to try to reach a deeper code path in the program, and yield more coverage. To achieve runtime monitoring, AFL will instrument the target application and inject code at compile time. This is done by substituting `gcc` or `clang` with AFL's wrappers: `afl-gcc` and `afl-clang`. The wrapper will call the normal compiler, then add the instrumentation code and produce a binary that can be monitored by `afl-fuzz`.

With its low-level compile-time or binary-only instrumentation, AFL provides an near-native fuzzing speeds against common real-world targets. Therefore, it can find bugs effectively with the fast speed. There are a few fuzzers for JS, e.g., *jsfuzz* [30] and *js-fuzz* [29], which are largely based on the fuzzing logic of AFL (American fuzzy lop) [22] and re-implemented it for JS. We view fuzzing and symbolic/concolic testing as complementing techniques: fuzzing for broader exploration of JS while symbolic/-

concolic testing for deeper exploration.

3 Concolic Execution of NMap Scripts for Honeyfarm Generation

3.1 Background

Methods of Honeyfarm Generation There are two ways for deploying honeyfarms: low-interaction honeyfarm and high-interaction honeyfarm. Low-interaction honeyfarm can monitor activities over millions of IP addresses at a time, such as KFSensor Honeypot [75] and Conpot [74]. This kind of scalability is achieved by emulating the network interface exposed by common services and requires low maintenance. However, such systems do not execute any code from applications; therefore, they may not be able to block attacks that have multiple phases of communication [97]. On the other hand, high-interaction honeyfarms run native application code, and therefore, is able to catch code behavior in its full complexity [80]. As a consequence, the implementation cost is quite high. Systems of high interaction honeyfarms include Honeynets [80], Sebek [57], Argos [78], etc. Our method is a light way of achieving the purpose of high-interaction honeyfarms.

3.2 Design

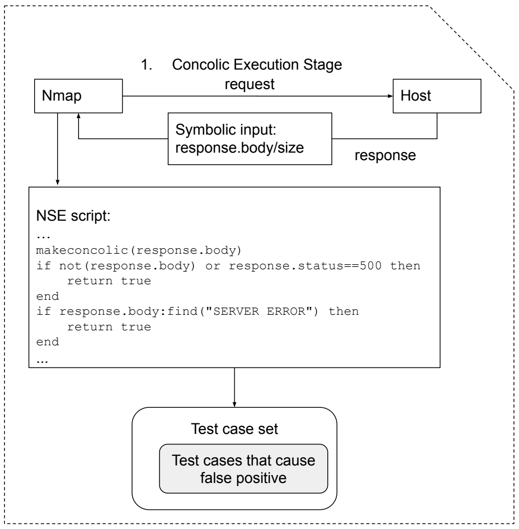
3.2.1 Overview

In this Chapter, we present e introduce an approach for applying concolic execution on NSE scripts in Nmap that are used for performing reconnaissance and scanning.

Our goal is to run `NSE` scripts using concolic execution to generate test cases to form decoys against the attackers. Because the nature of concolic execution is to explore every possible execution state along a concrete execution trace, this will allow us to focus our symbolic execution on discovering sets of network responses that force an `NSE` script into transitioning into each of its execution states. This generation is key for honeyfarms as it provides them with responses that can be used to control the `NSE` scripts' execution behavior. For example, returning an input that leads the script into believing the host is vulnerable would allow the interaction to continue in order to further consume an attacker's time and energy. Selecting a response that leads the script into believing the host is not vulnerable or has no resource of interest would send the attacker away. Randomly selecting from calculated inputs per connection would allow defenders to actively confuse the attacker. Finally, returning inputs that may leverage bugs and errors would allow defenders to potentially crash the attacking scripts and terminate the scan altogether. All of these synthesized responses would potentially allow defenders to slow down an attacker's workflow. In order to complete this process, the approach we take is broken into two stages:

- `Concolic Execution Stage`: As shown in Figure 3.1, in this stage, we perform concolic execution on `NSE` scripts to generate test cases for honeyfarm synthesis. For the case shown, we can set `response.body` and `response.size` to symbolic values for the engine to explore.

- `Defending Stage`: As shown in Figure 3.2, in this stage, we use the various test cases generated from concolic execution to synthesize honeyfarm responses with



```
…
makeconcolic(response.body)
if not(response.body) or response.status==500 then
    return true
end
if response.body:find("SERVER ERROR") then
    return true
end
...
```

Figure 3.1: Concolic Execution Stage

which we can then have an Intrusion Detection System (IDS) to respond upon detecting the corresponding NMap scan.

3.2.2 Concolic Execution Stage

There are several challenges when considering the use of symbolic and concolic execution on an NMap script. Most existing symbolic and concolic execution engines target low-level code compiled statically. `NSE` scripts use Lua as the base language and are not statically compiled, but rather interpreted by NMap's built-in Lua interpreter. The Lua interpreter itself is extended by NMap with a library for communication, which is responsible for providing additional information that `NSE` scripts need to execute. For example, `nmap.new_socket()` function supplied by the library returns a new socket wrapper object `NSE` scripts can use. The NMap library also takes care of initializing the

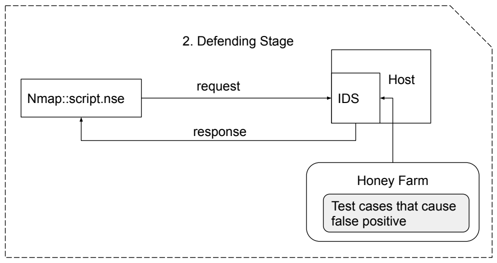
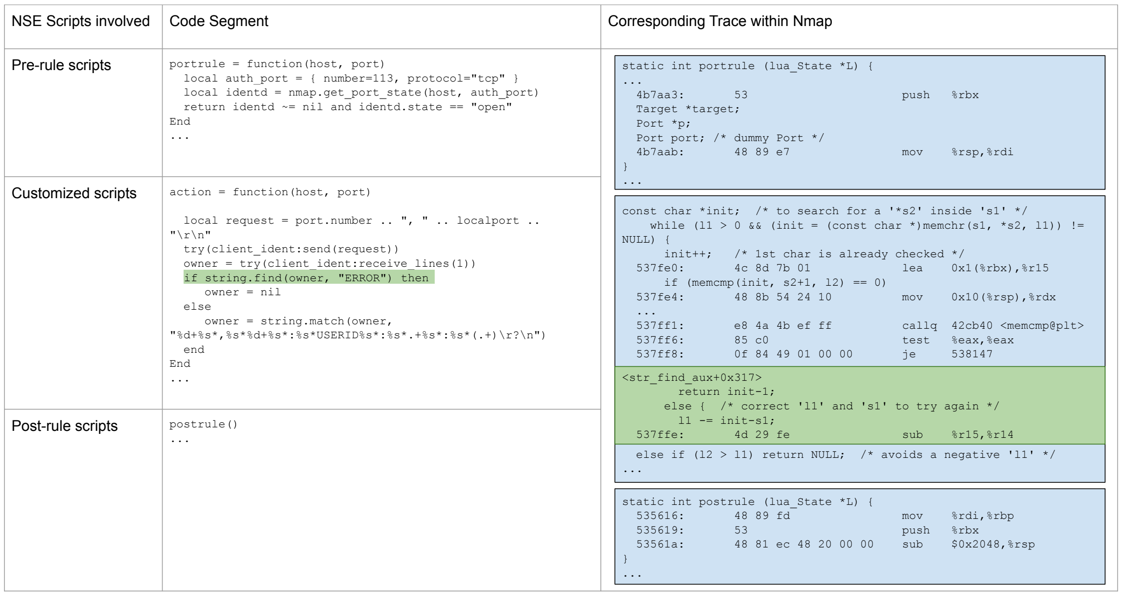


Figure 3.2: Defending Stage



NSE Scripts involved | Code Segment | Corresponding Trace within Nmap

Pre-rule scripts

```
portrule = function(host, port)
  local auth_port = { number=113, protocol="tcp" }
  local identd = nmap.get_port_state(host, auth_port)
  return identd ~= nil and identd.state == "open"
End
...
```

```
static int portrule (lua_State *L) {
...
  4b7aa3:       53                      push   %rbx
  Target *target;
  Port *p;
  Port port; /* dummy Port */
  4b7aab:       48 89 e7                mov    %rsp,%rdi
}
...
```

Customized scripts

```
action = function(host, port)

  local request = port.number .. ", " .. localport ..
"\r\n"
  try(client_ident:send(request))
  owner = try(client_ident:receive_lines(1))
  if string.find(owner, "ERROR") then
     owner = nil
  else
     owner = string.match(owner,
"%d+%s*,%s*%d+%s*:%s*USERID%s*:%s*.+%s*:%s*(.+)\r?\n")
  end
End
...
```

```
const char *init;  /* to search for a '*s2' inside 's1' */
    while (l1 > 0 && (init = (const char *)memchr(s1, *s2, l1)) !=
NULL) {
      init++;   /* 1st char is already checked */
  537fe0:       4c 8d 7b 01             lea    0x1(%rbx),%r15
      if (memcmp(init, s2+1, l2) == 0)
  537fe4:       48 8b 54 24 10          mov    0x10(%rsp),%rdx
  ...
  537ff1:       e8 4a 4b ef ff          callq  42cb40 <memcmp@plt>
  537ff6:       85 c0                   test   %eax,%eax
  537ff8:       0f 84 49 01 00 00       je     538147
<str_find_aux+0x317>
        return init-1;
      else {  /* correct 'l1' and 's1' to try again */
        l1 -= init-s1;
  537ffe:       4d 29 fe                sub    %r15,%r14
  else if (l2 > l1) return NULL;  /* avoids a negative 'l1' */
...
```

Post-rule scripts

```
postrule()
...
```

```
static int postrule (lua_State *L) {
  535616:       48 89 fd                mov    %rdi,%rbp
  535619:       53                      push   %rbx
  53561a:       48 81 ec 48 20 00 00    sub    $0x2048,%rsp
}
...
```

Figure 3.3: Explanation of Our Approach

Lua context, scheduling parallel scripts and collecting the output produced by completed scripts.

Because NSE scripts can utilize both the extended libraries in NMap and the default libraries of the Lua language, they are more complex than stand-alone Lua scripts.

Compounding this complexity is that statements of interpreted languages can encapsulate complex operations that are implemented in underlying compiled libraries written in lower-level languages. For example, the Lua language supports 7 string operations that are implemented in a string library of the Lua interpreter, which contains thousands of lines of C code interpretation [65]. Symbolically executing such code can easily cause path explosion. Consequently, symbolic execution of such scripts may require manual intervention to avoid this problem. Recent work has sought to automate this task, which involves changing the interpreter and building a new symbolic execution engine. Unfortunately, the implementation of a dedicated symbolic execution engine adds a significant amount of work for each language, requiring constant maintenance if the language is updated.

Therefore, we need to apply symbolic execution to analyze arbitrary `NSE` scripts in a way that avoids the path explosion problem as well as continually updating our execution engine when there are update to the Lua language. To meet this goal, we adapt CRETE, our concolic execution engine, using API calls in the glue layer of the built-in interpreter. Specifically, we use the interface provided by CRETE and modify glue layer in NMap to allow users to conveniently inject symbolic values from the scripts. Additionally, we modify the engine to provide interfaces that allow users to defer concolic execution of a program as needed in order to further limit the execution paths of the script to the minimum. The reason why we need to defer concolic execution is that we need to keep execution complete for the whole scanning process to guarantee completeness of the trace whiling ensuring that we only symbolically execute the portion of the trace that is of interest. We will show the significant reduction in execution time by deferring concolic execution in Section 3.4.

Figure 3.3 shows an example of `NSE` scripts involved in a NMap network scan, which has pre-rule scripts, customized scripts and post-rule scripts running in three scan phases respectively (script pre-scan, script scan and script post-scan). In each scan phase, more than one `NSE` script will be executed. In the script pre-scan phase, pre-rule scripts are executed to collect information for customized scripts which will be executed in the script scan phase. In most cases, users are interested in testing customized scripts because they can be modified, allowing the library to be extended. Testing them with concolic execution requires capturing the execution traces for all the `NSE` scripts that have been executed. The last column in Figure 3.3 gives an example of one such trace that shows the obstacles facing concolic execution, which is one to many code mapping from scripting language to low level code. The figure shows assembly code snippets for each phase of the scan. As concolic execution works with low-level code representation, path explosion can happen in the script pre-scan phase before the concolic engine can even reach the script scan phase for customized scripts. This situation worsens when the interpreted pre-scan script involves loops or nested pattern matching operations, which is quite common in `NSE` scripts for string manipulation. Therefore, being able to test the scripts users are actually interested in requires methods to defer symbolic execution to specific segments in order to prevent path explosion. As a result, our approach leverages the adapted interface of CRETE to allow user to customize concolic execution as needed.

In doing so, we make the observation that on such application, an embedded script conceptually executes both on the high level (e.g. at the script language) and the low level (e.g. at the host language). In most cases, applications use C and its interfaces for the host language. Figure 3.4 illustrates a typical structure for embedding scripts as of NMap.̇The base layer includes the host program of the application in C. The top layer consists of embedded scripts prepared by the user. By providing various scripts, the user can customize the application as their wish without recompiling the entire program. The glue layer, which is also written in C, contains the built-in interpreter and glues the gap between C and the scripting language.

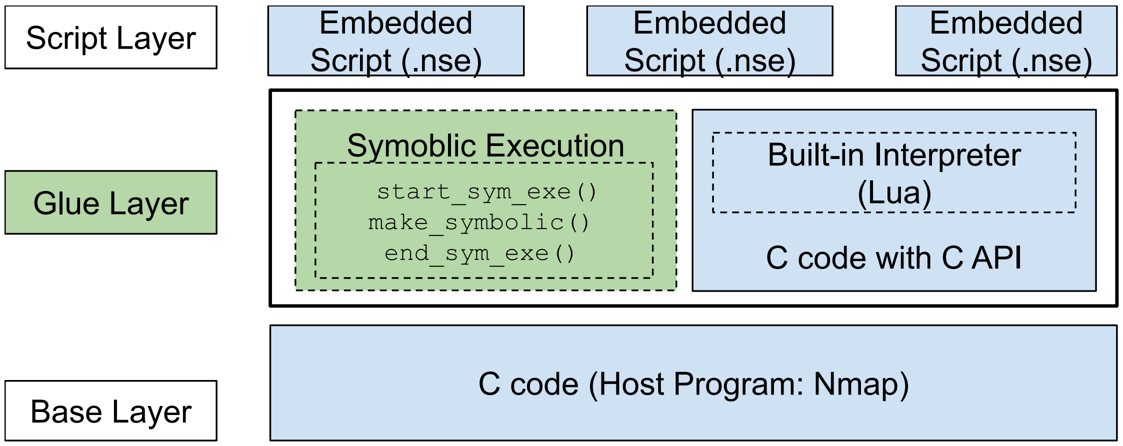


Figure 3.4: Structure for Applications with Embedded Scripts and Symbolic Execution Interface

We make use of the glue layer to achieve our goal of concolically executing `NSE` scripts. To gain control of concolic execution, we introduce three important interfaces for symbolic execution: `start_analysis()`, `mark_symbolic()`, and `end_analysis()`. These interfaces will allow us to customize concolic execution in scripts. Modifying the glue layer to include these interfaces allows users to start symbolic execution with a function call. At the same time, starting symbolic execution from the script layer generates a massive execution trace which leads to path explosion. Hence, we have `start_analysis()` and `end_analysis()` to allow us to delay the symbolic execution till later in the execution where we want it and stop it as wish. Therefore, when the target scripts invoke additional scripts of no interest to the analysis, we can easily avoid running unwanted scripts symbolically and only execute the target scripts symbolically by properly calling the above functions. This method has a potential to be applied on other application with the similar structure. In our case (NMap), embedded scripts and built-in interpreter refer to `NSE` scripts and Lua interpreter respectively. With these interfaces we can go through the entire execution for pre-rule scripts, customized scripts, and post-rule scripts but only symbolically execute the traces of customized scripts, thus reducing possible symbolic paths significantly.

3.2.3 Defending Stage

With the method explained above, we can apply concolic execution to any `NSE` script to get responses that can be leveraged by honeyfarm to control the execution state of the attacking scripts. To achieve our objectives, a range of selection rules targeting different application scenarios can be implemented. Two rules, in particular, include:

- **Early Termination Rule.** With this rule, responses selected will be the ones which will cause the attacking script to stop as soon as possible. We use script coverage as an indicator. We will consider test cases that achieve lower coverage on a script with higher priorities for the synthesis of a honeyfarm.

- **False Positive Rule.** The test cases selected for honeyfarm generation will be the ones which will cause the attacking script to believe that it has find a host with certain vulnerabilities. We will consider test cases that reach certain end-points in a script. These end-points can be annotated manually or identified through templates.

Upon selecting a response, the next step of the defending stage is handling the attacking connection and delivering the response back to the script. Intrusion detection systems (IDS) combined with templating systems provide a natural mechanism for doing so. For example, consider an `NSE` script seeking to find a vulnerable HTML form submission. An IDS running on a honeyfarm system can provide us hooks into the request being made by the script, while an HTML-templating engine such as Mustache [98], can allow us to use templates that we fill in with the test cases from concolic execution in order to complete the defending stage response.

3.3 Implementation

3.3.1 Concolic Script Execution

To support concolic execution of `NSE` scripts our implementation focuses on the glue layer of NMap. We use CRETE as the concolic execution back-end engine and modify the glue layer of NMap to allow users to customize concolic execution for the target application. This includes allowing users to start concolic execution, to introduce symbolic values and to stop concolic execution as needed.

By default, CRETE performs concolic execution on the entire execution trace of a

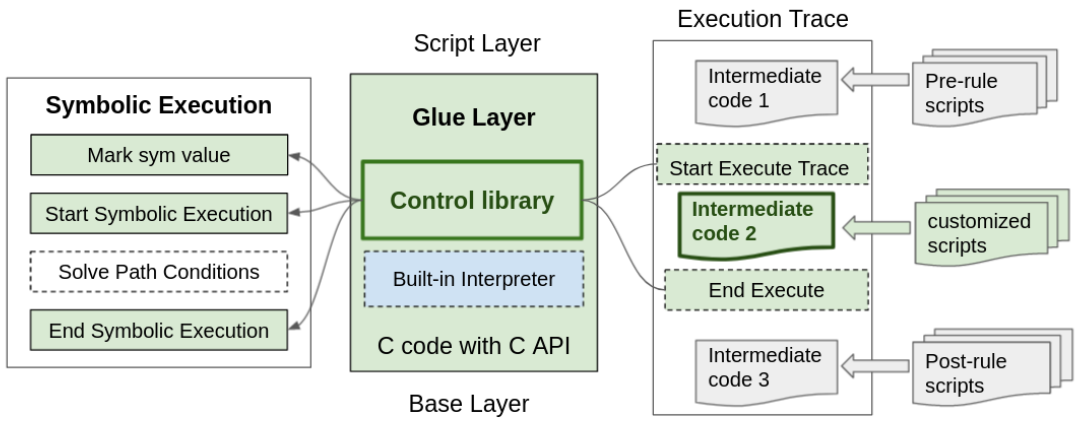


Figure 3.5: Control Library for Concolic Execution

program captured by the CRETE front-end in QEMU. Because we wish to finely control the parts that are symbolically executed, we modify CRETE to decouple concolic execution with a set of interface functions, namely `sendpid()`, `mconcolic()` and `exit()`. These functions pass control of concolic execution from CRETE to `NSE` scripts. For clarity, the naming convention we used in our implementation of the glue layer for `NSE` scripts is to keep consistent with the CRETE back-end engine: `sendpid()` is the interface function to start concolic execution if a symbolic variable is present (in corresponding to `start_analysis()`). `mconcolic()` is the interface function to mark symbolic variable (in corresponding to `mark_symbolic()`). `exit()` is to stop concolic execution (in corresponding to `end_analysis()`). As a result, we can defer the concolic execution in NMap until after the script pre-scan phase and end it before the post-scan phase. We use this control library to minimize symbolic execution on execution traces to address the path explosion problem when concolically executing an interpreted script as shown in Figure 3.5. The control library allows us to decide which segment of the intermediate code we want CRETE to execute symbolically.

