Master's Programme in Security and Cloud Computing

Differential Fuzzing the WebAssembly

Master's Thesis

Gilang Mentari Hamidy

Aalto University
School of Science

EURECOM

MASTER'S
THESIS

Aalto University - EURECOM
MASTER'S THESIS 2020

Differential Fuzzing the WebAssembly

Fuzzing Différentiel le WebAssembly

Gilang Mentari Hamidy

This thesis is a public document and does not contain any confidential information.

Cette thèse est un document public et ne contient aucun information confidentielle.

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology.
Antibes, 27 July 2020

Supervisor: Prof. Davide Balzarotti, EURECOM
Co-Supervisor: Prof. Jan-Erik Ekberg, Aalto University

Aalto University - School of Science
EURECOM

Master's Programme in
Security and Cloud Computing

Aalto University
School of Science

EURECOM
Sophia Antipolis

Abstract

Author
Gilang Mentari Hamidy

Title
Differential Fuzzing the WebAssembly

School School of Science

Degree programme Master of Science

Major Security and Cloud Computing (SECCLO) **Code** SCI3084

Supervisor Prof. Davide Balzarotti, EURECOM
Prof. Jan-Erik Ekberg, Aalto University

Level Master's thesis **Date** 27 July 2020 **Pages** 133 **Language** English

Abstract

WebAssembly, colloquially known as Wasm, is a specification for an intermediate representation that is suitable for the web environment, particularly in the client-side. It provides a machine abstraction and hardware-agnostic instruction sets, where a high-level programming language can target the compilation to the Wasm instead of specific hardware architecture. The JavaScript engine implements the Wasm specification and recompiles the Wasm instruction to the target machine instruction where the program is executed. Technically, Wasm is similar to a popular virtual machine bytecode, such as Java Virtual Machine (JVM) or Microsoft Intermediate Language (MSIL).

There are two major implementations of Wasm, correlated with the two most popular web browsers in the market. These two are the V8 engine by Chromium project and the SpiderMonkey engine by Mozilla. Wasm does not mandate a specific implementation over its specification. Therefore, both engines may employ different mechanisms to apply the specification. These different implementations may open a research question: are both engines implementing the Wasm specification equally?

In this thesis, we are going to explore the internal implementation of the JavaScript engine in regards to the Wasm specification. We experimented using a differential fuzzing technique, in which we test two JavaScript engines with a randomly generated Wasm program and compares its behavior. We executed the experiment to identify any anomalous behavior, which then we analyzed and identified the root cause of the different behavior.

This thesis covers the WebAssembly specification extensively. It discusses several foundational knowledge about the specification that is currently lacking in references. This thesis also presents the instrumentation made to the JavaScript engine to perform the experiment, which can be a foundation to perform a similar experiment. Finally, this thesis analyzes the identified anomaly found in the experiment through reverse engineering techniques, such as static and dynamic analysis, combined with white-box analysis to the JavaScript engine source code.

In this experiment, we discovered a different behavior of the JavaScript engine that is observable from the perspective of the Wasm program. We created a proof-of-concept to demonstrate the different behavior that can be executed in the recent web browser up to the writing of this thesis. This experiment also evaluated the implementation of both JavaScript engine on the Wasm specification to conclude that both engines implement the specification faithfully.

Keyword webassembly, fuzzing, c++, compiler, testing, programming language, static analysis, dynamic analysis

Aalto University
School of Science

EURECOM
Sophia Antipolis

Abstrait

Auteur
Gilang Mentari Hamidy

Titre
Fuzzing Différentiel le WebAssembly

Programme d'Études Double Diplôme de Master

Filière d'Attachement Security and Cloud Computing (SECCLO)

Encadrants Académiques Prof. Davide Balzarotti, EURECOM
Prof. Jan-Erik Ekberg, Aalto University

Categorie La Thèse de Master **Date** 27 July 2020 **Pages** 133 **Langue** Anglais

Abstrait

WebAssembly, familièrement connu sous le nom de Wasm, est une spécification pour une représentation intermédiaire qui convient à l'environnement Web, en particulier du côté client. Il fournit une abstraction de la machine et des jeux d'instructions indépendants du matériel, où un langage de programmation de haut niveau peut cibler la compilation sur le Wasm au lieu d'une architecture matérielle spécifique. Le moteur JavaScript implémente la spécification Wasm et recompile l'instruction Wasm en l'instruction machine cible où le programme est exécuté. Techniquement, Wasm est similaire à un bytecode de machine virtuelle populaire, comme Java Virtual Machine (JVM) ou Microsoft Intermediate Language (MSIL).

Il existe deux implémentations majeures de Wasm, en corrélation avec les deux navigateurs Web les plus populaires. Ces deux sont le V8 de Chromium et le SpiderMonkey de Mozilla. Wasm n'exige pas une implémentation spécifique sur sa spécification. Par conséquent, les deux moteurs peuvent utiliser des mécanismes différents pour appliquer la spécification. Ces différentes implémentations peuvent ouvrir une question de recherche: les deux moteurs implémentent-ils également la spécification Wasm?

Dans cette thèse, nous allons explorer l'implémentation interne du moteur JavaScript par rapport à la spécification Wasm. Nous avons expérimenté en utilisant une technique de fuzzing différentiel, dans laquelle nous testons deux moteurs JavaScript avec un programme Wasm généré de manière aléatoire et comparons son comportement. Nous avons exécuté l'expérience pour identifier tout comportement anormal, puis nous avons analysé et identifié la cause profonde des différents comportements.

Cette thèse couvre largement la spécification WebAssembly. Il aborde plusieurs connaissances fondamentales sur la spécification qui manquent actuellement de références. Cette thèse présente également l'instrumentation faite au moteur JavaScript pour effectuer l'essai, qui peut être une base pour effectuer une essai similaire. Enfin, cette thèse analyse l'anomalie identifiée trouvée dans l'essai grâce à des techniques d'ingénierie inverse, telles que l'analyse statique et dynamique, combinées avec une analyse en boîte transparente au code source du moteur JavaScript.

Dans cette essai, nous avons découvert un comportement différent du moteur JavaScript qui est observable au point de vue du programme Wasm. Nous avons créé une preuve de concept pour démontrer les différents comportements qui peuvent être exécutés dans le navigateur Web récent jusqu'à l'écriture de cette thèse. Cette essai a également évalué l'implémentation des deux moteurs JavaScript sur la spécification Wasm pour conclure que les deux moteurs implémentent fidèlement la spécification.

Mots Clés webassembly, fuzzing, c++, compilateur, essai, langage de programmation, analyse dynamique, analyse statique

Kupersembahkan karya terbaik ini untuk
Ibunda tercinta
Ayahanda tersayang
Naini, Rosma, Ema, Anizar, Abuzar
Adit, Faris, Aziz, Tawfiq, Dimas, Joan
Icha, Ina, Neyga, Rima, Azza, Ahda, Iid,
Levin, Dinda, Ajeng, Leo, Upiek,
dan semua keluargaku

And also dedicated to my best friend Enrico Budianto, whom I always wished I could still have a long endless discussion together about our shared interests. Let this be a prayer that I can continue what he left.

And to all my teachers.

Acknowledgement

The education of the author was supported by the European Union through the Erasmus Mundus Joint Master's Degree (EMJMD) Master's programme in Security and Cloud Computing (SECCLO) for the intake year 2018 (Project Reference: 586541-EPP-1-2017-1-FI-EPPKA1-JMD-MOB).

The education of the author was conducted at Aalto University, Finland, and EURECOM, France.

This thesis work was supported through an internship in the Software and System Security (S3) Group at EURECOM, France, from February 2020 to July 2020.

Forewords

I give all praise to Allah that I can complete this master's thesis successfully. It has been a marvelous journey in the last two years in Finland and France. Before, I did not imagine that I would study computer security. I always thought that this field was challenging, and I did not dare to work in this field. Yet, getting admitted to the Security and Cloud Computing (SECCLO) program was a sign that people are confident with my ability.

I learned various new knowledge in this program. From a mathematical foundation of cryptography to binary program exploits. Without enrolling in SECCLO, I would not have been able to learn that knowledge by myself. I met people who share the same interests as me. We did many things together. We met every week to study cryptography together. We traveled together for our summer school. And we got to enjoy a small farewell dinner before our path diverged. I want to say thank you to all **SECCLO 2018** batch to the time we shared for the last two years. I extend my gratitude to my classmate **Christian Yudhistira** for being a very nice roommate in France. I also thank you to **Eljon Harlicaj**, a SECCLO 2019 student, for many random conversations we had in the past one year.

I got a very fantastic opportunity to work on a research assignment at Aalto University. I finally got my chance to get my hands dirty in working with compilers, and I mastered the compiler more than I could learn in my classroom ten years ago. From this opportunity, I finally realized that I want to focus my expertise in the compiler field. I want to say thank you very much to **Prof. N. Asokan** for the opportunity. It began only with small talk during the HAIC Dinner event, where he advised me to look at the Secure System Group web page about special assignments.

That web page led me to the research task on pointer analysis. The goal of this task was to support research in Pointer Authentication. This research task also contributed to my seminar paper class. I could not complete this without the support from **Dr. Hans Liljestrand**. He and Prof. Asokan also supported me in getting a summer internship at Security Lab in Huawei Technologies Finland. I experimented extensively with the LLVM compiler there. I want to thank **Dr. Jan-Erik Ekberg** for his constant support there, also for co-supervising this thesis work.

EURECOM was not my first preference in the SECCLO program. But now I realized it was an obscured judgment from an amateur in computer security.

After comprehending every possibility I could learn there, I knew I was not sent there by coincidence. EURECOM is the best place to learn computer security. I studied exploit techniques, which previously I only knew the theory. Then, I once again got a chance to work with compilers for my semester project. Ultimately, I was offered to do my master's thesis working in something I like: compilers. I could not do all of it without the constant trust and support of **Prof. Davide Balzarotti**. He was continuously confident with my progress. Even to this day, I could not even perceive the reason for his generous attitude towards me. Perhaps he could see my potential beyond my own self-esteem.

I could not be in this position also without the recommendation from **Dr. Ruli Manurung**. He consistently motivated me to continue my study and offered his recommendation anytime I needed. I also thank **Risman Adnan** for his support. In the last two years, the SECCLO Consortium also provided their support. I want to thank **Eija Kujanpää**, **Laura Mursu**, and **Gwenaëlle Le Stir** from Aalto University and EURECOM. I want to extend my gratitude to **Prof. Tuomas Aura** from Aalto University to tirelessly running the SECCLO program. I also want to thank every professor who taught me in their classes. Special thanks to **Prof. Yanick Fratanotnio** for his Mobile Systems and Smartphone Security (MOBISEC) class. He hosted an exciting and inspiring class.

I also want to say thank you to my close friends: **Anggi**, **Bayu**, **Dolot**, **Gita**, **Glenn**, **Jonathan**, and **Satriawan**. All of them contributed to the constant laughter, from daddy jokes to juicy gossips, I had in the last two years. Although we are separated by $0.002c$ to $0.04c$ distance, they never stop cheering me with their friendly conversations and supports. It helped me to overcome the random homesickness I had in my journey.

To end these very long forewords, I want to say thank you to everyone who supported me during my study. I hope this thesis can present new knowledge to everyone who reads it. I recognized that I had an ambitious goal to write this manuscript. It contains the complete knowledge I obtained in the last six months. Nevertheless, I know the energy I spent is worthwhile. And I wish this can inspire everyone to keep their faith in whatever they are doing. Because I always believe that there will be ways for every good intention we have.

Antibes, 27 July 2020

Gilang Mentari Hamidy

Contents

List of Tables

List of Figures

Abbreviations

AJAX	Asynchronous JavaScript and XML
AOT	Ahead-of-Time
API	Application Programming Interface
CG	Community Group
GC	Garbage Collector
ISA	Instruction Set Architecture
JIT	Just-in-Time
JS	JavaScript
MVP	Minimum Viable Product
SUT	System Under Test
WASI	WebAssembly System Interface
Wasm	WebAssembly
W3C	World Wide Web Consortium

Chapter 1

Introduction

"In the last few years, a significant effort has been devoted to devising methods that exploit the technology base in a disciplined way. Although promising, they carry along preconceptions brought from non-Web application development. Overall, our study led us to believe that the most critical element at this point is to formulate a concise and simple model of what these applications are about and to build a programming system around such a model."

Survey of Technologies for Web Application Development
Barry Doyle and Cristina Videira Lopes [12]

1.1 Motivation

In recent years, computer technology has shifted to a web-based environment. The advent of cloud computing enables the outsourcing of computation from local to remote machines. It creates novel inventions in web platform technologies, both on server-side and client-side.

Web Technology Surveys estimates that the majority of the web uses PHP [42]. A smaller portion uses ASP.NET, Ruby, and Java. Multiple technology choices are available to use for server-side development. However, the situation differs radically in the client-side ecosystem. JavaScript is the only option for client-side scripting. A smaller user base uses ActionScript from Adobe Flash, which is soon reaching its End-of-Life by the end of 2020 [26]. It follows the Java applet and C#-based framework Microsoft Silverlight that were already deprecated several years earlier.

This situation limits the programming language choice of client-side developers. The client-side ecosystem does not allow the use of alternative language, which can be beneficial for app development. One main use-case is integrating legacy codes which are written in other programming languages. The JavaScript efficiency in executing power-intensive tasks also raises the concern since not

every task can be delegated to the cloud.

WebAssembly opens up the possibility of integrating other programming languages into the web-client ecosystem. WebAssembly, abbreviated as Wasm, has been actively developed since 2015 [1]. Major browsers have been fully supporting Wasm since the standardization of Wasm specification. Wasm allows the use of existing programs or libraries that are written in other languages to be run in the web browser environment. It also claims that it can harness the client machine performance to perform compute-intensive tasks.

As a new technology, various exciting opportunities to tinker and experiment with Wasm technology emerges. Major JS engine has pledged to support and implement Wasm specification. As multiple implementations are developed in parallel with different developers, different behavior between the implementation may be possible. Moreover, despite being incorporated as required testing steps, fuzzing has been primarily focused on the general JS engine components. Thus, several areas in Wasm implementation are open for investigation through fuzz-testing.

1.2 Problem Statement

This thesis aims to explore the Wasm implementation to observe any potential misbehavior. Wasm is a relatively young technology that opens rooms for exploration. Moreover, Wasm is targetted towards a wide use-case of the world wide web, which requires scrutiny over its claim on security aspects.

The research in this thesis explores the possibility of the differences between the implementation of the Wasm specification. We are interested in whether the implementor, namely the JS engine, is implementing the specification accurately. Also, we are interested in finding any possibility for unexpected behavior from any given Wasm program, which can lead to exploits and vulnerability.

1.3 Contribution

This thesis aims to provide an introduction to the WebAssembly specification and implementation. The writer recognizes the limited amount of academic references for Wasm at the moment. Many resources revolve around developer blogs, articles, specification documents, and technical documentation. Several papers provide a foundational background of the Wasm [23, 50]. However, these papers have been dated compared to the recent development and finalization of the Wasm specification. Moreover, many articles and references discuss the use of Wasm instead of its internal implementation. Thus, the thesis provides the necessary information for the Wasm engine experiment in general.

The thesis also proposes a software design to perform differential testing. Differential testing can be incorporated in the software development pipeline

to compare multiple implementations of the same specification. Some research and investigation conducted differential testing against compiler implementation, such as GCC vs. LLVM [6]. As it is still in its infancy, WebAssembly has a room to explore the differential testing on two major implementations.

Finally, the thesis gathered the experiment result. The thesis discusses the observation from the differential testing performed. Additionally, the thesis also presents the author's contribution to open-source development, which occurred during the research.

In summary, the thesis expects itself to be self-contained. It provides all the necessary information to perform the proposed Differential Fuzzing the WebAssembly experiment. For a more in-depth explanation beyond the scope of the thesis, readers are invited to also consult the cited references.

1.4 Structure of the Thesis

The thesis is organized into five main chapters apart from the first introductory chapter.

The second chapter introduces the reader to the WebAssembly. It discusses the conceptual overview of the Wasm to the specification of Wasm instruction sets. The discussion continues to the Wasm integration to the JavaScript program. Finally, the chapter concludes by introducing the two major implementations of Wasm and their comparison.

The third chapter introduces the foundation of software testing. It discusses software testing from its basic motives in the software development lifecycle. It also provides the general idea of fuzz testing and differential testing. This chapter concludes by providing the argument that becomes the basis of the experiment.

The fourth chapter explains the development of the Wasm differential fuzzing experiment. It suggests the approaches and the main requirements for the experiment system. Then, it tells the design rationale for the fuzzer infrastructure, from an architectural and workflow standpoint. The modification to the investigated Wasm implementation is also presented and detailed in this chapter.

The fifth chapter discusses the result of the experimentation. It presents the analysis of the experiment result using a common reverse engineering technique, namely static and dynamic analysis, combined with white-box analysis to the original JS engine implementation. It also proposes a proof-of-concept of the identified differences from the experiment.

The sixth chapter concludes the thesis and describes the contribution of the author that is made during the research. It also proposes brainstorming ideas to follow-up the research presented in this thesis.

1.5 Accompanying Source Code

This thesis is accompanied by the source code of the testing system. The source code is accessible through the link:

https://github.com/gilanghamidy/DifferentialFuzzingWASM

The source code itself is self-contained and linked to all dependencies, including the instrumented JS engines. The instrumented JS engine is forked from the original code, and resides in a separate forked repository. The reader can automatically update the dependency through `git submodule update` command.

This thesis discusses several implementations that can be found from the source code. The code listing mentioned in this thesis, also information that is cited from the code is indicated by the source code file name and line number. Note that the cited line number from the JS engine source code may change over the time of the development. Therefore, the reader is suggested to refer to the accompanying forked JS engine source for the reference, which is pinpointed at the specific commit level where the experiment is performed.

The testing system program is licensed under the MIT License. The dependent source codes are licensed under their original licenses.

Chapter 2

WebAssembly

> *"Of course, every new standard introduces new costs (maintenance, attack surface, code size) that must be offset by the benefits. WebAssembly minimizes costs by having a design that allows (though not requires) a browser to implement WebAssembly inside its existing JavaScript engine (thereby reusing the JavaScript engine's existing compiler backend, ES6 module loading frontend, security sandboxing mechanisms and other supporting VM components). Thus, in cost, WebAssembly should be comparable to a big new JavaScript feature, not a fundamental extension to the browser model."*
>
> *"Why create a new standard when there is already asm.js?"*
>
> *WebAssembly FAQ* [1]

WebAssembly is a relatively young technology. Therefore, researching the reference for this technology can be quite challenging. Several resources discuss the use-case of Wasm, especially to improve the web client experience. However, academic references for Wasm internals are hard to find. This chapter aims to cover the fundamental concepts behind Wasm and its internal mechanisms. The topic covered in this chapter is necessary for the experiment discussed in this thesis.

Section 2.1 introduces the WebAssembly from its historical, high-level, and specification perspectives. The Wasm specification details and its essential elements are discussed in Section 2.2 . This second section also provides examples that compare Wasm programs and their equivalent C programs. Section 2.3 covers the compilation of high-level language to Wasm. Section 2.4 describes the details to embed a Wasm program to a JavaScript program. This section also provides several code examples to conceive the Wasm in action. The chapter concludes with Section 2.5 and Section 2.6, which introduce the JS engines that implement the Wasm, and compares them.

2.1 WebAssembly Overview

2.1.1 History

In the past few decades, the major client-side scripting revolves around mostly on JavaScript programming language. However, many workarounds have been attempted to introduce different programming languages. For instance, Adobe Flash introduces ActionScript for their interactive web platform [37]. Another example is Java applets which embed Java application on the web page [11].

Microsoft DHTML allows a web browser to execute a compiled native library [24]. It gained popularity due to the ubiquitous use of Microsoft Windows and Internet Explorer. Although it mainly utilizes Microsoft ActiveX controls, it is also possible to use other native libraries. The browser downloads and attaches the native binaries from the server into the client page [12]. Still, this technology suffers from severe security issues. Hence, it gained popularity as a backdoor for web app [33].

Another issue with the development environment of the web client platform is the fragmentation of the web browser itself. Before the ECMAScript standardization, every major browser implemented its own version of JavaScript. Many of the implementations were browser-specific, thus created fragmentation in the JavaScript ecosystem [12]. A JavaScript code that was running on Internet Explorer was not guaranteed to run on Netscape Communicator and vice versa. It created significant issues for web developers to handle every variant to support multiple web browsers.

Mozilla introduced Emscripten around 2010. It is a compiler that uses LLVM toolchain to compile code to JavaScript language. It allows the compilation of existing programming languages compatible with LLVM to JavaScript [50]. Programming languages, such as C, C++, or Objective-C, can be directly compiled into JavaScript by using Emscripten. Other programming languages can also benefit from it by compiling their runtimes to JavaScript. In order to maintain program semantics, Emscripten generates JavaScript codes that emulate the source program execution.

Many web browsers can directly execute the Emscripten compiled programs. It also allows the use of existing libraries and runtimes written in languages other than JavaScript. Eventually, the Emscripten-generated JavaScript codes evolved into a specialized Embedded Domain-Specific Language (EDSL). This EDSL, which is called asm.js, is a strict subset of JavaScript. It employs several language constructs to emulate the behavior of native-targeting languages. Modern JavaScript engines can identify an asm.js script and optimize it to achieve higher performance. At the same time, it also allows the legacy JavaScript engine to run the program without any modification in the program itself.

Despite its usefulness, JavaScript faces several fundamental limitations to express native-targetted programs. For instance, the JavaScript engine requires the parsing and recompiling of the JavaScript representation. It creates more overhead than the actual native execution. Another issue is that JavaScript lacks a 64-bit integer representation. It raises issues for many native-targetted programs that rely on this data type. Although major JavaScript engine optimizes asm.js programs, further improvement remains a challenge without modifying the infrastructure of the intermediate representation itself.

Principal web browser vendors announced the WebAssembly Project (Wasm) around 2015. It marked the first step of standardizing bytecode that is suitable for the web environment. Wasm project aims to address the limitations of JavaScript as an intermediate representation of programs written in other programming languages [23]. It is designed to be compact, portable, fast, and secure to address the use case in the web environment that can outperform asm.js performance [51].

Wasm specification defines a virtual Instruction Set Architecture (ISA). It eliminates the overhead from using JavaScript as an intermediate representation. The ISA abstracts the original program and enables JavaScript engines to compile it to the target machine efficiently. The architecture emphasizes code simplicity by only providing basic fundamental instructions. It is up to the JavaScript engine's implementation to perform necessary optimization to the Wasm program.

The WebAssembly Community Group (CG) released its first version of WebAssembly specification in March 2017. The first version of Wasm specification consists of Minimum Viable Product (MVP) requirements, as defined in Wasm high-level goals [23]. This release marked the end of Wasm preview and a point where no further fundamental design changes in the future. As of late 2019, major JavaScript engines have provided support for WebAssembly programs. The community group is also actively discussing improvements to the standard to add more functionality and features to Wasm.

As Wasm is an emerging technology, the adoption rate is still limited to some extent. The majority of the use-cases, for example, revolve around high-performance multimedia application, porting non-browser application into a web-based application and reusing existing toolkits and libraries that do not have a JavaScript counterpart [1]. Recently, Microsoft also introduced support for integrating their .NET infrastructure with WebAssembly using the Blazor framework [36]. Blazor enables the compilation of .NET program written in C# to Wasm, then executing locally in the client browser. This kind of approach will enable further adoption of new applications to utilize Wasm.

2.1.2 High-level Workflow

All Wasm program originates in the source language. Currently, only C and C++ that are officially supported by Wasm standard. The Wasm standard aims to provide the minimum requirement to execute a C or C++ program without modifying its original sources. Wasm adheres to standard ANSI C and ISO C++ specification. Therefore, the source does not need any supplementary framework-specific syntaxes or markups.

The source program is compiled by the Wasm toolchain to produce a Wasm binary. The resulting binary, which is called the Wasm module, works similarly to a native shared library. It consists of the instruction codes and symbols, such as function information global variables. External code can then load the compiled binary and access it from its code. The user program can enumerate the available functions to call or invoke an entry point, such as the main function, if it is defined.

The host environment, typically the JavaScript engine, will load and compile the Wasm instructions to the target machine language. The host environment can either perform Just-in-Time (JIT) or Ahead-of-Time (AOT) compilation based on the configuration. JIT compilation compiles the function when it is called for the first time. On the other hand, the AOT compilation compiles the entire Wasm module at the time of loading. The compiled code resides in the host process, and it will execute the machine instruction when the host code calls it.

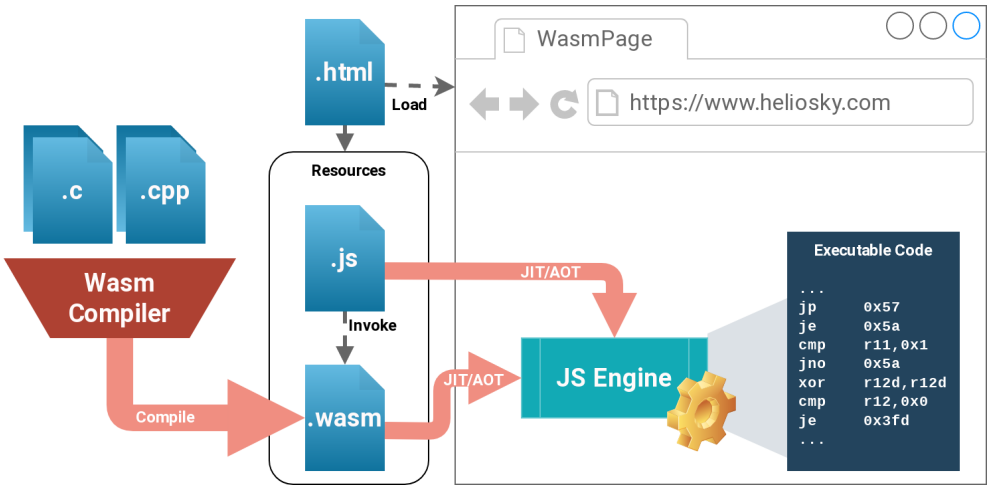

Figure 2.1. High-level workflow of Wasm program

When the host code wants to invoke a Wasm function, it needs to instantiate the Wasm module. The instantiation process will validate the module requirements, from required functions or other elements. The host environment will also perform necessary linking. Wasm module may require an external function to work, for instance, the standard library functions. Wasm specification does not specify any standard library, and a Wasm module may not include all the functions

it requires. Therefore, it is up to the Wasm program and the host environment to supply the required library. The WebAssembly CG, has a working group to standardize system API for Wasm. WebAssembly System Interface (WASI) provides a Wasm port of the libc standard library, which a Wasm program can link.