3.3.2 Lua Interpreter Instrumentation

The embedded Lua Interpreter in NMap interprets `NSE` scripts utilizing the string interning optimization. We disabled string interning so that CRETE can use taint analysis to make sure all the relative traces to the symbolic values are captured. Disabling string interning is relatively simple and can be done through a Lua configuration macro [65]. We also handled the Lua's two internal representations for numbers: float and integer. Specifically, we ignored numbers whose internal representation are float, as the underlying symbolic execution engine CRETE uses, namely KLEE, does not support floating point numbers. In addition, we modified Lua math library for all functions to support making internal integer representations symbolic. As an example, Listing 3.1 shows how we call the interface functions from a `NSE` script that allows for us to customize concolic execution. The script performs a form submission on a potential vulnerable site and obtains a response. It returns `true` if a null response body is received or if an error is returned. In this example, we choose to inject symbolic values and start symbolic analysis right when the relevant parts of the script are being executed to minimize path explosion.

3.3.3 Snort response

Once we have performed concolic execution on the script, we then use Snort [82], a network-based IDS to deliver the response. Snort can be configured to detect malicious

```
local function check_response(response)
  --crete start
  crete.sendpid()
  crete.mconcolic(response.body,12)

  if not(response.body) or response.status==500 then
    return true
  end
  if response.body:find("SERVER ERROR") then
    return true
  end

  --exit program
  crete.mexit(0)

  return false
end
```

Listing 3.1: http-form-fuzzer.nse instrumented with CRETE.

behaviors over the network with a set of rules in `snort.conf`. We leverage one such set of rules that is maintained, validated, and updated by Proofpoint [81] to allow Snort to detect NMap scans. Listing 3.2 shows the rule used to detect NMap web application attacks in the evaluation. As part of the Snort rule, we configure the rule's `react` option to deliver specific responses that are synthesized using the generated test cases from our concolic execution when the NMap scan is detected. For the Web Application Scan from Listing 3.1, an example of the synthesized response is shown in Listing 3.3. The string "SERVER ERROR" in line 3 has been generated by CRETE. Note that for this case, while the string appears in the page's title, one can place the string anywhere in the `response.body` to trick this particular script. The generation of the response HTML can be done using any automated templating system such as Mustache [98] that allows us to replace parts of the content with the test case generated from concolic execution.

Listing 3.2: Snort detecting rule for NMap web application scan

```
alert tcp any any -> any any (msg:"ET SCAN Nmap Scripting
Engine User-Agent Detected (Nmap Scripting Engine)";
flow:to_server,established;
content:"User-Agent3a Mozilla/5.0 (compatible3b Nmap
Scripting Engine";
react; fast_pattern:38,20; http_header;
nocase; reference:url,doc.emergingthreats.net/2009358;
classtype:web-application-attack; sid:2009358; rev:5;)
```

```
<!doctype html>
 <html lang="en">
   <head><title>SERVER ERROR</title></head>
   <body>
   <div style="color:red">
   </div>
    <form name="LoginForm" method="post"
    action="/loginclass/Login.do;jsessionid=D34B538055462B75E1CD6DFD18B9650E
    ">
     User Name:<input type="text" name="userName" value="">
     Password:<input type="password" name="password" value="">
     <input type="submit" value="Login">
    </form>
   </body>
  </html>
```

Listing 3.3: An Example of Synthesized Response in Snort

3.4 Evaluation

In this section, we first introduce the `NSE` scripts we target and the experimental setup for our approach including the CRETE settings which are used in the concolic execution stage. Then, we will summarize our preliminary results, which shows the type of test cases from running `NSE` scripts with our approach with a set of examples. Finally, we analyze why we are able to achieve these results.

3.4.1 Experimental Setup for `NSE` Scripts

Because a large majority of network protocols such as HTTP are string-based, string manipulation operations are some of the most frequently used in `NSE` scripts. As a result, our experiments mainly focus on string variables when injecting symbolic values into `NSE` scripts. We follow the simple heuristics below to select which variables are made symbolic:

- For host scan scripts, variables that are involved in `if-else` branches in scripts are set as symbolic values. Among string operations, substring finds and string pattern matching commonly appear in branch statements since such functions return values that are of `boolean` type.
- For web scripts, `response.body` and `response.size` are set as symbolic since they are commonly involved in branches as information they return is often of interest to `NSE` scripts.

To showcase our approach, we use `http-form-fuzzer.nse` as an example, which involves the `string.find` function. With the above heuristics, we set `response.body` and `response.size` as symbolic variables for the case where response is an HTML page.

3.4.2 Control Interface Evaluation

Naïve Case Our early attempt of applying concolic execution on `NSE` scripts is to run the `NSE` script concolically using CRETE without deferring concolic execution until when it is needed. The experiment setup for this case is that we simply use the interface of `crete.mconcolic()` to mark symbolic variables then run the `NSE` script directly. As expected, doing so causes the pre-scan stage to be involved in the concolic execution process, leading to excessive execution time. Across four executions of the script done in this manner, execution time averages 4519 seconds to explore each new feasible path in the script.

Customized Concolic Execution Case The advantage of our approach is the support for a control interface that allows the `NSE` script to defer concolic execution. For example, the segment of code in Listing 3.1 is from `http-form-fuzzer.nse`. It is frequently used to fuzz the fields of web page that contain `<form>` tags to try to find a certain request that will cause an ERROR in the web page [77]. Listing 3.1 shows an example of how we use the control interface to efficiently enable concolic execution when needed. In the listing, we wish to test line 6 to line 11, which contains two `if` statements and the symbolic value we wish to evaluate, `response.body`, whose type is a string. We then call function `crete.sendpid` to start symbolic analysis before we mark symbolic value with `crete.mconcolic` function. In this way, we have CRETE defer symbolic execution of the code until after we inject the symbolic value, thus avoiding the symbolic execution of pre-run scripts. Finally, we terminate symbolic execution using `crete.mexit` so that the symbolic execution only targets line 6 to 11 and avoids running post-run scripts symbolically.

When testing `http-form-fuzzer.nse`, with otherwise the same experimental setup as the naïve case, execution time is reduced from 4519 seconds on average to around 179 seconds per new feasible path in the script. This indicates the effectiveness of customized concolic execution. For the rest of our experiments, we apply this method for deferring concolic execution when testing `NSE` scripts.

3.4.3 `NSE` Script Evaluation

Test case generation for honeyfarms Our goal is to concolically execute a variety of `NSE` scripts in order to produce inputs that can be used to drive them to particular states. To demonstrate this, we initially select a collection of `NSE` scripts for HTTP shown in Table 3.1. For the script we have been using as an example, `http-form-fuzzer.nse`, concolic execution yields the test case with the content of "SERVER ERROR" that leads execution to go into the `if` branch in Listing 3.1, demonstrating that our approach can produce results at the script level despite the massive amount of interpreted code being executed. We use this test case in a Snort `react` defense rule and succeed in fooling NMap into thinking it identified a vulnerability, accomplishing the *False Positive* goal for the honeyfarm.

Listing 3.4: A code segment of http-title.nse

```
if display_title and display_title ~= "" then
  display_title
  = string.gsub(display_title , "[\n\r\t]", "")
  if #display_title > 65 then
    display_title
    = string.sub(display_title, 1, 62) .. "..."
  end
else ...
```

A more interesting case is the `http-form-brute.nse` script, in which a `string.m -atch` call tries to validate whether a certain value exists in a user information form returned by a scan. Furthermore, the script checks that the value `v` parsed from the form via `string.match(form[k], v)` has a pattern '`%d%d`'. To match this, concolic execution generates the test case with two digits in random combinations. Our concolic execution approach also uncovers invalid patterns that lead execution into an error state. For example, the value of '`username/(`' crashes the script since magic characters such as '`(`' need to be escaped in Lua in order to be taken literally or they must appear in pairs such as '`()`'. Such a crashing pattern can be used to trigger the *Early Termination* rule for the honeyfarm.

For `http-auth.nse`, we have test cases that have '`\0`' in the middle of the `name` variable, e.g., '`name = do\0in`'. This causes an error since '`\0`' is not considered as a terminator for a string in `NSE` with the Lua interpreter instead treating the character as an *embedded zero* instead. Therefore, in `NSE` the length of variable `name` is 6 but in C it is 2. This leads to an inconsistency in length which forces execution into the Lua error state, triggering another *Early Termination* situation.

Finally, our concolic approach exposes another similar bug in `http-grep.nse` by

NSE scripts	**Test Cases/Bugs**	**Defending Rules**
http-form-fuzzer	SERVER ERROR	False positive
http-form-brute	Invalid patterns	Early termination
http-auth	Embedded zero	Early termination
http-grep	Type inconsistency	Early termination

Table 3.1: Examples of interesting test cases and bugs discovered

generating input that triggers a type inconsistency bug in the script shown in Listing 3.6. Detailed explanation is given later in the Anlysis section. Our patch for this crashing bug has been accepted by the NMap team [1].

We use the generated test cases discussed above to form honeyfarm responses that fool the scripts. These test cases are expected to trick the scripts or stop them from running. We synthesized these test cases with templates and deliver them back to NMap using Snort configured with appropriate `react` rules. The test cases successfully cause NMap to reach the desired states, accomplishing the goals from Section 3.2: namely *Early Termination* and *False Positive* as summarized in Table 3.1.

Analysis We use a few scripts as an example to show how we generate such test cases. When testing `http-form-fuzzer.nse`, we have the desired test case with the content of "SERVER ERROR" that leads the execution to go into the `if` branch. We get to this particular test case at the 80th iteration, and we obtain the test cases that cover both `if` and `else` branches. We disassemble the relevant part of the NMap binary and show it in Listing 3.5. For this case, our approach only captures the basic block that has the branch ("`537ff6`") in shown in line 12, which matches the branch of `string.find` in the `NSE`

[1]https://github.com/nmap/nmap/issues/1931

script in Listing 3.1. Only this part of the execution trace is executed symbolically instead of the entire trace, thus allowing us to generate the desired test cases efficiently.

For testing of the `http-grep.nse` script shown in Listing 3.6, our approach enabled us to discover a bug in a local function within the script that implements `Luhn`, an algorithm that is used to validate a variety of identification numbers, such as credit card numbers. To understand how the bug works, we first describe the `Luhn` algorithm [56] in the following 4 steps:

1. Starting from the rightmost digit, double the value of every second digit.
2. If doubling a number results in a two-digit number, then add the digits of the product to get a single-digit number.
3. Take the sum of all the digits.
4. If the total modulo 10 is equal to 0 (if the total ends in zero) then the number is valid according to the Luhn formula; otherwise, it is not valid.

The two loops (in lines 5-7 and in lines 9-15) show the implementation of steps 2 and 3 in the `NSE` script and contain a bug. The bug is triggered by a test case that causes the value of variable `double` inside of `string.gsub` to be 14. When this happens, the returning value of the `string.gsub` call in line 12 becomes `5.0.0`, which cannot be coerced to a string by the code in line 13. Thus, our concolic approach allows us to easily reveal crashing bugs in `NSE` scripts that could be used to trigger the termination of the scan. In this case, however, the bug was reported and the NMap developers changed its implementation to fix the issue.

Listing 3.7 shows the captured execution trace that corresponds to the loop of the reverse function in C code that triggers the issue. This trace guides concolic execution to mutate the input string backwards (from the last position instead of the first position). In addition, it has the information about the two `for` loops, which increment `i` by 2 every iteration. This means that only mutating the bytes in odd positions of the input string after being reversed can trigger the bug in line 12 due to the step of 2 in each iteration. With this knowledge, our approach can make changes on the proper position of the string, which is every other character in the string after being reversed. The effective change is to flip the bits of the character to an ASCII code that can be converted to a number so it can pass line 11 to get to line 12 where the bug resides. The bug is triggered if the number (`doubled`) in an odd position is greater than 9. Our approach was able to make the right mutation after a few iterations to trigger the bug in line 12.

```
const char *init;  /* to search for a '*s2' inside 's1' */
    l2--;  /* 1st char will be checked by 'memchr' */
    l1 = l1-l2;  /* 's2' cannot be found after that */
    while (l1 > 0 && (init = (const char *)memchr(s1, *s2, l1)) != NULL) {
      init++;   /* 1st char is already checked */
  537fe0:       4c 8d 7b 01             lea    0x1(%rbx),%r15
      if (memcmp(init, s2+1, l2) == 0)
  537fe4:       48 8b 54 24 10          mov    0x10(%rsp),%rdx
  537fe9:       48 8b 74 24 18          mov    0x18(%rsp),%rsi
  537fee:       4c 89 ff                mov    %r15,%rdi
  537ff1:       e8 4a 4b ef ff          callq  42cb40 <memcmp@plt>
  537ff6:       85 c0                   test   %eax,%eax
  537ff8:       0f 84 49 01 00 00       je     538147 <str_find_aux+0x317>
```

```
          return init-1;
        else {  /* correct 'l1' and 's1' to try again */
          l1 -= init-s1;
537ffe:       4d 29 fe                sub    %r15,%r14
else if (l2 > l1) return NULL;  /* avoids a negative 'l1' */
```

Listing 3.5: Captured trace from http-form-fuzzer.nse script

```
1   function luhn (matched_ccno)
2     crete.mconcolic(matched_ccno, matched_ccno.len)
3     local n = string.reverse(matched_ccno)
4     local s1 = 0
5     for i=1, n:len(), 2 do
6       s1 = s1 + tonumber(n:sub(i,i))
7     end
8     local s2 = 0
9     for i=2, n:len(), 2 do
10      --conversion from string to double
11      local doubled = n:sub(i,i)*2
12      doubled = string.gsub(doubled,'(%d)(%d)',
13      function(a,b)return a+b end)
14      s2 = s2+doubled
15    end
16  end
```

Listing 3.6: A code segment of http-grep.nse script with string.reverse function: a type inconsistency bug is triggered in line 13 when trying to sum `doubled` with `s2`. This function (*luhn*) is used to validate credit card numbers

```
static int str_reverse (lua_State *L) {
  535616:       48 89 fd                mov    %rdi,%rbp
  535619:       53                      push   %rbx
```

```
    53561a:       48 81 ec 48 20 00 00    sub    $0x2048,%rsp
    size_t l, i;
    luaL_Buffer b;
    const char *s = luaL_checklstring(L, 1, 1);
    535621:       48 8d 54 24 08          lea    %rsp,%rdx
    else lua_pushliteral(L, "");
    return 1;
  }
```

Listing 3.7: The captured trace when testing http-grep.nse script. This trace segment contributes to finding the type inconsistency bug

3.5 Summary

This chapter illustrates an approach to test `NSE` scripts via concolic execution and to use the result to generate honeyfarms that can slow down attackers. Preliminary results have shown its efficiency in generating test cases that can stop NMap scans or return false positive responses. Our approach is effective with complicated programs such as NMap which runs embedded scripts where traditional concolic execution does not work at all. Our approach does so by avoiding path explosion by supporting customized concolic execution at specific locations in order to generate useful test cases efficiently. The implementation for our approach makes use of the glue layer that most embedded scripting languages provide to integrate the concolic execution engine and the interface functions for customizing concolic execution. In this way, the approach does not need to modify the built-in interpreter each time the language is updated. In the future, we aim to test more libraries in `NSE` since the effective concolic execution of more `NSE` scripts is the key to building diverse honeyfarms. Finally, we will further automate the process of synthesizing test cases with response templates so that the results of concolic execution can be included in honeyfarms with minimal manual intervention.

4 In-Situ Concolic Testing of JavaScript

In this Chapter, 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 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.

4.1 Background

In this section, we introduce the related background Node.js and V8 JS engine and explain the components we leveraging to implement our approach.

4.1.1 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.

4.1.2 V8 JS Engine

V8 is Google's high-performance JS and WebAssembly engine [21]. V8 can run standalone or can be embedded in C++ applications such as Node.js and Chrome. As shown in Figure 4.1, V8 supports two modes for executing a JS script: (1) *interpreted mode*

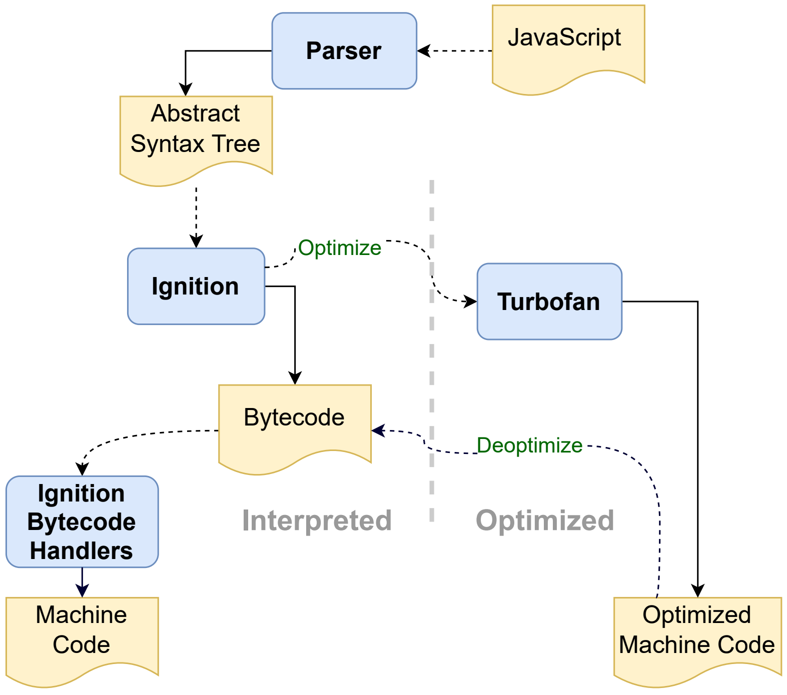


Figure 4.1: How V8 runs a JS Script: Interpreted vs Optimized

where the JS bytecode [20] translated from the JS script is interpreted by its interpreter, Ignition [8]; (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 [19], 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 [6]. 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 4.2. CSA allows us to create new V8 builtin functions to extend V8's functionality [19]. Ignition's bytecode handlers are written in CSA and compiled into binary by the unified code generation. V8's optimizing compiler, Turbofan, is also based on the unified code generation. We leverage this feature to integrate concolic execution into the V8 engine.

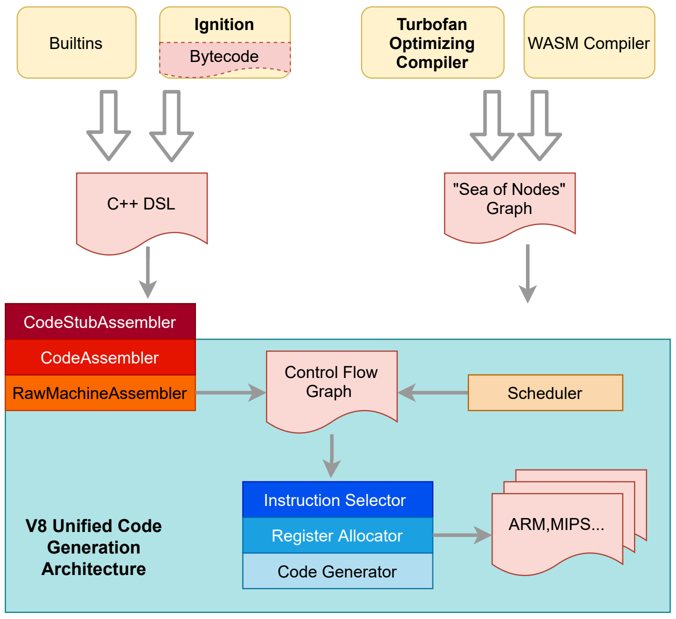


Figure 4.2: V8's Unified Code Generation

4.2 Design

4.2.1 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 [84], 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 [51], and bugs detected may also be false positives [48]. Thus, it is strongly desirable to analyze a JS script in its native environment under its normal usage.

Our approach conducts concolic testing on JS scripts in-situ, as illustrated in Figure 4.3. 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.