2.1.3 Virtual Instruction Set Architecture (ISA)

Wasm models the program instruction by defining its own Instruction Set Architecture (ISA). ISA is an abstraction of a machine that executes a program [39]. ISA defines necessary components that a program requires to run. It includes memory model, operations, instruction encoding, and so on.

Wasm specification defined a Virtual ISA. Virtual ISA does not have an actual physical machine implementation. Moreover, Wasm also does not expect the host environment to implement a virtual machine execution. Instead, it compiles the Wasm instruction to target machine instructions. It makes Wasm an intermediate representation rather than an interpreted instruction. Wasm uses this approach to remain machine and implementation agnostic [22].

Wasm uses a stack machine architecture, which is is a machine model where the instruction operates with the operands stored in the stack. A stack machine instruction pops operands from the stack, computes, and pushes back the result [40]. It is different from a register machine where the operands are stored in registers. Although the real hardware stack machine is relatively uncommon, modern virtual machines, JIT compilers, and interpreters model their architecture with a stack architecture. One advantage of stack architecture is it has a more compact instruction code compared to a register-based machine. The stack architecture also enables more efficient compilation and interpretation. It roughly models the Abstract Syntax Tree (AST) representation of the original source [22]. It allows the JIT compiler to perform more optimization, depending on the target machine.

Wasm defines a limited set of data types. Currently, Wasm only includes 32-bit and 64-bit integer and floating-point types [22]. It contrasts to x86 or ARM architectures, which provides a wide range of integer sizes, from 8-bit to 64-bit. Since C and C++ typically support various integer sizes, integer sizes lower than 32-bit, typically `short` and `char` type, needs to be promoted to 32-bit. However, this limitation does not conflict with C and C++ language specification, where the basic data types do not have a maximum size requirement. C and C++ only require that the basic data types follow a specified ordering and minimum size requirement.

6.8.1 Fundamental types [basic.fundamental]

1. There are five *standard signed integer types*: "`signed char`", "`short int`", "`int`", "`long int`", and "`long long int`". In this list, each type provides at least as much storage as those preceding it in the list. ...

4. ... Except as specified above, the width of a signed or unsigned integer type is implementation-defined.

ISO Standard - Programming Languages — C++ (2020) [29]

Table 2.1 lists Wasm basic instructions types. Most instructions need to specify the data types it operates on. The operands in the stack need to agree with the instruction data type, which the Wasm engine validates the agreement between the operand and the instructions. A well-formed Wasm program needs to have a valid stack structure according to the type and instruction agreements. Since most instructions also produce a result and push it to the stack, the data type agreement chains from the entry point to the end of the function.

Table 2.1. Wasm basic intruction types

Types	**Description**
Numeric	Arithmetic or logical operations on numerical operands
Parametric	Manipulation of operand stack
Variable	Load and store to local or global variable
Memory	Load and store to linear memory
Control	Manipulating control flow of the program

In the function scope, Wasm allows defining local variables. A local variable is stored independently from the operand stack. It enables persistent storage during function execution without being influenced by operand stack changes. It is analogous to a register in a register-based machine but with a larger limit of the register count. Function arguments are also stored as local variables. Apart from local variables, Wasm also provides global variables which have a module-level scope. Global variables are accessible from every function in the same module, similar to global or static variables in C and C++ programs. Wasm program access variables through the index number, which is encoded as an immediate value of instruction. Thus, it is not possible to dynamically address local or global variables.

Other than local and global variables, Wasm provides dynamic storage in

the form of linear memory. Linear memory is an addressable data storage that analogous to random-access memory in typical computer architecture. Wasm program access the linear memory using memory instructions. Memory instructions accept memory address as the operand, allowing dynamic memory access based on program logic. Each memory unit is addressable in a one-byte unit. And despite the Wasm only support 32-bit and 64-bit operand, It is possible to manipulate data in memory with bit-length lower than 32-bit. Wasm memory instructions provide load and store instructions with a specific bit length.

Wasm, however, does not allow a raw code pointer in programs. In a typical architecture, code and data reside in the same address space. Without conscientious programming, a data pointer can access code area and vice versa. Many severe security vulnerabilities originate from this problem. In an attempt to provide a more secure environment, Wasm defines a separate location to store indirections called a table. Wasm tables are similar to a jump table or virtual table but with a strict type checking. A Wasm table entry is tagged with the function signature of the target function. Wasm engine will perform indirect call validation before jumping to the target function. Therefore, it can detect and prevent an invalid indirect call due to error or malicious act.

Both memory and table addresses do not reflect the actual address space where the Wasm is executed in the target machine. Wasm linear memory always starts at address zero to the maximum available address. Wasm manages memory similar to paging in an operating system environment. Each memory page has 64 KiB of space. Wasm program can dynamically increase or decrease the available space in a way that is similar to `sbrk` system call in the POSIX system[1]. On the other hand, Wasm tables are always static. The Wasm program has no ability to manipulate table entries, which maintains the integrity of Wasm program validation.

2.1.4 Interface with External Codes

Wasm provides several interface mechanisms to enable information exchange to the host environment. The interface mechanism is analogous to Executable and Linkable Format (ELF)[2] file in typical operating system environments. It provides symbol information that the host can read to link with the Wasm module. Wasm symbol includes Exports, Imports, and Entry Point.

Export symbols are Wasm components that are accessible from the host environment. Wasm module can export its function to make it invokable from the host code. Unlike ELF format, Wasm function export is type-safe. The function export

[1] `sbrk` is an API function that performs a system call to the kernel. The call resizes the data segment of a program address space and makes it usable by the program. It is typically used within a memory management function, such as `malloc`. [13]

[2] Executable and Linkable Format (ELF) is an object file format which provides information about the executable. It typically includes symbol information (e.g., function references), program entry point, and external dependencies. [10]

information consists of the function signature of the target function. Therefore, the host environment can validate the function argument before the function invocation. It eliminates invalid argument passing, hence improving the overall program security.

Wasm module can also export global variables and memory. It enables information exchange between the host and Wasm module without invoking a Wasm function. Global variable export is also type-safe, but it only provides Wasm basic types. For larger data, memory export is the better choice as it allows efficient transfer of raw data between the Wasm module and the host. However, memory export is untyped and unstructured. It is similar to accessing and manipulating raw memory dump. The host code needs to make careful consideration when manipulating data through memory export. It ensures the access does not induce an error in the Wasm program.

Wasm table can also be exported. It enables the host environment to transfer function references from and to the Wasm module. It enables the host program to access reference information that is provided by the Wasm module. Similar to Wasm memory, Wasm table uses index-based access. The host program may obtain the element inside the Wasm table by supplying the index.

Apart from exporting symbols, Wasm module can also import symbols. Wasm module may import a function, memory, or table. Importing symbols require the host to supply the required element before instantiating the Wasm module. The host environment ensures that the host code provides all required elements, including correct memory allocation, table allocation, and function. The module instantiation process is analogous to the linking process in a compilation, where the code is linked to all required symbols before it can be executed.

Wasm module may define a data and element initializer within its module. It is used to initialize Wasm memory content and Wasm table value. Wasm module can also define a start function. The start function behaves similarly like an executable entry point, which is executed at the start of the program. The host environment automatically invokes the start function after module instantiation completes. An example of a start function use-case is to perform runtime initialization of a Wasm program that a simple data and element initializer cannot perform.

2.1.5 API and System Interface

Wasm specification does not provide a standard library or system interface. The specification is purely an ISA without higher-level infrastructure such as system call. However, many programs require this infrastructure to work properly. To put it simply, every application which utilizes the standard library requires access to the system call. For instance, a simple printf function requires a write system call to the standard input stream.

The current Wasm implementation emulates most of the low-level system call through the JS program. It usually provided by the toolchain, such as Emscripten, which will be discussed in the later chapter. Every toolchain and host environment may provide its own emulation implementation. It creates a possibility of fragmentation in the toolchain and host environment. This issue leads to the initiation of WebAssembly System Interface (WASI) specification [9].

WASI aims to standardize the system interface API for the Wasm environment. This standard ensures every system call implementation in Wasm adheres to the same specification. In a way, it is similar to the POSIX specification for a UNIX-based operating system. Being an architecture for a close-to-native language, Wasm requires low-level system API functionality. Nevertheless, Wasm has a different architecture where a typical operating system functionality does not exist. Wasm program also runs in a different execution environment. It introduces an additional boundary for a low-level system call.

WASI ensures that Wasm programs remain portable across different operating systems. WASI introduces a layer that is implemented in the standard library internals. Meaning that the libc implementation for Wasm calls the WASI API instead of specific operating system API. WASI expects that toolchains and compilers provides seamless interface with WASI interface when compiling to Wasm.

Nevertheless, not every system call can be emulated. For example, Wasm architecture does not recognize the notion of a multi-process environment. Therefore, low-level functionality, such as fork, is not going to be available in Wasm. Additionally, WASI must adhere to the sandboxed environment of Wasm. It ensures malicious programs do not have direct open access to the underlying system.

At the time of the writing, WASI specification is still in the drafting process. However, some toolchains, including Emscripten, have provided support to the current specification draft. Another implementation is Wasmtime, a Wasm runtime which can run Wasm program outside of the JS engine. It is analogous to running a Java program in a shell.

2.2 Wasm Semantics and Source Format

2.2.1 Wasm General Source Format

To provide human-readable text, Wasm specifies a text format to represent a Wasm program. The text format is analogous to a human-readable assembly text. Wasm text format uses a syntax based on s-expressions, a syntax style that was popularized by Lisp. S-expression is a nested list notation that enables to represent the abstract syntax tree closely. A line comment starts with double semicolon (`;;`), and block comments in the source are enclosed between `(;` and `;)`

token.

The module is the root element in the Wasm source. A module contains all Wasm elements: function definition; declaration of global variables, export, import, table, memory, and value initializers. Also, Wasm source can declare a function signature type separately. It identifies the signature of a function definition, validates call instruction, and validate table elements. It is comparable to the typedef in C programming language.

Listing 2.1. Wasm module elements

```
(module
  (import ... ) (type ... ) (memory ... ) (table ... )
  (func ... ) (global ... ) (export ... )
  (elem ... ) (data ... ) (start ... ))
```

All declaration in Wasm source is indexed in the order of appearance. However, Wasm source can also define an identifier to refer to the declaration. An identifier begins with a dollar ('$') symbol. Wasm specification allows the identifier to be any printable ASCII characters except whitespace, quotation mark, comma, semicolon, or bracket.

Listing 2.2. Using identifier in Wasm source

```
(module
  (global $glob_1 ... )
  (func $addi32 ... ))
```

2.2.2 Wasm Function and Signature Type

A function signature is defined using the `type` keyword. The function signature consists of parameter type lists and result type lists. The specification supports multiple return values. However, no host environment supports this feature at the moment. A function definition can either use type signature or define the signature inline with the definition. But, a function needs to declare the parameter inline to assign an identifier to it.

A single declaration can contain multiple anonymous declarations. However, only a single declaration can be made when using an identifier. The following declaration must be made in a separate expression. The order of the parameter follows the order of the appearance on the list.

Listing 2.3. Function signature type and function definition

```
(module
  ;; Function type sig1 with 2 i32 parameter and return i32
  (type $sig1 (func (param i32 i32) (result i32)))

  ;; Function addi32 uses type sig1
  (func $addi32 (type $sig1) ... )

  ;; Function subi32 declares parameter with identifier
  (func $subi32 (param $p1 i32) (param $p2 i32)
                (result i32) ... ))
```

A function may declare local variables. Local variable declaration resembles

parameter and return value declaration of the function signature. However, it can only appear in the function definition. All local variable declaration must appear before the first function instruction. It resembles old versions of the C programming language, where every variable declaration must appear at the beginning of the block.

Listing 2.4. Function local variables

```
(module
  (func $compute (param $p1 i32) (param $p2 i32)
    (result i32)
    (local $val1 i32) ;; With identifier
    (local i64 f32)   ;; Anonymous, access using index number
    ... ))
```

2.2.3 Instructions in a Wasm Function

Wasm instruction can be written in a procedural sequence similar to a regular assembly program. Wasm executes the program sequentially from the function entry point. The validation also occurs following each instruction step according to Wasm validation rules. This instruction writing is equivalent to writing expression in Postfix Notation. In postfix notation, operands appear before the operator, and the ordering of the operands follow the order in the stack.

Apart from regular writing conventions, Wasm also allows program writing in folded form. It uses Prefix Notation and encloses an instruction and its operands in parentheses. Prefix notation itself is an expression notation where the operator appears at the beginning of the expression. The operand can also be a nested instruction itself. Note that the nested operand must also be written in the folded form if it contains a complex expression. It is similar to Lisp programming style. Both writing conventions are equivalent. However, the folded form is considered as syntactic sugar.

Listing 2.5. Writing Wasm instruction

```
(module
  (func $add_regular (param i32 i32) (result i32)
    local.get 0
    local.get 1
    i32.add)

  (func $add_folded (param i32 i32) (result i32)
    (i32.add (local.get 0) (local.get 1))))
```

Most Wasm instruction requires to specify the operand type which it operates on. The operand that is stored in the stack must agree with the operand type specified by the instruction. The operand type appears at the beginning of the instruction keyword before the instruction keyword itself. For example in Listing 2.5, the add instruction operates on `i32` type which is a 32-bit integer. Wasm does not allow implicit conversion of values. Hence, the program needs to cast the operand to the correct instruction type explicitly. Wasm provides conversion

operators between value types that the program can use to cast the operand.

2.2.4 Wasm Control Structure

Unlike regular assembly programs, Wasm has a different control instruction logic. Wasm does not allow an arbitrary jump to any instruction point. Instructions are grouped into blocks, and control instructions navigate through the block. Wasm program has three different blocks: regular block, conditional, and loop. A block may specify the result value type after the block completes executing. Thus, a block is analogous to a closure. A block begins with block keyword and terminates at end instruction. end instruction is a pseudo-instruction that marks the end of the block.

When the program enters a block, Wasm creates a new operand stack. An inner block cannot access the operand in the outer block. The program needs to spill the value via a local variable to transfer the value inside the block. The final state of the operand stack must also agree with the result type defined in the block. When the execution escapes a block, Wasm validates the stack state with the result type of the block. Execution can exit from a block by reaching the end instruction, or by executing branch instruction.

Wasm has three types of branches: unconditional branch (`br`), conditional branch (`br_if`), and table branch (`br_table`). An unconditional branch is always executed regardless of any external state. A conditional branch, on the other hand, consumes one operand from the stack to determine whether a branch needs to be taken or not. A conditional branch takes the branch if the operand value is non-zero. It is usually paired with test or comparison instructions. The target blocks are supplied as an array in the immediate value. The table branch then consumes an integer operand as an index to select the target block.

Branch instruction behaves differently for every block type. However, all branch instructions can only target their enclosing blocks. A branch instruction cannot target other blocks that do not have parent-child relations. It is due to Wasm treats block context also as a stack. When an execution enters a block, Wasm pushes a block context to a stack. The context is popped from the stack when the execution leaves the block.

A regular block combines multiple instructions into a single scope. The keyword to define regular block is block-end. It is analogous to scope block in C and C++ programming languages. In Wasm, however, block groups function as an escape label. A branch to a regular block exits the block, effectively skipping the rest of the block instructions. It is called a forward jump. The end pseudo-instruction in a regular block is the actual label where the branch instruction jumps. Listing 2.7 presents a simple example of Wasm code that is equivalent to a **`goto`** instruction in Listing 2.6.

Listing 2.6. C program with **goto** instruction

```c
int simple_goto(int val) {
  int res = 3;
  if(val % 5 == 0) goto EXIT;
  val *= 20;
EXIT:
  res *= val;
  return res;
}
```

Listing 2.7. Regular block from C code in Listing 2.6

```
(module
  (func $simple_goto (param $val i32) (result i32)
    block $EXIT
      (i32.eqz (i32.rem_s (local.get $val) (i32.const 5)))
      br_if $EXIT ;; if(val % 5 == 0) goto EXIT
      (i32.mul (i32.const 20) (local.get $val))
      local.set $val ;; val *= 20
    end ;; EXIT:
    (i32.mul (local.get $val) (i32.const 3)) ;; res *= val
  ))
```

A conditional block is similar to a regular block, but it consumes an i32 operand from the stack. The keyword sequences for a conditional block is if-else-end. If the operand is non-zero, the execution enters the block. Otherwise, the execution either enters an else block or skip the block altogether if an else block is not defined. Listing 2.8 shows the use of conditional block that is equivalent with Listing 2.7.

A loop block is used to define an iteration. The keyword sequence for the loop block is loop-end. However, the Wasm loop block is not iterative by itself. It requires a branch instruction targetting to the loop block to loop back. It is called a backward jump. Without the backward jump, the loop block falls through the end instruction and exit the block. Due to its semantic, a Wasm loop block is analogous to a do-while-loop in the C and C++ programming language. Do-while-loop checks the loop condition at the end of the loop. The backward jump is also analogous to continue statement in C and C++ language. Listing 2.9 and Listing 2.10 show equivalent loop instruction in C language and Wasm.

Listing 2.8. Equivalent code with Listing 2.7 using conditional block

```
(module
  (func $simple_conditional (param $val i32) (result i32)
    (i32.rem_s (local.get $val) (i32.const 5))
    if ;; if(val % 5 != 0)
      (i32.mul (i32.const 20) (local.get $val))
      local.set $val ;; val *= 20
    end
    (i32.mul (local.get $val) (i32.const 3)) ;; res *= val
  ))
```

Listing 2.9. Do-While loop in C

```c
int do_while(int a, int b) {
  int res = 3;
  do {
    a++;
    res *= b;
  } while(a <= 100);
  return res * a;
}
```

Listing 2.10. Loop block code from C code in Listing 2.9

```
(module
  (func $do_while (param $a i32) (param $b i32) (result i32)
    (local $res i32) (local $cond i32) (local $temp_a i32)
    i32.const 3
    local.set $res     ;; res = 3
    loop $L0
      (i32.mul (local.get $res) (local.get $b))
      local.set $res  ;; res *= b
      (i32.lt_s (local.get $a) (i32.const 100))
      local.set $cond ;; a < 100
      (i32.add (local.get $a) (i32.const 1))
      local.tee $temp_a
      local.set $a     ;; a++
      local.get $cond
      br_if $L0        ;; while(a <= 100)
    end ;; POSTCONDITION: 101 <= temp_a <= (a + 1)
    (i32.mul (local.get $res) (local.get $temp_a)) ;; res * a
  ))
```

Representing while-loop and for-loop statements in Wasm is more complicated than do-while-loop. Do-while-loop checks the condition at the end of the loop. Consequently, do-while-loop at least executes the loop instruction once. In contrast, while-loop and for-loop condition is located in the prelude of the loop. Thus, the equivalent Wasm program must check the condition before entering the loop block. Loop block does not provide precondition checks before entering the loop. Hence, the equivalent Wasm program must combine regular or conditional block with the loop block. Listing 2.11 and Listing 2.12 show the C and its equivalent Wasm program that contains a loop.

Listing 2.11. C program with **`while`** instruction

```c
int while_loop(int a, int b) {
  int res = 3;
  while(a <= 100) {
    a++;
    res *= b;
  }
  return res * a;
}
```

Listing 2.12. Wasm code from C code in Listing 2.11

```
(module
  (func $while_loop (param $a i32) (param $b i32) (result i32)
    (local $res i32) (local $temp_a i32) (local $cond i32)
    i32.const 3
    local.set $res
    block $LOOP_END
      block $LOOP_START
        (i32.le_s (local.get $a) (i32.const 100))
        br_if $LOOP_START ;; if a < 100 then enter loop
        local.get $a
        local.set $temp_a
        br $LOOP_END      ;; otherwise, skip the loop
      end ;; LOOP_START:
      loop $LOOP
        (i32.mul (local.get $res) (local.get $b))
        local.set $res    ;; res *= b
        (i32.lt_s (local.get $a) (i32.const 100))
        local.set $cond   ;; a < 100
        (i32.add (local.get $a) (i32.const 1))
        local.tee $temp_a ;; a++
        local.set $a
        local.get $cond
        br_if $LOOP       ;; while(a < 100)
      end
    end ;; LOOP_END:
    (i32.mul (local.get $res) (local.get $temp_a)) ;; res * a
  ))
```

Enclosing a loop block with a regular block is also required to allow escaping the loop block in an arbitrary location. A loop block on its own can only exit the loop when the execution reaches the end instruction. By enclosing the loop with a regular block, a branch instruction inside the loop can target the regular block. A branch inside a loop block that targets a regular block effectively exits the loop. It is equivalent to a break statement in C and C++ language. Listing 2.14 shows Wasm code of C program in Listing 2.13 when compiled with Clang.

Listing 2.13. C program with **break** instruction

```
int break_loop(int a, int b) {
  int res = 3;
  while(a <= 100) {
    a++;
    if(a * b == 100) break;
    res *= b;
  }
  return res * a;
}
```

Listing 2.14. Wasm code from C code in Listing 2.13

```
(module
  (func $break_loop (param $a i32) (param $b i32) (result i32)
        (local $res i32) (local $a_inc i32) (local $a_temp i32)
    i32.const 3
    local.set $res
    block $END_LOOP
      (i32.gt_s (local.get $a) (i32.const 100))
      br_if $END_LOOP       ;; if a > 100 skip the loop
      (i32.add (local.get $a) (i32.const -1))
      local.set $a_inc
      (i32.add (i32.mul (local.get $b) (local.get $a))
               (i32.const -100))
      local.set $a          ;; a to store a * b value
      block $BREAK_LOOP
        loop $LOOP
          (local.set $a_temp (local.get $a_inc))
          (i32.add (local.get $a) (local.get $b))
          local.tee $a      ;; store a * b result
          i32.eqz
          br_if $BREAK_LOOP ;; if(a * b == 100) break
          (i32.mul (local.get $res) (local.get $b))
          local.set $res    ;; res *= b
          (i32.add (local.get $a_temp) (i32.const 1))
          (i32.lt_s (local.tee $a_inc) (i32.const 100))
          br_if $LOOP       ;; while(a <= 100)
        end
      end ;; BREAK_LOOP:
      (i32.add (local.get $a_temp) (i32.const 2))
      local.set $a          ;; reset a value to original
    end ;; END_LOOP:
    (i32.mul (local.get $a) (local.get $res)) ;; res * a
  ))
```

2.2.5 Memory, Table, and Initializer

Wasm module may contain a memory declaration. Memory declaration sets the minimum memory capacity that a Wasm module can use. Optionally, a Wasm module can also set the maximum memory size. If a Wasm module does not specify the maximum memory size, the maximum memory size depends on the host implementation. The syntax to declare memory is `(memory [MIN] [MAX])`. The minimum and maximum values are in Wasm page units. One page unit is 64 KiB of space. In the current specification, only one memory declaration can appear in a module.

Wasm table declaration is similar to memory declaration. Wasm module must specify the minimum and optionaly specify the maximum number of table elements. Wasm table declaration may also only appear once in a module. The syntax to declare table is `(table [MIN] [MAX] anyfunc)`. The keyword `anyfunc` specifies the element type as a function reference of any type. Future Wasm specification may extend the element type beyond function references.

Listing 2.15. C program with some string

```
char const* get_text(int v) {
  if(v % 2) return "it is an even number";
  else if(v % 3) return "it is divisible by three";
  else if(v % 5) return "it is divisible by five";
  else return "it is some odd number";
}
```

Listing 2.16. Wasm code with data initializer from C code in Listing 2.15

```
(module
  (memory 1)
  (func $get_text (type $t0) (param $v i32) (result i32)
    (local $ret i32)
    (local.set $ret (i32.const 0))
    block $END_IF
      (i32.and (local.get $v) (i32.const 1))
      br_if $END_IF  ;; if (v % 2) ret = 0
      (local.set $ret (i32.const 21))
      (i32.rem_s (local.get $v) (i32.const 3))
      br_if $END_IF  ;; else if(v % 3) ret = 21
      i32.const 46
      i32.const 70
      (i32.rem_s (local.get $v) (i32.const 5))
      select         ;; ret = v % 5 ? 46 : 70
      local.set $ret
    end ;; END_IF:
    local.get $ret)  ;; ret is address to the string
  (data $d0 (i32.const 0)  "it␣is␣an␣even␣number\00")
  (data $d1 (i32.const 21) "it␣is␣divisible␣by␣three\00")
  (data $d2 (i32.const 46) "it␣is␣divisible␣by␣five\00")
  (data $d3 (i32.const 70) "it␣is␣some␣odd␣number\00"))
```

Wasm module can initialize memory and table elements. The memory initializer section specifies the data offset and the data content. The content of the memory can be written in the UTF-8 string. The program can also specify a raw hexadecimal value by using the escape sequence. A memory initializer is used to initialize static data in Wasm memory. A typical use-case is a string table. Listing 2.15 and Listing 2.16 show the C program with string and the equivalent Wasm program that is compiled using Clang.

Listing 2.17. C program with function pointer

```c
typedef int (*func_t)(int, int);
int compute(func fPtr, int a, int b) { return fPtr(a, b); }
int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
int mul(int a, int b) { return a * b; }
int div(int a, int b) { return a / b; }

func_t getfunc(int idx) {
  if(idx % 2) return &add;
  else if(idx % 3) return &sub;
  else if(idx % 5) return &mul;
  else return &div;
}
```

Listing 2.18. Wasm code with data initializer from C code in Listing 2.17

```
(module
  (type $func_t (func (param i32 i32) (result i32)))
  (table 4 funcref)
  (func $f0
    (param $fPtr i32) (param $a i32) (param $b i32) (result i32)
    (call_indirect (type $func_t)
             (local.get $a) (local.get $b) (local.get $fPtr)))
  (func $add (type $func_t) (i32.add (local.get 0) (local.get 1)))
  (func $sub (type $func_t) (i32.sub (local.get 0) (local.get 1)))
  (func $mul (type $func_t) (i32.mul (local.get 0) (local.get 1)))
  (func $div (type $func_t) (i32.div_s (local.get 0) (local.get 1)))
  (func $getfunc (param $idx i32) (result i32)
    (local $ret i32)
    i32.const 1
    local.set $ret
    block $B0
      (i32.and (local.get $idx) (i32.const 1))
      br_if $B0 ;; if(idx % 2) return [1]
      (local.set $ret (i32.const 2))
      (i32.rem_s (local.get $idx) (i32.const 3))
      br_if $B0 ;; if(idx % 3) return [2]
      i32.const 3
      i32.const 4
      (i32.rem_s (local.get $idx) (i32.const 5))
      select    ;; if(idx % 5) return [3] else return [4]
      local.set $ret
    end
    local.get $ret)
  (elem (i32.const 1) $add $sub $mul $div))
```

As explained in Section 2.1.3, the main purpose of the Wasm table is to exchange reference information. Wasm does not allow a raw pointer to an instruction address, i.e., a code pointer. Consequently, Wasm needs to represent a function pointer with a different reference model. Wasm table stores a reference to a function which Wasm program can use to perform a dynamic or indirect call.

Typically, the compiler populates the table as required. The compiler detects dynamic function call using a function pointer as requiring a table. A different case, such as using function pointer as a return value, also requires a table. The

compiler populates the table statically and uses the table index as the reference from the instruction. Currently, Wasm does not allow table manipulation from within a Wasm program. Listing 2.17 and Listing 2.18 show the use of Wasm table for dynamic dispatch and function pointer return.

2.2.6 Import and Export

Import and Export declaration is used to expose symbols from a module to the host environment. An import is an element that a module requires before it can run. The host code needs to supply all imported aspects before it can instantiate a module. On the other hand, export exposes elements inside a module to be accessible from the external domain. No necessary action by the host code before the module can run. The host code uses export as the available API of the module.

A module may import or export functions, memory, table, or global variables. Wasm provides two different flavors to export or import module elements. A Wasm module may declare export or import in a separate instruction and use an identifier to refer to the element. Another way is declaring export and import inside the element declaration.

The import declaration has two string arguments. The first string is the "module-name" part, while the second one is the actual identifier. Wasm groups the identifier by its "module-name." Its primary purpose is to allow a logical structure of import components. For example, a module import functions from two different shared libraries. The compiler may group the function into two "module-name," and the host code can link the shared libraries accordingly.

Listing 2.19. Some import and export variations in Wasm

```
(module
  ;; Import with memory or table declaration
  (import "env" "memory" (memory 1 2))
  (import "env" "table" (table 10 20 funcref))

  ;; Exporting memory (Note: memory definition can only appear
  ;; once in Wasm module)
  (memory $mem 1 2)
  (export "memory" (memory $mem))

  ;; Import global
  (import "env" "x" (global $x i32)) ;; Global declaration inside
  (global $y (import "env" "y") i32) ;; Import inside

  ;; Export global using separate export declaration
  (export "v1" (global $v1))
  (global $v1 i32 (i32.const 5))

  ;; Export global inline
  (global (export "v2") i32 (i32.const 0))

  ;; Import function
  (import "env" "getfloat" (func $getfloat (param f32)))
  (func $getint (import "env" "getint") (param i32))

  ;; Export function using separate declaration
  (export "addone" (func $addone))
  (func $addone (param i32) (result i32)
        (i32.add (local.get 0) (i32.const 1)))

  ;; Export function inline
  (func $subone (export "subone") (param i32) (result i32)
        (i32.sub (local.get 0) (i32.const 1)))
)
```

2.3 Compiling to WebAssembly

2.3.1 LLVM

LLVM compiler toolchain supports compilation to Wasm. Programmers can specify Wasm as the target architecture, and LLVM generates the Wasm binary file. We can use the LLVM compiler frontend, Clang, to process the entire compilation pipeline from C and C++ source to the Wasm binary. The pipeline is similar to cross-compilation to a different target architecture.

However, LLVM does not provide the Wasm system environment required to perform full compilation. The system environment, or sysroot, contains required headers and their respective libraries [43]. Without sysroot, LLVM cannot compile a program that uses standard libraries. Hence, it restricts the usability of the compiler itself.

Developers can download Wasm sysroot separately. The WASI project pro-

vides the sysroot that we can use to compile a program that requires system libraries to Wasm. Developers can specify the sysroot to the compiler during compilation. With this, the compiler can compile and link programs that use standard libraries.

Another limitation of using LLVM directly is writing the JS binding manually. Loading a Wasm module to a web page requires a JS code to load and launch the module explicitly. Wasm program cannot run independently on a web page without using a JS script. The complexity increases when the Wasm requires to emulate standard library functions such as input and output. Therefore, LLVM produced Wasm module is not directly usable in every case.

2.3.2 Emscripten

Emscripten was initially an LLVM-to-JavaScript compiler. Its primary purpose was to compile C and C++ program to JavaScript. In 2015, Emscripten introduced support compilation to Wasm. This early introduction is to expedite the adoption process of the new standard. As of version 1.39.14, Emscripten provides solid support in compiling C and C++ programs to Wasm. Emscipten handles end-to-end compilation from the source to a runnable web page. Emscripten can also link to the WASI, allowing the use of standard library functions.

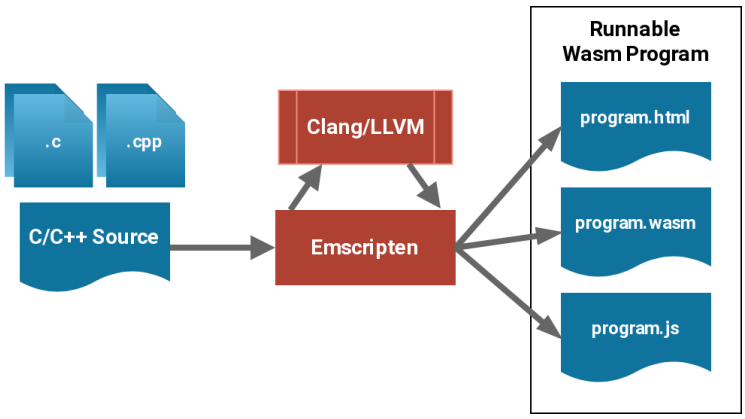

Figure 2.2. High-level workflow of Emscripten

Emscripten is bundled with EMSDK. It uses Node.js and Python scripts as its backend. The default installation also includes its own LLVM compilers, eliminating local dependency. Developers can download the toolchain by checking out the EMSDK git repository and run the configuration script.

Enscripten processes the input source by passing it to the LLVM compiler. LLVM compiles and generates Wasm binaries from the input. Emscripten then generates necessary JavaScript binding to glue the Wasm module and web environment. Emscripten also performs linking with the required libraries. Emscripten provides the WASI library, which allows the program to use most of the standard library functions. Emscripten automatically links the required library when

compiling the source program.

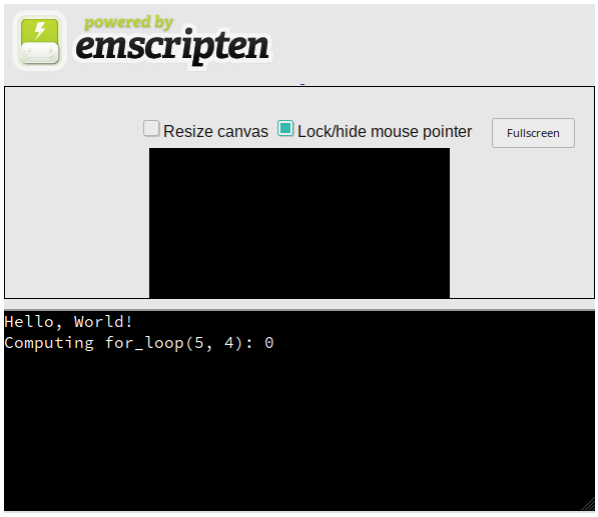

Figure 2.3. The default page generated by Emscripten. It consists of a 'console' which display the program standard output.

The JS script generated by Emscripten is a Wasm launcher. It encapsulates the launching sequence from the module binary fetch to module instantiation. Emscripten also generates a JS shell function that links to the Wasm function. It allows user codes to utilize Wasm function from the JS program directly. It covers the use case of utilizing C and C++ codes as an external library for a JS program.

2.4 Execution Environment

2.4.1 Embedding WebAssembly to JavaScript

Besides Wasm specification, W3C also standardizes the WebAssembly JavaScript Interface. It specifies the JS API, which connects the JS and the Wasm environment. The API provides the infrastructure to load, compile, and execute Wasm modules from JS programs. Wasm requires the host environment implementation to implement this specification.

The specification defines a WebAssembly namespace in the JS environment. This namespace consists of two main functions: compile and instantiate. The compile function compiles Wasm binaries that are stored in a BufferSource object. The compile function produces a Module object, which can then be instantiated. The instantiate function itself comes with two flavors. JS code can provide either the compiled Module object or the Wasm binaries. The latter simplifies the compilation process without having a separate compile-instantiate pipeline. The compile and instantiate function use the JS Promise framework. It allows asynchronous operation and conforms to the asynchronous nature of JS language.

The Module object provides interfaces to import and export information. JS code can use this information to reflect the content of a Wasm module. Apart

Listing 2.20. Compiling and Instantiating Wasm from JS

```
fetch('thesis.wasm')
    .then(response => response.arrayBuffer())
    .then(bytes => WebAssembly.compile(bytes))
    .then(module => WebAssembly.instantiate(module))
    .then((result) => { /* do something with the instance*/ });
```

from the Module object, the WebAssembly namespace also provides several other object types. The Instance object, which is produced by the instantiate function, provides a list of exported functions. A JS program can use this API to invoke Wasm function, similar to calling a regular JS function.

A Memory object represents a memory in Wasm environment. It allows a JS program to initialize and supply the memory to a Wasm module. A JS program can also manipulate the memory buffer without using a Wasm function. It is necessary in case of transferring extensive data between the JS and the Wasm realm. Table and Global objects also serve the same purpose for the Wasm table and global variable.

Besides JS Interface, W3C also defines an additional specification for hosting Wasm in the web ecosystem. This separate specification extends the JS Interface specification for the web environments. The separation lets a non-web environment, such as Node.JS, to implement the JS Interface without the Web API.

The Web API provides standardization for streaming the Wasm module through the network. Streaming compilation enables Wasm compilation without waiting for a complete module download. It improves the efficiency of the pipelining between data transfer and program compilation. The additional Web API requirement is the serializability of the Module object. It enables the Module object to be stored or serialized to local storage. The specification mandates the host to try reusing the compiled code by caching. The module caching avoids recompilation and improving the entire efficiency of the Wasm ecosystem.

2.4.2 Accessing Wasm Exports

The Wasm instance object provides an interface to the export list. The export list allows a JS program to access the Wasm module exported members. The export element is directly callable by using its export names. For example, Listing 2.19 shows an exported function addone and subone. Listing 2.21 below shows the JS code that invokes those exported functions.

Listing 2.21. Invoking exported functions

```
var thesisInstance; // consists of a Wasm instance object
var addOneRes = thesisInstance.exports.addone(5);
var subOneRes = thesisInstance.exports.subone(addOneRes);
```

Exported memory, tables, and globals also appear in the export list. Their respective JS objects reference these exports and provide access from JS code. The JS Interface provides Memory, Table, and Global object to enable accessing these Wasm elements from the JS program.

The Memory object provides access to the raw memory buffer. JS code can manipulate Wasm memory, similar to use an array buffer. A JS program can access the buffer by instantiating the Uint8Array object and supplying the array buffer reference. A JS program can also use other unit sizes, such as Uint16 and Uint32. However, since the memory is addressable in a one-byte unit, using larger than the one-byte unit can be challenging to convey. Additionally, the JS code can also grow the Wasm memory.

Listing 2.22. Wasm code with memory export

```
(module
  (memory $mem 16 32)
  (func $getandmul (param i32) (result i32)
    local.get 0
    i32.load offset=0
    i32.const 3
    i32.mul)
  (export "mem" (memory $mem))
  (export "getandmul" (func 0))
```

Listing 2.23. Accessing exported memory from Wasm in Listing 2.22

```
var thesisInstance; // consists of a Wasm instance object
var memoryBuffer = new Uint8Array(thesisInstance.exports.mem.buffer);
// Wasm memory always zero-initialized. Hence: buffer[0] * 3 = 0
var zero = thesisInstance.exports.getandmul(0);
// Modify the content of Wasm memory
memoryBuffer[0] = 25;
// No longer zero. buffer[0] * 3 = 75
var val = thesisInstance.exports.getandmul(0);
```

JS code can manipulate Wasm Table through the `WebAssembly.Table` object. The object provides an accessor and mutator method to get and set table elements. The accessor function accepts a table index and returns the function reference pointed by the table element. The mutator function, on the other hand, accepts the function reference to be set as the table element. The mutator may only accept an exported WebAssembly function. Otherwise, it will raise an error.

WebAssembly.Global object provides access to Wasm global variables. The object represents a global variable that a JS program can access. However, only immutable globals can be exported from Wasm. If a JS code needs to access to modifiable global values, the Wasm module needs to declare it through import instead of export.

2.4.3 Supplying Import to Wasm Module

Instantiation of Wasm module requires the JS code to supply imported elements. The import object is a regular JSON object organized with module and import

name hierarchy. The key on the first-level of the import object is the module name, as declared in the Wasm module. The second-level key is the import name. The second-level contains the respective Wasm object that is supplied for the import.

The supplied Wasm object is the same with objects that are used in accessing exports. However, the JS code is responsible for instantiating and providing that object. In Wasm export, the JS engine handles the object instantiation.

The supplied import objects remain accessible and modifiable from the JS code. Any change made to the object content from the JS code reflects automatically in the Wasm realm. Moreover, imported global variables are mutable, unlike the immutable export counterpart. In a sense, import and export only differ by its requirement in the module instantiation.

Listing 2.24. Wasm code with imports

```
(module
  (import "thesis" "mem" (memory $mem 16 32))
  (import "thesis" "glob" (global $global1 (mut i32)))
  (import "thesis" "tbl" (table 1 funcref))
  (type $op (func (param i32 i32)(result i32)))
  (func $add (type $op) (i32.add (local.get 0)(local.get 1)))
  (func $sub (type $op) (i32.sub (local.get 0)(local.get 1)))
  (func $compute (param i32) (result i32)
    local.get 0 ;; Memory index 0
    i32.load offset=0
    global.get $global1
    i32.const 0 ;; Table index 0
    call_indirect (type $op))
  (export "compute" (func $compute))
  (export "add" (func $add))
  (export "sub" (func $sub)))
```

Listing 2.25. Supplying import for Wasm module in Listing 2.24

```
var importObjects;
WebAssembly.compileStreaming(fetch("imports.wasm"))
  .then((x) => {
    importObjects = {
      thesis:{
        mem: new WebAssembly.Memory({initial: 16, maximum: 32}),
        glob: new WebAssembly.Global({value:'i32', mutable: true}),
        tbl: new WebAssembly.Table({initial: 1, element: "anyfunc"})
    }};
    return WebAssembly.instantiate(x, importObjects)
  }).then((x) => {
    // Set the table entry
    importObjects.thesis.tbl.set(0, x.exports.add);
    // Set global value
    importObjects.thesis.glob.value = 25;
    // Set memory content
    var memoryBuffer =
            new Uint8Array(importObjects.thesis.mem.buffer);
    memoryBuffer[0] = 10;
    // Prints 'Result: 35'
    console.log("Result: " + x.exports.compute(0));
  });
```

Also, a Wasm module can also require a function import. JS code can supply

a function, including a JS function, to be imported. The JS engine automatic marshal the function argument and return value between Wasm and JS realm. Wasm program can call the imported function in the same way with the regular Wasm function. In addition, the imported function can also be re-exported by the module. This re-exported function can be used as a table element from the JS program.

The primary use case for function import is to provide callback function from Wasm module to its user. Many programs require a callback function in their design. In Wasm, however, a callback function can only be supplied through the explicitly declared import function. At the moment, Wasm does not provide a mechanism to dynamically extend an existing Wasm module. Without the explicit import declaration, JS program is unable to provide the callback function to the Wasm module.

Listing 2.26. Extending Listing 2.24 with functions import

```
(module
  ;; Similar to previous Wasm
  (import "thesis" "ext_mul" (func $ext_mul (type $op)))
  (export "mul" (func $ext_mul)))
```

Listing 2.27. Supplying function import for Listing 2.26

```
var importObjects;
WebAssembly.compileStreaming(fetch("imports.wasm"))
  .then((x) => {
    importObjects = {
      thesis:{
        // Same with previous implementation
        ext_mul:  function(v1, v2) {
                    console.log("Imported function call");
                    return v1 * v2;
                  }
    }};
    return WebAssembly.instantiate(x, importObjects)
  }).then((x) => {
    // Same with previous implementation
    // Prints 'Imported function call'
    //        'Result: 250'
    console.log("Result: " + x.exports.compute(0));
  });
```

2.5 Host Environment Implementation

2.5.1 Mozilla SpiderMonkey

SpiderMonkey development began in late 1990 during the rise of the world wide web. It was the original implementation of the JS engine that is used in Netscape Navigator [17]. The engine is open-sourced in 1998, along with the creation of the Mozilla Project by several Netscape members. It is the JS engine used in Mozilla products, mainly Firefox browser. Throughout two decades of

development, SpiderMonkey has gone through long and continuous evolution.

SpiderMonkey is a JS implementation in C that can be embedded in other programs [18]. The web browser is one of several programs that integrate the JS engine for web client scripting. SpiderMonkey provides a C-based API interface to interact with the JS engine. The embedder invokes the engine, and the engine prepares the necessary runtime for the JS environment. Every script will run on a unique Context, which defines the internal stack size for the script execution. The embedder instantiates and manages the Context manually and explicitly. It is a low-level object that is considered as a resource. Therefore, the embedder needs to ensure to release the Context after it is no longer used.

For higher-level objects, such as JS variables and objects, the JS engine provides an automatic Garbage Collector (GC). The GC tracks the lifetime of every JS object. The JS object lives in a different realm with the embedder object. Although the embedder code, which is in C, can interact with an object created in the JS realm, the embedder code needs to inform the GC when referencing it. It is to ensure that the object is not released while being referenced in the GC code. This different realm creates the requirement for an embedder code to marshal[3] all data when interacting with the JS engine.

When asm.js became popular, Mozilla incorporated an optimizing AOT compiler for the asm.js program [5]. The compiler, which is named OdinMonkey, precompiles the asm.js program to the machine instruction before the execution [34]. It dramatically improves the performance of asm.js applications compared to using the regular JS pipeline. This optimization is benefitting the characteristic of asm.js as a strict subset of JS. Asm.js can be considered as a JS without dynamic types, runtime features, and GC requirements. Ultimately, this optimization preluded the Wasm development as a true intermediate code for the JS engine.

In the Wasm compilation, SpiderMonkey incorporates a tiered compilation [8, 23]. It is a multi-level compilation process which allows a program to pass through a different level of optimization. Program optimization takes time to process, and it may delay the start-up of the program. By incorporating the tiered-compilation, the JS engine can expedite the response time of the program start-up. In the background, the engine recompiles the program with more optimization processes and yields an optimized compiled program. The engine can then hot-swap the optimized program to improve the overall performance of the execution.

Mozilla is planning to incorporate a new code generator called Cranelift in its future release. Cranelift is a new optimizing code generator that is written in Rust. The JS engine compiles the Wasm to Cranelift's intermediate representation (IR). Then, Cranelift generates the target machine instruction based on the IR.

[3]Packaging data to be transmitted across application boundaries. It is typically used in Remote Procedure Call (RPC), interprocess communication, or invoking a component that uses a specific data format. [2]

Cranelift is a part of the ongoing development of Wasmtime, an external runtime for Wasm programs. It is sponsored by the Bytecode Alliance, which the Mozilla Foundation is also a part of.

2.5.2 Chromium V8

The Chromium project was published in 2008 along with the release of Google Chrome [19]. Chromium is the base code for the Google Chrome web browser without proprietary features. Initially, the project started with a multi-process web browser based on the WebKit engine. Along with the first release, Google also published the V8 JavaScript engine, which is used in the entire Chromium ecosystem.

The V8 engine has been incorporated since the very first version of Chromium release. Google claimed V8 outperformed the contender of the JS engine, including JScript from Microsoft, SpiderMonkey from Mozilla, and JavaScriptCore from Apple. This performance gave significant benefits for the web browser in executing web page scripts. The performance benefit became more necessary since the rise of Web 2.0, where more web sites implement AJAX and interactive web client.

V8 addressed three major areas in their early releases [20]. V8 implemented efficient property access. It improved the property access by eliminating dynamic lookup using a dictionary data structure. Many prior JS engines used a dictionary since JS is a dynamically-typed language, and internal members can be introduced at any point. Instead, V8 generated a hidden class that is similar to the inheritance model in an object-oriented language. It enabled the object to be stored similar to a struct in the memory. It allowed the object member to be accessed using an offset lookup.

The next significant improvement in V8 is the Dynamic Machine Code Generation. This capability enabled V8 to compile the JS code directly to the target machine instruction. It equals to Just-in-Time compilation in major virtual machine architecture. V8 does not use intermediate byte codes, which improves the overall performance of the execution. Previously many JS engine implementation interpreted the JS code instead of compiling it. It created a performance hit compared to the JIT-based engine. The final improvement in the V8 was the more efficient garbage collection.

Along with other JS engines, the V8 engine introduced its first experimental support in WebAssembly in March 2016 [48]. This experimental support used existing infrastructures of the V8 engine. In addition to using the existing JIT compiler, V8 added a Wasm decoder to read and validate Wasm modules. This pipeline generates machine instruction from the Wasm program. In 2018, V8 developers replaced the existing JIT compiler with a specialized Wasm compiler called Liftoff.

The Liftoff compiler simplified the machine code generation [25]. It bypasses

the complex pipeline of the previous compiler. The last compiler, TurboFan, is designed to process the JS program, which requires more complex intermediate representation. Wasm, on the other hand, has a simpler architecture and instruction variants compared to the JavaScript. As discussed in previous sections, Wasm itself acts similarly to an intermediate representation. By eliminating the redundant intermediate representation, V8 performs more efficiently when processing Wasm programs.

Apart from its use in the Chromium web browser, V8 is also used in the Node.js environment [15]. Node.js is a JS environment outside of a web browser that can be used for general purposes. Node.js expands the coverage of the JS language in software development. A web application can be developed in a single programming language instead of two different languages for frontend and backend. With the development of Wasm, Node.js benefited the native Wasm support from the V8 engine. It allows a Node.js program to depend on external non-JS library without platform boundaries. Previously, Node.js uses the native extension to integrate external native libraries to the Node.js environment [16].

2.6 Comparison of the Host Environment

This section discusses the relevant comparison between the V8 and SpiderMonkey for the thesis. More details are also discussed in the later chapter about the instrumentation of the JS engine for the fuzzing experiment. This comparison is also relevant only to the time of the writing of this thesis. Since both JS engine is very active projects, the information here may evolve throughout time.

2.6.1 Project Structure and Compilation

The SpiderMonkey JS engine is integrated with the source tree of the Gecko web browser engine. The JS engine resides in the subfolder of the source. Developers need to clone the entire source tree even if they are only interested in the JS engine. The Mozilla developers no longer maintain a separate standalone download for SpiderMonkey sources. Therefore, third-party developers have no choice other than cloning the entire Gecko source tree.

Despite being integrated with the entire Gecko source tree, the SpiderMonkey source is isolated in its subfolder. Although the source is mainly written in C++, the recent development incorporates Rust sources, especially the Cranelift instruction generator. The build script prepares the Rust dependency from a separate third-party folder in the main source tree. Hence, the developers do not need to handle the dependency manually.

On the other hand, the V8 engine is a completely independent and isolated source. The source tree is maintained separately with the rest of the Chromium web browser engine development. Thus, interested developers do not need to pull

the entire Chromium source tree to play with the V8 engine. The V8 engine is also written entirely in C++.

Yet, the V8 engine requires additional dependency to build properly. It requires a custom build tools called *depot_tools*. The *depot_tools* is a collection of custom build tools developed by Google for their projects. It consists of a build generator, along with infrastructures that integrate with their source control system. V8 requires this tool to generate the actual build script.

The SpiderMonkey build uses the Makefile build automation tool, while V8 uses Ninja. Since both projects use a customized build generator, it is impossible to switch the build automation tool. Both projects use a C++ compiler to build the project, and they use the system default compiler. Besides, SpiderMonkey requires the Rust toolchain to compile the Rust dependencies. The SpiderMonkey build script automatically calls the Rust toolchain to build the dependencies.

2.6.2 Embedding the Engine

Both engines compile into a shared library. An external program can embed the engine to utilize the JS engine features by linking to the engine shared library. Such external program is called an embedder. An embedder can access the API exposed by the engines by including its public header file. Through the available API, the embedder can call the necessary functions to utilize the JS engine features.

The embedding workflow is relatively similar in both engines. The embedder needs to initialize the JS engine before accessing the rest of the API. SpiderMonkey has a more straightforward process compared to V8. In V8, the embedder needs to invoke several functions before the engine is fully initialized.