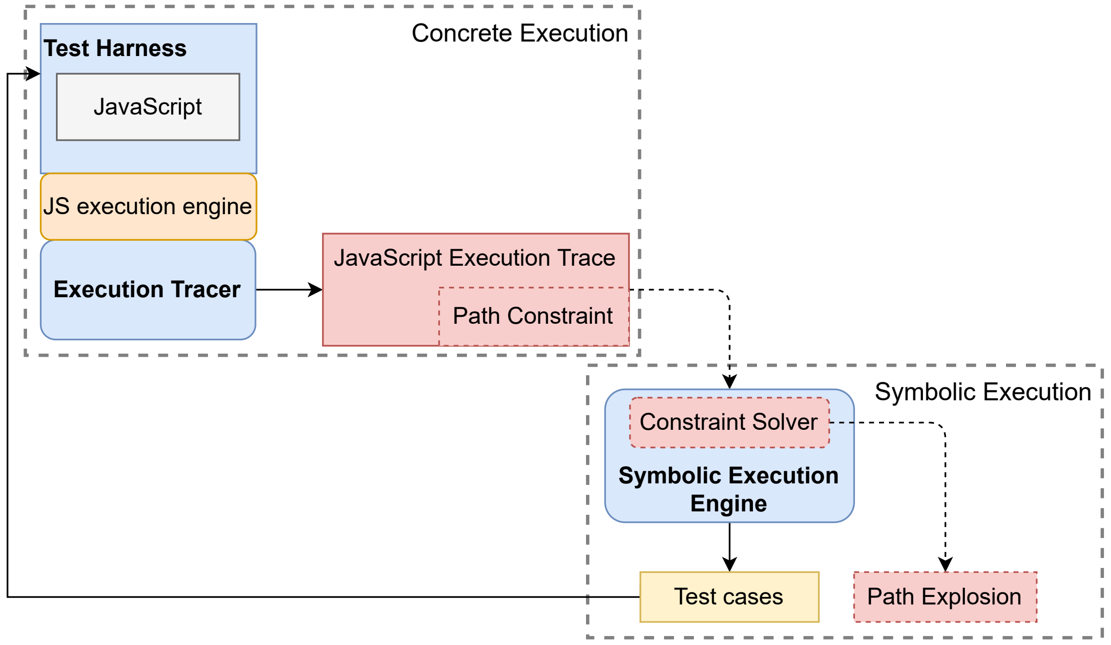


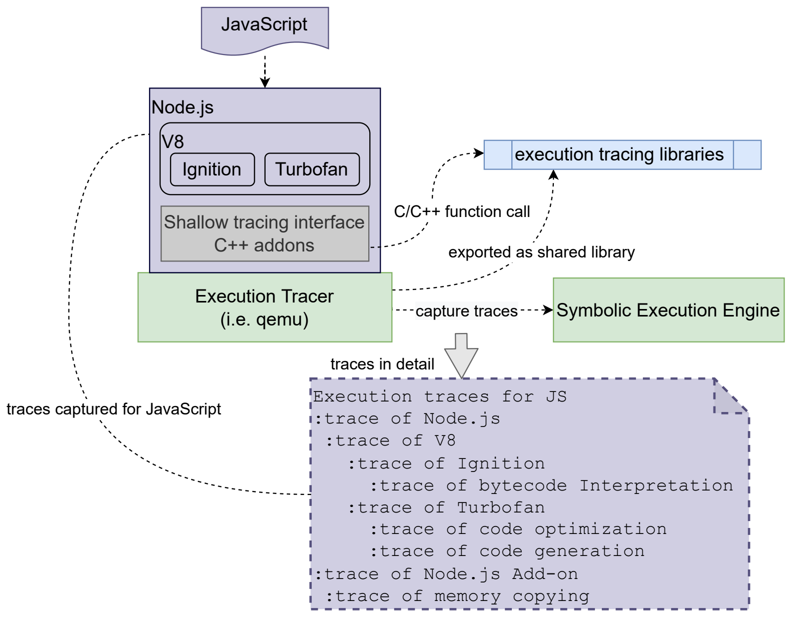
Figure 4.3: Workflow for Concolic Testing of JavaScript

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 [87].

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.


JavaScript
Node.js
V8
Ignition
Turbofan
Shallow tracing interface
C++ addons
execution tracing libraries
C/C++ function call
exported as shared library
Execution Tracer
(i.e. qemu)
capture traces
Symbolic Execution Engine
traces in detail
traces captured for JavaScript

```
Execution traces for JS
:trace of Node.js
 :trace of V8
   :trace of Ignition
     :trace of bytecode Interpretation
   :trace of Turbofan
     :trace of code optimization
     :trace of code generation
:trace of Node.js Add-on
 :trace of memory copying
```

Figure 4.4: Shallow Integration of Tracing in Node.js

4.2.2 Shallow Integration of Tracing in Node.js

As shown in Figure 4.4, 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 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.

4.2.3 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 byte-code of JS script under test. Therefore, the best place to trace such binary code is inside the V8 engine. As shown in Figure 4.5, 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 4.4.

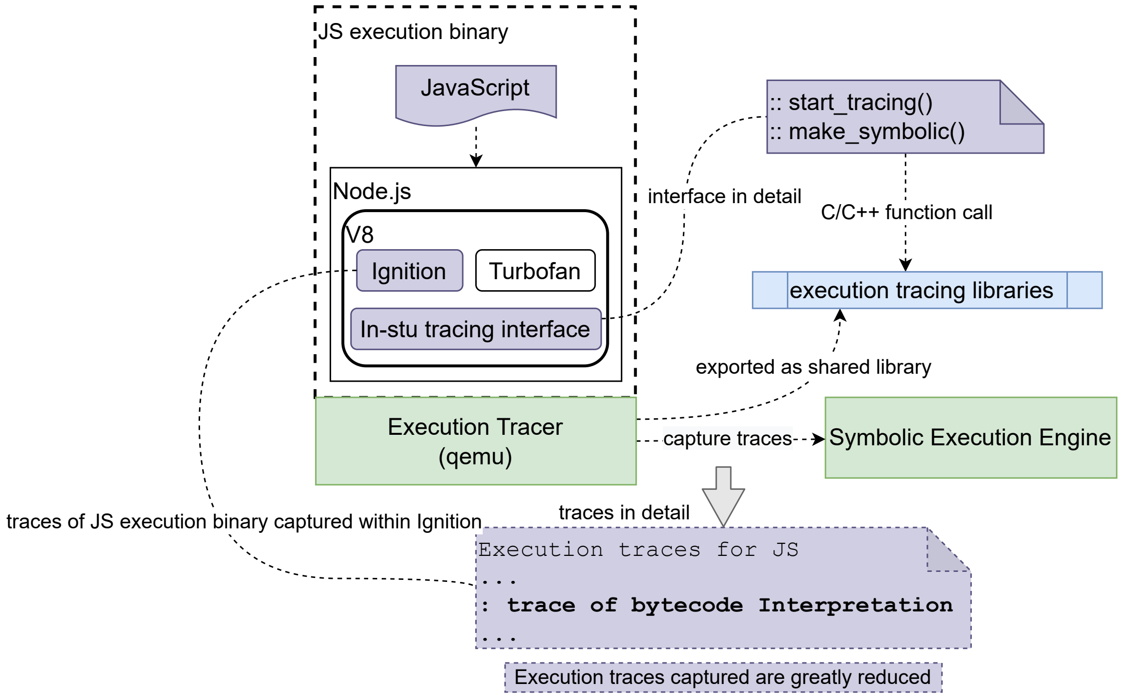
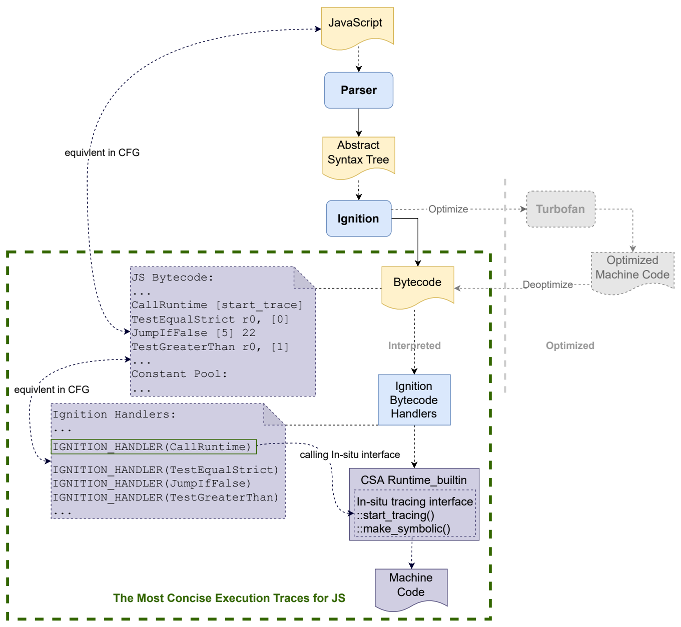


Figure 4.5: Deep Integration of Tracing in V8

JS bytecode interpretation happens in V8's Ignition interpreter. As shown in Figure 4.6, for each JS statement in bytecode, there is a corresponding bytecode handler in Ignition for its interpretation [3]. 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 4.6. 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 4.6. Thus, 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.

4.3 Implementation

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



```
JS Bytecode:
...
CallRuntime [start_trace]
TestEqualStrict r0, [0]
JumpIfFalse [5] 22
TestGreaterThan r0, [1]
...
Constant Pool:
...
```

```
Ignition Handlers:
...
IGNITION_HANDLER(CallRuntime)

IGNITION_HANDLER(TestEqualStrict)
IGNITION_HANDLER(JumpIfFalse)
IGNITION_HANDLER(TestGreaterThan)
...
```

Figure 4.6: How Deep Integration of Tracing Captures the Most Concise Execution Traces

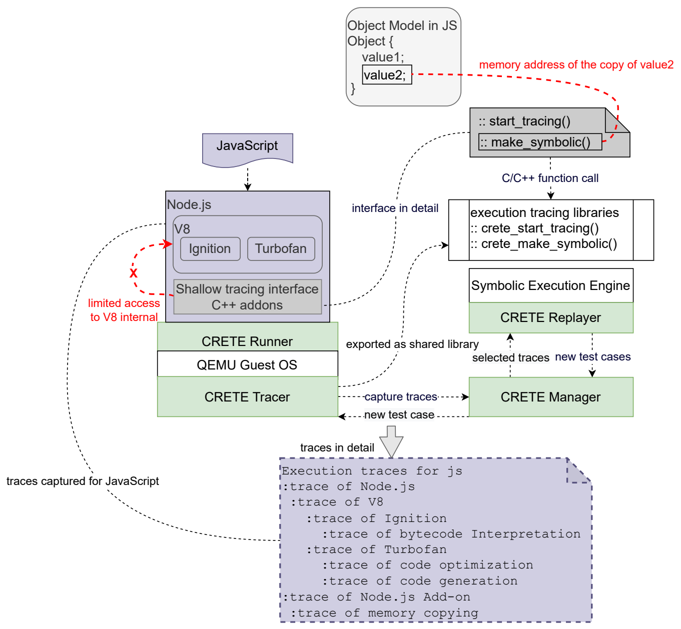
implement a new `addon` library in C++, which is later loaded into the Node.js runtime during JS script execution. This `addon` library wraps around the CRETE's tracing control interfaces and provides them to the JS script running on Node.js. This implementation requires no modification on Node.js, but 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 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.

4.3.1 Shallow Tracing Interface as C++ Addons

Figure 4.7 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 are first exported from CRETE execution tracer and can later be invoked from the JS script to mark symbolic variables

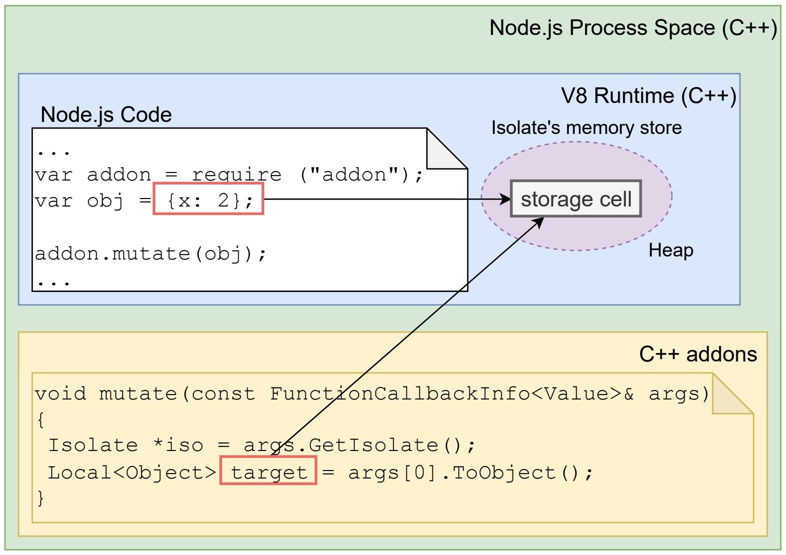


```
Execution traces for js
:trace of Node.js
 :trace of V8
   :trace of Ignition
     :trace of bytecode Interpretation
   :trace of Turbofan
     :trace of code optimization
     :trace of code generation
:trace of Node.js Add-on
 :trace of memory copying
```

Figure 4.7: Implementation of Shallow Tracing using Addons

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 4.8, the `addon` library has a different address space from V8 while V8 allocates JS variables within its own address space as storage cells [10]. 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.



```
...
var addon = require ("addon");
var obj = {x: 2};

addon.mutate(obj);
...
```

```
void mutate(const FunctionCallbackInfo<Value>& args)
{
 Isolate *iso = args.GetIsolate();
 Local<Object> target = args[0].ToObject();
}
```

Figure 4.8: 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.

4.3.2 Deep Tracing Interface as V8 Builtins

Figure 4.9 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 4.1.2, *V8 JS Engine*). 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

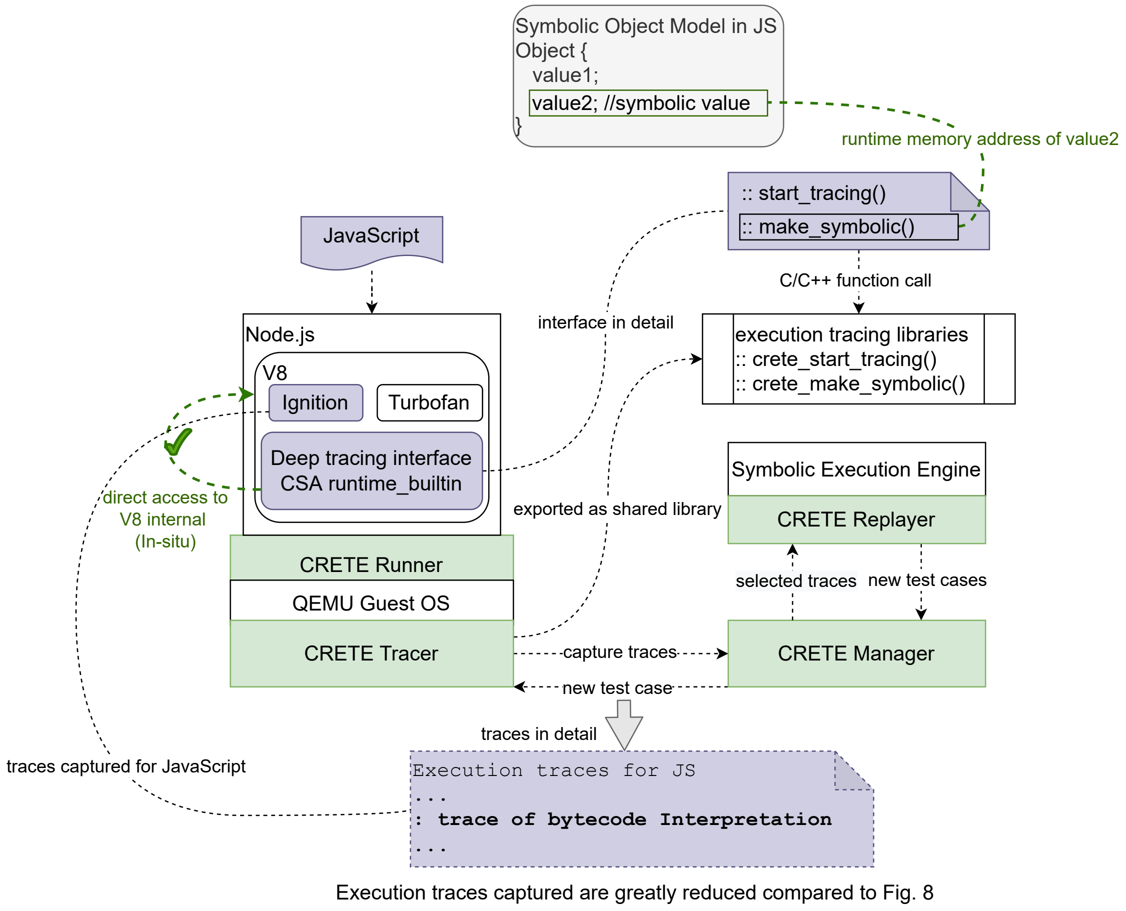


Figure 4.9: Deep Tracing Interface in V8

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 4.1. 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.

4.3.3 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 [5]. As shown in Figure 4.10, 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 4.10 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.

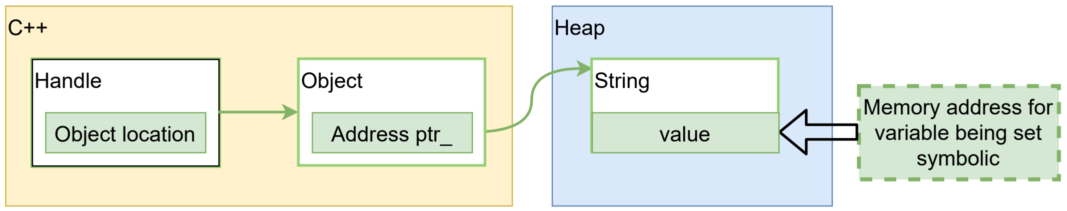


Figure 4.10: V8 Object Memory Model

We encountered four cases when attempting to retrieve the memory address of the actual value of the `String` object for symbolic execution [4], 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 4.1 and Listing 4.2 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 4.2. The actual value stored in this runtime address is loaded at line 5 of Listing 4.2 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 4.2, as the concrete execution trace. The captured trace also preserves all constraints corresponding to the JS script of Listing 4.1. Thus, the traces captured with our method are concise and accurate for symbolic execution.

```
var str_var = "init";

%MakeSymbolic(str_var);
%StartTracing();

if( str_var === "tests")
    return "tests";

if( str_var > "tests1"){
    return "tests1";
}else{
    return "tests2"
}
```

Listing 4.1: A Simple Example of JavaScript and Calling Convention of In-Situ Tracing Interfaces

4.4 Evaluation

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 a 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

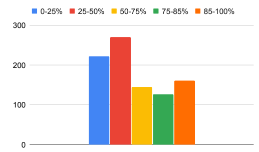
```
[generated bytecode for function: ]
Parameter count 6
Frame size 8
  0x40b8ec2c9a StackCheck
  0x40b8ec2c9b LdaConstant [0]
  0x40b8ec2c9d Star r0
  0x40b8ec2c9f CallRuntime [MakeSymbolic],r0-r0
  0x40b8ec2ca4 CallRuntime [StartTracing]
  0x40b8ec2ca9 LdaConstant [1]
  0x40b8ec2cab TestEqualStrict r0,[0]
  0x40b8ec2cae JumpIfFalse [5](0x40b8ec2cb3)
  0x40b8ec2cb0 LdaConstant [1]
  0x40b8ec2cb2 Return
  0x40b8ec2cb3 LdaConstant [2]
  0x40b8ec2cb5 TestGreaterThan r0,[1]
  0x40b8ec2cb8 JumpIfFalse [5](0x40b8ec2cbd)
  0x40b8ec2cba LdaConstant [2]
  0x40b8ec2cbc Return
  0x40b8ec2cbd LdaConstant [3]
  0x40b8ec2cbf Return
  0x40b8ec2cc0 LdaUndefined
  0x40b8ec2cc1 Return
Constant pool (size = 4)
- map: 0x01eccde023c1 <Map>
- length: 4
       0: 0x34ecf6d42849 <String[4]: init>
       1: 0x0040b8ec2949 <String[5]: tests>
       2: 0x0040b8ec2969 <String[6]: tests1>
       3: 0x0040b8ec2989 <String[6]: tests2>
```

Listing 4.2: Bytecode for JS Script in Listing 4.1

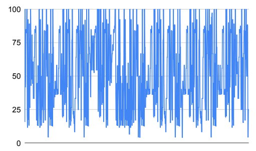
in CRETE for each Node.js library automatically. With the test harness and automation pipeline we can set up concolic testing for Node.js libraries conveniently and have applied our approach to 995 Node.js libraries which include approximately 9000 JS files. Our current study focuses on string-intensive 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.