Both engines also require the embedder to instantiate a Context object. This object represents a JavaScript execution in the engine. A context encapsulates every global value declared in the JS script. A script from a different context

Table 2.2. Comparison of the Host Environment Projects

Category	**SpiderMonkey**	**V8**
Source Control	Mercurial and automatic mirror in Git	Git
Project Structure	Integrated with the web engine parent project (Gecko)	Independent and self-contained
Build Generator	Self-contained shell and Python script with custom build configuration file in `build.moz` file	Third-party build configuration tool (*depot_tools*) with custom build configuration file in `BUILD.gn` file
Build Tool	Makefile	Ninja
Programming Language	C++ with Rust Dependency	C++

Listing 2.28. Embedding the SpiderMonkey

```
#include "jsapi.h"
#include "js/Initialization.h"

// The class of the global object
static JSClass global_class = {
    "global",
    JSCLASS_GLOBAL_FLAGS,
    &JS::DefaultGlobalClassOps
};

int main(int argc, const char *argv[])
{
  int ret = 0;
  JS_Init();
  JSContext *cx = JS_NewContext(8L * 1024 * 1024);
  if (!cx)
      return 1; // Failed instantiating context
  if (!JS::InitSelfHostedCode(cx))
      return 1; // Failed initializing selfhosted code

  { // Scope for various stack objects
    JS::RealmOptions options;
    JS::RootedObject global(cx, JS_NewGlobalObject(cx,
                                &global_class, nullptr,
                                JS::FireOnNewGlobalHook, options));
    if (!global)
        return 1;
    JS::RootedValue rval(cx);
    { // Scope for JSAutoRealm
      JSAutoRealm ac(cx, global);
      JS::InitRealmStandardClasses(cx);
      // JS script operations starts here ...
    }
  }
  JS_DestroyContext(cx);
  JS_ShutDown();
  return ret;
}
```

cannot access value from a script in another context. It effectively creates logical isolation between scripts. V8 engine, however, added additional layer above context called Isolate.

Isolate is a layer of isolation similar to a process in an operating system. An Isolate has its own memory allocator, and it may not share resources with another Isolate. An Isolate is considered thread-safe isolation. Thus, it can run in a different thread. It is particularly vital in a multi-threading ecosystem, such as web engine rendering.

After all necessary initialization and context instantiation, the embedder can invoke the desired JS script. The embedder can supply the script, as well as communicating with the JS realm by passing and receiving data. When the embedder completes the execution, it needs to clean up the JS engine by releasing all resources properly. The API provides all necessary clean-up functions for this purpose.

Listing 2.29. Embedding the V8

```
#include "libplatform/libplatform.h"
#include "v8.h"

int main(int argc, char const* argv[]) {
  int ret = 0;

  // Initialize V8.
  v8::V8::InitializeICUDefaultLocation(argv[0]);
  v8::V8::InitializeExternalStartupData(argv[0]);
  std::unique_ptr<v8::Platform> platform =
                              v8::platform::NewDefaultPlatform();
  v8::V8::InitializePlatform(platform.get());
  v8::V8::Initialize();

  // Create a new Isolate and make it the current one.
  v8::Isolate::CreateParams create_params;
  create_params.array_buffer_allocator =
      v8::ArrayBuffer::Allocator::NewDefaultAllocator();
  v8::Isolate* isolate = v8::Isolate::New(create_params);

  { // Scope for Isolate
    v8::Isolate::Scope isolate_scope(isolate);
    // Create a stack-allocated handle scope.
    v8::HandleScope handle_scope(isolate);
    // Create a new context.
    v8::Local<v8::Context> context = v8::Context::New(isolate);

    { // Scope for context
      v8::Context::Scope context_scope(context);
      // JS script operations starts here ...
    }
  }
  // Dispose the isolate and tear down V8.
  isolate->Dispose();
  v8::V8::Dispose();
  v8::V8::ShutdownPlatform();
  delete create_params.array_buffer_allocator;
  return 0;
}
```

Apart from the embedder code implementation, the V8 embedder is also required to provide the `snapshot_blob.bin` file. `snapshot_blob.bin` is an additional file generated by the V8 build that is required by the V8 engine. By default, the V8 engine searches the file in the same directory as the embedder executable. The embedder can supply the file by copying the `snapshot_blob.bin` file. The embedder can also put a symlink to the file in the same directory with the executable. The embedder can also modify the argument that is passed to the `InitializeExternalStartupData` function. The V8 engine refuses to work if it does not find the `snapshot_blob.bin` file.

2.6.3 Internal Data Structure

The JavaScript and C++ programming language have different data representations. The fundamental difference in both languages is the type-safeness. C++ is

a strongly-typed language, meaning that every variable has a known type during the compile time [44]. JavaScript, on the other hand, is a dynamically-typed language [14]. The value type stored in a variable may be unknown until the execution evaluates the variable.

Another difference is the primitive types of the two languages, in which both languages have a different concept of primitive types. C++ types are naturally closer to low-level binary types. In C++, primitive values are backed by simple integral values stored in memory. The primitive types vary by their storage size. For example, an `int` value is an integral type that is stored precisely 4 bytes in an x86 machine[4].

JavaScript primitive types are more high-level and abstract. JavaScript only has a single universal Number type, which is backed by 64-bit floating-point value [14]. String value is also considered a primitive type in JavaScript, unlike in C++. This differing typing discipline requires a bridge between the C++ and the JS representation. The dynamic nature of the JavaScript language implies that every value is polymorphic. Therefore, a single root representation of value needs to exist. Both SpiderMonkey and V8 defines a root Value object to represent this dynamic value. SpiderMonkey defines it as the `JS::Value` class, while V8 defines it as `v8::Value`.

These value classes provide the necessary functions to interchange between JS and C++ primitives. Those functions include accessor and mutator from and to C++ primitives. The main difference between the two implementations is the way to store the value in the object. A SpiderMonkey value object is mutable. The value stored inside the value object can be modified using the available mutator function. V8 does not provide API to manipulate a primitive value after it has been instantiated.

This difference in value object mutability is due to its internal implementation of storing the value in the memory. V8 stores all value directly in its internal heap. V8 API provides interfaces to instantiate the `v8::Value` object based on the desired type. The object must be stored inside a specialized container called `Local`. `Local` acts as a reference container for heap-allocated objects. The GC tracks the liveness of an object through the `Local` object. It is unsafe to track a raw pointer to a heap-allocated object since the GC can move the object at any moment. The reference container acts as a safe reference in which the GC can inform the new location of an object that has been moved.

In contrast, SpiderMonkey does not always store every value in the heap. SpiderMonkey allows a value to live outside the heap, i.e., in the stack. SpiderMonkey differentiates the reference using `Root` and `Handle`. In principle, it is similar to the GC management in V8. However, SpiderMonkey specialized the

[4] C++ does not specify the exact storage size for the primitive data types. Therefore, primitive types in a different architecture and system may have different storage size. For example, `long` is stored as 4 bytes in a 64-bit x86-64 Windows system, while it is 8 bytes in Linux/POSIX family. See **6.8.1 Fundamental types [basic.fundamental]**[29, 4]

case for a `RootedValue`, which does not involve any heap allocation.

2.6.4 Internal Wasm API

Both implementations encapsulate the Wasm infrastructure in a separate namespace. It is relatively easy to isolate Wasm-related codes as most of the codes are isolated and decoupled from the rest of the JS engine. The main components are the Wasm parser, validator, compiler, code representation, and element representations. The element representation includes the Wasm memory, table, and global variable objects.

The compile function accepts a binary Wasm module. It handles all compilation steps in a single call, from parsing to the machine instruction generation. The returned instruction bytes is encapsulated in a module object, which includes the entry point accessible from the JS realm. However, only exported functions that are included in the entry point lists. The unexported function is invisible from the external observer. Nevertheless, the Wasm internal keep tracks of every declared function in the module.

The module needs to be instantiated before the JS engine can execute it. The workflow equals to the Wasm JS API that is discussed in the previous section. The instantiation function requires the import object that is required by the Wasm module. If the caller does not supply complete import elements, the instantiation function fails. Both engines have a different mechanism in supplying the import elements. V8 uses a JS object to supply the import, analogous to constructing the object from a JS script. While SpiderMonkey provides a specialized C structure named `js::wasm::ImportValues` to encapsulate the import elements.

The instantiation produces an instance object. At this point, the Wasm module is prepared and ready to be executed. Wasm function can be invoked from the entry point provided by the instance object. SpiderMonkey has more direct access to the invocation function. SpiderMonkey's `wasm::Instance` object provides a `callExport` function, which can be called by supplying the exported function index and arguments. V8 uses a JS function handle to invoke the Wasm function. Hence, every Wasm function is invoked through the regular JS function invocation pipeline.

Both engines accept a vector of Value object as an argument. For SpiderMonkey, however, the call argument vector also provides a slot to store the return value of the function. This is different from V8, where the return value of the called function is returned by the invoker function.

Chapter 3

Software Testing

> *"Human errors can cause a defect or fault to be introduced at any stage within the software development life cycle and, depending upon the consequences of the mistake, the results can be trivial or catastrophic. Rigorous testing is necessary during development and maintenance to identify defects, in order to reduce failures in the operational environment and increase the quality of the operational system."*
>
> *"Foundation of Software Testing: ISTBQ Certification"*
> *Dorothy Graham, Erik van Veenendaal, Isabel Evans, Rex Black* [21]

This chapter presents the foundational knowledge related to software testing. This knowledge is essential in building the argument and basis of the experiment presented in this thesis. This chapter summarizes the essential key points around software testing (Section 3.1), fuzz testing (Section 3.2), and differential testing (Section 3.3). This chapter concludes by presenting the analysis of the experiment background by applying the knowledge provided in this chapter.

3.1 Testing in General

3.1.1 Overview

We can describe testing as a process to evaluate expectations. The process begins by defining the expectation of the subject. Then, the expectation is evaluated to conclude whether the subject meets the expectation or not. From the conclusion, we can decide on the next action to take against the subject.

We can take a simple example of purchasing a car. The car dealer provides the potential buyer to test drive the car they sell. The buyers have certain expectations of the car, which affects their consideration to make the purchase. The buyers then test the car by driving it directly and decide whether the car meets their expectations. If the car does not meet the expectations of the buyers, the buyer can then reiterate the process of searching for a new car model and

perform the test drive again.

Testing is an integral part of human activities that involve expectations. Software engineering deals greatly with expectations. Even, software development lifecycle always starts with defining the expectation, in the form of requirements [41]. Intuitively, the lifecycle must also involve testing to validate if the expectation has been satisfied. All software development process methodology, from the waterfall model to agile, incorporates testing in the process. With this fact, it is irrefutable that testing is an inseparable part of software development.

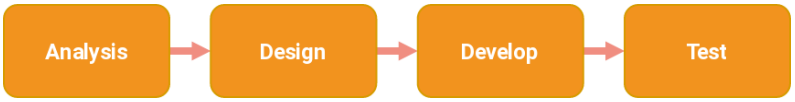

Figure 3.1. A simple software development lifecycle [41]

Dorothy Graham et al. describe that testing is performed to evaluate the requirement and specification of a product is satisfied [21]. From the evaluation, we can measure the quality of the product and perform improvement if necessary. Through testing, we can also identify a missing requirement, and more importantly, a defect. It is crucial to prevent a defect before a product is released so that it can be addressed before the product deployment.

Testing also requires proper planning to achieve an accurate result. The planning enables a proper understanding of the context of the testing, and define the strategy of the testing [21]. The strategy itself covers the technique, scope, and tools. The test plan also defines the exit criteria, which determine the success of the test activity.

From the test activity, we can obtain the result that we can use for analysis. The result enables us to develop the defect report for fixing and mitigation. We can also obtain metrics and statistics of the software quality, which can indicate the quality of the software and the development process. This information becomes feedback about the development cycle to improve the overall software development process [41].

3.1.2 Types of Testing

Software testing can be classified based on its scope and target [21]. The scope classification adheres to the V-model in the software engineering practices, which illustrates the relationship between software development phases and the respective testing [30]. Each level of design has specific testing to validate that the design is satisfied by the implementation. Figure 3.2 shows the illustration of the V-model.

Component testing, or commonly known as unit testing, is typically conducted at the lowest level of implementation. The test is targetted to the smallest unit of the program, typically a function. A unit test validates the implementation of the

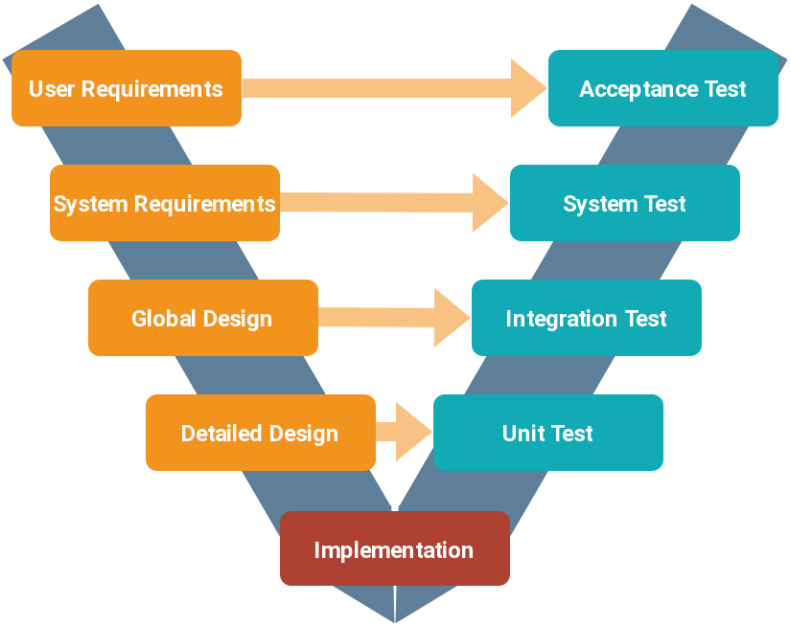

Figure 3.2. V-model according to ISTQB Standard [21]

detailed design to work according to its specification. For example, it validates a function is producing a correct output by the given input.

The integration test validates the interconnectivity between component units. It ensures every component works correctly together, and the integration is implemented correctly. A system test is conducted after the integration test. It validates the system requirement and checks the entire system implementation. Finally, the high-level user requirement is validated through the user acceptance test. It ensures the high-level use case is implemented correctly according to the user demand. After completing the user acceptance test, the software can be considered fit for purpose and ready for deployment.

Dorothy Graham et al. also classifies the testing target in several aspects [21]. Functional testing focuses on system functionality based on the requirements and business processes, in which defines the system specification. Typically, the test cases are derived from the user processes and high-level logical operation.

However, not every system functionality is documented as use cases. A system also consists of non-functional attributes, which are carried by the software itself. These attributes, such as performance and reliability, also affect the usability of the system. Therefore, these attributes are also tested through non-functional testing.

We can consider both functional and non-functional testing as black-box testing. In black-box testing, we do not consider the details of the implementation. We only focus on the behavior of the System Under Test (SUT) from any given test cases and expect that the SUT produces valid and accurate behavior. However, we are more interested to consider on how-it-works rather than the final result only. For this purpose, we use the white-box testing technique.

The white-box testing considers the implementation of the SUT. It evaluates the architecture of software implementation. The main purpose of white-box testing is to cover the entire implementation of a system. It includes every control flow, every exit criteria, and every part of the program. It prevents the implementation from being unchecked via testing. Also, it prevents software defect that may not be visible from regular black-box testing.

Finally, the confirmation and regression testing target the changes in a system. It is typically conducted in highly active system development. A new code introduced to a system can affect other parts of the system. Therefore, it is crucial to test not only the newly introduced code but also the entire system itself. Confirmation testing is conducted by reevaluating the existing test cases to validate if the outcome is unaffected by the new component.

3.1.3 Automated Testing

Software testing is a continuous and repetitive process. Relying on human solely to perform this task can be daunting and inefficient. The automated testing tool provides the solution by allowing the engineer to design test specifications and let the tool to execute the test specification against the SUT. An automated testing system can execute tests more accurately compared to humans. It is important in a test-driven development, which typically involves a unit-testing for every component units in the system [21].

Various tools are available to support automated testing in software development [30]. Typically the tool is tied to a specific development environment and programming language. For example, Google Test is a testing framework which is targetting the C and C++ software development. Similar unit testing frameworks for other programming languages are also available, for example, JUnit for Java and Mocha for JavaScript. The unit testing framework can be integrated into the software build script. It enables the test to be performed automatically during the software build through Continuous Integration (CI) tools.

However, we must also consider the side effect of the automated testing tools on the software [21]. It is apparent, especially for testing non-functional requirements such as system performance. The automated testing tools may present additional overhead, which we need to take into account when interpreting the result. Also, a code that is specifically instrumented for the testing purpose may have a different behavior compared to the original behavior. This different behavior may impact the outcome of the test and yields a diverging conclusion that may not present in the original program path.

3.1.4 Test Cases

Test cases define the elements to be evaluated in the testing process. It describes the precondition of the SUT, the input, and the expected output and postcondition.

We must develop test cases along with the system design process since the test case itself must reflect the system specification and requirement.

A single test case may have multiple input cases. The input case adheres to the input specification defined in the test case, which defines the boundaries and scope of the input. The input case can be generated through this specification, creating multiple input cases for a single test case specification.

The output of the test case is also an essential element in the testing activity. It provides us an insight of system behavior against specific input. An input may have multiple correct results, however, it may not be the most optimum one. By evaluating the result, we can also improve the quality of the system in order to achieve the most optimal result.

A test case is a vital part of software testing. Paul Jorgensen noted in his book that it is as valuable as the source code itself [30]. Hence, it is important to develop test cases thoroughly, as well as keeping it maintained and checked throughout the software development processes.

3.2 Fuzz Testing

3.2.1 Overview

Fuzz testing, or also known as fuzzing, is a form of testing a system by supplying irregular data with the intention to fail and crash the system [46]. It is commonly used to identify corner cases that typically affect system reliability and security.

Fuzzing is considered black-box testing. In a typical fuzzing process, the system is put under stress by receiving unexpected input. Fuzzing is also considered as negative testing. Because, in fuzzing, we are not interested in whether the system is well-behaved. Instead, we are looking for the case where the system behaves unintentionally.

This unexpected behavior is a symptom of a fault within an implemented system. Fuzzing aims to trigger this unexpected behavior to arise, which a typical usual test case and regular input fail to detect. Fuzzing is also considered as a low-cost system evaluation because it does not involve a thorough system analysis to evaluate a system.

Fuzzing can simply utilize a random test case generator to generate the input test case. The fuzzer performs the test automatically without human intervention by repeating the same process all over again until a system failure is detected. Therefore, it is typically common to see system evaluation that tests a system under a fuzzer for an amount of time.

3.2.2 Type of Fuzzer

Ari Takanen et al. categorizes fuzzer based on two criteria [46]. The first criterion is the injection vector, which differentiates fuzzer by its entry point to the system. Every system has different interfaces that yield different fuzzer types to be used. Since fuzz testing is black-box testing, we need to adjust our fuzzer type according to the black-box input.

The other criterion is the complexity of the test case. An appropriately crafted fuzz test case can target a specific part of the system. It is important with the previous injection vector criteria because the specific internal part of the system is not directly exposed to the external domain. We need to penetrate several layers before our test can reach the target vector we aim. A simple random input fuzzer is not going to work well with this case.

Let us take an example of an image processing library that accepts a binary image file. A random fuzzer can generate a random binary image file and supply it to the library interface. Typically, the library interface checks for the basic binary structure before proceeding to the image processing steps. For example, the library may check a magic value at the beginning of the file. A simple random fuzzer may never pass beyond this point as the library automatically rejects the input file altogether. This type of fuzzer may not be efficient enough to expose the hidden defect inside the system. Therefore, a crafted fuzzer can generate a more valid structure that can be accepted by several layers of the system and reach the target component we desired.

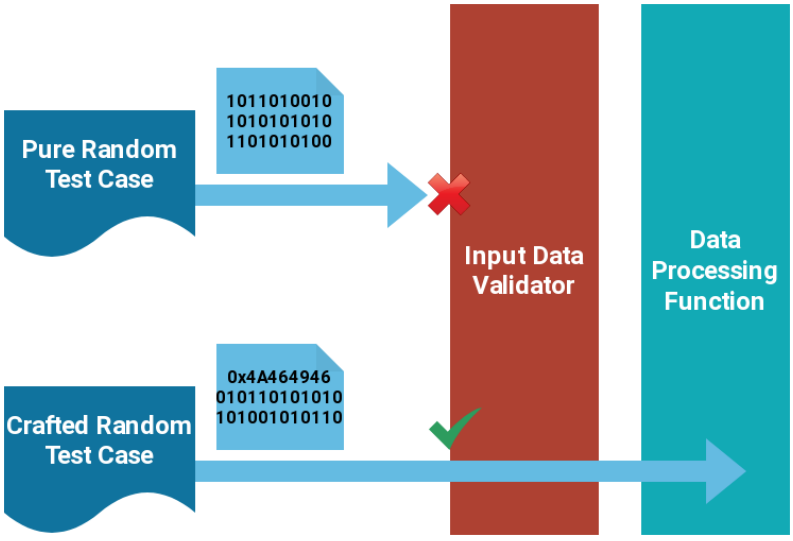

Figure 3.3. Illustration for crafting test case to target a specific internal functionality

A fuzzer can also craft its test case dynamically. This type of fuzzer learns the behavior of the system during the testing sequence, and craft the subsequent input based on this information. A more sophisticated fuzzer also involves a

full protocol of a system to target a system that has a complex internal state. This type of fuzzer is used in testing a protocol-based system, such as a network protocol stack.

3.2.3 Fuzzing Process

Fuzzing process is similar to a regular testing procedure. The SUT receives the input test case, in which the observer collects the information of the SUT. The test result can be either valid, error, anomalous, or system failure. The result must be completely recorded for further investigation.

The examination of the result is often be done manually by analyzing the detected fail cases. The fail cases are reproduced by supplying the exact test case input to the SUT. Generally, this process involves a debugging and dynamic analysis of the program. These fail cases form a corpus of test cases that can be reused for future testing to validate the defect is no longer present in the system.

Several fuzzing frameworks are available on the market. It can be used to incorporate fuzz testing into a software development project. However, Ari Takanen et al. reported that the on-the-market framework does not usually provide a ready-to-use input for every system [46]. Significant development time is also required in order to implement a fuzz testing that covers the entire SUT, since it requires the knowledge of the SUT itself in order to develop a test case that can possibly trigger corner cases.

3.3 Differential Testing

3.3.1 Purpose of Differential Testing

Differential testing is testing against multiple comparable systems to find the difference between implementation. According to William McKeeman, differential testing originates in search of an oracle in testing [35]. An oracle, or a gold standard, is used to evaluate a test result.

For a simple unit testing, a test result can be trivially evaluated. However, for a more complex integration testing, the complexity of the test result rises and introduces difficulty in interpreting the test result. It is even more impractical when random test cases are involved. It is unlikely to derive high-level reasoning to be applied to a random test case.

Differential testing alleviates the problems of test result evaluation [35]. It involves multiple systems which have the same functionality, and compare the result and behavior from those systems by providing them the same input. A small set of test cases may not adequate enough to produce a visible difference. Nevertheless, through a large number of automatically generated test cases, differences can be exposed, which can be a sign of a software defect in any of the

SUT involved.

3.3.2 Real-Case Examples

Differential testing is possible for a system that has multiple different implementations. A notable example is testing multiple implementations of compilers [35]. C and C++ language enjoy the public adoption of its standard, where several different implementations are available, such as GCC and LLVM. Many differential fuzzing experiments on C-family compilers, and proposes different methods of test case generations. A compiler is a mission-critical tool that requires thorough testing and examination. Through differential testing, compiler implementations can benefit the check-and-balance mechanism to identify possible bugs in one another.

The Csmith differential testing tool for C compilers proposes a heuristic voting technique to identify possible bugs [49]. The tool generates a test case, which is a randomly crafted C source code, and send the source to multiple compilers to produce executable programs. The tool then executes the resulting executables and compares their output. From the collected result, the tool analyzes and identifies a possible bug by detecting differences in the output. The compiler that produces different output from the majority of the other compilers is considered to be the faulty compiler. Hence, the majority of the compilers which produce equal output become the oracle.

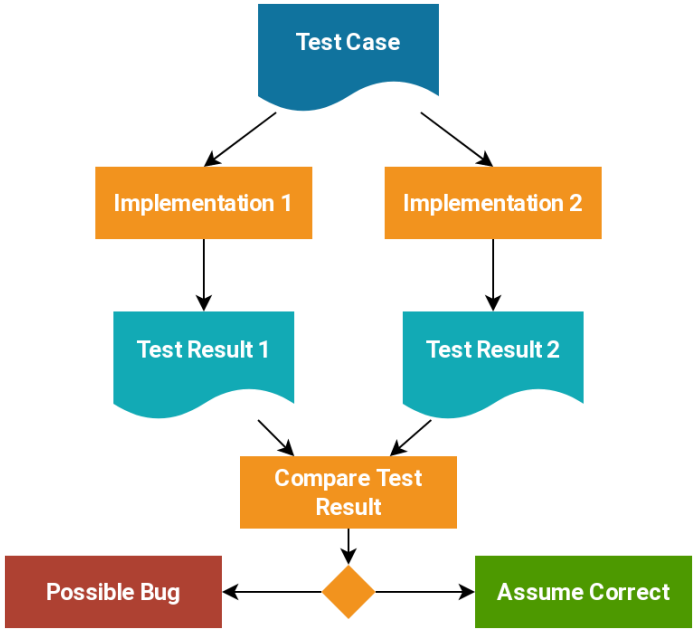

Figure 3.4. Differential testing against two different implementation, adapted from [49]

Differential testing can also detect differences in hardware behavior. A research proposed by Roberto Paleari et al. suggested that differential testing can be used to identify possible differences in hardware behavior that can be

exploited for malicious purposes [38]. The research uses the differences found in machine behavior to detect the environment where the executable runs. From this behavior, the executable can use it to identify the environment it runs, e.g., virtualized or bare-metal. This technique, which is called red-pills as a reference to The Matrix movie, is used by malware to protect itself from dynamic analysis.

Roberto Paleari et al. research uses randomized machine instruction and executes the instruction in separate execution environments [38]. The red-pill candidate is detected when the same instruction yields different behaviors, such as different exception state, register value, or memory value. Those different behaviors are possible to be distinguishable from the perspective of the running program. For such cases, the red-pills are found, and an executable can use them to distinguish its execution environment.

3.4 Analyzing Wasm with Fuzzer

This section discusses the experiment this thesis aims by incorporating testing techniques presented in this chapter. It provides the grounds and reasoning of selecting techniques and experiment model, which influenced the design decision of the experiment implementation discussed in the later chapter.

Possible Fuzzing Technique

Wasm specification is implemented in the JS engine, which is a mission-critical system. Therefore, the JS engine itself must have been undergone thorough testing, including fuzzing. However, we can explore the differential testing against multiple JS engine implementations. Similar to the C programming language that has multiple compilers, the JavaScript language is implemented by multiple different JS engines. With this situation, it is possible to perform differential testing against JS engines that implements the Wasm specification.

Targeting Specific Component

JS engine is a massive and monolithic system. Wasm implementation resides within the JS engine itself, mixed with the rest of JavaScript infrastructure. Therefore, it is essential to target the specific component related to Wasm for our experiment. We need to isolate the component we are interested in minimizing the convolution on the experiment process. The testing can be appropriately mapped to an integration testing, in which we aim to target a specific functionality that connects multiple components in the system.

We are interested in the full pipeline of Wasm program, from the compilation stage to the execution stage. We want to observe the behavior of the compiled Wasm program during the execution. Therefore, the experiment must include all Wasm components related to this pipeline.

Building Test Cases

As discussed in Section 3.2.2, the random test case must be adequately designed to target the desired component. Wasm program itself is in the binary format.

Although we can throw random binary to the Wasm interface, it is not going to yield any meaningful result to observe. Therefore, we need to craft the test case to follow a proper Wasm semantics so that the test case can produce a desirable result to analyze.

Since we aim to test the full Wasm pipeline, particularly the execution stage, the crafted test case must be at least pass the compilation stage. Therefore, the test case generator must be able to produce a valid compilable Wasm program.

Result Analysis and the Oracle

We are interested in observing the behavior of the JS engine for any random Wasm program. From each test case, we observe the behavior of the Wasm engine and collect the information. We then compare the observation between multiple implementations to search for any possible different behavior.

At this point, only two major JS engine that can be involved in the differential testing. Therefore, it is impossible to use voting heuristic as an oracle, as discussed in the Csmith research [49]. For any difference found, we consider this as an anomaly that we must investigate from both engine perspective. We must assume that both engines have an equal probability of containing the defect.

The investigation includes reproducing the identified anomalous test case, static analysis of the produced machine instruction, and dynamic analysis of the JS engine execution.

Chapter 4

Fuzzing the WebAssembly

> *"Different browsers handle compiling WebAssembly differently. Some browsers do a baseline compilation of WebAssembly before starting to execute it, and others use a JIT. Either way, the WebAssembly starts off much closer to machine code."*
>
> *"What makes WebAssembly fast?"*
> *Lin Clark - Mozilla* [7]

It is necessary to design the testing system properly. A proper test system enables to test to be performed accurately, which yields a valid and reasonable result. This chapter aims to explain the developed test bench system to execute the differential fuzz-testing on the WebAssembly implementation. It provides a thorough design explanation and analysis, which becomes the basis of the testing system development.

This chapter also presents various code examples necessary to explain the internal of the test system, particularly for the JS engine instrumentation. The JS engine instrumentation requires an in-depth inspection of the original source code, as the engine in many parts is not documented. Hence, this chapter can guides readers who are interested in performing a similar experiment to the JS engine discussed in this thesis.

Section 4.1 preludes the system development by introducing the approach used in the development. Section 4.2 discusses the general architecture design, following a good software development practice. Section 4.3 presents the development environment for system development, mainly to automate the build process. Section 4.4 and Section 4.5 discusses the instrumentation of the SpiderMonkey and the V8 JS engine for the experiment purposes thoroughly. Section 4.6 discusses the random Wasm program generator, which borrows the V8 implementation. Finally, Section 4.7 and Section 4.8 discusses the embedder program and the control program which perform the actual experiment.

4.1 Approach

The experiment aims to find a different behavior of Wasm implementations. The SUTs of this experiment are the JS engine that implements the Wasm specification. Every SUT needs to execute the same input in a controlled environment to ensure the validity of the experiment. Accordingly, the experiment requires a well-prepared infrastructure to conduct.

The testing expects the Wasm engine to accept a valid binary Wasm module. The engine performs the compilation of the Wasm module and executes it using a specified argument. The test is iterated over several times, and the testing tools collect the observable behavior of the Wasm engine.

In order to maximize the efficiency of the testing process, the testing system needs to craft a proper input binary Wasm. The main focus of the testing itself is the behavior of the Wasm execution engine. Therefore, the test system needs to ensure that every test case is a valid compilable Wasm module. Without a proper module generator, the testing will be counterproductive as the randomized test case will most - if not all - of the time produces an invalid Wasm module.

The test system also needs to isolate the component of the testing. Other elements in the JS engine, such as JS parser and compiler, is out of the scope of the experiment. The test system needs to minimize its interaction with the rest of the JS component to get a more isolated and reproducible result. Hence, the test system needs to bypass several JS pipeline to access and communicate with the Wasm components directly.

Finally, the test needs to collect the behavior of the SUT. The experiment aims to observe all observable behaviors of a Wasm program from the Wasm module standpoint. Since Wasm itself is a limited and isolated architecture, Wasm program can only observe limited information. At the moment, only memory, global variables, and tables that can be observed internally. The Wasm Table, however, is not modifiable from inside the Wasm program. Therefore, the experiment can assume that no Wasm function can modify a Wasm Table entry. Another observable side effect is the execution timing and the return value of the execution.

The test system must store the collected behavior for further analysis. It needs to provide a structured database system to ease the analysis process. Considering that the test involves a large number of tests, it is essential to employ a relational database approach for the result storage.

4.2 Designing Fuzzer

4.2.1 Requirement Specification

The requirement specification for the fuzzer infrastructure can be derived from the approach described in the previous section:

§1 Develop a random test case generator to generate the input Wasm module for fuzz-testing the Wasm engine
  §1.1 Generates a consistent random test case for any configuration
  §1.2 Always generates a valid Wasm module
  §1.3 Isolated from the SUT

§2 Instrument the JS engine to execute Wasm program with minimal overhead
  §2.1 Allow bypassing the JS script compiler to access Wasm infrastructure
  §2.2 Allow behavior observation of the JS engine during the execution of the Wasm module
  §2.3 Enable access to the generated machine instruction for inspecting the generated code

§3 Develop a shell program to embed the JS engine
  §3.1 Embed and isolate the JS engine
  §3.2 Provide a communication channel with the external program to inform the result
  §3.3 Enable an interactive session to support inspection and reproducing test cases
  §3.4 Single and unified interface for every tested JS engine

§4 Develop a program to control the testing pipeline
  §4.1 Perform automatic test case generation and execution
  §4.2 Collect and store the result of every test cases
  §4.3 Isolate every test case execution

The subsequent sections in this chapter refer to a requirement item by its requirement number.

4.2.2 Architecture Design

According to the requirement, the system requires three main components: a Test Case Generator (**§1**), Shell Programs (**§3**), and a Control Program (**§4**). Each component needs to be isolated from each other to minimize the side effect caused by every component. The side effect may reduce the precision of the test. Consequently, it needs to be minimized to produce an accurate observation of the SUT. Figure 4.1 presents the overall architecture stack for the fuzzing test system.

The Control Program governs the entire testing infrastructure. It communicates with the Shell Programs and the Test Case Generator via a prepared communication channel. The Control Program also relies on an IPC Support Library to assist the interprocess communication. This component uses a JSON library, which allows a structured text-based communication. Additionally, the Control Program also uses a Database Library to store the test result data.

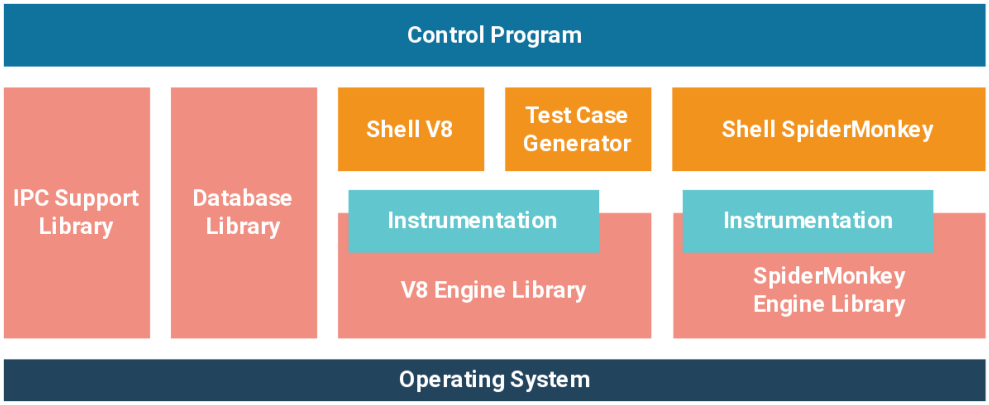

Figure 4.1. Architecture stack for the fuzzing test system

The Shell Program embeds the instrumented JS engine libraries. The shell isolates the JS engine and only accepts basic input commands to perform its tasks. Since the experiment evaluates two different engines, the JS engine is embedded into two different shell programs. The Shell Program also uses IPC Support Libary to communicate the test result to the Control Program.

The JS engine libraries are instrumented to access the internal Wasm infrastructure from the Shell Program. The instrumentation, as defined in **§2**, becomes an integral part of the JS engine library. The modification to the JS engine needs to be as minimal as possible to minimize the impact of the instrumentation on the entire JS engine. It is vital since the experiment demands the original behavior of the engine instead of the modified one.

Finally, the Test Case Generator is built on top of the instrumented V8 Engine Library. It will be explained further in Section 4.6.

4.2.3 Workflow

The test needs to be repeated to maximize the coverage of the test. Hence, the system requires a well-designed workflow cycle to describe the testing process. This cycle also describes the interconnection between testing system components. This description provides a better high-level picture of the communication requirements. Figure 4.2 shows the workflow of the fuzzing test cycle.

The test cycle begins in the control program to trigger a new test case generation. The control program, as specified in **§4**, invokes the test case generator to generate a new test case (1). The random program generator, specified in **§1**, generates a new test case (2).

The control program starts a new shell program to perform the testing (3). The shell program then consumes the generated test case, which consists of a Wasm module and an initial state of memory content (4). The shell program processes the test case and generates the output report containing the observed behavior. Finally, the control program collects the output (5), and the cycle repeats.

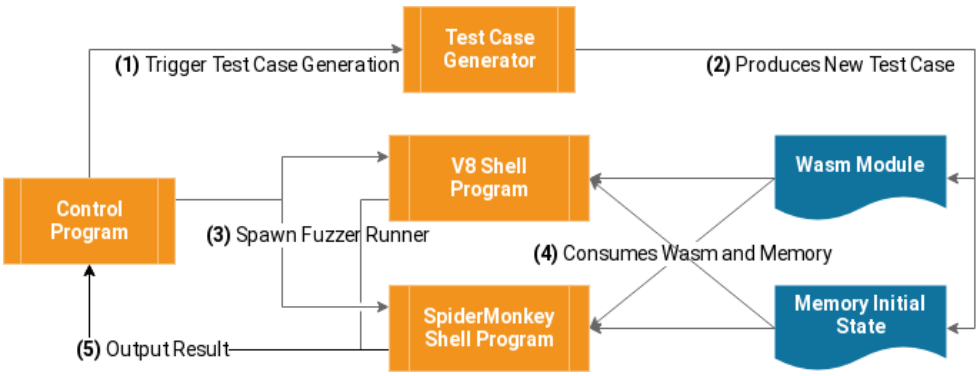

Figure 4.2. Workflow for the fuzzing process

This testing workflow creates a test circuit which performs the test automatically without the test operator intervention. The test operator can start the control program and begins the test cycle. After several test attempts, the test operator can collect and analyze the result.

4.3 General Development Environment

4.3.1 Code Organization

A well-defined code organization is mandatory for a smooth software development project. Its main principle is to separate different codes by their domain and functionality to avoid convolution. Although organizing code can be done merely through managing source code directory, a source control tool plays an essential role in managing a large codebase. This test system uses the Git source control tool. Moreover, this thesis, particularly this section, heavily uses Git terminologies. However, it does not limit the reader to use other source control solutions to implement the design presented here.

The importance of code organization became visible when dealing with a large number of source codes involved in the project. The test system discussed in this thesis incorporates two JS engines, which are major open-source projects. These engines require customizations, and the test system depends on the customization. Consequently, the engine source codes became an inseparable part of the entire test system.

In order to reduce the complexity of managing the source code, the test system incorporates the dependent JS engine source codes as the project submodule. The JS engine source code is forked and maintained separately from each other. This forked source code contains the modification made to the JS engine specifically for the test system and the experiment. By maintaining the modified JS engine in the forked repository, it can keep track of the original project and apply the update through rebasing the modification to the newer version of the main project.

As described in Section 2.6.1, the V8 project requires an external dependency,

called *depot_tools*, to build the project. This dependency is also added as a submodule. It avoids unnecessary decoupling and separate downloads, particularly when the test system integrates the entire compilation process. Other dependencies in the form of source codes are also included as subprojects. For this thesis implementation, all subprojects are stored in the third-party folder of the main project.

4.3.2 Build Automation

Build automation is necessary to reduce the time spent on configuring and compiling the program. A software project may contain a large number of source files. Without build automation, the developers need to compile every file manually. Moreover, an active development typically needs to recompile for every change made to the code. It is particularly crucial in C and C++ development where typically each source file compiles into separate object files before finally linked into a final executable. By using build automation tools, the modified code can be automatically identified and recompiled instead of recompiling the entire source file. It saves much time compared to a non-automated build. Some notable examples of build automation tools are Make and Ninja.

Table 4.1. Code organization for the test system project

Source Folder	**Content**	**Description**
third-party/v8	V8 Source	Root source tree for V8 engine. Only contains the JavaScript engine and its dependencies.
third-party/ depot_tools	Google Depot Tools	Google tools required to compile V8 engine. Consists of Python scripts to generate build script for Google project, in this case, V8.
third-party/ gecko-dev	Mozilla Firefox Source	Root source tree for SpiderMonkey engine. The project is integrated with the entire Firefox browser code, but can be compiled separately.
third-party/ wabt	WASM Binary Tools	Collection of tools to assemble/dissamble WASM using command line, as well as other additional interesting tools.
third-party/ rapidjson	RapidJSON	Header-only library to generate and parse JSON text format. It is used to aid interprocess communication between components.
third-party/ quince & third-party/ quince-sqlite	Quince ORM Library	Object-Relational Library to simplify database development. It is used to aid storing the result information from each test run.
src	Fuzz Testing Component Sources	Source folder for tools that will be developed for fuzzing.
wasm	WASM Codes	Hand-crafted or permanent WASM code that will be used for testing.

On top of build automation tools, there is a build generator tools. A build generator does not perform the actual build. Instead, it generates a build script to be processed by the build automation tools. Although one can build a build automation script manually, a full-fledged build generator tool offers a rich set of features to improve the software development process without the complexity of writing a build script manually. For example, a build automation tool simplifies the steps to add a new source code file in a project. The tool automatically updates its build script when a new source file is introduced. Some popular build generators include Autotools and CMake.

The test system developed for this experiment uses CMake. CMake allows a build-script agnostic development, instead of sticking to a specific build automation system. CMake can generate several popular build automation script, including Make, Ninja, Visual Studio Project, and XCode. It is also fairly popular for C and C++ software development and cross-platform projects. CMake also enables a build folder to be isolated from the source folder. It allows multiple builds to exist on the same project without conflicting.

Since the test system also includes several dependent projects, the test system build needs to include these projects in the build process. As mentioned in Section 2.6.1, V8 and SpiderMonkey have their own build script generator. Consequently, the test system build generator must invoke their build generator to generates the build script correctly. CMake provides a mechanism to invoke an external command during the script generation process.

4.3.3 Building the JS Engine Projects

It is necessary to examine each JS engines build step before integrating it into the entire test system build. Since both engines have their own build generator, both of them have different sets of commands to trigger the build script generation. Moreover, the engines have configurable parameters that need to be set accordingly to suit the experiment system.

The SpiderMonkey build generator uses a combination of shell and Python script, which is located in the `js/src/configure.in`. As stated in Section 2.6.1, although the SpiderMonkey project resides within the entire Gecko project, it can be built independently without building the entire Gecko project. Hence, this build generator script specifically generates the SpiderMonkey build and its required dependency.

Several configurations need to be passed to the build generator. The configuration can be supplied through the command-line argument when invoking the build generator. The build needs to be configured to disable automated testing to save build time. The *jemalloc* memory allocator also needs to be disabled as it breaks the embedding from external programs. Without disabling the *jemalloc*, the embedder always crashes with Segmentation Fault signal during the

embedding initialization. Finally, the debugging is enabled to allow inspecting the internals through a debugger. Listing 4.1 presents the command to invoke the SpiderMonkey build generator.

Listing 4.1. SpiderMonkey build generation script

```
cd gecko-dev/js/src
mkdir build_OPT.OBJ
cd build_OPT.OBJ
/bin/sh ../configure.in --enable-debug --disable-tests
↪  --disable-gtest-in-build JS_STANDALONE --disable-jemalloc
```

The V8 build generator employs a generator provided in the *depot_tools* named gn. The command to invoke the generator is similar to the SpiderMonkey generator. The build needs to disable the custom libc++ implementation. V8 uses a non-default C++ Standard Library implementation, and it can cause a conflict with the embedder, which usually compiles using the system default standard library. Disabling this feature forces the V8 to use the system provided standard library. Some features, such as disassembler and testing features are enabled. The testing features itself is essential as it orders the build generator to include the fuzzer components in the V8, which is used in the test system. More details about these fuzzer components are discussed in Section 4.6.

Listing 4.2. V8 build generation script

```
cd v8/
V8_BUILD_PATH=out/build.x86
gn gen $V8_BUILD_PATH --args='target_cpu="x64" v8_target_cpu="x64"
↪  v8_enable_disassembler=true v8_enable_v8_checks=true
↪  v8_expose_symbols=true v8_optimized_debug=true is_component_build=true
↪  use_custom_libcxx=false v8_enable_test_features=true'
```

The main difference between both build generators is the target build directory. SpiderMonkey generates the build script in the current working directory where the generator is invoked. The generated build script automatically specify the source folder relative to the working directory. On the other hand, V8 generates the build script relative to where the project is located. The gn tool requires a target directory to be supplied, and this target directory is relative to the project directory. This provision must be taken into account to ensure the build script is correctly generated when building V8 outside of its project directory.

4.3.4 Integrating the Build

Integrating the JS engine build is essentially executing the build generator command in the test system build generator. The previous section describes the command to invoke the build generator for each JS engine. The test system generator script must call these commands using `execute_process` command in CMake script.

The `execute_process` command in CMake starts an external process on behalf of the generator script. The build script can use this functionality to trigger external tools during the build script generation. It includes generating a build script for an external project. The build script can also configure the working directory and output channel of the command. Setting the working directory is important since both build generator behaves according to the working directory where they are executed.

After invoking the external build generator, the test system build generator specifies the target object to build for each JS engine. Both engines use different build tools, which also have to be invoked differently and separately. CMake provides `add_custom_target` command to add a custom build step. This custom target can invoke external command during the actual build. Through this functionality, the JS engine build can be automatically started via the test system build script.

To isolate the build, all build artifacts, including generated object files, intermediary files, and executables, must reside under the same location. Intuitively, the JS engines build artifacts must be stored in the subfolder under the build folder. Consequently, the working directory for `execute_process` and `add_custom_target` must be specified correctly to the respective JS engine build folder. It is relatively simple for SpiderMonkey since the build generator generates the build script in the working directory. Listing 4.3 shows the CMake script for SpiderMonkey build integration.

Listing 4.3. SpiderMonkey build integration (CMakeLists.txt)

```
# Make new directory for the build artifact
file(MAKE_DIRECTORY ${CMAKE_BINARY_DIR}/third-party/spidermonkey)

# Generate the build script in the build directory
execute_process(COMMAND /bin/sh
↪  ${CMAKE_SOURCE_DIR}/third-party/gecko-dev/js/src/configure.in
↪  --enable-debug --disable-tests --disable-gtest-in-build JS_STANDALONE
↪  --disable-jemalloc
    WORKING_DIRECTORY ${CMAKE_BINARY_DIR}/third-party/spidermonkey
    RESULT_VARIABLE RES_MOZ_GEN
    OUTPUT_FILE "/proc/self/fd/0")

# Add target to SpiderMonkey: run the make inside the SM's build dir
add_custom_target(build-moz make -j$$(nproc)
    WORKING_DIRECTORY ${CMAKE_BINARY_DIR}/third-party/spidermonkey
    USES_TERMINAL)
```

One particular behavior of SpiderMonkey build is that it generates a shared library with a version numbering in its file name. In order to ensure the test system can link appropriately to the library, the build script must also capture the correct shared library name. SpiderMonkey build generator generates additional metadata, which provides the file name for this purpose. Listing 4.4 shows the CMake script to obtain this information.

Listing 4.4. Getting SpiderMonkey version (CMakeLists.txt)

```
# Read SpiderMonkey version
file(READ ${CMAKE_BINARY_DIR}/third-party/spidermonkey/binaries.json
↪  MOZ_BINARIES_JSON)
string(REGEX MATCHALL "libmozjs-([A-Za-z0-9]+)\\.so" MOZ_VERSIONS
↪  ${MOZ_BINARIES_JSON})
list(GET MOZ_VERSIONS 0 MOZ_VERSION)
string(REGEX REPLACE "libmozjs-([A-Za-z0-9]+)\\.so" "\\1" MOZ_VERSION
↪  ${MOZ_VERSION})
message(INFO "Moz Version: ${MOZ_VERSION}")
```

While for the V8 build integration, the build generator must compute the relative path of the build directory. The address is relative to the V8 project directory. The build generator script is aided by a simple Python command to produce a relative path from a given absolute path. The computed path is then passed to the V8 build generator to generate the build script. Listing 4.5 shows the V8 build integration in the test system CMake script.

Listing 4.5. V8 build integration (CMakeLists.txt)

```
# Compute the target build directory
execute_process(COMMAND python -c "import os.path; print
↪  os.path.relpath(\"${CMAKE_BINARY_DIR}/third-party/v8\",
↪  \"${CMAKE_SOURCE_DIR}/third-party/v8\")"
    OUTPUT_VARIABLE V8_BUILD_PATH
    COMMAND_ECHO STDOUT
    OUTPUT_STRIP_TRAILING_WHITESPACE
)

# Generate the build script in the source directory
execute_process(COMMAND ${CMAKE_SOURCE_DIR}/third-party/depot_tools/gn gen
↪  ${V8_BUILD_PATH} --args=${V8_GEN_ARGS}
    WORKING_DIRECTORY ${CMAKE_SOURCE_DIR}/third-party/v8
    COMMAND_ECHO STDOUT
    RESULT_VARIABLE RES_MOZ_GEN
    OUTPUT_FILE "/proc/self/fd/0")

# Add target to V8: run the ninja inside the V8's build dir
add_custom_target(build-v8 ninja
    WORKING_DIRECTORY ${CMAKE_BINARY_DIR}/third-party/v8
    DEPENDS build-moz # Ordering the build
    USES_TERMINAL)
```

Other than the JS engines, the build script also needs to build the dependencies. The WABT project already uses CMake, which can be included directly in the test system CMake script. The original Quince library does not use a proper build system. Hence, the project is forked to provide a CMake build script, which then can be integrated with the rest of the project.

4.4 Instrumenting the SpiderMonkey

4.4.1 General Analysis

Before the engine can be instrumented for the experiment, we need to perform a thorough analysis of the engine itself. The engine does not provide direct access to Wasm infrastructure from its embedding API. Therefore, we need to search and expose the internal API so that it can be accessible from the embedder side.

SpiderMonkey declares Wasm related functions under the `js::wasm` namespace. The source code for the namespace also resides under the `js/wasm` directory. The first functionality that needs to be identified is the module compilation function.

It is reasonably apparent that the compilation functionality is declared in `WasmCompile.h` header file. It declares the `CompileBuffer` function [`WasmCompile.h`:103], which processes a Wasm binary to produce a Module representation. As already discussed in Section 2.6.4, the Wasm compilation process begins by compiling the module binary and stored in the internal representation for further operation. This `CompileBuffer` function performs precisely the first step in Wasm compilation, which became our first entry point for analyzing the internal engine operation.

The second step we need to perform is searching for the user of this function. Through the use of an Integrated Development Environment (IDE), we can find all references to this function. We identified several references to this function: the native function that is exposed through JS API; and the fuzzing component in SpiderMonkey. The particularly interesting one is the `js::wasm::Eval` function [`WasmJS.cpp`:619]. This function is marked as testing and fuzzing support function and process the entire workflow from the compilation to the instantiation of a Wasm module. Hence, this entire function definition satisfies the requirement **§2.1** for the test system.

Another aspect to consider is to instantiate the Wasm module properly. The instantiation process requires the imported element to be supplied. As discussed in Section 2.6.4, SpiderMonkey uses the `ImportValues` object to pass the imported elements. It is defined in the `WasmModule.h` header [`WasmModule.h`:42]. It encapsulates all import types as the member field of the object. Imported functions, tables, and global variables are stored in the vector of pointers. The imported elements must be ordered according to its index in the Wasm module. The memory import, on the other hand, is pointed using a single pointer instead. It is adhering to the current Wasm specification, where only one memory declaration can appear in a Wasm module.

For the global variables, the import can be either supplied through a `WasmGlobalObject` object, which is stored in the `globalObjs` vector, or via a `Val`

object, which is stored in the `globalValues` vector. The latter is a simplification of initializing global import using a value constant. The Wasm internal API automatically populates the respective `WasmGlobalObject` during the instantiation if it is not supplied by the caller [`WasmModule.cpp`:905].

The test system also needs access to the Wasm module metadata, such as function names. It is required to develop a reflection-like API to inspect the content of a Wasm module as required by requirement **§2.3**. The `Module` object returned by the `CompileBuffer` function[5] contains a reference to a metadata object [`WasmModule.h`:174]. This metadata object, called `MetadataTier` [`WasmCode.h`:420], contains information about exported functions and the location of the functions in a module. The function location information is stored in a vector of `CodeRange` and stores important information about the compiled instruction, including the function entry point.