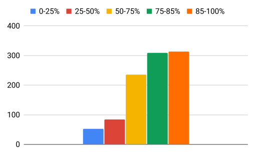
The overall statement coverage on all 995 Node.js libraries for shallow tracing and deep tracing is shown in Figure 4.11. Figure 4.11d and Figure 4.11b show that deep tracing via V8 builtin functions performs significantly better than shallow tracing via Node.js `addons` in terms of statement coverage. The darker shadow between 75% and 100% in Figure 4.11d indicates that more libraries achieved the coverage between 75% and 100% with deep tracing. Figure 4.11a and Figure 4.11c show the exact number of libraries in each coverage range.



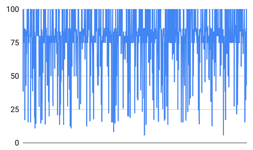
(a) Shallow Tracing: Number of libs in each coverage range



(b) Shallow Tracing: Statement coverage distribution



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



(d) Deep Tracing: Statement coverage distribution

Figure 4.11: 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 [11] and compatible with most JavaScript testing frameworks, e.g., Mocha [13] and Node-Tap [14]. 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 4.1 shows the demographics of the selected libraries. The LoC (lines of code) for a library under test is calculated with *github-loc* [9]. The number of weekly downloads of a library under test is calculated with

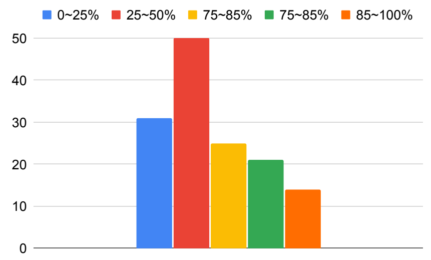
Metric	**Range**	**Average**
Line of Code	[93, 16910]	1687
Weekly Downloads	[3, 37491350]	9552965
Dependencies	[3, 18154]	282

Table 4.1: Demographics for Libraries under Test

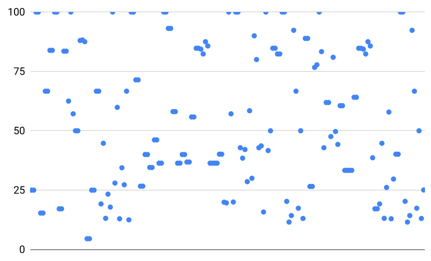
npm-stats-api [17]. The number of dependencies is the number of dependent libraries that the library under test has. We calculated it with *dependent-counts* [7].

4.4.1 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 4.12a, 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 4.12b 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 4.13, which shows the average coverage growth trends when the number of libraries grows. It can be observed from Figure 4.11a and Figure 4.12a 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

Figure 4.12: Coverage Achieved by Shallow Tracing

4.4.2 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 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 4.14b. 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 4.14a, 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.

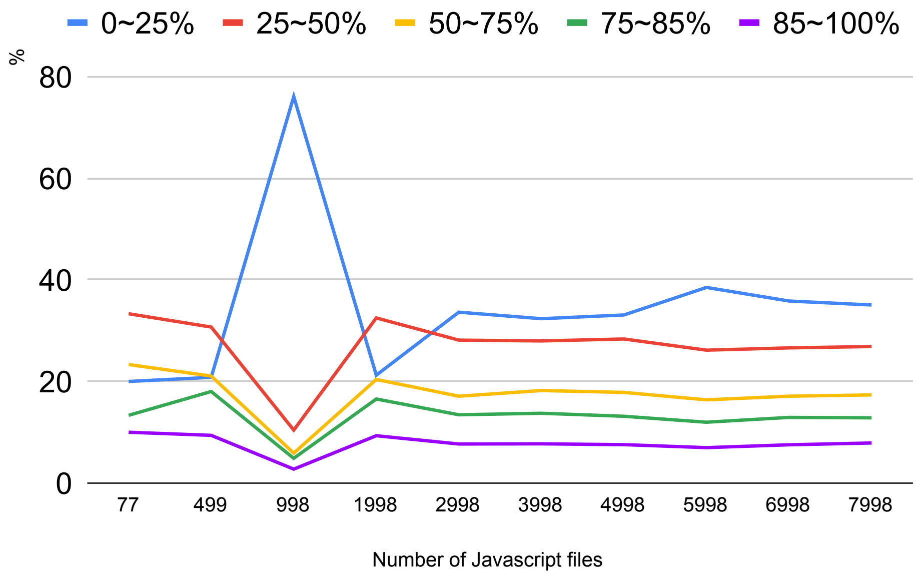
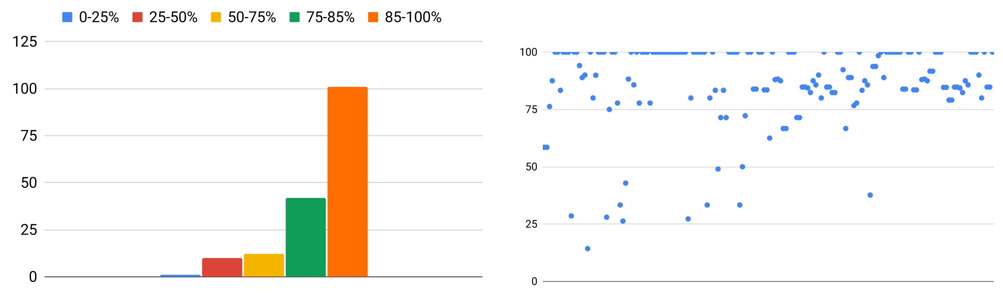


Figure 4.13: Coverage Growth Trend with Shallow Tracing

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 4.3.2 and 4.3.3. It can be observed from Figure 4.11c and Figure 4.14a that the overall coverage on 995 libraries closely resembles that of 180 representative libraries randomly selected.

4.4.3 Comparisons

Test Coverage Achieved by NPM Test Suites A systematic investigation on test coverage of hand-craft test suites in NPM [93] is illustrated in Figure 4.15. 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%,



(a) Number of Libraries in Each Coverage Range

(b) Statement Coverage Distribution

Figure 4.14: Coverage Achieved by Deep Tracing

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.

Performance Comparison between Shallow and Deep Tracing For comparison, it can be observed from Figure 4.16a and Figure 4.16b that the number of libraries achieving code 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

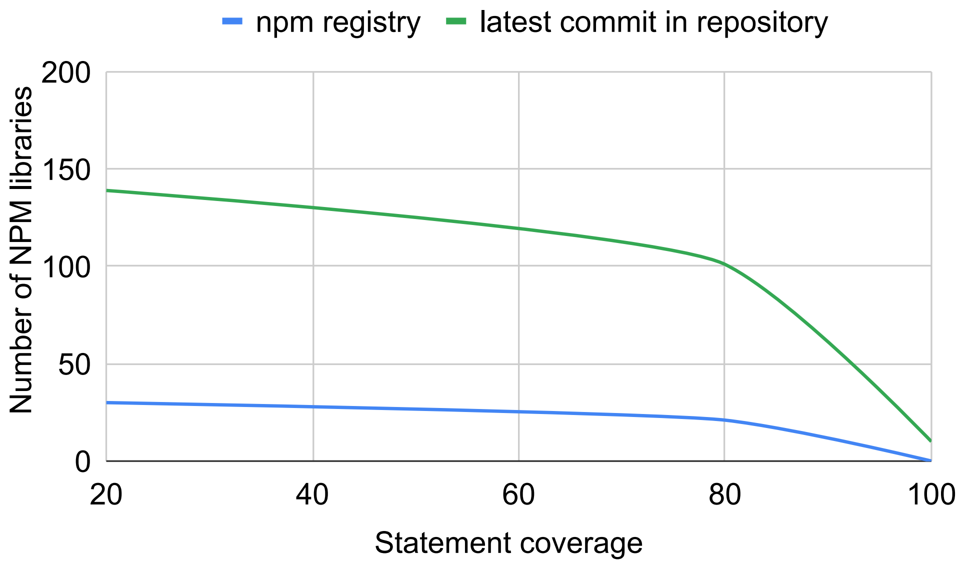
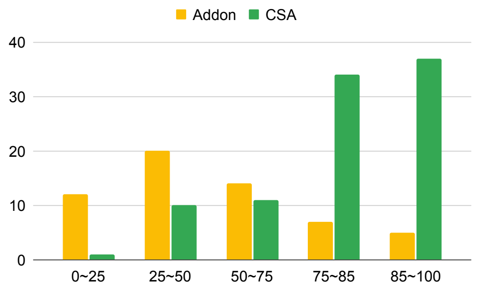


Figure 4.15: Coverage by Hand-Crafted NPM Test Suites

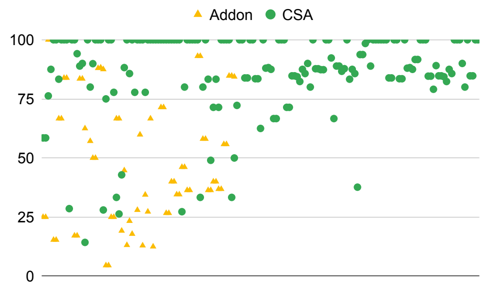
ability to achieve higher coverage in JavaScript libraries at the cost of extending the V8 engine with new builtins.

Comparison with Related Work We have compared our approach with an existing tool, ExpoSE [71]. ExpoSE has been evaluated on 4 JS libraries shown in Figure 4.17. 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.

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



(a) Number of Libraries in Each Coverage Range



(b) Statement Coverage Distribution

Figure 4.16: Coverage Comparison: Shallow vs. Deep Tracing

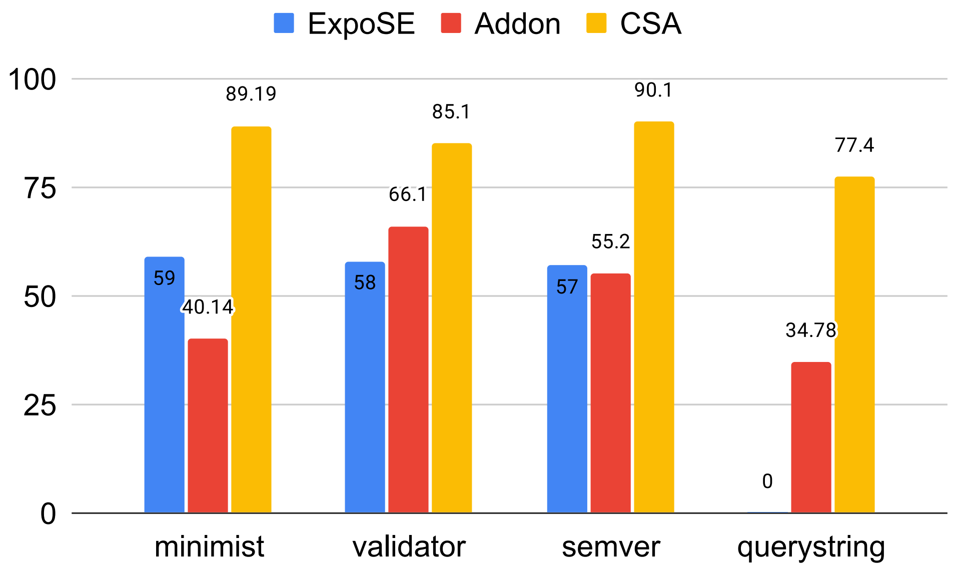


Figure 4.17: Comparison with ExpoSE

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 4.2: Bugs Detected in 12 NPM Libraries

Table 4.2 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.

4.4.4 Discussions and limitations

A test case typically has two parts: test inputs and output assertions. In our study, we focus on generating test inputs and utilize default assertions for testing, e.g., exceptions and reuse assertions that user previously defined for existing test cases. 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 [87]. 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. The effectiveness of our approach also depends on the symbolic object model that we can handle. As we make more types of objects symbolic, our approach can potentially become more powerful.

Our implementation is based on CRETE which uses QEMU as its tracing platform [44]. 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, this could lead to path explosions. Additional algorithms are needed to handle multi-threaded executions efficiently, which is not the focus of this paper.

4.5 Summary

In this Chapter, we have presented a novel approach to in-situ concolic testing of JS scripts. This approach enables concolic execution of JS scripts in their native environments and is able to automatically generate test cases that achieves comparable, if not better, code coverage than manually crafted unit test suites for Node.js libraries and discover previously unknown bugs in these libraries. We will further extend this approach to support a wider range of JS scripts, e.g., those executing in web browsers and those following application frameworks that build on Node.js, e.g., Express in Chapter 5. 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, which we will explain in Chapter 6.

5 Concolic Execution of Front-end JavaScript

In this Chapter, we introduce a novel approach to concolic testing of front-end JS web applications based on in-situ concolic testing. This approach leverages widely used JS testing frameworks such as *Jest* and *Puppeteer* and conducts concolic execution on JS web functions for unit testing [95]. These testing frameworks isolate the web function under test in the context of its embedding web page by mocking the environment, and provide the test data that drives the function. This isolation of web function provides an ideal target for the application of concolic testing. We integrate concolic testing APIs into these testing frameworks. The seamless integration of concolic testing allows the injection of symbolic variables within the native execution context of a JS web function and the precise capture of concrete execution traces of this function. As the testing framework executes the function under test with test data, parts or all of the test data can be made symbolic and the resulting execution traces of the function are captured for later symbolic analysis. Concise execution traces greatly improve the effectiveness and efficiency of the subsequent symbolic analysis for test generation. The new test data generated by the symbolic analysis is again fed back to the testing frameworks to drive further concolic testing.

5.1 Background

5.1.1 Front-end JavaScript Testing Frameworks

In a general software testing framework, a test case is designed to exercise a single, logical unit of behavior in an application and ensure the targeted unit operates as expected [39]. Typically, it is structured as a tuple $\{P, C, Q\}$:

- P are the preconditions that must be met so that the function under test can be executed.
- C is the function under test, containing the logic to be tested.
- Q are the post assertions of the test case that are expected to be true.

As shown in Figure 5.1, a front-end JS testing framework inspects the web application in the browser for JS functions to test.

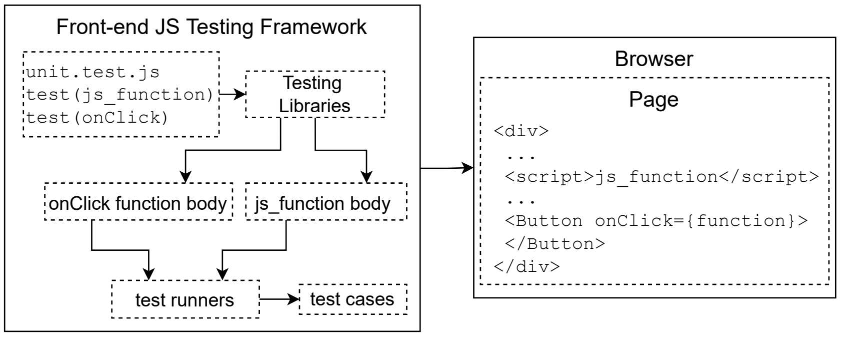


Figure 5.1: Front-end JS testing framework workflow

It utilizes testing libraries to obtain the web pages, parses them and stores page functions and their context information individually so that test runners can run the functions browser-less [24]. The test runner sets up the three parts of a test case for each JS function under test and then executes the test case. The front-end JS testing framework helps isolate the JS function under test and provides the execution context for testing the function, which is an ideal entry for our application of the concolic testing to front-end JS.

5.1.2 In-situ Concolic Testing of Backend JavaScript

In Chapter 4, we introduced to applying concolic testing to backend JS in-situ [70], i.e., scripts are executed in their native environments (e.g., Node.js) as part of concolic execution and test cases generated are directly replayed in these environments [15]. As illustrated in Figure 4.3, the concrete execution step of concolic testing as indicated by the dashed box on top is conducted in the native execution environment for JS, 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 that are then fed back into the native concrete execution to drive further test case generation. This approach has been implemented on the Node.js execution environment and its V8 JS engine [21]. As a script is executed with Node.js, its binary-level execution trace is captured and later analyzed through symbolic execution for test case generation. It also offers the flexibility of customizing trace as needed. We leverage this functionality in our approach.

5.2 Design

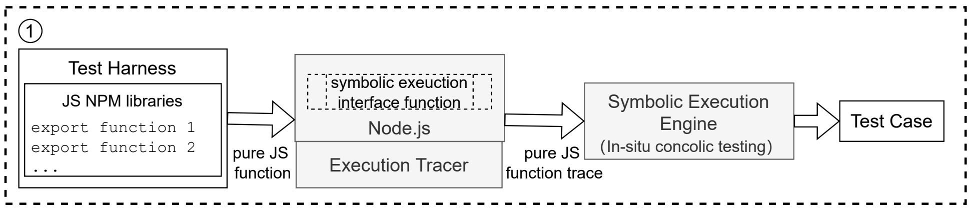
5.2.1 Overview

Our approach strives to apply concolic testing on front-end JS web applications to generate effective test data for unit testing of these applications. Below are the specific design goals for our approach:

- **Front-end JS Extraction.** JS web functions need to be extracted from web pages to execute independently to reduce complexity for concolic testing.
- **Execution Context Construction.** JS web functions under test need to have the same execution environments as they are executed in the web pages.
- **Non-intrusive and Effective Concolic Testing.** Concolic execution on JS web applications needs to require minimal changes on both the applications and the symbolic engine and generate useful test cases effectively.

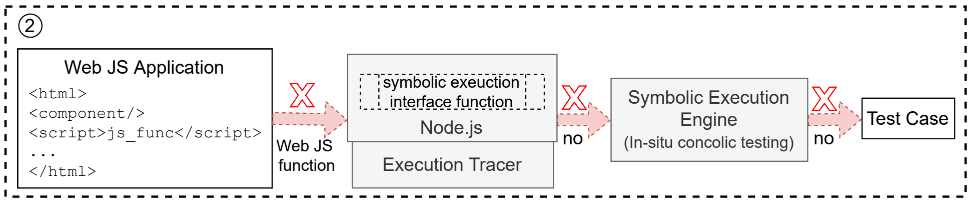
With the above goals in mind, we design an approach to concolic testing of front-end JS web applications, which leverages the JS testing frameworks such as Jest and Puppeteer and conducts concolic execution on JS web functions for unit testing. The seamless integration of concolic testing with these testing frameworks is achieved through extending in-situ concolic testing of backend JS applications. Figure 5.2 illustrates how the integration is realized:

1. Workflow 1 in Figure 5.2a illustrates the original capability of in-situ concolic testing of backend JS applications. It tests pure JS functions from NPM JS libraries.



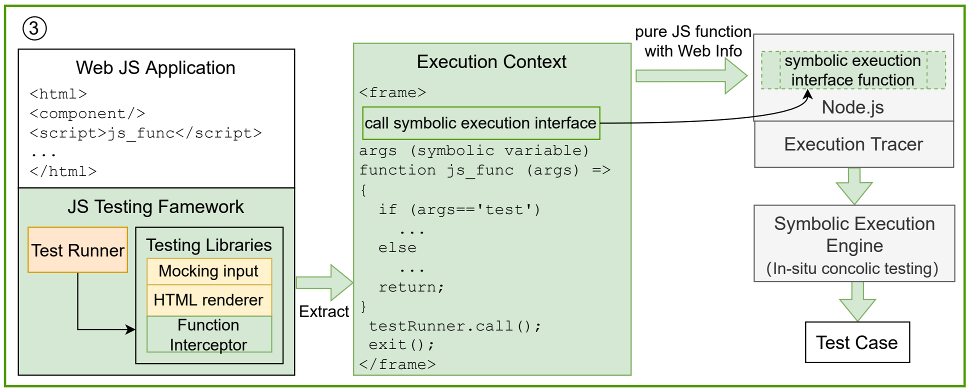
```
export function 1
export function 2
...
```

(a) Original workflow of in-situ concolic testing


2
Web JS Application
Web JS function
symbolic execution interface function
Node.js
Execution Tracer
no
Symbolic Execution Engine
(In-situ concolic testing)
no
Test Case

```
<html>
<component/>
<script>js_func</script>
...
</html>
```

(b) Naïve execution of JS web function directly with in-situ concolic testing



```
<html>
<component/>
<script>js_func</script>
...
</html>
```

```
<frame>
args (symbolic variable)
function js_func (args) =>
{
  if (args=='test')
    ...
  else
    ...
  return;
}
 testRunner.call();
 exit();
</frame>
```

(c) Workflow for enabling effective in-situ concolic testing on front-end JS

Figure 5.2: Overview for concolic testing of front-end JS

The execution tracer captures the traces of the pure JS functions and feeds them to the symbolic execution engine to generate new test data.

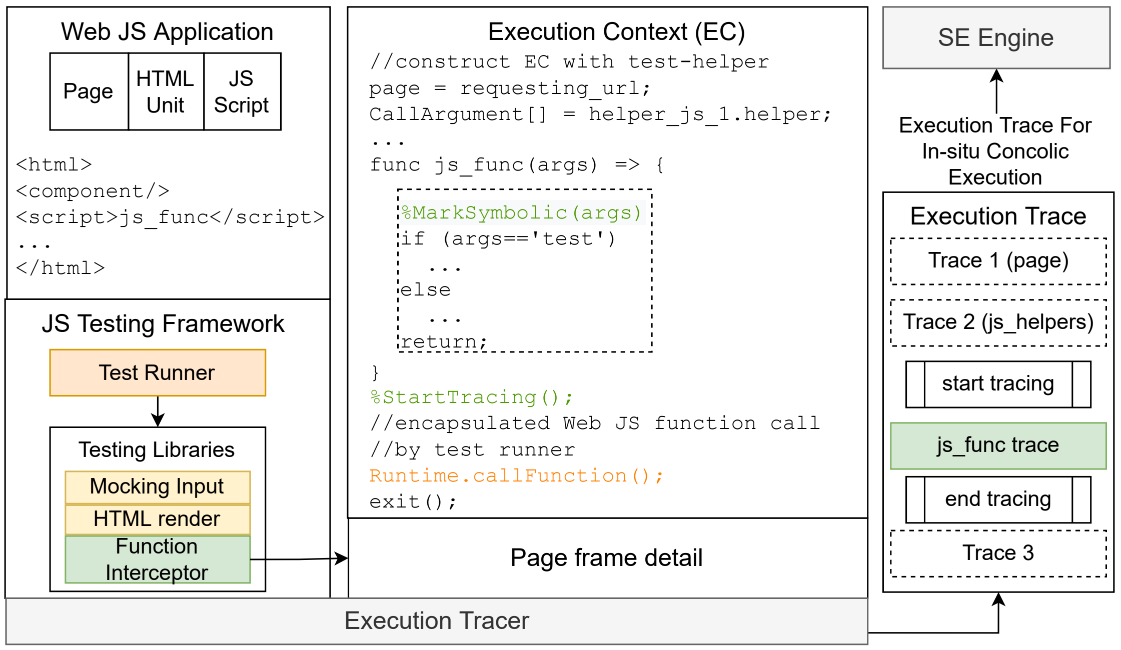
2. Workflow 2 in Figure 5.2b illustrates a naïve application of in-situ concolic testing to a JS web application. However, in-situ concolic testing cannot handle web elements, e.g., <HTML> tags, without the capability of a browser.

3. Workflow 3 in Figure 5.2c illustrates how we leverage a JS testing framework to extract the front-end JS web function and its execution context from the web page. In the extraction, we encapsulate them as a pure JS function augmented with the web page information, inject symbolic values and capture execution traces for later symbolic analysis by calling the symbolic execution interface functions within the extracted execution context. We then utilize the test runner of the JS testing framework to initiate and drive concolic testing within the execution context to generate new test data.

This workflow allows faithful simulation of the execution context of a JS web function without the presence of a web browser. It enables injection of symbolic variables and captures of concrete execution traces within the execution context of the JS web function under test. A concise and accurate concrete execution trace can greatly improve the effectiveness and efficiency of the following symbolic analysis for test generation. We explain how to decide the starting point of tracing within the native execution context and what difference it makes in Section 5.2.4.

5.2.2 Concolic Testing of JS Web Function within Execution Context

A front-end JS web function is invoked from a web page and its execution depends on the execution context from the web page [59]. The core of our approach is to enable concolic testing on the JS web function within its native execution context from the web page in a manner same as in-situ concolic execution of back-end JS. We can achieve this by the following three steps: execution context extraction, execution context tracing customization (including symbolic value injection and tracing control), and concolic execution within execution context.



```
<html>
<component/>
<script>js_func</script>
...
</html>
```

```
//construct EC with test-helper
page = requesting_url;
CallArgument[] = helper_js_1.helper;
...
func js_func(args) => {
  %MarkSymbolic(args)
  if (args=='test')
    ...
  else
    ...
  return;
}
%StartTracing();
//encapsulated Web JS function call
//by test runner
Runtime.callFunction();
exit();
```

Figure 5.3: Concolic testing of JS Web function within execution context

5.2.3 Execution Context Extraction

To transform a JS web function to a pure JS function without losing the context of a web page, we introduce a `function interceptor` to the JS testing framework to serve this purpose. As shown in Figure 5.3, the `function interceptor` completes the following tasks to finish this transformation in order to suit later in-situ concolic testing in the back end:

- First, the `function interceptor` requests the page frame detail of the web page where the targeted JS web function resides, utilizing the existing mocking data and the `HTML render` function. The mocking data and the `HTML render` function are usually created manually and included in the unit test suite.

- Second, from the page frame detail, the `function interceptor` identifies the function body in a pure JS form given the function name. To preserve the JS function's native web environment, it extracts the associated execution context of the web page. This is realized by calling helper functions provided by the testing libraries of the JS testing framework. The execution context contains everything that is needed for the pure JS function to be executed in the web page, which includes the arguments of the function, its concrete dependency objects set by mocking data and the function scope.

- Third, the `function interceptor` delivers a complete function in the pure JS form encapsulated with its associated web execution context by assembling them, and then makes it accessible for the test runner of the JS testing framework so that the test runner can initiate the concolic execution in the execution context when running the test suite.

5.2.4 Execution Context Tracing Customization

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 [20,70]. The conciseness of an execution trace determines the efficiency and effectiveness of later symbolic analysis and test case generation. Therefore, to make the most of this capability, we pinpoint the locations of where to introduce symbolic values and start tracing during the extraction of the execution context, before we commence concolic testing on the encapsulated JS web function with its execution context. In-situ concolic testing provides interface functions for introducing the symbolic values (`MarkSymbolic()`) and tracing control (`StartTracing()`). We use these interface functions to customize execution context tracing as needed.

Symbolic Value Injection and Tracing Control A JS testing framework uses a test runner to execute its test suites. As shown in Figure 5.4, the test runner prepares the dependencies for setting up the testing environment and loads the JS libraries the test suites need before starting run the individual function under test. In order to avoid

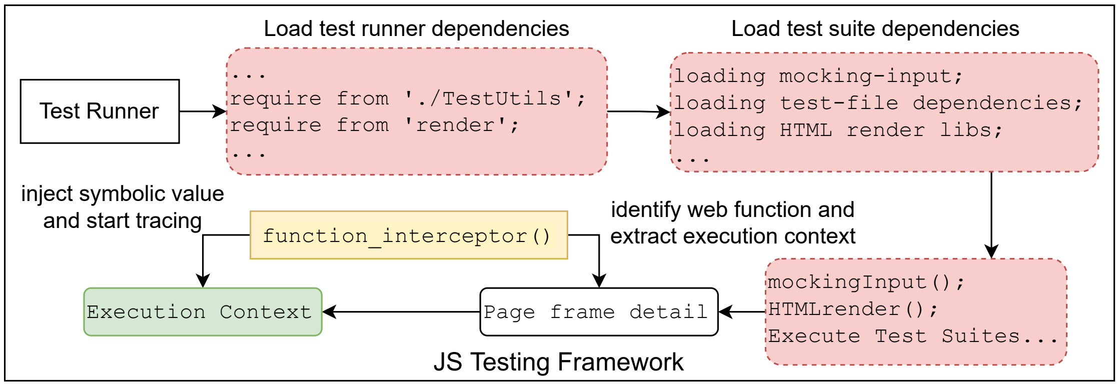

Load test runner dependencies
Load test suite dependencies
Test Runner
...
require from './TestUtils';
require from 'render';
...
loading mocking-input;
loading test-file dependencies;
loading HTML render libs;
...
inject symbolic value and start tracing
function_interceptor()
identify web function and extract execution context
Execution Context
Page frame detail
mockingInput();
HTMLrender();
Execute Test Suites...
JS Testing Framework

Figure 5.4: How to avoid unnecessary tracing of the test runner setup by delaying the injection of symbolic values and the start of tracing

tracing the unnecessary startup overhead of the test runner (indicated by the red box in Figure 5.4), we choose to inject symbolic values inside the execution context and start tracing when the test runner actually executes the encapsulated function, by calling the interface functions the in-situ concolic testing provides. This way the execution tracer only captures the execution trace of the encapsulated JS web function. The locations for injecting symbolic values and starting tracing are indicated in the ”Execution Context

(EC)” box in Figure 5.3 and the captured execution trace is indicated by the ”Execution Trace” box in the right corner of Figure 5.3.

Most Concise Execution Trace Figure 5.5 shows why our approach can obtain the most concise execution trace for the JS web function driven by the test runner of the JS testing framework.

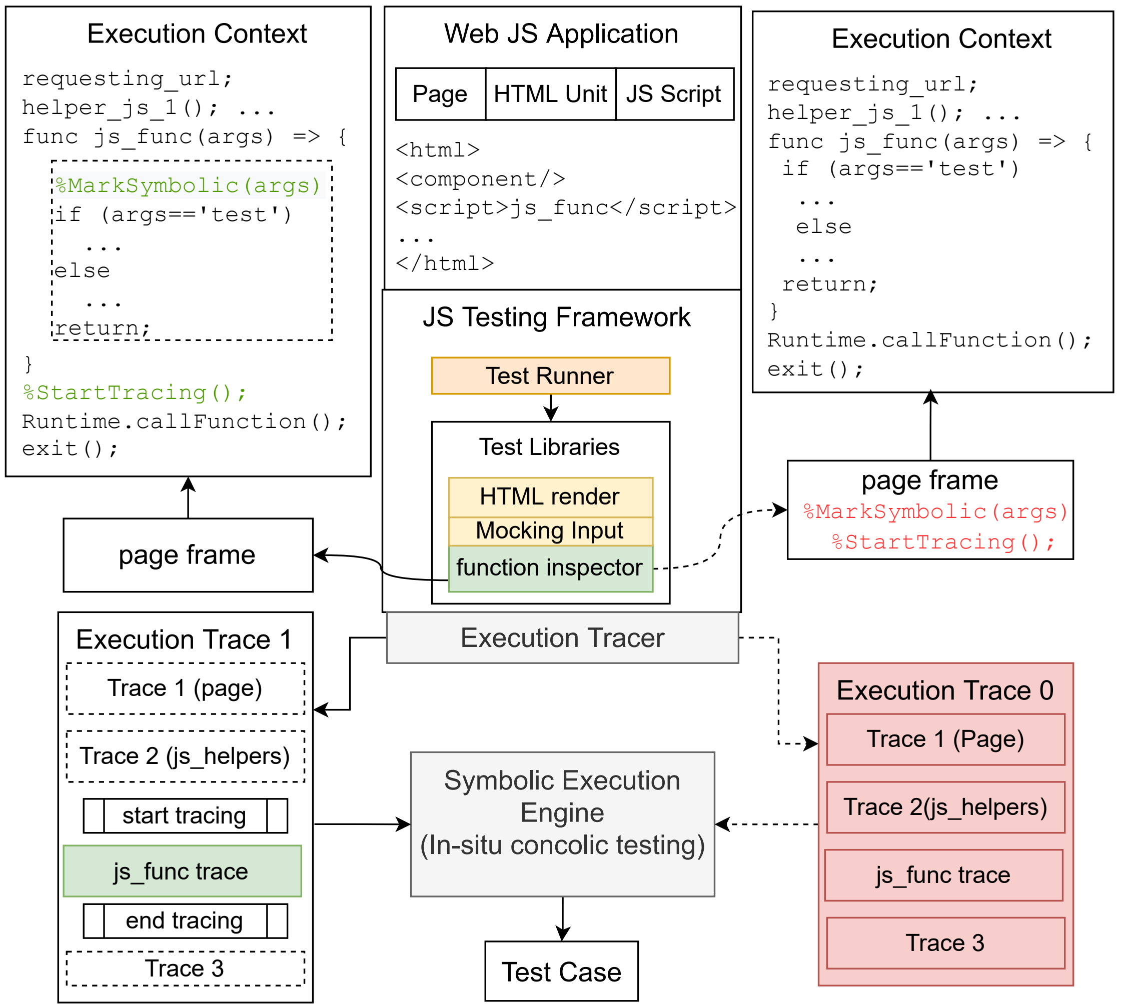


Figure 5.5: How we obtain the most concise concrete execution trace

Apart from the overhead caused by the test runner, the extraction of the execution context for the JS web function involves calling a set of JS helper functions to collect web page information, such as `helper_js_1` and `JSHandle_js_1`. If we directly apply symbolic execution within the test runner where the JS function is intercepted along with the execution context extraction, the execution tracer will also capture the execution traces of the test runner and the testing helper functions from the testing libraries shown as ”Execution Trace 0” in the right-hand side of Figure 5.5. We modified the test runner to mark symbolic variables and enable tracing control within the execution context. Instead of starting tracing when the test runner starts, we defer the tracing of the execution to when and where the test runner actually executes the encapsulated function under test in the extracted execution context, indicated by the ”Execution Trace 1” in the left-hand side of Figure 5.5. This way we minimize the extend of execution tracing needed.

5.2.5 Concolic Testing within Execution Context

We leverage the test runner of the JS testing framework to initiate and start the in-situ concolic testing of the JS web function under test. Typically the test runner starts running the JS web function with an existing unit test. In our approach, the execution of the unit test triggers the `function interceptor`, which starts the process of extracting the execution context and encapsulating the target JS web function. During this process, symbolic values are injected and tracing is started in the right place as described in previous sections. The resulting pure JS application is then executed by in-situ concolic testing. Newly generated test data is fed back to the JS testing framework to drive further concolic testing.

5.3 Implementation

In this section, we demonstrate the feasibility of our approach to concolic testing of front-end JS functions by implementing it on two popular JS testing frameworks, namely Puppeteer and Jest assisted by the *React* testing library [33, 37].

5.3.1 Implementation on Puppeteer

Puppeteer is a testing framework developed by the Chrome team and implemented as a Node.js library [33]. It provides a high-level API to interact with headless (or full) Chrome. It can simulate browser functions using testing libraries. Puppeteer can execute JS functions residing in a web page without a browser. Puppeteer allows us to easily navigate pages and fetch information about those pages. In the implementation of our approach on Puppeteer, we augment it with the implementation of the `function interceptor` to identify the targeted web JS functions and extract their execution contexts from the web pages and encapsulate them for in-situ concolic testing.

Encapsulating JS Web Function with Execution Context As shown in Figure 5.6, Puppeteer communicates with the browser [34]. One browser instance can own multiple browser contexts. A `Browser Context` instance defines a browsing session and can have more than one page. The `Browser Context` provides a way to operate an

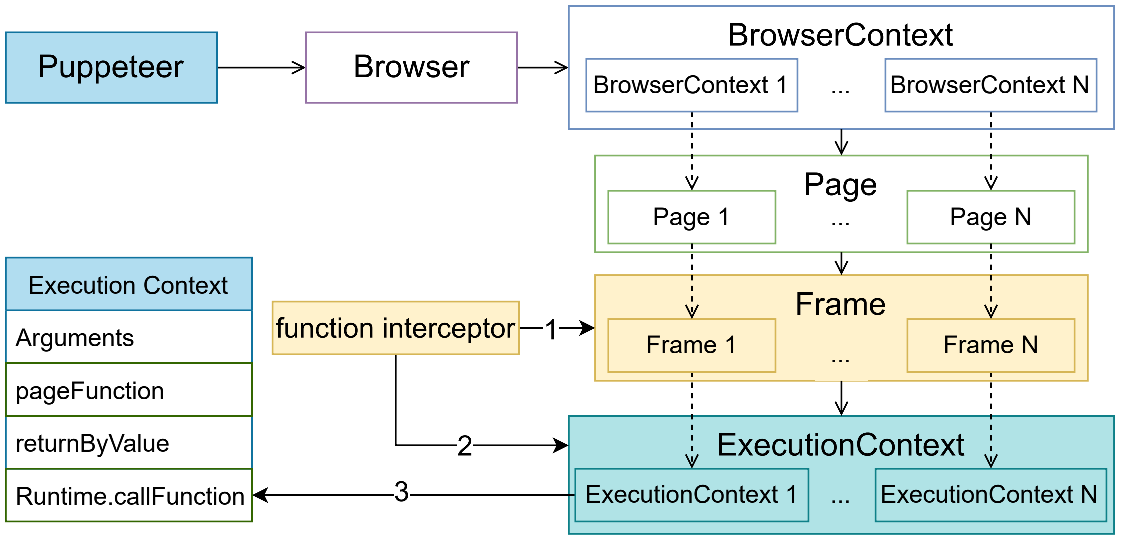


Figure 5.6: How Puppeteer executes a JS function in a web page

independent browser session [23]. A `Page` has at least one frame. Each frame has a default execution context. The default execution context is where the frame's JavaScript is executed. This context is returned by `frame.executionContext()` method, which gives the detail about a page frame. We implement the `function interceptor` in the *Execution Context* class under the browser context to collect necessary information for encapsulating a JS function with its associated web execution context. The *Execution Context* class represents a context for JS execution in the web page. We modified it to identify the page function, its arguments and return value [25]. The `pageFunction` is the function in the `HTML` page to be evaluated in the execution context, which is in a pure JS form. For example, Listing 5.1 shows a front-end application example written with the *Express* web development framework [26]. This example contains a web page (from line 7 to line 17) with a JS web function marked by `<script>` tag in line 15. The `${path}` points to the JS file that contains the implementation of the JS web function, as shown in Listing 5.2. Our approach is able to encapsulate the pure JS form of the web JS function (its implementation) with its associated web execution context.

Listing 5.1: An example of a front-end web application using Express framework

```
const app = express()
  .use(middleware(compiler, { serverSideRender: true }))
  .use((req, res) => {
    const webpackJson = res.locals.webpack.devMiddleware.stats.toJson()
    const paths = getAllJsPaths(webpackJson)
    res.send(
      `<!DOCTYPE html>
      <html>
        <head>
          <title>Test</title>
        </head>
        <body>
          <div id="root"></div>
          ${paths.map((path) =>
          `<script src="${path}"></script>`).join('')}
        </body>
      </html>`
    )
  })
```

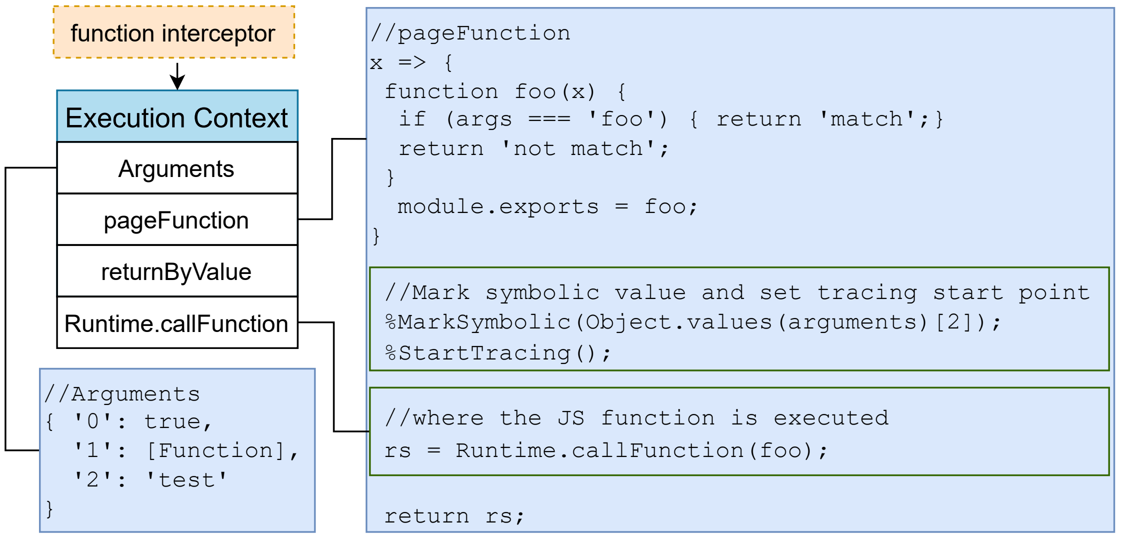
Listing 5.2: An example of a front-end JS script under Express framework

```
function foo(args) {
  if(args === 'foo'){
    return 'match';
  }
  return 'not match';
}
module.exports = foo;
```

Execution Context Tracing Customization We utilize the *page.evaluate* function of the *Puppeteer* testing framework to drive the JS function under test and extend it with the `function interceptor`. As described in Figure 5.7, to enable customized execution context tracing, the `function interceptor` introduces symbolic variables and set the starting point for tracing within the web execution context of the JS function wrapped by the `<script>` tag in the web page. This way, we make it possible for the test runner to initiate concolic testing when it starts running the test suites so that JS function can be tested concolically and automatically without tracing additional overheads. Since the `Execution Context` is triggered by the `evaluate` function in unit tests. We target applications from GitHub that uses *Puppeteer* to test front-end features and utilizes `evaluate` in unit testing. We will discuss the results later in Section 5.4.