The `funcNormalEntry` accessor function of the `CodeRange` object provides the information of the function entry point [`WasmTypes.h`:2463]. Moreover, the `end` accessor function of the same object provides the end position of the function [`WasmTypes.h`:2405]. These locations are relative locations, which is an offset of the code section store in the memory. The actual address of this code section can be obtained from the `ModuleSegment` object, which can also be obtained from the Module object [`WasmModule.h`:172]. The `ModuleSegment` object, which inherits the `CodeSegment` class, stores the base address of the instruction address [`WasmCode.h`:142]. By computing the base address and the entry point obtained previously, we can obtain the actual memory location of the compiled instruction. This mechanism is essential to allow dumping and inspecting the compiled Wasm program for the experiment result analysis.

4.4.2 Introducing a New API Header

The instrumentation requires a public API header to allow the embedder to access the customized functions. The SpiderMonkey provides `jsapi.h`, which is the main header file to embed the engine. It is possible to extend the existing header with the new customized functions. However, it may introduce a complexity, especially during the update of the code from the original repository. It is a good practice to separate the instrumented and customized function into a separate header and source file. Therefore, the customization remains isolated and separately tracked.

The test system introduces a new header file and a source file to contain the customized function. The file is located in the source root of the SpiderMonkey project, the same location as the original `jsapi.h` file. Then, the new header and source must be specified in the build generator script, the `moz.build` file. The header file is added to the array of `EXPORTS` variable, while the source file is added

[5] The `CompileBuffer` returns the reference to the `Module` object inside a shared pointer object (`RefPtr`/`SharedModule`). The object members can be accessed through dereference operator, similar to dereferencing a pointer.

to the array of `UNIFIED_SOURCES`. We give the header and source file name as `jsapi-ext.h` and `jsapi-ext.cc`, respectively.

4.4.3 Designing API for the Instrumented Function

Exposing the internal data structure to the embedder needs to be avoided at all costs since the definition is not visible from the embedder side. Therefore, it is necessary to define a new custom data structure to maintain the information between the JS engine side and the embedder side. It abstracts the information and the operation that is introduced by the customized function developed for this experiment.

The test system embedder requires an abstraction of a Wasm module. It can be implemented by encapsulating the Module object inside an opaque data structure that is only defined from the JS engine internal code. In C++, it is achieved through Pointer to Implementation (*pImpl*) idiom [45]. This idiom breaks the compile-time dependency between different compilation units. Through this idiom, the public header does not require the definition of internal data structures to compile correctly.

Listing 4.6. `CompiledInstruction` definition (`jsapi-ext.h`)

```
class CompiledInstructions {
private:
  class Internal;
  std::unique_ptr<Internal> internal;
  std::list<Function> functions_;
  std::map<std::string, WasmType> globals_;
  std::map<std::string, std::reference_wrapper<Function>> functionsByName;

public:
  CompiledInstructions(JSContext*);
  ~CompiledInstructions();

  std::list<Function> const& Functions() {
    return functions_;
  }

  std::map<std::string, WasmType> const& Globals() {
    return globals_;
  }

  bool InstantiateWasm(JSContext* cx);
  void NewMemoryImport(JSContext* cx);
  void NewGlobalImport(JSContext* cx);

  void SetGlobalImport(JSContext* cx, std::string const& name,
                       WasmGlobalArg value);

  auto GetGlobalImport(JSContext* cx, std::string const& name)
      -> std::pair<WasmType, WasmGlobalArg>;

  WasmMemoryRef GetWasmMemory();
};
```

Listing 4.6 shows the definition for `CompiledInstructions`. This class encapsulates the compiled Wasm module returned from the compilation function. It provides mechanisms to inspect the compiled Wasm module, specifically the exported functions and global variables. The `CompiledInstruction` object also provides the function to instantiate the module. Since the instrumentation is designed only for the testing purpose, it does not support multiple instantiations of a Wasm module. The module can only be instantiated once to simplify the instrumentation code complexity.

In addition to the instantiation function, the `CompiledInstruction` object also tracks and maintains the imported memory and global variables. As specified in requirement **§2.2**, the embedder needs a mechanism to observe the behavior of the JS engine from the perspective of a Wasm program. Hence, the embedder needs direct access to observe the Wasm memory and global variables to monitor changes in each testing sequence.

Listing 4.7. Supporting data types definition (`jsapi-ext.h`)

```
enum class WasmType {
  Void,
  I32,
  I64,
  F32,
  F64
};

struct WasmMemoryRef {
  uint8_t* buffer;
  size_t length;
};

union WasmGlobalArg {
  uint32_t i32;
  uint64_t i64;
  float   f32;
  double  f64;
};
```

The instrumentation also defines several support data types to represent Wasm types, Wasm memory, and Wasm values. Listing 4.7 shows the definition of the support data types. An enum class type is defined to represent a Wasm type. By default, the SpiderMonkey public header does not expose the definition for Wasm types. Consequently, a new data type needs to be defined to transfer the information to the embedder side.

The Wasm memory is also encapsulated in a POD[6] type. Wasm memory reference is generally a regular pointer to a memory location. In order to ensure safe memory access, the API must retain and enforce the length information. This encapsulation also simplifies the accessor function for Wasm memory. Finally, the `WasmGlobalArg` union handles the value transmission from and to global

[6]Plain Old Data, which is a data type which only consists of fields without member function.

variables. This type uses a union type to minimize the memory footprint for a simple polymorphic object definition. Since the SpiderMonkey uses C++14 standards, a discriminated union type such as `std::variant` is not yet available to be used in the code.

The module representation also contains module function representation. The Function object encapsulates a Wasm function. It contains the metadata, cached instruction binary, and a function to invoke the Wasm function. Listing 4.8 shows the definition of the Function object. A vector of `WasmType` stores the function parameters. The test system uses this information to prepare a crafted argument for the fuzzing process. The return type is also stored as `WasmType` for the same purpose.

Listing 4.8. `Function` object definition (`jsapi-ext.h`)

```
struct Function {
  std::string name;
  std::vector<uint8_t> instructions;
  bool exported;
  uint32_t index;
  CompiledInstructions* parent;
  std::vector<WasmType> parameters;
  WasmType returnType;

  std::tuple<bool, uint64_t> Invoke(JSContext* cs, std::vector<JS::Value>&
  ↪  argsStack);
};
```

Another function to expose is the Wasm compile function, which produces the `CompiledInstruction` object. Additionally, the instrumented API also needs to expose a function to create and obtain a `BigInteger` value. For some reason, the SpiderMonkey embedder API does not provide a function to set and get `BigInteger` value. Hence, it is necessary to expose this function to allow utilizing 64-bit integers with the Wasm function.

Listing 4.9. Static function declarations (`jsapi-ext.h`)

```
extern std::unique_ptr<CompiledInstructions> CompileWasmBytes(JSContext* cs,
↪  uint8_t const* arr, size_t size);

extern JS::Value CreateBigIntValue(JSContext* cs, uint64_t val);

extern uint64_t GetBigIntValue(JS::Value val);
```

The Internal object used in the pImpl idiom is defined in `jsapi-ext.cc` source file. The definition consists of references to every Wasm internal objects. It includes the instance object, module object, Wasm memory object, and Wasm global objects. This Internal object is used in every instrumented functionality developed for the experiment. Listing 4.10 shows the definition of the Internal object.

Listing 4.10. Internal object declarations (jsapi-ext.cc)

```
struct GlobalEntry {
  js::ext::WasmType type;
  js::WasmGlobalObject* global_object;
};

class CompiledInstructions::Internal {
  js::RootedWasmInstanceObject instance;
  js::wasm::SharedModule module;
  js::WasmMemoryObject* wasm_memory { nullptr };
  std::map<std::string, GlobalEntry> wasm_global_list;
  js::WasmGlobalObjectVector global_vector;
  bool global_import_processed { false };
public:
  Internal(JSContext* cx) : instance(cx) { }
};
```

4.4.4 Wasm Module Compilation Function

As outlined in Section 4.4.1, the compilation function copies the implementation of the Eval function. In addition to that, the function also collects the metadata information from the compiled module to provide it to the embedder. The metadata information is cached in the public data structure, as described in Section 4.4.3.

Collecting the metadata information, as explained in Section 4.4.1, uses the information provided by the `MetadataTier` object. It contains the collection of `CodeRange` object, which can be iterated to obtain every declared function in the Wasm module. Listing 4.11 shows the snippet of the iteration to obtain the compiled function data.

Listing 4.11. Code snippet to iterate Wasm functions (jsapi-ext.cc)

```
// Returned CompiledInstruction object
auto retObj = std::make_unique<js::ext::CompiledInstructions>(cx);

// Get all required internal data structures
auto& wasmCode = module->code();
auto& codeTierMeta = wasmCode.metadata(js::wasm::Tier::Optimized);
auto& codesegment = wasmCode.segment(js::wasm::Tier::Optimized);
auto& moduleExports = module->exports();
auto baseaddress = codesegment.base();
auto& funcExports = codeTierMeta.funcExports;

for(auto& codeRange : codeTierMeta.codeRanges) {
  if(codeRange.isFunction()) {
    auto& dumpedFunction = retObj->functions_.emplace_back();
    dumpedFunction.parent = retObj.get();
    dumpedFunction.index = codeRange.funcIndex();
    // Get instruction bytes
    // Get function names
    // Get function signatures
  }
}
```

From the iteration, the function can obtain the actual machine instruction bytes by accessing the information inside the `CodeRange`. After getting the memory

address where the instruction is located, the function can perform a simple buffer copy to the cached instruction bytes in the `CompiledInstruction` object. Listing 4.12 shows the snippet to copy the instruction bytes.

Listing 4.12. Getting the compiled instruction bytes (`jsapi-ext.cc`)

```
auto codeBegin = baseaddress + codeRange.funcNormalEntry();
auto codeEnd = baseaddress + codeRange.end();
dumpedFunction.instructions.insert(dumpedFunction.instructions.end(),
↪  codeBegin, codeEnd);
```

Function names are stored in a different location from the metadata information. It is stored in the `Export` object, which is managed on a different list. The function needs to search the list of exported elements and match them by its function index. The matched `Export` object contains the function name, which can be copied to the instrumentation cache. Listing 4.13 shows the sinppet for obtaining the function name.

Listing 4.13. Code snippet to obtain function name (`jsapi-ext.cc`)

```
auto funcExport =
  std::find_if(moduleExports.begin(), moduleExports.end(),
               [idx = codeRange.funcIndex()] (Export const& a) { return
               ↪  a.kind() == DefinitionKind::Function && a.funcIndex() ==
               ↪  idx; });

if(funcExport != moduleExports.end()) {
  dumpedFunction.exported = true;
  dumpedFunction.name = funcExport->fieldName();
  retObj->functionsByName.emplace(dumpedFunction.name,
  ↪  std::ref(dumpedFunction));
}
```

The function signature, which consists of parameter types and return type, is stored inside the `FuncExport` list. Similar to obtaining the function names, the `FuncExport` list must be searched manually by matching its function index. The found object contains the parameter and return type information, which can be stored in the instrumentation data structure. Listing 4.14 shows the snippet of getting the function signature.

4.4.5 Wasm Memory and Global Variable Accessors

SpiderMonkey encapsulates memory and global variable in internal objects. It is possible to track these objects to allow behavior observation of Wasm execution towards the memory and global variable. Although it is possible to obtain memory and global variable objects for every Wasm module, it is easier to use the Wasm import feature.

Wasm import provides a complete set of API to supply memory and global variable object explicitly to a Wasm instance. Through this API, the instrumentation does not have to modify the internal Wasm data structure to expose the memory

Listing 4.14. Code snippet to obtain function signature (`jsapi-ext.cc`)

```
auto funcExportMeta =
  std::find_if(funcExports.begin(), funcExports.end(),
               [idx = codeRange.funcIndex()] (FuncExport const& a) { return
               ↪  a.funcIndex() == idx; });

if(funcExportMeta != funcExports.end()) {
  FuncType const& funcType = funcExportMeta->funcType();
  // Get Return Type information
  decltype(auto) resultTypes = funcType.results();
  if(resultTypes.empty()) {
    dumpedFunction.returnType = WasmType::Void;
  } else {
    // Only take the first one
    dumpedFunction.returnType = ValTypeToExtType(resultTypes[0].kind());
  }

  // Get parameters
  for(decltype(auto) kind : funcType.args()) {
    dumpedFunction.parameters
        .push_back(ValTypeToExtType(kind.kind()));
  }
}
```

and global variable object. Moreover, it enables the test system to initialize the environment and observe the change accordingly.

The first instrumentation required is memory and global variable object instantiation. We can find some code example inside the engine which describe the process of instantiating the memory object. We previously identified the class name for the memory object as `WasmMemoryObject`. This class provides a static method `create`, which instantiates the object by supplying a memory buffer.

Intuitively, we look for any reference that uses this function to understand the proper way to use it. We identified that the `instantiateMemory` function in `Module` class uses the function to prepare the memory object if it is absent during the module instantiation process. From this code, we can craft the instrumented function to instantiate a memory object explicitly. Listing 4.15 shows the function that implements this mechanism.

The raw buffer of the memory is accessible by using the accessor function provided by `WasmMemoryObject`. Listing 4.16 shows the implementation of `GetWasmMemory` function, which returns the raw memory pointer and its length. The raw buffer can be modified directly, and the changes automatically reflected in the Wasm realm.

Global variable object instantiation is more complex than the memory object since it involves multiple items. The instantiation function needs to iterate the import list and instantiate a global variable object for each global import. The module metadata also provides the import list, similar to the export list used to obtain exported function information. Listing 4.17 displays the main loop for instantiating global import.

Listing 4.15. Code snippet to obtain function signature (`jsapi-ext.cc`)

```
void CompiledInstructions::NewMemoryImport(JSContext* cx) {
  // Get the memory size requirement from metadata
  const wasm::Metadata& metadata = this->internal->module->metadata();
  uint32_t declaredMin = metadata.minMemoryLength;
  mozilla::Maybe<uint32_t> declaredMax = metadata.maxMemoryLength;

  // Define te memory limit and instantiate the buffer object
  RootedArrayBufferObjectMaybeShared buffer(cx);
  wasm::Limits l(declaredMin, declaredMax, wasm::Shareable::False);
  if (!CreateWasmBuffer(cx, l, &buffer)) {
    std::cerr << "Error CreateWasmBuffer\n";
    return;
  }

  // Build the WasmMemoryObject
  RootedObject proto(cx,
  ↪  &cx->global()->getPrototype(JSProto_WasmMemory).toObject());
  this->internal->wasm_memory = WasmMemoryObject::create(cx, buffer, proto);
}
```

Listing 4.16. Code snippet to get memory buffer (`jsapi-ext.cc`)

```
WasmMemoryRef CompiledInstructions::GetWasmMemory() {
  if(this->internal->wasm_memory == nullptr ) {
    return { nullptr };
  }
  auto data = this->internal->wasm_memory->buffer().dataPointerEither();
  return { data.unwrap(), this->internal->wasm_memory->volatileMemoryLength()
  ↪  };
}
```

Listing 4.17. Iterating the global variable import list (`jsapi-ext.cc`)

```
void CompiledInstructions::NewGlobalImport(JSContext* cx) {
  // Process only once
  if(this->internal->global_import_processed)
    return;
  this->internal->global_import_processed = true;

  // Obtain the metadata
  auto& moduleImports = this->internal->module->imports();
  const wasm::Metadata& metadata = this->internal->module->metadata();
  const wasm::GlobalDescVector& globals = metadata.globals;

  uint32_t globalIndex = 0;
  for(js::wasm::Import const& importEntry : moduleImports) {
    if(importEntry.kind == js::wasm::DefinitionKind::Global) {
      // Instantiate global variable object
    }
  }
}
```

For each global variable import, a new global variable instance must be instantiated. The global variable instance must be stored in a vector, ordered by its index. It is necessary because SpiderMonkey enforces the strict ordering of the items during the instantiation of the module. Besides, the instrumented function also stores the instantiated global object in a key-based collection for

easy retrieval by the embedder. Listing 4.18 presents the instantiation of a global variable object for each iteration in Listing 4.17.

Listing 4.18. Instantiating the global variable object for import (`jsapi-ext.cc`)

```
// The iteration of import orders the index value
uint32_t this_index = globalIndex++;

// Get global description
wasm::GlobalDesc const& this_desc = globals[this_index];
wasm::ValType this_type = this_desc.type();

// Create initial value based on type(source: WasmJS.cpp:550-557)
wasm::RootedVal val(cx);
val.set(wasm::Val(this_type));

// Create WasmGlobalObject with the specified global type
RootedObject proto(cx);
proto = GlobalObject::getOrCreatePrototype(cx, JSProto_WasmGlobal);
WasmGlobalObject* this_global = WasmGlobalObject::create(cx, val,
↪  this_desc.isMutable(), proto);

// Store in WasmGlobalObjectVector based on its index
if (this->internal->global_vector.length() <= this_index &&
↪  !this->internal->global_vector.resize(this_index + 1)) {
  ReportOutOfMemory(cx);
  return;
}

// Store internally
this->internal->global_vector[this_index] = this_global;
this->internal->wasm_global_list.emplace(
  std::string{importEntry.field.get()},
  GlobalEntry {ValTypeToExtType(this_type.kind()), this_global});
this->globals_.emplace(std::string{importEntry.field.get()},
↪  ValTypeToExtType(this_type.kind()));
```

The global variable object stores its value in a `Val` object. The object is polymorphic internally, and stores all types of Wasm value. It provides accessor and mutator for every Wasm type and must be called according to the actual value type it stores. Since our internal data also stores the Wasm type, the function can use a switch structure to pick the correct value to obtain or supply. Listing 4.19 and 4.20 shows the implementation of global value setter and getter.

The setter function accepts the `GlobalWasmArg` union declared in the instrumented API. The function assumes the union is storing the value according to the global type. It discriminates the union value by the global type information stored in the internal data structure. If the caller supplies a mismatched value, the behavior is undefined according to the C++ standard. The `Val` object constructor is called according to the argument type passed to its constructor, which finally is passed to the global object via `setVal` function.

The instrumented getter function selects the appropriate getter function of the `Val` object based on the global types. It then stores the value to the `GlobalWasmArg` union, which then passed to the function return value. Since the

Listing 4.19. Global value setter implementation (`jsapi-ext.cc`)

```
void CompiledInstructions::SetGlobalImport(JSContext* cx, std::string const&
↪  name, WasmGlobalArg value) {
  auto& global_list = this->internal->wasm_global_list;
  auto global_iter = global_list.find(name);
  if(global_iter != global_list.end()) {
    using E = js::ext::WasmType;
    GlobalEntry& global_ = global_iter->second;
    wasm::RootedVal val(cx);

    switch(global_.type) {
      case E::I32: val.set(wasm::Val(value.i32)); break;
      case E::I64: val.set(wasm::Val(value.i64)); break;
      case E::F32: val.set(wasm::Val(value.f32)); break;
      case E::F64: val.set(wasm::Val(value.f64)); break;
      case E::Void: MOZ_CRASH();
    }
    global_.global_object->setVal(cx, val);
  }
}
```

`Val` object may be tracked by the GC, accessing the Val object stored inside the global variable object must use the `Rooted` container. `RootedVal`, the specialization of `Rooted` container for `Val` object, holds the reference to the `Val` object owned by the global variable object. The value then can be safely accessed from our getter function.

Listing 4.20. Global value getter implementation (`jsapi-ext.cc`)

```
auto CompiledInstructions::GetGlobalImport(JSContext* cx, std::string const&
↪  name)
  -> std::pair<WasmType, WasmGlobalArg> {
  auto& global_list = this->internal->wasm_global_list;
  auto global_iter = global_list.find(name);
  if(global_iter != global_list.end()) {
    GlobalEntry& global_ = global_iter->second;
    wasm::RootedVal val(cx);
    global_.global_object->val(&val);

    WasmGlobalArg value;
    switch(global_.type) {
      case WasmType::I32: value.i32 = val.get().i32(); break;
      case WasmType::I64: value.i64 = val.get().i64(); break;
      case WasmType::F32: value.f32 = val.get().f32(); break;
      case WasmType::F64: value.f64 = val.get().f64(); break;
      case WasmType::Void: MOZ_CRASH();
    }
    return {global_.type, value};
  } else {
    return {WasmType::Void, {}};
  }
}
```

4.4.6 Wasm Function Invoker

As mentioned in Section 2.6.4, SpiderMonkey provides access to call exported functions through its instance object. The function, which is named `callExport`, accepts the function index and the array of the arguments encapsulated in `CallArgs` object. SpiderMonkey defines a specific convention on the calling argument to be passed via the `CallArgs` object.

The `CallArgs` object originates from an array of `Value` objects. The `CallArgs` object does not specify the kind of allocation types for the array. Therefore, a heap-based array, such as `std::vector`, can be used to store the array. It is also preferable to use dynamically allocated and managed array because the argument count for every function call may be different. The `CallArgsFromVp` function encapsulates the raw array of Value pointer to the `CallArgs` object, which can then be passed to the `callExport` function. Listing 4.21 shows the complete implementation of the `Invoke` function. Note that the function also tracks the elapsed time of the function execution, which is one of the behavior observed by the experiment.

Listing 4.21. Implementation of the `Invoke` function (`jsapi-ext.cc`)

```
auto CompiledInstructions::Function::Invoke(JSContext* cs,
↪  std::vector<JS::Value>& argsStack)
  -> std::tuple<bool, uint64_t> {
  // Uninstantiated module
  if(parent->internal->instance.get() == nullptr)
    return {false, 0};
  auto& moduleInstance = parent->internal->instance;
  js::wasm::Instance& instance = moduleInstance->instance();

  // Build CallArgs
  auto callargs = JS::CallArgsFromVp(argsStack.size(), argsStack.data());

  // Call
  std::chrono::steady_clock::time_point start_time =
  ↪  std::chrono::steady_clock::now();
  bool res = instance.callExport(cs, this->index, callargs);
  std::chrono::steady_clock::time_point end_time =
  ↪  std::chrono::steady_clock::now();
  uint64_t elapsed =
  ↪  std::chrono::duration_cast<std::chrono::nanoseconds>(end_time -
  ↪  start_time).count();

  // true if call is successful, false otherwise
  return {res, elapsed};
}
```

Naturally, the array contains the function argument according to the function parameter order. However, the array also needs to contain a space for the function return value. The first element of the array is reserved for this purpose and must be initialized with an empty `Value` object. The second element of the array is a reserved space and must also be initialized with an empty `Value`. SpiderMonkey uses this second element to indicate several internal JS functions, such as the

constructor function of a JS object. The actual function arguments begin at the third element onwards. Listing 4.22 shows the proper steps to prepare the vector of `Value` objects before it is passed to the Invoke function.

Listing 4.22. Using the `Invoke` function (`runner-spidermonkey.cpp`)

```
std::vector<JS::Value> callStack;
callStack.emplace_back(); // Return value
callStack.emplace_back(); // MAGIC (empty)
// This function puts the arguments to the callStack vector
// It will be explained in later section
MarshallArgs(callStack, args);
auto [invokeRes, elapsed] = (*compiled_wasm)[name]->Invoke(context,
↪  callStack);
```

4.5 Instrumenting the V8

4.5.1 General Analysis

Similar to the SpiderMonkey, we need to analyze the engine to identify the internal Wasm infrastructure. V8 also isolates all Wasm infrastructure under a single directory, which makes code examination easier. V8 also groups the Wasm infrastructure under the same `v8::internal::wasm` namespace.

The examination begins by searching the compilation function. V8 provides a `module-compiler.h` header, which the `CompileToNativeModule` function resides [`module-compiler.h`:42]. This function has complex parameters, which suggests that this function is not intended to be directly called to compile a binary Wasm module. The search continues by finding the references to this function. We finally reach the `WasmEngine` class, which has `SyncCompile` function [`wasm-engine.cc`:464].

The `SyncCompile` function uses the `CompileToNativeModule` function in its process. The `SyncCompile` function itself accepts more simple parameters compared to the `CompileToNativeModule` function. These parameters are are the Isolate object, a Wasm feature configuration, error notifier, and the module bytes. The function returns a `WasmModuleObject`, which obviously, a representation of a compiled Wasm module. Therefore, we can conclude that this function is the endpoint for other codes to compile a binary Wasm module.

From the `SyncCompile` function, we continue the search to find an example of its uses. The function is referred to in the `wasm-js.cc` file. It contains the native implementation of the constructor for `WebAssembly.Module` JS object [`wasm-js.cc`:637]. This function is specifically designed to accept invocation from the JS realm, as indicated by its parameter. It is more difficult to attach to this function as the embedder must craft the proper arguments for this function. The argument structure is specified according to the JS to Native calling convention defined in the V8 engine. Therefore, it is more reasonable to take some portion of its code

that is essential for the Wasm module compilation.

From the code in `wasm-js.cc` file, it is apparent that the Isolate object provides the `WasmEngine` object [`wasm-js.cc`:668]. It implies that any part of the program that has access to an Isolate object can also access the `SyncCompile` function. We also identified the proper steps to prepare the raw binary data to be passed to the `SyncCompile` function. It only involves a simple raw buffer copy, which suggests that a simple heap-based array can also be used for this purpose.

We also identified the instantiate steps in the `wasm-js.cc` file. The instantiation process also involves the `WasmEngine` object, which provides the `SyncInstantiate` function [`wasm-js.cc`:757]. `WasmEngine` object provides synchronous and asynchronous functions for the module instantiation. For the testing system, we must use the synchronous function because the test workflow itself is executed in a single-threaded environment.

The next element to search is the memory and global variable representation. The `wasm-object.h` header contains all related header to Wasm element representation. It includes `WasmMemoryObject` [`wasm-objects.h`:282] and `WasmGlobalObject` [`wasm-objects.h`:320], which is the internal representation of the JS `WebAssembly.Memory` and `WebAssembly.Global` object, respectively. Both classes provide a factory function to instantiate the object that can be used in the instrumented function.

Unlike in SpiderMonkey, the module metadata information is stored directly in the module object. The metadata is available directly in the `WasmModule` object returned by the compilation function. The `WasmModule` object stores the export and import list in its `export_table` and `import_table` members [`wasm-module.h`:333]. `WasmModule` object also stores the function information, including its signature. The signature is accessible by accessing the `FunctionSig` object through the `WasmFunction` object [`wasm-objects.h`:54]. The `WasmFunction` object is stored in the `WasmModule` in its functions field, which is ordered by the function index.