5.3.2 Implementation on Jest with React Testing Library

Another implementation of our approach is on the *Jest* testing framework assisted by the *React* testing library for unit testing. The *React* testing library is a lightweight library for testing *React* components that wrap the JS functions with the `HTML` elements [37]. As shown in Figure 5.8, there are three components in the application as indicated by the numbers. Components allow splitting of an UI into independent, reusable pieces, and



```
//Arguments
{ '0': true,
  '1': [Function],
  '2': 'test'
}
```

```
//pageFunction
x => {
 function foo(x) {
  if (args === 'foo') { return 'match';}
  return 'not match';
 }
  module.exports = foo;
}
 //Mark symbolic value and set tracing start point
 %MarkSymbolic(Object.values(arguments)[2]);
 %StartTracing();

 //where the JS function is executed
 rs = Runtime.callFunction(foo);

 return rs;
```

Figure 5.7: How we set symbolic variables in the execution context and enable customized execution context tracing in *Puppeteer*

designing each piece in isolation. *React* is flexible; however, it has a strict rule: all *React* components must act as pure functions with respect to their inputs [35]. We refer to them as ”functional components”. They accept arbitrary inputs (called “props”) and return React elements describing what should appear on the web page [36]. An individual component can be reused in different combinations. Therefore, the correctness of an individual component is important with respect to the correctness of their compositions. In our implementation, we only consider components that have at least one input. Jest has a test runner, which allows us to run tests from the command line. Jest also provides additional utilities such as mocks, stubs, etc., besides the utilities of test cases, assertions, and test suites. We use Jest's mock data to set up the testing environment for the front-end components defined with *React*. Figure 5.9 shows how we leverage and extend *Jest* assisted by *React* testing library to apply the in-situ concolic testing to React component.

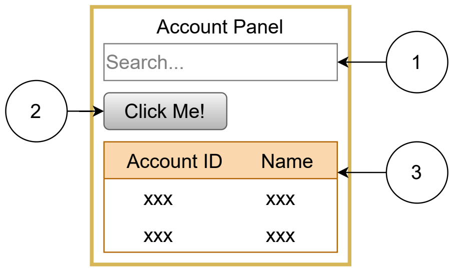
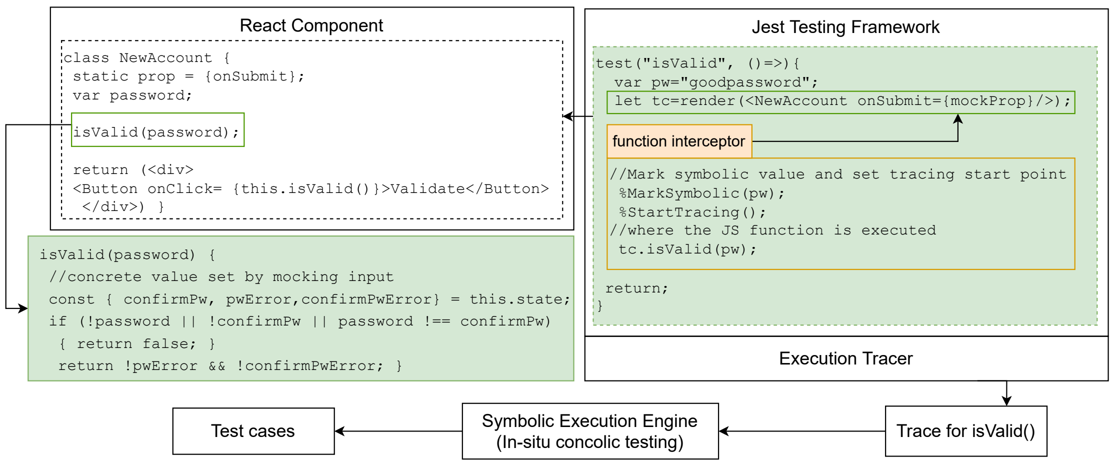


Figure 5.8: Example React Components

To encapsulate the JS function in the component with its execution context, we augmented the `render` function, whose functionality is to render the *React* component function and props as an individual unit for *Jest* to execute from the web page, with the `function interceptor`. Through the `render` function, the `function interceptor` extracts a complete execution context for the functional component and intercepts the JS function wrapped in the functional component indicated by the arrows in Figure 5.9. To enable customized execution context tracing, the `function interceptor` then marks symbolic variables and starts tracing after the completion of the encapsulation. At last, we configure *Jest*'s test runner to run each unit test individually while initiating in-situ concolic execution so that we can obtain the most concise execution traces for later symbolic analysis.

5.4 Evaluation

For evaluations, we apply our approach to in-situ concolic testing on front-end JS web application projects that come with unit test suites. They are utilizing Jest with *React* testing library and Puppeteer. In these evaluations, we target the `String` and `Number` types as symbolic variables for the functions under test.



```
class NewAccount {
 static prop = {onSubmit};
 var password;

 isValid(password);

 return (<div>
 <Button onClick= {this.isValid()}>Validate</Button>
  </div>) }
```

```
isValid(password) {
 //concrete value set by mocking input
 const { confirmPw, pwError,confirmPwError} = this.state;
 if (!password || !confirmPw || password !== confirmPw)
  { return false; }
  return !pwError && !confirmPwError; }
```

```
test("isValid", ()=>){
  var pw="goodpassword";
  let tc=render(<NewAccount onSubmit={mockProp}/>);

 //Mark symbolic value and set tracing start point
  %MarkSymbolic(pw);
  %StartTracing();
 //where the JS function is executed
  tc.isValid(pw);

 return;
}
```

Figure 5.9: How to apply in-situ concolic testing on React components using *Jest*

5.4.1 Evaluation of *Puppeteer* Implementation on Github Projects

We have selected 21 GitHub projects utilizing *Puppeteer*. We test them using the *Puppeteer* framework extended with our concolic testing capability. As a result, we discovered 4 bugs triggered from their web pages and 2 of them originated from their dependency libraries.

Evaluation Setup We selected GitHub projects with the following properties as as our targets:

(1) They use *Puppeteer* for unit testing of their JS web features;

(2) They have JS functions in web pages and such functions have at least one argument whose type is string or number;

(3) They utilize `evaluate` in their unit tests.

We have developed a script based on such properties and used the searching API provided by GitHub to collect applicable projects [38]. 21 projects were collected. Table 5.1 summarizes the demographics of the 21 GitHub projects collected by our script.

Table 5.1: Selected Projects that utilize *Puppeteer* for unit testing

name	LoC/JS	LoC/HTML	LoC/unit test	test ratio
keepfast	15835	514	58	8.61
DragAndScale*	982	16	370	77.49
affiliate	306	13	197	64.37
ecowetrics	3363	339	0	0
phantomas	5973	655	1440	24.10
polymer*	5399	157	2045	37.87
Insugar*	1967	32	410	20.84
wolkenkit*	1618	15	0	0
vidi	192048	2430	3505	1.82
vue*	849	125	0	0
weatherzen	333	45	0	0
querystringme	835	12	191	22.87
avocode*	5330	9	0	0
Odoo	1303	528	92	7.06
easy	54620	25141	4081	7.47
drag-and-scale	1100	24	548	88.09
My-first*	22729	808	0	0
boxtree	2033	48	1434	70.53
foundation	967	0	18	1.86
treezjs	109975	1475	8519	7.74
TicTacToe	543	442	243	44.75

We calculated the statistics using *ls-files* [27] combined with *cloc* provided by GitHub [28]. The LoC/JS is the LoC (lines of code) of all JS files, which includes the JS files of the libraries the project depends on. The LoC/HTML is the LoC of HTML files, which indicates the volume of its front-end web contents. The LoC of unit tests (LoC/unit test) includes the unit test files ending with `.test.js`. The test ratio is the ratio between the LoC/unit test over the LoC/JS, indicating the availability of unit tests for the projects. Before evaluation, we configure these projects to use the extended Puppeteer framework instead of the original one.

Result Analysis We ran each project with our approach for 30 minutes. On average, our implementation generates 200 to 400 test cases for each function. Table 5.2 summarizes the bugs detected. For polymer, our method generates two types of test cases that trigger two different bugs in user password validation functionalities of the project: 1) a generated test case induces execution to skip an *if* branch, which causes the password to be `undefined`, leading to the condition `!password || this.password === password` to return true, which should have returned false. We have fixed this bug by changing the operator `||` to `&&`. 2) test cases containing *unicode* characters fail password pattern matching using regular expression without g flag, i.e., `/[!@#$%^&*(),.?":|<>]/.test(value)`. For InsugarTrading, a test case of a string not containing `comma` is generated for `str.split(',')` function. The return value of an empty array causes errors in the dependency library `cookie-connoisseur`. A number out-of-bound error is discovered in the `changeCell()` function of TicTacToe. For phantomas, function `phantomas` has a check for `url` to be the string type but does not have pattern matching for it. A generated test case with an invalid `url` causes an exception in function `addScriptToEvaluateOnNewDocument` of *chromeDevTools*.

We identified two traits of the projects for which we did not detect bugs in. (1) A project does not fit the design of our Puppeteer implementation, i.e., `evaluate` is not

Table 5.2: Bugs detected in web applications using Puppeteer from Github

GitHub Projects	**Bugs**	**Error Sources**
polymer	Passwords fail validate and match	validator-match.js
InsugarTrading	Empty array caused by invalid string	cookie-connoisseur
TicTacToe	changeCell() out of bound	game.js
phantomas	Invalid string for url due to lack of pattern matching	chromeDevTools

used in the test suite. (2) The applicable JS part is small and well tested.

5.4.2 Evaluation of Jest Implementation on *Metamask*

In evaluation of the implementation of our concolic testing approach on Jest, we focus on Metamask's browser extension for Chrome. MetaMask is a software crypto-currency wallet used to interact with the Ethereum blockchain. It allows users to access their Ethereum wallet through a browser extension or mobile app, which can then be used to interact with decentralized applications [31]. *Metamask* extension utilizes the `render` functionality for testing JS functions in *React* components. We focus on front-end JS web functions, *React* component functions in particular. They reside in the `ui` folder of the *metamask-extension* project.

Testing Coverage Statistics of *Metamask* We select the `ui` folder as our evaluation target for two reasons: (1) *React* components of *metamask-extension* are mostly defined and implemented under this folder; (2) the functions in this folder is under tested. Figure 5.10 shows the current testing coverage statistics of the `ui` folder of *metamask-extension* [12]. We can see that only one sub-folder of `ui` (which also happens to be named as `ui`) has a relatively high coverage of 82.03%. Most other folders have coverage under 70% or even lower coverage.

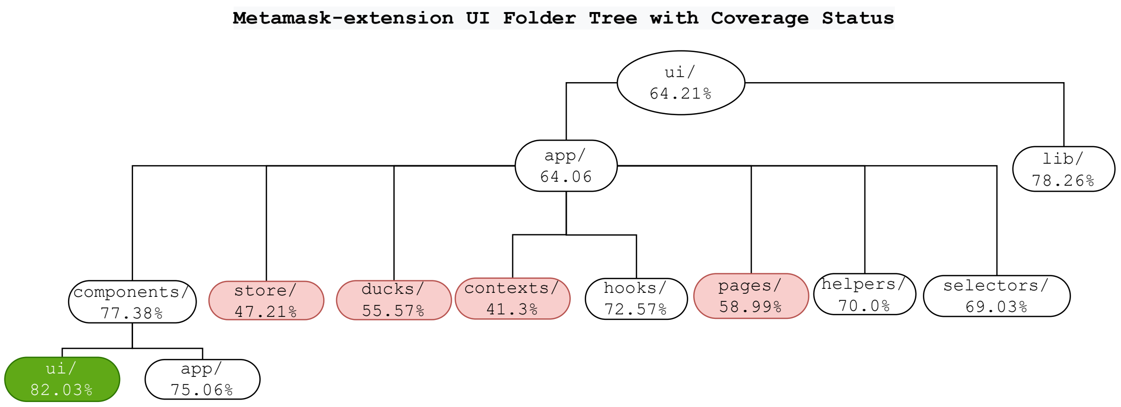


Figure 5.10: Coverage statistics of `ui` folder of Metamask-extension

Evaluation Setup In the unit testing workflow of *metamask-extension*, there is a global configuration for all unit test suites of UI components. This is because one component's functionality may depend on other components. Therefore, *metamask-extension* needs to be executed as an instance to support unit testing. To evaluate the implementation of in-situ concolic testing for *React* components, we need an independent environment for each component function wrapped with a single test file. This test file only contains one function under test. Therefore, each test file is an independent in-situ concolic testing runner for a function in a component. We implement an evaluation setup script to complete this task. This script automatically prepares the evaluation environment for in-situ concolic testing of a *React* component. Specifically, it does the following work under the folder where the target component resides:

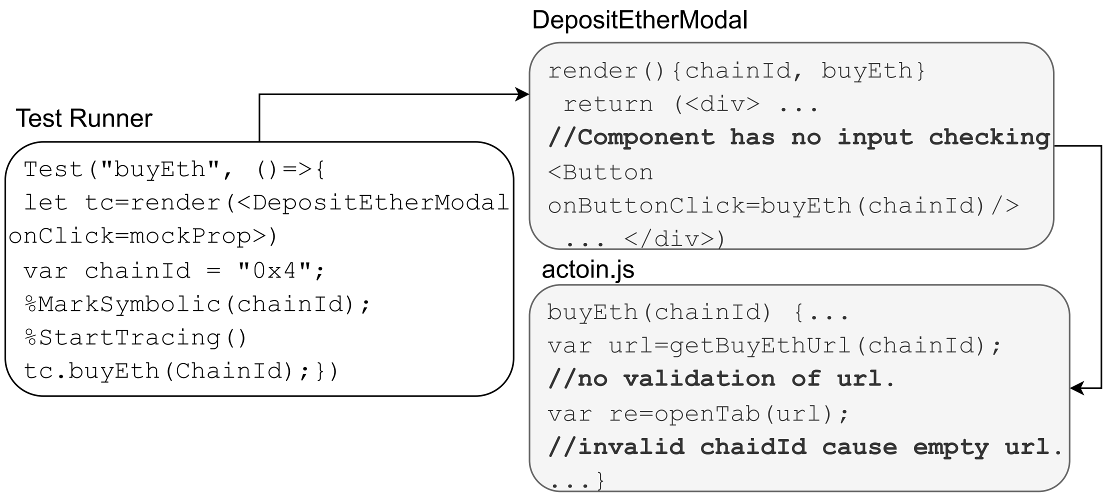
- *Jest* Configuration. Configure *Jest* for the individual component test file with an independent `jest.config.js`
- Babel Configuration. Configure `Babel` for the component test file to take JS native syntax, which is required by in-situ concolic testing. This is because *metamask-extension* JS source files are transformed using `Babel`.
- Dependency Installation. Collect and install dependencies for the target component. Such dependencies can be components or libraries.

Result Analysis After we set up the evaluation environment, we can conduct our evaluation in a sandbox on the test network of *Metamask*. We have uncovered 3 bugs and 1 test suite improvement as shown in Table 5.3. We have filed them as bug reports through GitHub. They have been accepted by *Metamask* developers. Along the way, we also found some similar test cases that *Metamask*'s bot reported.

Table 5.3: Bugs Detected in Metamask under UI folder

Features	**Bugs**	**Functions**
buy-eth	Missing checks for if the returned url is null causes page to return 500 in test network	buyEth
token-search	Syntax error without boundary checking	isEqualCase-Insensitive
ens-input	No NULL check for function argument	isValidDomainName
advanced-gas-fee	Show error if gas limit is not in range	gasLimit

For the buy-eth feature as shown in Figure 5.11, a test network error with a respond code of 500 was triggered when testing the Ether deposit functionality. Concolic testing generates a test case of an invalid `chainId` for `buyEth()`, which is defined in the `DepositEtherModal` component. It is wrapped by a `<Button>` tag and can be triggered by `onClick()`. `buyEth()` calls into `buyEthUrl()`, which retrieves a url for `buyEth()` function. Because `buyEthUrl()` did not check if the url is valid or null before it calls `openTab(url)` with the returned url. And there is also no validation for input in the component implementation. Additionally, this process was not wrapped in a `try/catch` block. We caught this error in our evaluation. We tested 16 component folders and discovered that *metamask-extension* most likely will ignore input checking if inputs are not directly from users. `chainId` is retrieved from mock data in this case, which is generated by our concolic engine. For the token-search feature,



```
Test("buyEth", ()=>{
 let tc=render(<DepositEtherModal
onClick=mockProp>)
 var chainId = "0x4";
 %MarkSymbolic(chainId);
 %StartTracing()
 tc.buyEth(ChainId);})
```

```
render(){chainId, buyEth}
 return (<div> ...
//Component has no input checking
<Button
onButtonClick=buyEth(chainId)/>
 ... </div>)
```

```
buyEth(chainId) {...
var url=getBuyEthUrl(chainId);
//no validation of url.
var re=openTab(url);
//invalid chaidId cause empty url.
...}
```

Figure 5.11: Error trace of the bug discovered in buy-eth

we uncovered a bug triggered by an empty string. In the `TokenSearch` component, function `handleSearch()` is wrapped by `<TextField>` with `onChange` method. It calls `isEqualCaseInsensitive()` with an empty string as its second argument without boundary checking. Function `isEqualCaseInsensitive` is defined in `utils.js`, which provides shared functions. We found that the unit testing for `utils.js` does not have test suites for that function, while the same bug is not found in the experiment conducted on the `send.js` file. In `send.js`, function `validateRecipientUserInput` also calls the incorrect function `isEqualCaseInsensitive`. However, since `send.js` checks for both empty string and null inputs before calling the faulty function, it avoids the potential error in `utils.js`.

For the ens-input feature, in the `onChange` method of component `EnsInput`'s `<input />`, the function `isValidDomain` is called. Our approach generated test cases with unacceptable ASCII characters in the domain name, e.g., `%ff.bar`. We replay this test case, function `isValidDomain` returns true when it should return false. In Listing 5.3, function `isValidDomain` returns the value of the condition `match!==undefined`. This test case made through regex matching and returned `null` but `null` is not equal to `undefined` in JS.

Listing 5.3: A code segment of `utils.js` with function `isValidDomain` showing incorrect behavior in line 8

```
1 function isValidDomainName(\%ff.bar) {
2   var match = punycode
3     .toASCII(address)
4     .match(
5       /^(?:[a-z0-9](?:[-a-z0-9]*[a-z0-9])?\.)+[a-z0-9][-a-z0-9]*[a-z0-9]$/u
    ,
  );
  //After match function, returning string match=null; therefore, match !==
    undefined return true.
  return match !== undefined;
}
```

For the advanced-gas-fee feature, we found the `updateGasLimit(gasLimit)` function (expecting a numeric input) in the `<FormField>` component has wrong behavior when given a string input containing only digits such as `"908832"`. The function simply sets the gas limit to 0 without emitting errors. We do not consider this as a bug since component `<FormField>` restricted the input to be `numeric` in the HTML element. After we filed it, this has been marked with the `area-testSuite` tag on GitHub by developers as a test suite improvement.