Accessing the machine instruction in V8 is also more straightforward than in SpiderMonkey. The machine instruction is accessible through the `WasmCode` object [`wasm-code-manager.h`:114]. This object can be obtained from `NativeModule` object, which the Wasm module object provides [`wasm-objects.h`:135]. `WasmCode` object is unique per Wasm function and can be retrieved by function index from the `NativeModule`'s `GetCode` function [`wasm-code-manager.h`:497].

4.5.2 Introducing a New API Header

Similar to the SpiderMonkey, it is a better approach to separate the instrumented source from the original code. Hence, we need to introduce a new public header file and a new source file to contain the implementation.

The public header must be stored in the include folder, along with the original `v8.h` header. This include folder contains all publicly accessible header files

required by embedders. In order to add the new source file to the compilation chain, we need to add the new source file in the sources list in the `BUILD.gn` file. Introducing a new source file is straightforward. However, since the instrumentation also requires the fuzzer component from V8 to be exposed, it requires more complicated build script modification. Therefore, this build script configuration is covered later in Section 4.6.3.

4.5.3 Designing API for the Instrumented Function

The API design for the V8 instrumentation is similar to the SpiderMonkey counterpart. We use the *pImpl* idiom to break compile-time dependency between internal and external code. We create the CompiledWasm class to encapsulate the compiled Wasm module. The instance of this class stores the function and global variables information. Internally, the CompiledWasm object stores the reference to the respective Wasm objects, which are used for all Wasm operations. Listing 4.23 presents the designed CompiledWasm class.

Listing 4.23. The `CompiledWasm` definition (`v8-ext.h`)

```
class CompiledWasm {
  struct Internal;
  std::unique_ptr<Internal> internal;
  std::vector<CompiledWasmFunction> functions;
  std::map<std::string, size_t> function_names;
  std::map<std::string, WasmType> globals;

public:
  CompiledWasm();
  ~CompiledWasm()

  std::vector<CompiledWasmFunction> const& Functions() {
    return functions;
  }
  std::map<std::string, WasmType> const& Globals() {
    return globals;
  }

  bool InstantiateWasm(Isolate* i);
  void NewMemoryImport(v8::Isolate* i);
  void NewGlobalImport(v8::Isolate* i);

  void SetGlobalImport(std::string const& name, WasmGlobalArg value);
  auto GetGlobalImport(std::string const& name)
      -> std::pair<WasmType, WasmGlobalArg>;

  WasmMemoryRef GetWasmMemory();
  size_t GetWasmMemorySize();
};

Maybe<CompiledWasm> CompileBinaryWasm(Isolate* i, const uint8_t* arr, size_t
↪  len);
```

The instrumentation also defines the same supporting data types as in SpiderMonkey. Basically, the V8 instrumented header also contains the same definition

Listing 4.24. The `Internal` definition (`ext-api.cc`)

```
struct CompiledWasm::Internal {
  Handle<WasmModuleObject> module_object { Handle<WasmModuleObject>::null()
  ↪ };
  Handle<WasmInstanceObject> module_instance {
  ↪ Handle<WasmInstanceObject>::null() };
  Handle<WasmMemoryObject> wasm_memory_object {
  ↪ Handle<WasmMemoryObject>::null() };
  std::map<std::string, GlobalEntry> wasm_global_list;
  bool global_import_available { false };
};
```

as in Listing 4.7. Since it is more difficult to unify the instrumentation code into a single source and definition, duplicating the definition for every JS engine is the more viable option.

Also similar to the SpiderMonkey instrumentation design, the V8 instrumentation also encapsulates Wasm functions in its own object. The Function object provides access to function information, including function name and its signature. Unlike the SpiderMonkey instrumentation, however, the signature information is not cached in the Function object. It is because the V8 engine provides a relatively more trivial way to obtain the function signature. Also, V8 provides direct access to the instruction bytes without complex computation. Therefore, the machine instruction byte does not need to be cached manually as in the SpiderMonkey instrumentation. Listing 4.25 shows the `Function` class definition.

Listing 4.25. The `Function` definition (`v8-ext.h`)

```
class CompiledWasmFunction {
  struct Internal;
  std::unique_ptr<Internal> internal;
  uint32_t func_index;
  std::reference_wrapper<CompiledWasm> parent;
  std::string name;

public:
  CompiledWasmFunction(CompiledWasm& parent);
  ~CompiledWasmFunction();

  WasmType ReturnType() const;
  std::vector<WasmType> Parameters() const;
  std::string const& Name() const { return name; }

  auto Invoke(Isolate* i, std::vector<Local<Value>>& args) const
      -> std::tuple<MaybeLocal<Value>, uint64_t>;

  std::vector<uint8_t> Instructions() const;
};
```

4.5.4 Wasm Module Compilation Function

Section 1 discusses the SyncCompile function that is used in the Wasm JS API implementation. The instrumented compilation function uses the portion of this

implementation, which we copy the code to our instrumented function. Listing 1 shows the implementation of our compilation function.

As mentioned before, the WasmEngine object is available from the `Isolate` object. Hence, the caller must pass the Isolate object to the compilation function. Externally, the `Isolate` object is stored in an opaque pointer. This opaque pointer, `v8::Isolate`, does not have a definition, meaning that the compiler does not recognize the content of the object[7]. Therefore, the internal function must cast the external Isolate object to the internal `Isolate` object. The internal `Isolate`, `v8::internal::Isolate`, has a complete definition that is accessible from internal codes. Casting the pointer to this type is done through `reinterpret_cast`, which "change" the pointer type[8]. It effectively allows the pointer to access its content and operation.

Listing 4.26. The compilation function implementation (`api-ext.cc`)

```
namespace i = v8::internal;
v8::Maybe<CompiledWasm> CompileBinaryWasm(v8::Isolate* i, const uint8_t* arr,
↪  size_t len) {
  i::Isolate* isolate = reinterpret_cast<i::Isolate*>(i);
  i::WasmJs::Install(isolate, true);
  auto enabled_features = i::wasm::WasmFeatures::FromIsolate(isolate);
  i::wasm::ErrorThrower interpreter_thrower(isolate, "Interpreter");
  i::wasm::ModuleWireBytes wire_bytes(arr, arr + len);

  // Force enable the Liftoff compiler
  bool prev = i::FLAG_liftoff;
  i::FLAG_liftoff = true;
  auto wasm_engine = isolate->wasm_engine();

  // Compile the binary WASM
  i::MaybeHandle<i::WasmModuleObject> compiled_module_res =
                wasm_engine->SyncCompile(isolate, enabled_features,
                                         &interpreter_thrower, wire_bytes);
  i::FLAG_liftoff = prev;

  if(compiled_module_res.is_null()) {
    return v8::Nothing<v8::ext::CompiledWasm>();
  }
  auto compiled_module = compiled_module_res.ToHandleChecked();
  v8::ext::CompiledWasm ret;
  ret.internal->module_object = compiled_module;
  // Get function names ...
  return v8::Just<v8::ext::CompiledWasm>(ret);
}
```

Before the code can use the Wasm API, the code must initialize the Wasm infrastructure through its `Install` function. The `Install` function initializes all required Wasm infrastructure for a given `Isolate` object. It only needs to be called once per `Isolate` object, as the V8 implements an on-demand approach for

[7] The opaque pointer means that the type is declared, but it does not have a definition. Declaring a pointer type to a declared-but-undefined type is allowed in C++. However, it does not allow any operation, including pointer dereference. It is the same principle as in *pImpl* idiom, which breaks the compilation dependency.

[8] `reinterpret_cast` does not translate into any actual instruction. It is an instruction to the compiler to treat the value as if it is another type.

enabling the Wasm feature to the engine.

The instrumented compilation function calls the `SyncCompile` function by passing the required arguments. The function also forces enabling the Liftoff compiler by setting the flag manually. It may not be necessary if the compilation flag enables the Liftoff compiler by default. The `SyncCompile` function returns a `MaybeHandle` object, which may contain the `WasmModuleObject` value. If the `MaybeHandle` object is empty, the compilation fails, and the error message can be obtained via the `ErrorThrower` object.

From the module object, we can obtain the exported function to collect its information. For getting string value from the Wasm module, V8 has a more complicated step compared to SpiderMonkey. It seems that V8 does not cache the string value declared in a Wasm module. Instead, it directly all string in a single allocation, which can be referred to by offset and length. The string allocation in a module is accessible through the `ModuleWireBytes`, while referring to a single string value uses the `WireByteRef`. Listing 4.27 shows the procedure to access the name of the function, which can be copied to a `std::string` object.

Listing 4.27. Getting exported function names (`api-ext.cc`)

```
i::wasm::ModuleWireBytes
↪  module_bytes(compiled_module->native_module()->wire_bytes());
auto& export_table = compiled_module->module()->export_table;

for(auto& exported : export_table) {
  auto name = module_bytes.GetNameOrNull(exported.name);
  ret.function_names.emplace(std::string { name.data(), name.length() },
  ↪  exported.index);

  CompiledWasmFunction& func = ret.AddOneFunction();
  func.name = std::string { name.data(), name.length() };
  func.func_index = exported.index;
  func.internal->function_handle =
  ↪  decltype(func.internal->function_handle)::null(); //the_function;
}
```

4.5.5 Wasm Memory and Global Variable Accessor

The implementation for memory and global variable accessor in V8 is equivalent to the SpiderMonkey. Memory and global variable have their object representation internally, which can be supplied as import arguments.

Listing 4.28 shows the implementation to initialize a memory object. Similar to SpiderMonkey, a buffer object is backing the memory object. This buffer object is passed to the factory function for `WasmMemoryObject`. The buffer must be initialized according to the memory size as specified by the module.

Getting the memory buffer from the `WasmMemoryObject` is straightforward. The function needs to access the backing store of the memory object via its accessor function. The accessor function returns a managed pointer in a `std::shared_ptr` object. The `std::shared_ptr` object is a reference-counted managed pointer, which

Listing 4.28. Initializing Wasm memory in V8 (api-ext.cc)

```
namespace i = v8::internal;
void CompiledWasm::NewMemoryImport(v8::Isolate* i) {
  i::wasm::WasmModule const* wasm_module =
  ↪  this->internal->module_object->module();
  i::Isolate* isolate = reinterpret_cast<i::Isolate*>(i);

  // Initialize backing store
  auto initial_pages = wasm_module->initial_pages;
  auto maximum_pages = wasm_module->has_maximum_pages ?
  ↪  wasm_module->maximum_pages : initial_pages * 10;
  auto shared_flags = wasm_module->has_shared_memory ? i::SharedFlag::kShared
  ↪  : i::SharedFlag::kNotShared;
  auto backing_store = i::BackingStore::AllocateWasmMemory(isolate,
  ↪  initial_pages, maximum_pages, shared_flags);

  // Allocate JSArrayBuffer
  auto array_buffer_mem =
  ↪  isolate->factory()->NewJSArrayBuffer(std::move(backing_store));

  // New WasmMemoryObject
  auto wasm_memory_object = i::WasmMemoryObject::New(isolate,
  ↪  array_buffer_mem, maximum_pages);

  this->internal->wasm_memory_object = wasm_memory_object;
}
```

tracks the number of reference to the object. Therefore, it is crucial to transfer the reference counting to the returned buffer address.

To maintain this property, the V8 instrumentation does not return a naked pointer. Instead, it encapsulates the buffer pointer inside a `std::shared_ptr`, referring to the counter managed by the `BackingStore` object. Listing 4.29 presents the implementation of accessing the Wasm memory buffer.

Listing 4.29. Initializing Wasm memory in V8 (api-ext.cc)

```
struct WasmMemoryRef {
  std::shared_ptr<uint8_t> buffer;
  size_t length;
};

WasmMemoryRef CompiledWasm::GetWasmMemory() {
  if(this->internal->wasm_memory_object.is_null()) {
    return { nullptr, 0 };
  }
  auto array_buffer = this->internal->wasm_memory_object->array_buffer();
  std::shared_ptr<v8::BackingStore> backing_store_ptr =
  ↪  array_buffer.GetBackingStore();

  return { { backing_store_ptr, (uint8_t*)backing_store_ptr->buffer_start()
  ↪  }, backing_store_ptr->byte_length() };
}
```

The global variable initialization process in V8 is also equivalent to SpiderMonkey. The implementation iterates the list of imports and initializes the global variable object for every global variable import. Listing 4.30 shows the

implementation of initializing the global variable object.

Listing 4.30. Initializing Wasm global variable in V8 (`api-ext.cc`)

```
void CompiledWasm::NewGlobalImport(v8::Isolate* i) {
  if(this->internal->global_import_available)
    return; // Already imported

  i::wasm::WasmModule const* wasm_module =
  ↪  this->internal->module_object->module();
  i::Isolate* isolate = reinterpret_cast<i::Isolate*>(i);
  auto native_module = this->internal->module_object->native_module()
  i::wasm::ModuleWireBytes module_bytes(native_module->wire_bytes());

  // Iterate all global imports
  for(auto& import : wasm_module->import_table) {
    if(import.kind == i::wasm::kExternalGlobal) {
      auto name_b = module_bytes.GetNameOrNull(import.field_name);
      std::string global_name { name_b.begin(), name_b.end() };

      auto global =
        std::find_if(wasm_module->globals.begin(),
        ↪  wasm_module->globals.end(),
                    [idx = import.index] (auto& global)
                    { return global.index == idx; } );

      if(global == wasm_module->globals.end()) {
        std::cerr << "Null: " << global_name << std::endl;
      }

      auto global_type = ExtTyFromInternalTy(global->type.kind());
      auto global_object = i::WasmGlobalObject::New(isolate, {}, {},
      ↪  global->type, 0, global->mutability).ToHandleChecked();
      this->globals.emplace(global_name, global_type);
      this->internal->wasm_global_list.emplace(
            std::move(global_name),
            GlobalEntry { global_type, global_object });
    }
  }
  this->internal->global_import_available = true;
}
```

The global variable accessor and mutator implementation are also more straightforward than in SpiderMonkey. The global variable object provides a simple accessor and mutator function without any proxy object. Hence, the global variable can be read and modified directly using a simple function call. Listing 4.31 displays the implementation of the accessor and mutator function for the global variable.

4.5.6 Wasm Function Invoker

As mentioned in Section 2.6.4, Wasm function is invoked through a regular JS function pipeline. The caller must obtain the function handle to the target Wasm function, and invoke it through the static `Call` function from the JS execution component. The V8 engine does not differentiate regular JS function and Wasm function from the perspective of the embedder, although internally, V8 handles

Listing 4.31. Accessor and mutator of Wasm global variable in V8 (`api-ext.cc`)

```
void v8::ext::CompiledWasm::SetGlobalImport(std::string const& name,
↪  WasmGlobalArg value) {
  auto& global_list = this->internal->wasm_global_list;
  auto global_iter = global_list.find(name);
  if(global_iter != global_list.end()) {
    auto& global_ = global_iter->second;
    switch(global_.type) {
      case WasmType::I32: global_.global_object->SetI32(value.i32); break;
      case WasmType::I64: global_.global_object->SetI64(value.i64); break;
      case WasmType::F32: global_.global_object->SetF32(value.f32); break;
      case WasmType::F64: global_.global_object->SetF64(value.f64); break;
      case WasmType::Void: UNREACHABLE();
    }
  }
}

auto v8::ext::CompiledWasm::GetGlobalImport(std::string const& name)
      -> std::pair<WasmType, WasmGlobalArg> {
  WasmGlobalArg value;
  auto& global_list = this->internal->wasm_global_list;
  auto global_iter = global_list.find(name);
  if(global_iter != global_list.end()) {

    auto& global_ = global_iter->second;
    switch(global_.type) {
      case WasmType::I32: value.i32 = global_.global_object->GetI32(); break;
      case WasmType::I64: value.i64 = global_.global_object->GetI64(); break;
      case WasmType::F32: value.f32 = global_.global_object->GetF32(); break;
      case WasmType::F64: value.f64 = global_.global_object->GetF64(); break;
      case WasmType::Void: UNREACHABLE();
    }
    return std::make_pair(global_.type, value);
  }
  return std::make_pair(E::Void, value);
}
```

both differently.

Listing 4.32 shows the procedure to obtain the Wasm function handle. V8 prepares the handle on-demand. It means that the V8 does not create the JS function handle during the Wasm module compilation. The caller obtains the function handle through the `GetOrCreateWasmExternalFunction`, which accepts the Wasm module object and the desired function index. This function internally caches the function handle. Hence, it is safe to be called directly multiple times. However, our implementation also caches the function handle internally to save several function calls.

The invocation of the function handle uses the `Call` function provided by the `Execution` class. The `Call` function accepts the Isolate object, the function handle, and the pointer to the argument array. The third parameter of the function is a receiver object and must be supplied with an undefined value. A receiver object is an instance to a JS object that is analogous to a `this` object in an object-oriented language. It is only relevant in the JS context. The caller can supply the argument array through a heap-based array, including from `std::vector` object.

Listing 4.32. Obtaining the function handle (`api-ext.cc`)

```
namespace i = v8::internal;
auto CompiledWasmFunction::Invoke(v8::Isolate* i, std::vector<Local<Value>>&
↪  args) const
  -> std::tuple<MaybeLocal<Value>, uint64_t> {
  i::Isolate* isolate = reinterpret_cast<i::Isolate*>(i);

  // Check if the function handle is already obtained
  if(this->internal->function_handle.is_null()) {
    auto module_instance = this->parent.get().internal->module_instance;
    if(!module_instance.is_null()) {
      i::Handle<i::WasmExternalFunction> the_function =
              i::WasmInstanceObject::
              GetOrCreateWasmExternalFunction(isolate, module_instance,
              ↪  this->func_index);
      this->internal->function_handle = the_function;
    } else {
      // Module is not instantiated yet
      return { MaybeLocal<Value>{}, 0 };
    }
  }

  // Invoke the actual function
}
```

The argument array element type must be a Value object. Listing 4.33 presents the procedure to invoke the Wasm function handle.

Listing 4.33. Invoke the function (`api-ext.cc`)

```
i::Handle<i::Object> undefined = isolate->factory()->undefined_value();

std::chrono::steady_clock::time_point start_time =
↪  std::chrono::steady_clock::now();

i::MaybeHandle<i::Object> retval =
  i::Execution::Call(isolate, this->internal->function_handle, undefined,
  ↪  (int) args.size(),
  ↪  reinterpret_cast<i::Handle<i::Object>*>(args.data()));

std::chrono::steady_clock::time_point end_time =
↪  std::chrono::steady_clock::now();

uint64_t elapsed =
↪  std::chrono::duration_cast<std::chrono::nanoseconds>(end_time -
↪  start_time).count();

// Null return value means an error
if (retval.is_null()) {
  isolate->clear_pending_exception();
  return { MaybeLocal<Value> {}, elapsed };
}

v8::Local<v8::Value> result;
if(!v8::ToLocal(retval, &result))
  return { MaybeLocal<Value> {}, elapsed };
else
  return { result, elapsed };
```

4.5.7 Wasm Function Metadata and Instruction Bytes

As discussed in Section 4.5.1, obtaining a function signature in V8 is more straightforward. The function signature is stored in a FunctionSig object that is directly accessible from the WasmFunction object. The Wasm module stores all WasmFunction objects in a vector ordered by its function index. After obtaining the pointer to the respective FunctionSig, the accessor function can access the return type and parameter type of the function. Listing 4.34 presents the implementation for getting the function return type and parameter list.

Listing 4.34. Getting function paremeters and return type (`api-ext.cc`)

```
WasmType CompiledWasmFunction::ReturnType() {
  auto wasm_func_sig = this->parent.get().internal
                            ->module_object->module()
                            ->functions[this->func_index].sig;
  if(wasm_func_sig->return_count() == 0)
    return ext::WasmType::Void;
  return ExtTyFromInternalTy(wasm_func_sig->GetReturn().kind());
}

std::vector<WasmType> CompiledWasmFunction::Parameters() {
  auto wasm_func_sig = this->parent.get().internal
                            ->module_object->module()
                            ->functions[this->func_index].sig;

  std::vector<v8::ext::WasmType> ret;
  for(auto& kind_obj : wasm_func_sig->parameters()) {
    ret.push_back(ExtTyFromInternalTy(kind_obj.kind()));
  }
  return ret;
}
```

Getting the instruction bytes in Wasm is also straightforward. The program needs to obtain the `WasmCode` object from the `NativeModule` object. The `WasmCode` object contains the pointer to the actual location of the machine instruction, which then can be copied trivially through a `memcpy` function. Listing 4.35 presents the operation to obtain the machine instruction of a Wasm function.

In fact, V8 provides a mechanism to dump a Wasm instruction to the standard output. The `WasmCode` class provides a `Print` function, which can disassemble and print the machine instruction. However, this functionality is absent in SpiderMonkey. It is preferable to obtain the actual instruction bytes and disassemble it using a uniformed mechanism, which will be discussed in the later chapter.

4.6 Test Case Generator

4.6.1 V8 Fuzzer Suite

The V8 projects already incorporate fuzz testing to its software development. The V8 engine is equipped with several fuzzing toolkits that are integrated to the V8

Listing 4.35. Getting machine instruction of a function (`api-ext.cc`)

```
std::vector<uint8_t> CompiledWasmFunction::Instructions() const {
  i::wasm::WasmCodeRefScope ref_scope;
  auto wasm_code = this->parent.get().internal
                    ->module_object->native_module()
                    ->GetCode(this->func_index);

  // Marshall out the data
  std::vector<uint8_t> ret;
  size_t len = wasm_code->instructions().length();
  ret.resize(len);
  std::memcpy(ret.data(), wasm_code->instructions().data(), len);
  return ret;
}
```

engine. The particular exciting toolkits are the Wasm fuzzing tools, which we can find in the `test/fuzzer` folder of the source.

V8 uses the LLVM fuzz testing framework, the libFuzzer. The V8 fuzzer is developed using the libFuzzer programming standards, which can integrate smoothly with the entire libFuzzer pipeline. Although our experiment uses a customized fuzzing pipeline, the libFuzzer pattern simplifies the investigation of the fuzz testing library.

libFuzzer uses a very simplistic approach to code a fuzz tester for a program. libFuzzer produces a fuzzer program that links to a predefined C-based function called `LLVMFuzzerTestOneInput`. The fuzz tester program must define this function, which becomes the entry point for the fuzz tester. This function has only two parameters: a pointer to a random data bytes, and the length of the random data. libFuzzer automatically generates the random bytes and calls the fuzz function. The fuzz test function must handle the test sequence accordingly. For example, pass the random bytes as an input to the SUT.

The investigation starts from the folder that stores the fuzzer toolkit. The fuzzer toolkit for Wasm is clearly named, which makes it easily distinguishable. The particular interesting code is in the `wasm-compile.cc`. This fuzzer toolkit contains an extensive code that generates a Wasm program from input bytes. The user of this code can call `GenerateModule` function to create a valid Wasm module binary for any given input bytes [`wasm-compile.cc`:1605]. It is directly apparent that the experiment can use this code to generate a random Wasm program. It simplifies many development processes for the entire experiment.

By examining the entire fuzzer code, V8 performs fuzz testing to its Wasm compiler component. The `wasm-compile.cc` fuzzer contains the test sequence to generate a random Wasm module and pass it to the compile function [`wasm-fuzzer-common.cc`:264]. If the compiler successfully compiles the module, the test sequence instantiates the module. Then, the test executes the main function that the Wasm generator always generates [`wasm-compile.cc`:1655]. Overall, the test evaluates a full Wasm workflow, from the compilation, instantiation, and

execution. However, we can conclude that the test does not check for any behavior monitoring, and more to validate that the Wasm workflow does not fail or crash. It becomes one of the motivations for the experiment presented in this thesis.

4.6.2 Modifying the Wasm Generator

The V8 Wasm generator is not directly applicable to the experiment. The generator is being progressively developed along with the Wasm component development. The generator utilizes and generates several novel Wasm instruction that is yet to be implemented in the JS engines. Therefore, we need to modify the random generator to satisfy our experiment requirement.

The `GenerateModule` function uses the `WasmModuleBuilder` class for writing the module binary. This class provides functions to write Wasm module elements, which include global variables, memories, and functions. Firstly, the function generates the number of functions for the module and their signatures. Then the function generates the global variables for that module. The number of functions and global variables depends on the random bytes provided by the caller.

Listing 4.36 shows the `GenerateModule` function and its prologue procedures. The important parameters to consider are zone, data, and buffer. Zone is a specialized allocator that the generator uses for its operation. Zone performs memory allocation management, separate from the system-default memory allocation function. It aims to save resource-consuming memory allocation for allocating small-but-many memory space, and perform a faster memory reclaiming when the memory is no longer required. Zone itself does not provide a deallocator or `delete` function. It can only allocate memory. The deallocation occurs when the Zone object is destroyed, which also destroys all memory allocated inside the Zone.

The `ZoneBuffer` object stored in the buffer parameter is the location to store the binary module. The caller of the `GenerateModule` function can expect the module bytes to be written in the buffer. Finally, the data parameter contains the random bytes for generating the Wasm module. The data is encapsulated in the `DataRange` object. `DataRange` object performs an internal accounting to supply random bytes as requested. It provides a get function that can obtain a random byte according to the requested bit sizes. Listing 4.36 also shows the uses of obtaining bytes from the `DataRange` object that is used to compute the number of generated functions. This instruction is used throughout the Wasm module generation process that involves a randomized decision-making process.

Nevertheless, the original generator does not generate an imported or exported global variable. In order to satisfy requirement §2.2, the generated module must have a visible global variable. Therefore, we need to modify the generator to produce an imported global variable declaration. Instead of using `AddGlobal` function from `WasmModuleBuilder`, we use the `AddGlobalImport` function to declare an imported global variable. Listing 4.37 shows the modified snippet to add this

Listing 4.36. The GenerateModule function and its prologue snippets (`wasm-compile.cc`)

```
bool WasmCompileFuzzer::GenerateModule(
    Isolate* isolate, Zone* zone, Vector<const uint8_t> data,
    ZoneBuffer* buffer, int32_t* num_args,
    std::unique_ptr<WasmValue[]>* interpreter_args,
    std::unique_ptr<Handle<Object>[]>* compiler_args) {
  // Make WasmModuleBuilder object
  WasmModuleBuilder builder(zone);

  // Randomized bytes
  DataRange range(data);
  std::vector<FunctionSig*> function_signatures;

  // Setting memory configuration
  uint8_t min_memory = 10;
  uint8_t max_memory = 20;
  builder.SetMinMemorySize(min_memory);
  builder.SetMaxMemorySize(max_memory);
  builder.SetHasMemoryImport();

  // Generates functions
  static_assert(kMaxFunctions >= 1, "need min. 1 function");
  int num_functions = 1 + (range.get<uint8_t>() % kMaxFunctions);

  for (int i = 0; i < num_functions; ++i) {
    function_signatures.push_back(GenerateSig(zone, &range));
  }
  // ...
}
```

functionality.