5.5 Summary

In Chapter, we have presented a novel approach to apply concolic execution to front-end JS. The approach makes use of an in-situ concolic executor for JS and leverages the functionality of JS testing frameworks as test runners and web content extractors. Our approach works in three steps: (1) extracting JS functions from web pages using with JS testing framework; (2) integrating the in-situ concolic testing interface in the execution context for the JS Web functions; (3) utilizing the testing framework's test runner and its mock data as the driver for concolic execution to generate additional test data for the JS web function under test.

We have conducted evaluation on open-source projects from Github and on *Meta-*

mask's UI features, which are proper targets for our implementations on Puppeteer and Jest respectively. We have found bugs in each evaluation, whose bug reports have been accepted on GitHub. This contributes to both bug finding and test suite improvement for the applications tested. The results show that our approach to concolic testing frontend JS is both practical and effective.

6 Concolic Testing of JavaScript using Sparkplug and Remill

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 those libraries [70]. 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 [85]. 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 [94]. In-situ concolic testing for JavaScript using JavaScript's native execution environments becomes its biggest strength. However, this approach has several limitations [70]. It utilized the tracing engine of CRETE, which leverages the interpreted mode of *Qemu*, a dynamic translator [44], to capture the execution trace of JS scripts and employs `KLEE` as the backend symbolic execution engine. During this process, the concrete execution trace is converted from the original code to the host instruction set. Subsequently, the tiny code generator (TCG) of *Qemu*, serving as the dynamic translator, translates the instruction set to `qemu-ir`. This process impedes the efficiency of the tracing process greatly because execution tracing uses the interpreted mode of TCG. For example, 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.

In this chapter, we 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` translator using *remill* libraries to improve the efficiency of the execution tracer, reduce the complexity of translation stages, and conduct concolic testing in their native environments like the in-situ approach at the same time. 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 is 160 Node.js libraries. They heavily utilize the `String` type and its operations. The results show our improvement achieves comparable statement coverage (within 10% difference on average) on these libraries, detects all bugs that are discovered by the in-situ method, and only uses a fraction of the time needed by the in-situ approach.

6.1 Background

6.1.1 Sparkplug

Sparkplug is a non-optimizing JavaScript compiler of V8 [67]. It is engineered for swift compilation, which enables us to compile at our convenience. A couple of techniques are employed by the Sparkplug compiler to achieve its impressive speed. Firstly, Sparkplug utilizes a shortcut; the functions it compiles are already processed into bytecode in 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 typical step in most compilers. Instead, it directly translates bytecode into machine code in a single linear pass using bytecode handlers [3], aligning the emitted code with the execution flow of the bytecode. We will discuss the bytecode handler in detail in Section 6.3. This feature guarantees that the emitted execution trace in the form of machine code 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 optimization opportunities to localized peephole optimizations as shown in Figure 6.1, we heavily this feature of Sparkplug to improve execution tracing.

6.1.2 Interpreter Stack Frame Mirroring

V8 JavaScript engine supports two modes for executing a JS script, namely *interpreted mode* and *optimized just-in-time compilation mode*. The *interpreted mode* is where the JS bytecode [20] translated from the JS script is interpreted by its interpreter, Ignition [8], which is the foundation of in-situ concolic testing for JavaScript [70]. The *optimized just-in-time compilation mode* is where the bytecode is compiled by the V8

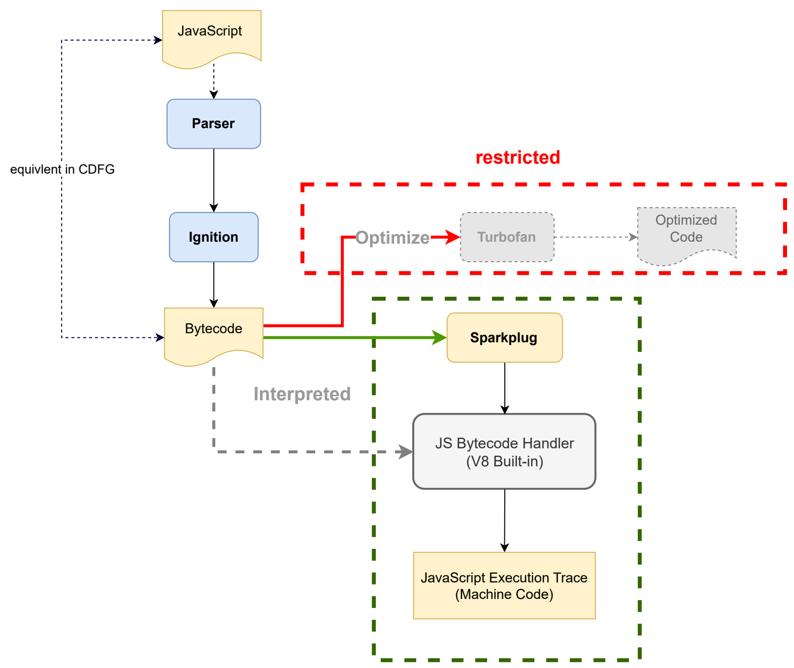


Figure 6.1: Sparkplug's restricted optimization feature

engine into optimized machine code using its just-in-time compiler, Turbofan [19], 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 allows us to improve the efficiency of 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 Ignition does in the in-situ approach. Furthermore, Sparkplug effectively pre-compiles 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 later be used for code translation from machine code (assembly code) to LLVM.

6.1.3 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 [41], the instruction translation functionality is powered by the *Remill* library. Unlike other tools, *Remill* exclusively handles machine code translation into LLVM IR [18].

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 [47]. KLEE, which performs symbolic execution, typically operates on the LLVM IR generated from source code using the LLVM toolchain [66]. By utilizing *Remill* to translate 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 IR. 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 IR 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.

6.2 Design

6.2.1 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 6.2, the concrete execution step in the leftmost box of concolic testing is conducted in the native execution environment for

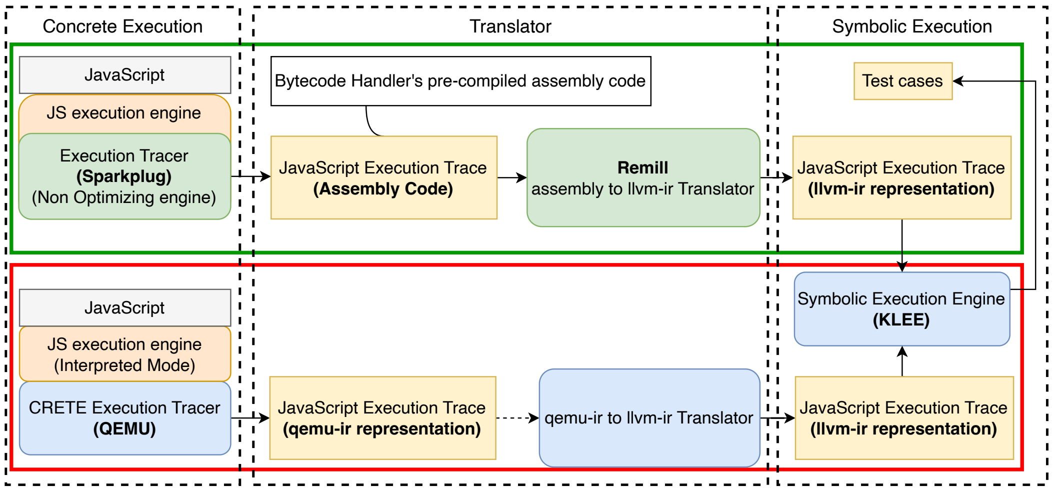

Concrete Execution
Translator
Symbolic Execution
JavaScript
JS execution engine
Execution Tracer (Sparkplug) (Non Optimizing engine)
Bytecode Handler's pre-compiled assembly code
JavaScript Execution Trace (Assembly Code)
Remill assembly to llvm-ir Translator
JavaScript Execution Trace (llvm-ir representation)
Test cases
Symbolic Execution Engine (KLEE)
JavaScript
JS execution engine (Interpreted Mode)
CRETE Execution Tracer (QEMU)
JavaScript Execution Trace (qemu-ir representation)
qemu-ir to llvm-ir Translator
JavaScript Execution Trace (llvm-ir representation)

Figure 6.2: Workflows of In-situ Concolic Testing Based on Sparkplug and CRETE

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 IR.

- **Symbolic Analysis.** The execution trace represented by LLVM IR is fed into a symbolic execution engine to generate test cases.

6.2.2 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 [20, 70]. 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.

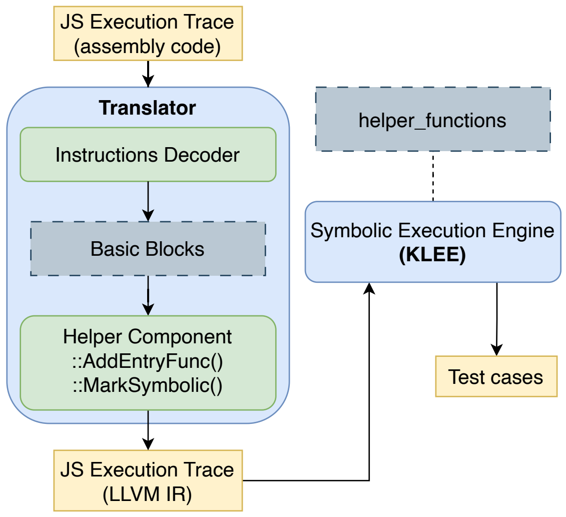

JS Execution Trace (assembly code)
Translator
Instructions Decoder
Basic Blocks
Helper Component ::AddEntryFunc() ::MarkSymbolic()
JS Execution Trace (LLVM IR)
helper_functions
Symbolic Execution Engine (KLEE)
Test cases

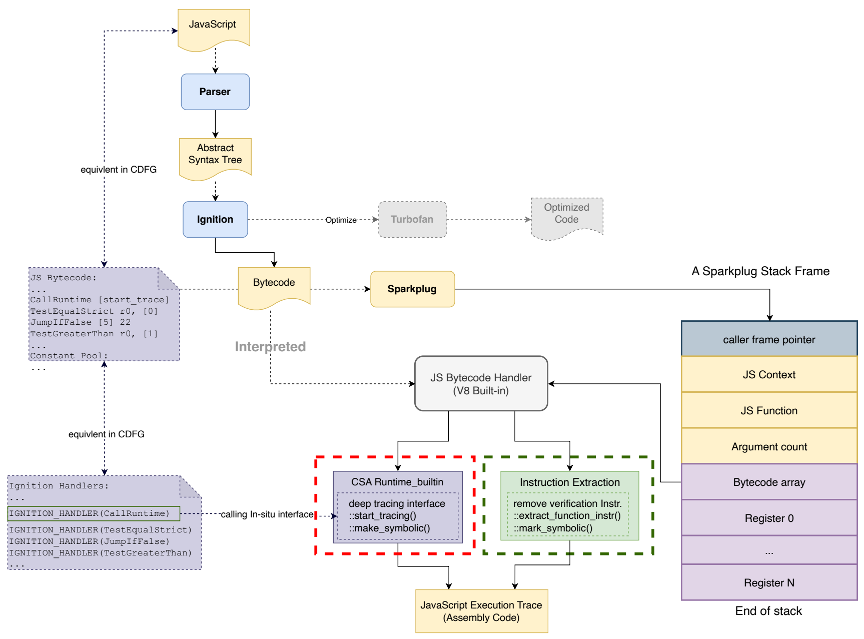
Figure 6.3: Workflow of the Translator

Our approach improves in-situ concolic testing in 2 aspects. In Figure 6.2, the in-situ approach is represented by the diagram in the red box and our approach in the green box. 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. 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.

6.2.3 Why we choose Sparkplug?

Sparkplug disables the Turbofan path naturally. It compiles from bytecodes that Ignition emits as shown in Figure 6.4. 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 6.1.2, Sparkplug mirrors Ignition's execution for JavaScript, Sparkplug and Ignition have almost identical stack frame [67]. This simplifies the design by removing the deep tracing control interface used in the in-situ approach shown in the red box. Instead, it captures execution traces 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 verification-related bytecodes in the generated execution trace without influencing the verification workflow of Sparkplug.

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 6.2. 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



```
JS Bytecode:
...
CallRuntime [start_trace]
TestEqualStrict r0, [0]
JumpIfFalse [5] 22
TestGreaterThan r0, [1]
...
Constant Pool:
...
```

```
Ignition Handlers:
...
IGNITION_HANDLER(CallRuntime)
IGNITION_HANDLER(TestEqualStrict)
IGNITION_HANDLER(JumpIfFalse)
IGNITION_HANDLER(TestGreaterThan)
...
```

Figure 6.4: Workflow of Execution Tracer between In-situ Approach and Our Approach

and marking symbolic variables as shown in Figure 6.3. As a result, the output of the translator forms a complete concrete execution trace for later symbolic execution engine to generate test cases.

6.3 Implementation

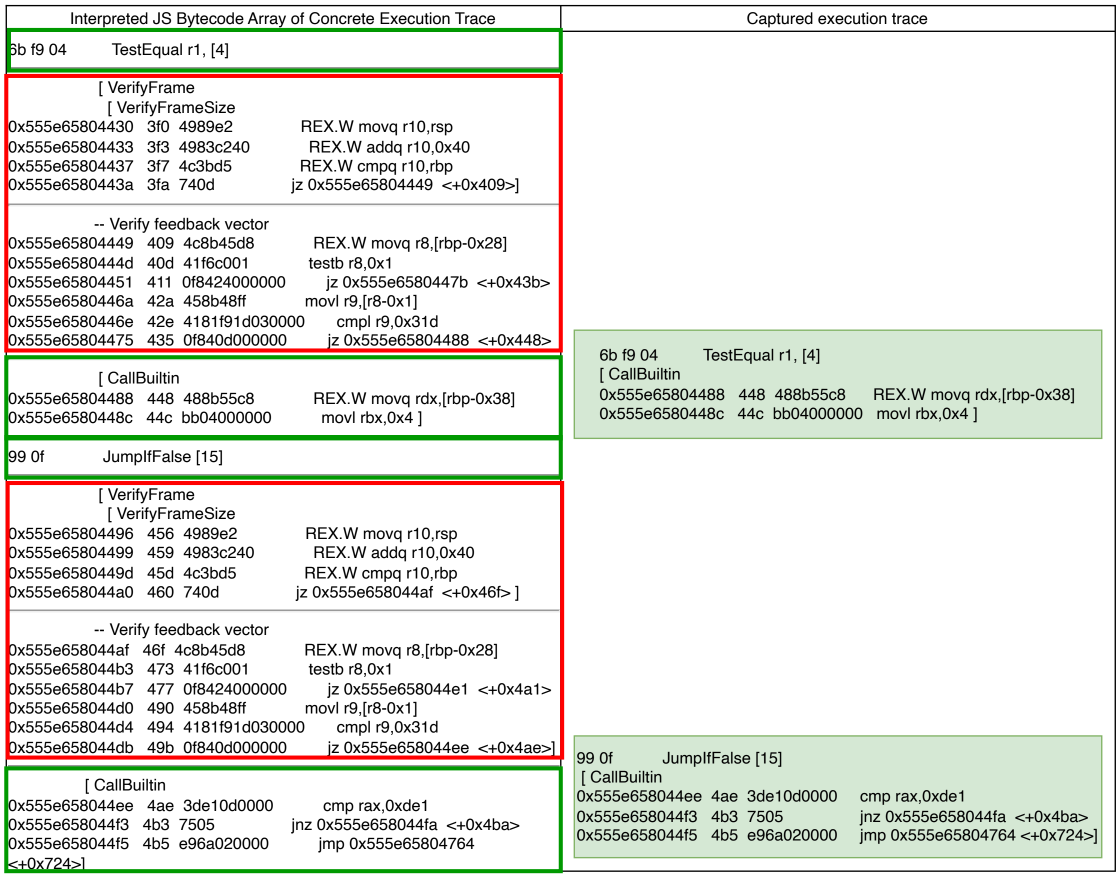
In this section, we demonstrate the feasibility of our approach by implementing its complete workflow with an execution tracer based on V8's Sparkplug, a translator leveraging *Remill*, and a symbolic execution engine using KLEE [47].

6.3.1 Modification on Bytecode Handlers of Sparkplug

To capture the most concise execution trace, we implemented the function `extract_fun -ction_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 6.5 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 `MARK_SYMBOLIC` is implemented to cache the symbolic value in the execution trace for later symbolic analysis.

6.3.2 Implementation on Remill translator

We utilized *remill* library to implement an assembly-to-LLVM translator. Figure 6.6 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



Interpreted JS Bytecode Array of Concrete 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>

                     [ CallBuiltin
0x555e65804488   448  488b55c8              REX.W movq rdx,[rbp-0x38]
0x555e6580448c   44c  bb04000000              movl rbx,0x4 ]

99 0f            JumpIfFalse [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              movl r9,[r8-0x1]
0x555e658044d4   494  4181f91d030000        cmpl r9,0x31d
0x555e658044db   49b  0f840d000000          jz 0x555e658044ee  <+0x4ae>]

                   [ CallBuiltin
0x555e658044ee   4ae  3de10d0000              cmp rax,0xde1
0x555e658044f3   4b3  7505              jnz 0x555e658044fa  <+0x4ba>
0x555e658044f5   4b5  e96a020000              jmp 0x555e65804764
<+0x724>]
```

Captured execution trace

```
6b f9 04         TestEqual r1, [4]
[ CallBuiltin
0x555e65804488   448  488b55c8       REX.W movq rdx,[rbp-0x38]
0x555e6580448c   44c  bb04000000   movl rbx,0x4 ]

99 0f            JumpIfFalse [15]
 [ CallBuiltin
0x555e658044ee  4ae  3de10d0000     cmp rax,0xde1
0x555e658044f3  4b3  7505           jnz 0x555e658044fa  <+0x4ba>
0x555e658044f5  4b5  e96a020000     jmp 0x555e65804764 <+0x724>]
```

Figure 6.5: How the execution tracer only extracts the execution traces that contribute to the main control flow of JS scripts

symbolic analysis. The main function then calls into the basic blocks LLVM functions. At last, the resulting trace is readily consumable by 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.

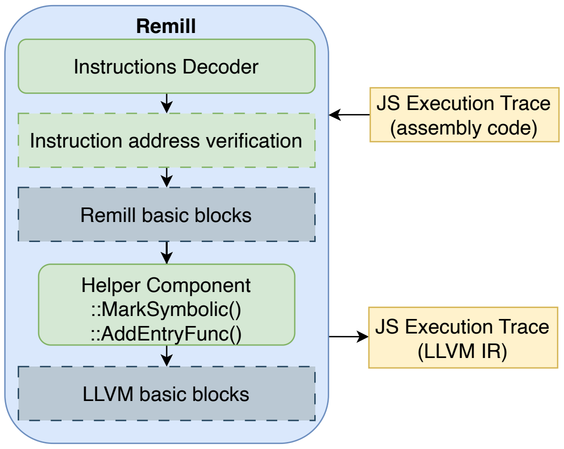


Figure 6.6: How the execution tracer only extracts the execution traces that contribute to the main control flow of JS scripts

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

6.4 Evaluation

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 [11] and compatible with most JavaScript testing frameworks, e.g., Mocha [13] and Node-Tap [14]. 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 6.1 shows the demographics

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

Table 6.1: Demographics for Libraries under Test

of the selected libraries. The LoC (lines of code) for a library under test is calculated with *github-loc* [9]. The number of weekly downloads of a library under test is calculated with *npm-stats-api* [17]. The number of dependencies is the number of dependent libraries that the library under test has. We calculated it with *dependent-counts* [7].

6.4.1 Coverage Analysis

Figure 6.7 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 6.8 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 50% 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.

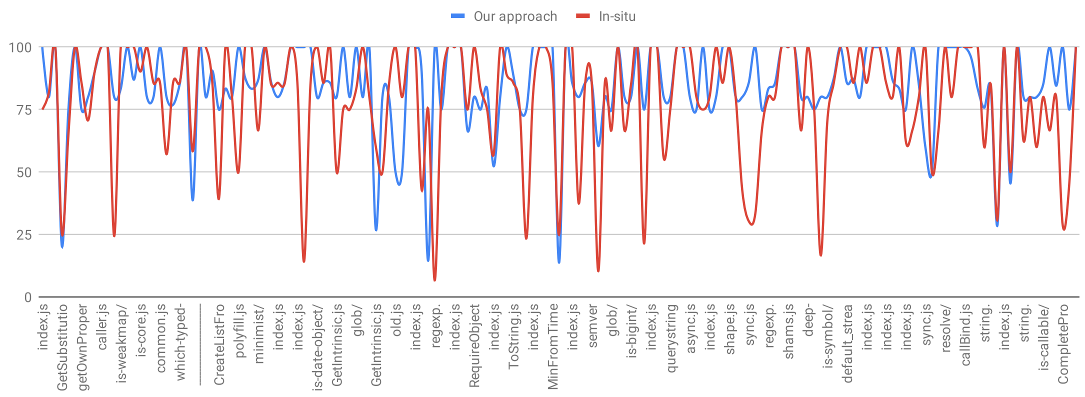


Figure 6.7: Statement Coverage Comparison between our approach and In-situ approach

We also compared our approach with an existing tool, ExpoSE [71], by testing the same set of libraries as shown in Figure 6.9, on which ExpoSE has been applied. Our

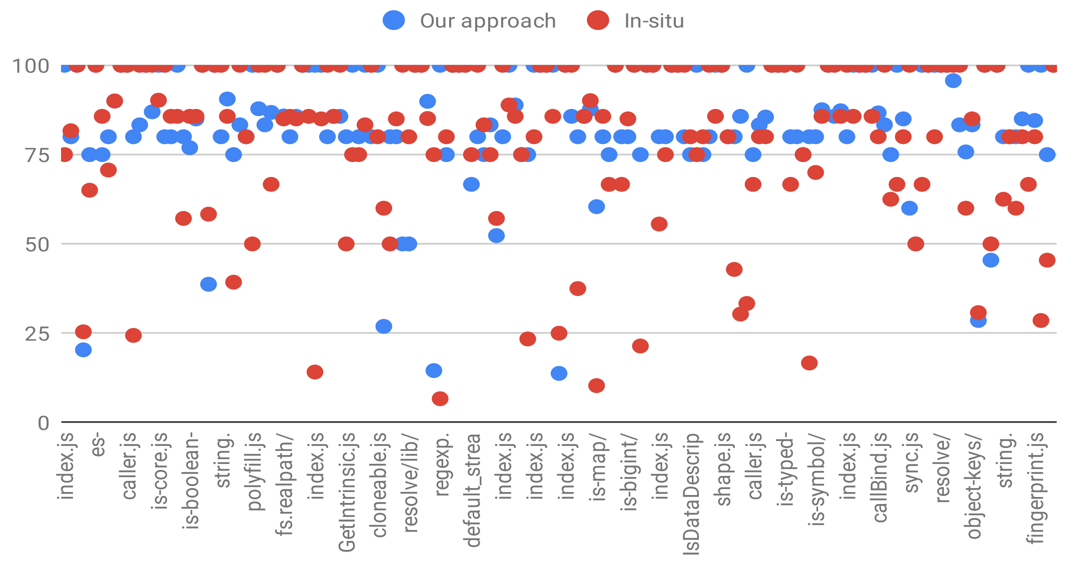


Figure 6.8: Coverage Distribution Comparison between our approach and In-situ approach

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.

6.4.2 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 with execution tracing on OS-VM level takes about 3 to 5 minutes to

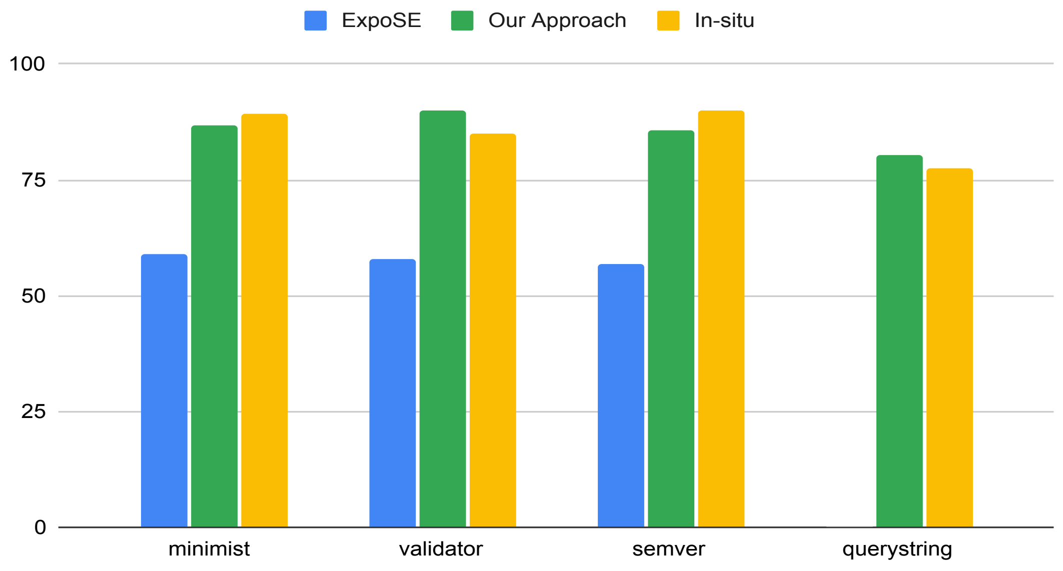


Figure 6.9: Statement Coverage Comparison among our approach, In-situ approach, and ExpoSE

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

Table 6.2: Bugs detected in functions

complete an iteration of test case generation and it only needs about 5 to 10 seconds to complete an iteration with our approach. Table 6.2 shows a summary of the bugs that we detected again. The detailed explanation for each bug can be found in Section 4.4.3.

6.5 Summary

In this Chapter, we introduced improvements to the in-situ 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 translator 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 improvements achieve similar statement coverage on these libraries within no more than 10% difference on average and can detect all bugs that are detected by the in-situ method, which only uses a fraction of the time needed by the in-situ approach.

7 Conclusions

In this dissertation, we introduced a holistic framework for applying concolic testing to applications in scripting languages, which extended the applicability and flexibility of traditional concolic execution (binary-level concolic execution) to scripting languages, especially JavaScript and Lua applications. We also presented the designs, implementations, and evaluations of several systems and frameworks based on the proposed approach to make popular modern applications using JavaScript and Lua languages more reliable, including network scan tools, Node.js applications, and several front-end applications. To conclude, this chapter summarizes the main contributions and highlights some directions for future research.

7.1 Summary of Contributions

Broadly, this dissertation pushed the boundaries of testing automation techniques, in particular concolic testing, enriched the research communities of both academia and industry (specially for software engineering and testing) by developing a framework and implementing several prototype systems, and contributed directly to build more reliable applications in the real world by detecting and fixing unknown bugs in some important software applications. In summary, this dissertation makes the following contributions:

- Provide and implement the approach of concolic execution for scripting languages and use it on Lua and JavaScript.

- Design and develop concolic testing on NSE scripts to generate honey farms automatically to provide defense against attackers.
- Design, develop and apply in-situ concolic testing on backend JavaScript in the environment of Node.js and perform concolic testing NPM libraries.
- Design and implement in-situ concolic testing to test front-end JavaScript based on JS testing framework. Eventually, we mprove the execution tracer using Sparkplug to remove its dependence on a virtual machine and adapt the translator accordingly.

7.2 Future Directions

In this section, we highlight some interesting future directions. More detailed discussions and future work can also be found in previous sections (Section 3.5, Section 4.4.4 and Section 5.5)

Refine symbolic interfaces of JavaScript object. Currently, our in-situ concolic testing framework for JavaScript, supports `String` and `Integer` JavaScript Objects. Covering more JS Objects will make the symbolic interfaces more comprehensive and enable testing for a wide range of JS scripts. One future work is to further extend the symbolic object model to make the framework more effective.

Refine execution context extraction for front-end JS Our implement currently covers *Puppeteer* test framework and *React* testing library of *Jest*. We can further extend our implementation based on our approach to more libraries of *Jest* and also more testing frameworks, such as *Jasmine* [2], *Cypress* [1], etc. As a result, more applications using various testing frameworks can benefit from the approach.

Improve the Sparkplug-based execution tracer and Remill-based translator. The current workflow of the two components is connected by automation harness scripts and also does not exploit the potential of multiprocessing and parallelism. Improving the automation of the workflow and encouraging parallelism will provide opportunities for future optimization.

Bibliography

[1] Cypress web testing framework. https://www.browserstack.com/guide/cypress-framework-tutorial, June 2010.

[2] Jasmine, simple javascript testing. https://jasmine.github.io/, June 2010.

[3] Ignition: V8 interpreter. https://docs.google.com/document/d/11T2CRex9hXxoJwbYqVQ32yIPMh0uouUZLdyrtmMoL44/mobilebasic, March 2016.

[4] V8 stringobject. https://v8docs.nodesource.com/node-0.8/d9/d38/classv8_1_1_string_object.html, March 2016.

[5] V8's object model using well-defined c++. https://docs.google.com/document/d/1_w49sakC1XM1OptjTurBDqO86NE16FH8LwbeUAtrbCo/edit, January 2019.

[6] Codestubassembler builtins. https://v8.dev/docs/csa-builtins, August 2021.

[7] dependent-counts. https://www.npmjs.com/package/dependent-counts, June 2021.

[8] Firing up the ignition interpreter. https://v8.dev/blog/ignition-interpreter, August 2021.

[9] github-loc. https://www.npmjs.com/package/github-loc, June 2021.

[10] 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, August 2021.

[11] Istanbul. https://istanbul.js.org/, August 2021.

[12] Metamask-extension coveralls. https://coveralls.io/github/MetaMask/metamask-extension, March 2021.

[13] Mocha: simple, flexible, fun. https://mochajs.org/, August 2021.

[14] Node-tap. https://node-tap.org/, August 2021.

[15] Node.js. https://nodejs.org/en/, August 2021.

[16] Npm. https://www.npmjs.com/, August 2021.

[17] npm-stats-api. https://www.npmjs.com/package/npm-stats-api, June 2021.

[18] Remill, 2021. https://github.com/lifting-bits/remill.

[19] Turbofan: A new code generation architecture for v8. https://docs.google.com/presentation/d/1_eLlVzcj94_G4r9j9d_Lj5HRKFnq6jgpuPJtnmIBs88/htmlpresent, August 2021.

[20] Understanding v8's bytecode. https://medium.com/dailyjs/understanding-v8s-bytecode-317d46c94775, August 2021.

[21] v8. https://v8.dev/, August 2021.

[22] Afl: American fuzzy lop. http://lcamtuf.coredump.cx/afl/, January 2022.

[23] Browser context. https://pptr.dev/api/puppeteer.browser, October 2022.

[24] Building a javascript testing frameworkbuildtestframework. https://cpojer.net/posts/building-a-javascript-testing-framework#building-a-testing-framework, October 2022.

[25] Executioncontext class. https://pub.dev/documentation/puppeteer/latest/puppeteer/ExecutionContext-class.html, October 2022.

[26] Express. https://expressjs.com/, October 2022.

[27] git-ls-files. https://git-scm.com/docs/git-ls-files, October 2022.

[28] git-ls-files. hhttps://github.com/AlDanial/cloc, October 2022.

[29] js-fuzz. https://github.com/connor4312/js-fuzz, January 2022.

[30] Jsfuzz: coverage-guided fuzz testing for javascript. https://github.com/fuzzitdev/jsfuzz, January 2022.

[31] Metamask. https://metamask.io/, October 2022.

[32] Module counts. http://www.modulecounts.com/, August 2022.

[33] Puppeteer. https://pptr.dev/, October 2022.

[34] Puppeteer architecture. https://devdocs.io/puppeteer, October 2022.

[35] React component. https://reactjs.org/docs/react-component.html, October 2022.

[36] React component. https://reactjs.org/docs/components-and-props.html, October 2022.

[37] React testing library. https://testing-library.com/docs/react-testing-library/intro/, October 2022.

[38] Search: The search api lets you to search for specific items on github. https://docs.github.com/en/rest/search, October 2022.

[39] Six essential frameworks for creating automated tests. https://dzone.com/refcardz/javascript-test-automation-frameworks, October 2022.

[40] Usage statistics of javascript as client-side programming language on websites. https://w3techs.com/technologies/details/cp-javascript, October 2022.

[41] Sparkplug. https://github.com/lifting-bits/mcsema, August 2023.

[42] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. Why do developers use trivial packages? an empirical case study on npm. pages 385–395, 08 2017.

[43] Ars Technica. NSA-leaking Shadow Brokers Just Dumped Its Most Damaging Release Yet, April 2017. https://arstechnica.com/information-technology/2017/04/nsa-leaking-shadow-brokers-just-dumped-its-most-damaging-release-yet/.

[44] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. Califor-nia, USA, 2005.

[45] Bloomberg Technology. Equifax Suffered a Hack Almost Five Months Earlier Than the Date It Disclosed, September 2017. `https://www.bloomberg.com/news/articles/2017-09-18/equifax-is-said-to-suffer-a-hack-earlier-than-the-date-disclosed`.

[46] Stefan Bucur, Johannes Kinder, and George Candea. Prototyping symbolic execution engines for interpreted languages. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 239–254, 2014.

[47] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[48] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE way to test openflow applications. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 127–140, San Jose, CA, April 2012. USENIX Association.

[49] Bo Chen, Christopher Havlicek, Zhenkun Yang, Kai Cong, Raghudeep Kannavara, and Fei Xie. Crete: A versatile binary-level concolic testing framework. In Alessandra Russo and Andy Schürr, editors, *Fundamental Approaches to Software Engineering*, pages 281–298, Cham, 2018. Springer International Publishing.

[50] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices*, 46(3):265–278, 2011.

[51] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Trans. Comput. Syst.*, 32(4), jan 2015.

[52] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (redos) in practice: An empirical study at the ecosystem scale. ESEC/FSE 2018, page 246–256, New York, NY, USA, 2018. Association for Computing Machinery.

[53] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 181–191, New York, NY, USA, 2018. Association for Computing Machinery.

[54] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[55] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[56] Hans Peter Luhn. Luhn algorithm, 2021. `https://en.wikipedia.org/wiki/Luhn_algorithm`.

[57] Pei-Sheng Huang, Chung-Huang Yang, and Tae-Nam Ahn. Design and implementation of a distributed early warning system combined with intrusion detection system and honeypot. In *Proceedings of the 2009 International Conference on Hybrid Information Technology*, pages 232–238, 2009.

[58] Xiao-ou JIN, Bao-yan ZHONG, and Xiang LI. Research and implementation of interpreting javascript dynamic web page based on rhino engine [j]. *Computer Technology and Development*, 2(002), 2008.

[59] Jordan Jueckstock and Alexandros Kapravelos. Visiblev8: In-browser monitoring of javascript in the wild. In *Proceedings of the Internet Measurement Conference*, IMC '19, page 393–405, New York, NY, USA, 2019. Association for Computing Machinery.

[60] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[61] Ronny Ko, James Mickens, Blake Loring, and Ravi Netravali. Oblique: Accelerating page loads using symbolic execution. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 289–302. USENIX Association, April 2021.

[62] Igibek Koishybayev and Alexandros Kapravelos. Mininode: Reducing the attack surface of node.js applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 121–134, San Sebastian, October 2020. USENIX Association.

[63] Saparya Krishnamoorthy, Michael S Hsiao, and Loganathan Lingappan. Tackling the path explosion problem in symbolic execution-driven test generation for programs. In *2010 19th IEEE Asian Test Symposium*, pages 59–64. IEEE, 2010.

[64] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. *SIGPLAN Not.*, 47(6):193–204, June 2012.

[65] LabLua. Lua reference manuals. `https://www.lua.org/manual/`, June 2021.

[66] 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.*, pages 75–86, 2004.

[67] Leszek Swirski. Sparkplug — a non-optimizing JavaScript compiler, 2021. `https://v8.dev/blog/sparkplug`.

[68] Guodong Li, Esben Andreasen, and Indradeep Ghosh. Symjs: automatic symbolic testing of javascript web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 449–459, 2014.

[69] Yuan-Fang Li, Paramjit K Das, and David L Dowe. Two decades of web application testing—a survey of recent advances. *Information Systems*, 43:20–54, 2014.

[70] Z. Li and F. Xie. In-situ concolic testing of javascript. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 236–247, Los Alamitos, CA, USA, mar 2023. IEEE Computer Society.

[71] Blake Loring, Duncan Mitchell, and Johannes Kinder. Expose: practical symbolic execution of standalone javascript. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, pages 196–199, 2017.

[72] Gordon Lyon. Nmap: the network mapper. `https://nmap.org/`, June 2021.

[73] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In Eran Yahav, editor, *Static Analysis*, pages 95–111, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[74] Microsoft. CONPOT ICS/SCADA Honeypot, 2021. `http://conpot.org/`.

[75] Microsoft. KFsensor: Advanced Windows Honeypot System, 2021. `http://www.keyfocus.net/kfsensor/`.

[76] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. Jseft: Automated javascript unit test generation. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, 2015.

[77] Piotr Olma and Gioacchino Mazzurco. Nse script description. `https://nmap.org/nsedoc/scripts/http-form-fuzzer.html`, June 2021.

[78] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *ACM SIGOPS Operating Systems Review*, 40(4):15–27, 2006.

[79] Bobby Powers, John Vilk, and Emery D. Berger. Browsix: Bridging the gap between unix and the browser. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 253–266, New York, NY, USA, 2017. Association for Computing Machinery.

[80] Honeynet Project. *Know Your Enemy: Revealing the Security Tools, Tactics, and Motives of the Blackhat Community*. Addison-Wesley Professional, 2001.

[81] Proofpoint. Proofpoint Emerging Threats Rules, 2021. `https://rules.emergingthreats.net/`.

[82] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *LISA*, pages 229–238, 1999.

[83] José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. Symbolic execution for javascript. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, pages 1–14, 2018.

[84] Samir Sapra, Marius Minea, Sagar Chaki, Arie Gurfinkel, and Edmund Clarke. Finding errors in python programs using dynamic symbolic execution. volume 8254, pages 283–289, 11 2013.

[85] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *2010 IEEE Symposium on Security and Privacy*, pages 513–528, 2010.

[86] Edward J. Schwartz, Thanassis Avgerinos, and David 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*, pages 317–331, 2010.

[87] Marija Selakovic and Michael Pradel. Performance issues and optimizations in javascript: an empirical study. In *Proceedings of the 38th International Conference on Software Engineering*, pages 61–72, 2016.

[88] Koushik Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572, 2007.

[89] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon 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*, pages 488–498, 2013.

[90] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. *ACM SIGSOFT Software Engineering Notes*, 30(5):263–272, 2005.

[91] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*, pages 1–25. Springer, 2008.

[92] Matt Staats and Corina Pundefinedsundefinedreanu. Parallel symbolic execution for structural test generation. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, page 183–194, New York, NY, USA, 2010. Association for Computing Machinery.

[93] Haiyang Sun, Andrea Rosà, Daniele Bonetta, and Walter Binder. Automatically assessing and extending code coverage for npm packages. *arXiv preprint arXiv:2105.06838*, 2021.

[94] Sümeyye Süslü and Christoph Csallner. Spejs: A symbolic partial evaluator for javascript. In *Proceedings of the 1st International Workshop on Advances in Mobile App Analysis*, A-Mobile 2018, page 7–12, New York, NY, USA, 2018. Association for Computing Machinery.

[95] Mohit Thakkar. *Unit Testing Using Jest*, pages 153–174. Apress, Berkeley, CA, 2020.

[96] Neline van Ginkel, Willem De Groef, Fabio Massacci, and Frank Piessens. A server-side javascript security architecture for secure integration of third-party libraries. *Security and Communication Networks*, 2019, 2019.

[97] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. Snoeren, G. Voelker, and S. Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *ACM SOSP*, pages 148–162, October 2005.

[98] Chris Wanstrath. Mustache Processor, 2009. `https://mustache.github.io/`.

[99] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 995–1010, Santa Clara, CA, August 2019. USENIX Association.