The `WasmModuleBuilder` class does not generate the random instruction bytes. Instead, the `WasmGenerator` class is responsible for generating random Wasm instruction. The random generated random instruction is then passed to the module builder for writing. For every function signature generated, the `GenerateModule` calls the `WasmGenerator` to generate the instruction for the function. Listing 4.38 presents the snippet to generate the random Wasm function.

We modify the `WasmGenerator` class constructor to include the maximum memory size information. It is used to create safe memory access bound, which in turn generates a valid memory access instruction. The iteration also adds all generated functions to the export list. In the original implementation, only the first function is added to the export list as the main function. In our experiment, we try to invoke the function randomly multiple times to increase the probability of yielding a different behavior.

The random generator uses random bytes value to direct the random generation process. The generator takes some bytes value and uses it to select the alternative instruction to generate. The process continues until it consumes the entire random bytes or reaches a specific complexity limit. The generation process exploits Wasm ISA that resembles a tree structure, which allows the generation

Listing 4.37. Generating imported global variables (`wasm-compile.cc`)

```
// Get how many globals to generate
int num_globals = range.get<uint8_t>() % (kMaxGlobals + 1);
std::vector<ValueType> globals;
std::vector<uint8_t> mutable_globals;
globals.reserve(num_globals);
mutable_globals.reserve(num_globals);

// Important to maintain the allocated string that is passed to the
// WasmModuleBuilder. WasmModuleBuilder does not take the ownership
// of the passed string, nor make any copy of it. Therefore, the
// string passed to the builder must be retained until the module
// bytes are written.
std::list<std::string> globalNames;

for (int i = 0; i < num_globals; ++i) {
  // Get a new random value type
  ValueType type = GetValueType(&range);

  // Generate new global variable name
  std::string newGlobalName { "global" };
  newGlobalName += std::to_string(i);
  globalNames.emplace_back(std::move(newGlobalName));

  // Add the actual global import
  builder.AddGlobalImport(CStrVector(globalNames.back().c_str()), type,
  ↪  mutability, CStrVector(""));
  globals.push_back(type); // Push the type of the global

  // All imported globals are mutable
  mutable_globals.push_back(static_cast<uint8_t>(i));
}
```

process to use a Depth-First Search (DFS) based algorithm.

The generation begins by calling the `WasmGenerator`'s `Generate` function. The `GenerateModule` function calls the `Generate` function overload with a `Vector` argument. This `Vector` consists of the expected return type of the function. In our experiment, the return type is limited to one only, as the multiple return type is not yet implemented in all JS engines. Limiting the return type size is done by setting the constant `kMaxReturns` to 1, which is located in the beginning of the source file. Accordingly, this `Generate` function then invokes another `Generate` function overload by providing the expected return type.

This overloaded `Generate` function selects the appropriate `Generate` function for the expected return type. The implementation has several variants of specialized `Generate` function according to the Wasm types. For every variant, the function selects a new instruction by taking a random value and produce the required argument by recursively calling `Generate` function again. The recursive call terminates when the `Generate` function choose a constant operand or reaches the recursion limit.

This recursive process ensures the generator produces a valid Wasm module according to the specification. The `Generate` function only generates a valid

Listing 4.38. Generating Wasm functions (`wasm-compile.cc`)

```
// Cache function names
std::list<std::string> funcNames;
for (int i = 0; i < num_functions; ++i) {
  std::string newFuncName { "func" };
  newFuncName += std::to_string(i);
  funcNames.emplace_back(std::move(newFuncName));
  auto& funcName = funcNames.back();

  DataRange function_range = i == num_functions - 1 ? std::move(range) :
  ↪  range.split();

  FunctionSig* sig = function_signatures[i];
  WasmFunctionBuilder* f = builder.AddFunction(sig);

  WasmGenerator gen(f, function_signatures, globals, mutable_globals,
  ↪  &function_range, max_memory);

  Vector<const ValueType> return_types(sig->returns().begin(),
  ↪  sig->return_count());
  gen.Generate(return_types, &function_range);

  f->Emit(kExprEnd);

  // Add function to the export list
  builder.AddExport(CStrVector(funcName.c_str()), f);
}
// Write all Wasm module to buffer
builder.WriteTo(buffer);
```

instruction sequence, which eventually produces a specified stack state. Each `Generate` function specialization also specifies every instruction alternatives that can be taken by the chosen random path. Considering the alternatives may contain an unimplemented Wasm instruction, we disabled several instruction alternatives. Those instructions include atomic instructions, SMID instructions, and floating-point saturation instruction. This procedure satisfies our requirement §1.2 for the random test case generator.

The final modification made to the Wasm generator is the memory operation. The original Wasm generator always generates a random operand for the memory operation. This behavior generates invalid memory access for almost every generated Wasm program. Since the experiment expects the module to behave correctly most of the time, the generator must produce a correct runtime behavior of the program. The original generator only ensures the valid static properties of the program, i.e., a well-formed Wasm program adhering to the validation rules.

To achieve this behavior, we modify the memory operation generator to generate a bounded offset value for the memory index operand. It is achieved by adding a modulo operation to the maximum memory size for every memory index operand. It ensures the memory index always within the valid memory range, thus, allowing the generator to produce a program that terminates correctly. Listing 4.39 shows the modification introduced to the memory instruction generator.

Listing 4.39. Generating bounded memory instructions (`wasm-compile.cc`)

```
// Recursive template, generate index also with the trailing extra
// arguments, e.g., some value to be stored to the memory
template<typename ValueType::Kind kind, typename ValueType::Kind...
↪  arg_types>
void WasmGenerator::GenerateWithBound(DataRange* data) {
  GenerateWithBound<ValueType::kI32>(data);
  Generate<arg_types...>(data);
}

// Base template, generate index without trailing extra arguments
template<>
void WasmGenerator::GenerateWithBound<ValueType::kI32>(DataRange* data) {
  Generate<ValueType::kI32>(data);
  builder_->EmitI32Const((int32_t)max_memory_);
  builder_->Emit(kExprI32RemU); // Emit modulo operation
}

template <WasmOpcode memory_op, ValueType::Kind... arg_types>
void memop(DataRange* data) {
  const uint8_t align = data->get<uint8_t>() % (max_alignment(memory_op) +
  ↪  1);
  const uint32_t offset = data->get<uint32_t>() % max_memory_;

  // Generate the index and the arguments, if any.
  GenerateWithBound<ValueType::kI32, arg_types...>(data);

  if (WasmOpcodes::IsPrefixOpcode(static_cast<WasmOpcode>(memory_op >> 8)))
  ↪  {
    builder_->EmitWithPrefix(memory_op);
  } else {
    builder_->Emit(memory_op);
  }
  builder_->EmitU32V(align);
  builder_->EmitU32V(offset);
}
```

Since the generated memory address is bounded to the maximum declared memory in the module, there are still 50% chances the instruction yields invalid memory access. It is due to the initial memory is set as half of the maximum memory. Therefore, the remaining half of the memory address space is invalid unless the memory is expanded through a `memory_grow` instruction inside the Wasm program. Through this property, the generated program also covers the cases of programs with invalid memory access.

4.6.3 Exposing the Fuzzer Library

The V8 fuzzer library is not intended to be accessible through any part of the JS engine. Although it utilizes V8 engine components, its code is isolated on its own compilation unit. The V8 build script also generates the fuzzer executable separately. Therefore, we must refactor the fuzzer library to allow external code accessing its API.

The first required step is exposing the `GenerateModule` function in

`WasmCompileFuzzer` class. V8 defines the `WasmCompileFuzzer` class in the source file, meaning that it is inaccessible from any other part of the engine. We need to introduce the `WasmCompileFuzzer` class in an accessible header file, allowing our instrumentation code to access the class. By moving the `WasmCompileFuzzer` class definition to the wasm-fuzzer-common.h header file, the class is accessible from other source files.

The second important step is introducing the fuzzer library compilation unit to the rest of V8 engine library. Since we want to introduce the random generator function in the instrumented API, the engine must be able to link to the fuzzer library. It is done by modifying the build script by combining the V8 compilation unit and the fuzzer compilation unit. Without this modification, the compilation breaks as the compiler is unable to link the instrumented API to the fuzzer library.

The build script modification is done by introducing a new V8 build artifact. The build script uses `v8_component` macro to introduce a new build artifact. In this artifact, we include our instrumentation API sources and the dependencies, which are the V8 base library and the fuzzer components. This build script produces a new shared library artifact in the build directory. The embedder must link to this separate artifact in order to utilize the instrumentation API. Listing 4.40 presents the new build script section.

Listing 4.40. New build script for the V8 instrumentation (`BUILD.gn`)

```
v8_component("v8_ext_diff_fuzz") {
  sources = [
    "src/api/ext-api.cc",
    "src/api/ext-api.h",
  ]

  deps = [
    ":v8_base_without_compiler",
    ":v8_compiler",
    ":v8_snapshot",
    ":lib_wasm_fuzzer_common",
    ":wasm_module_runner",
    ":wasm_compile_fuzzer",
    ":fuzzer_support"
  ]

  configs = [ ":internal_config" ]
}
```

Similar to other API, we introduce a function in our instrumented API. The function accepts a random byte array and produces a valid Wasm binary stored in an array. The function simply calls the `GenerateModule` function of the `WasmCompileFuzzer` class with the proper arguments. The resulting module binary is copied to the output array. Listing 4.41 shows the implementation of the API.

Listing 4.41. GenerateRandomWasm function (`ext-api.cc`)

```
namespace i = v8::internal;
std::tuple<bool, size_t> GenerateRandomWasm(v8::Isolate* i,
↪  std::vector<uint8_t> const& input, std::vector<uint8_t>& output) {
  i::Isolate* isolate = reinterpret_cast<i::Isolate*>(i);

  // Wrap the vector to V8 Vector
  i::Vector<const uint8_t> data { input.data(), input.size() };

  // Zone objects
  i::AccountingAllocator allocator;
  i::Zone zone(&allocator, ZONE_NAME);
  i::wasm::ZoneBuffer buffer(&zone);
  int32_t num_args = 0;

  // Placeholders out variable
  std::unique_ptr<i::wasm::WasmValue[]> interpreter_args;
  std::unique_ptr<i::Handle<i::Object>[]> compiler_args;

  i::wasm::fuzzer::WasmCompileFuzzer compilerFuzzer;
  if (!compilerFuzzer.GenerateModule(isolate, &zone, data, &buffer,
  ↪  &num_args, &interpreter_args, &compiler_args)) {
    return {false, 0}; // Failed
  }

  // Fast marshall to output
  auto generatedSize = buffer.size();
  output.resize(generatedSize);
  std::memcpy(output.data(), buffer.data(), generatedSize);

  // The compiler_args is "instrumented" to carry the memory size
  decltype(auto) mem_size_ret = compiler_args[0];
  decltype(auto) mem_size = i::Handle<i::Smi>::cast(mem_size_ret);
  auto mem = mem_size->value();

  return {true, mem};
}
```

4.6.4 Embedding the Generator

For simplifying the development, we do not develop the generator program within the V8 engine. Instead, we use the fuzzer generator component as if we use the V8 engine itself. Therefore, we step through the similar V8 engine embedding steps to build the Wasm generator.

In order to satisfy requirement §1.1 for test case consistency, the generator must use a stable random number generator to generate the input random bytes. A stable random number generator ensures the test case can be reproducible after the test has been performed. With this capability, the experiment does not need to store the entire test case data. Reproducing a test case requires only a valid configuration, i.e., the seed to regenerate the random test case.

The generator uses the C++ standard library for the pseudo-random generator (PRG). The standard library provides a PRG based on the Mersenne Twister algorithm, which a program can directly utilize without an additional library.

The PRG API is very straightforward and accepts a user-defined seed. Therefore, reproducing a state of the PRG to regenerate the test case is trivial. Listing 4.42 shows the implementation of a single test case generator.

Listing 4.42. Generator function in the embedder (`ext-api.cc`)

```
size_t GenerateRandomWASM(CommandLineArgument& args,  std::vector<uint8_t>&
↪  randomizedData, std::mt19937& re, v8::Isolate* isolate) {
  // PRECONDITION: randomizedData is already pre-initialized with the
  // correct total bytes requested

  // Cast to uint32_t* pointer, to access it as uint32_t array
  auto dataBeginAsInt32 = (uint32_t*) randomizedData.data();

  // The array length is 1/4 the original size
  auto dataEndAsInt32 = dataBeginAsInt32 + randomizedData.size() /
  ↪  sizeof(uint32_t);

  // Generate the random bytes
  std::for_each(dataBeginAsInt32, dataEndAsInt32, [&] (uint32_t& val) { val =
  ↪  re(); });

  // Call random instrumented random Wasm generator
  std::vector<uint8_t> generatedWasm;
  auto [success, mem_size_ret] = v8::ext::GenerateRandomWasm(isolate,
  ↪  randomizedData, generatedWasm);

  if(!success)
    std::abort();

  // Write the Wasm bytes to output file
  std::ofstream output(args.outfile, std::ios::out);
  output.write((char const*)generatedWasm.data(), generatedWasm.size());
  output.flush();

  return mem_size_ret;
}
```

We can take several design considerations to minimize the wasted random bytes and seed. The PRG API generates a random 32-bit that is returned as an unsigned integer. The generator maximizes the utilization of all random bits by storing the entire 32-bit value to the random byte buffer. Since the buffer is managed in an 8-bit value vector, the generator must cast the vector to a 32-bit vector to simplify the assignment instruction.

Additional efficiency can be achieved by using the seed to generate multiple random Wasm module. The random generator program is designed to continue running during the entire test sequence. Therefore, the random program can keep using the seed to generate the subsequent random case. In order to reproduce the specific test case, the generator skips the bytes of the previous test case. Listing 4.43 shows the implementation of the generator program.

The generator program accepts a command from the standard input. The command triggers the test case generation to the output file. Therefore, the generator program can run independently across multiple test sequences and be

Listing 4.43. Generator function in the embedder (`ext-api.cc`)

```
std::mt19937 re(args.randomSeed);
size_t mem_size;

// Go to specific test case point
if(args.skipCount != 0) {
  int skippedByteCount = args.skipCount * args.randomSize /
  ↪  sizeof(uint32_t);
  for(int i = 0; i < skippedByteCount; ++i)
    re();
}

std::vector<uint8_t> randomizedData;
randomizedData.resize(args.randomSize);

std::string input;
if(!args.repro) {
  while(true) {
    // Accept command
    std::getline(std::cin, input);
    if(input == "q")
      break;
    else if(input == "w")
      mem_size = GenerateRandomWASM(args, randomizedData, re, isolate);
  }
} else {
  // Only reproduce a single test case
  mem_size = GenerateRandomWASM(args, randomizedData, re, isolate);
}
```

isolated from the SUT as specified by requirement §1.3.

In order to improve the efficiency and the speed of test case generation, the output file for the testing should be stored in the `/dev/shm` folder in Linux. `/dev/shm` folder is a temporary file system that is stored in the memory. It dramatically reduces the disk latency of test case generation. It also simplifies the communication between the generator and the SUT because it behaves similarly to a regular file.

4.7 Shell Program

4.7.1 Common Interface

Two SUTs involved in the experiment have different instrumentation and embedding mechanism. To minimizing the development complexity, the test environment is designed with a common interface in mind. The common interface serves as the unification of the test sequence logic, which defines the general infrastructure to execute the test procedure. This common interface also satisfies requirement §3.

The main interface developed for the experiment is the shell program for the JS engine embedder. The shell program provides the functionality to execute test sequences, also to reproduce the experiment interactively. The shell program provides an concept class specification that needs to be implemented by each JS

engine embedder. Listing 4.44 shows the definition of the concept class, and Table 4.2 presents the details of the specification.

Listing 4.44. Concept class for shell program (`runner-common.h`)

```
class FuzzerRunnerImplementation {
public:
  bool InitializeModule(dfw::FuzzerRunnerCLArgs const&);
  bool InitializeExecution();
  std::vector<FunctionInfo> const& Functions();
  std::optional<std::vector<uint8_t>> DumpFunction(std::string const&);
  std::tuple<std::optional<dfw::JSValue>, uint64_t>
  ↪  InvokeFunction(std::string const&, std::vector<JSValue> const&);
  bool MarshallMemoryImport(uint8_t*, size_t);
  std::vector<MemoryDiff> CompareInternalMemory(std::vector<uint8_t>&
  ↪  buffer);
  std::vector<GlobalInfo> Globals();
  void SetGlobal(std::string const& arg, JSValue value);
  JSValue GetGlobal(std::string const& arg);
};
```

Table 4.2. Specification details of the concept class

Interface	Action	Returns
InitializeModule	Compile the module bytes supplied in the file specified in the command line argument	Success status of the compilation
InitializeExecution	Instantiate the Wasm module to start receiving invoke function command	Success status of the instantiation
Functions	Obtain the list of exported functions in the module	Exported function list
DumpFunction	Get the function bytes for the specified function name	Instruction bytes of the requested function
InvokeFunction	Call a function by its function name with the specified function arguments	The called function return value, which is null in case of error, and the elapsed time of the function execution.
MarshallMemory Import	Copy the specified byte array to the Wasm memory	Success status of the memory copying process
CompareInternal Memory	Compare the internal memory supplied by the caller and update the memory cache with the updated state in the Wasm memory	The difference found between the memory values
Globals	Obtain the list of visible global variables through the Wasm import	List of global variables
SetGlobal	Set global variable	n/a
GetGlobal	Get global variable	The global variable value from the Wasm

The interface also standardizes the data structure used in the test system.

Listing 4.45 presents the important data structure definition used in the shell program. The interface redefines the enum class type to represent a Wasm type. The enum class is defined to be equal to the defined type in the JS engine instrumentation to allow easy conversion between the types. The interface also defines a polymorphic value type. This type is implemented using a discriminated union pattern to simplify the implementation without involving intricate object-oriented design. The interface also defines additional data types to encapsulate the information used in the testing process. It includes the function information, global variable information, and memory differences.

Listing 4.45. Standard data structure for the shell interface (`runner-common.h`)

```
enum class WasmType {
  Void,
  I32,
  I64,
  F32,
  F64
};

struct JSValue {
  WasmType type;
  union {
    uint32_t i32;
    uint64_t i64;
    float    f32;
    double   f64;
  };
};

struct FunctionInfo {
  std::string function_name;
  WasmType return_type;
  std::vector<WasmType> parameters;
};

struct GlobalInfo {
  std::string global_name;
  WasmType type;
};

struct MemoryDiff {
  uint32_t index;
  uint8_t old_byte;
  uint8_t new_byte;
};
```

The JS embedder program that implements the interface can start the shell by calling the `Run` function. The `Run` function is the entry point to the shell program, which processes the command line arguments and executes the user command. Note that the `Run` function must be called after the embedding procedure has been done, and the JS engine is ready to accept the command.

4.7.2 Interactive Shell

The test system implements an interactive shell to allow experimenting with Wasm module independently without following the test sequence. It helps the user to experiment with Wasm module compilation, function invocation, and machine instruction inspection. It is inspired by the interactive JS shell provided by JS engine implementation. However, this shell is more specialized and only accepts a limited set of commands to interact with Wasm modules. This interactive shell satisfies requirement §3.3 to support interactive experiment and test case reproduction.

The interactive shell uses a main-loop design pattern. The main loop waits for user command from the standard input and executes it accordingly. Table 4.3 presents the command that is implemented in the interactive shell. The shell implements the command by utilizing the instrumented JS engine via the standardized interface. In this way, both JS engines have the same interactive shell functionality, providing the embedder implements the standard interface correctly.

Table 4.3. Commands for the interactive shell

Command	**Action**
dump	Dump the disassembled machine instruction of the specified function
list	List the exported function in the module and its signature
instantiate	Instantiate the module for execution
invoke	Invoke the specified function and the provided arguments. The module must be instantiated first through instantiate command
memimport	Load a binary file and import it into the Wasm memory
listglobal	List the global variables available to access
setglobal	Set the value of global variables
getglobal	Get the value of global variables

For commands that accept value arguments, such as `invoke` and `setglobal` commands, the shell parses and box the value before it is passed to the JS engine via the standard interface. The shell also validates the argument type, particularly for the arguments provided for a function invocation. The shell rejects the invalid argument, such as supplying a floating-point string number to an integer argument or supplying a non-numerical character altogether.

4.7.3 Single Test Sequence

The test system uses the single test sequence workflow to execute the fuzz experiment. The workflow consumes a Wasm module binary, compiles the module, initializes the execution environment, and invokes exported function randomly

multiple times. The process produces the observation result, which can be stored and processed by the test coordinator.

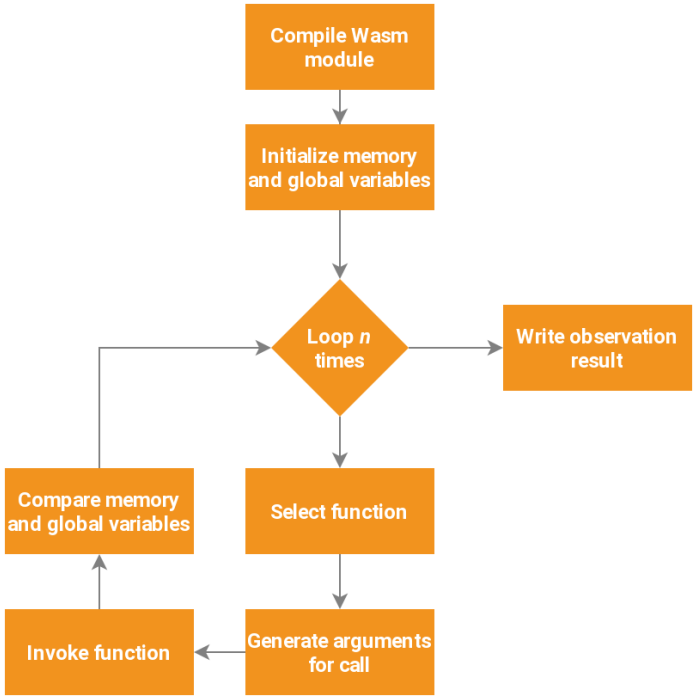

Figure 4.3. Workflow for single test sequence

Figure 4.3 shows the process workflow of the test sequence. The implementation can be seen on `SingleRun` function in `runner-common.cc` source file. The test starts by compiling the Wasm module through the instrumented API. If the compilation succeeds, the test system initializes the test environment. The environment is prepared by loading the Wasm memory from a pre-provided memory file. The test system also prepares a PRG using a pre-provided seed value, which is used to generate random values for the environment. The system uses this PRG to generate random global variable values, to select a function randomly to invoke during the test sequence, and to generate random function argument for the function invocation.

In each test iteration, the test system observes the post-invoke state of the Wasm. The observation involves memory and global variable comparison. The memory is scanned for each byte to observe the change, while the global variable is scanned per item. The system records the change and updates its internal cache to perform the subsequent observation. At the end of the test sequence, the system prepares the observation report, which is detailed by the test iteration. It allows the experiment to observe the progressive behavior caused by sequences of operations.

One crucial factor to consider for the test is the floating-point value used in the testing. The PRG generates a random 64-bit unsigned integer value, which does not always map accurately to the 64-bit floating-point value, especially the

Not-a-Number (NaN) value. The floating-point value system specifies a range of values that indicates a NaN value.

This particular property is exploited by JS engines to encode extra data within the NaN space. It allows the JS engine to save space for allocating primitive value. However, it raises an issue when an embedder tries to pass a NaN value with this bit combination. Therefore, the test system must flatten the NaN value to a "safe" value that is portable between system and environment. C++ library provides a standardized mechanism to detect a NaN value and supply a portable NaN value. In this way, all randomly generated floating-point value that falls into the NaN region is properly encoded an supplied to the JS engine. Listing 4.46 shows the snippet of this functionality.

Listing 4.46. Flattening NaN random value (`runner-common.h`)

```
template<>
inline double RandomGenerator::get<double>() {
  // Try getting value first
  uint64_t temp = get<uint64_t>();

  // This is undefined behavior, but works in many cases
  // double val = *reinterpret_cast<double*>(&temp);

  // Until C++20 standard is released with std::bit_cast,
  // this is the defined-behavior way to copy bitwise value
  // see: https://en.cppreference.com/w/cpp/language/reinterpret_cast
  double val;
  std::memcpy(&val, &temp, sizeof(val));

  // Check NaN value, and flatten the NaN if it is.
  if(std::isnan(val))
    return std::numeric_limits<double>::quiet_NaN();
  else
    return val;
}
```

Besides, the system also uses a bitwise value comparison when comparing the pre and post Wasm states. It is crucial since the comparison involves a floating-point value, which may involve imprecision if the system uses a regular comparison instruction.

4.8 Control Program

4.8.1 Integration Specification

The experiment runs the test independently and isolated from each other. It implies that it requires an integration which handles the coordination between independent test components to perform the entire test workflow. The integration standardizes the communication specification between components, particularly to transmit command and obtain the test result.

The test system provides a controller program that integrates all test compo-

nents, adhering to requirement §4. The controller program uses a basic interprocess communication and process management technique to control multiple test components. The controller program uses command-line arguments to transmit the test command to the shell program and retrieve the result.

For simplifying the development, the output transmission uses a file-descriptor based pipe. The file-descriptor based pipe allows programs to read and write data between process boundaries using simple stream-based operations. It also benefits the condition that the shell program is the child process of the controller program. This attribute allows the controller program to control the file descriptor of the child program directly.

Since the file-descriptor pipe is a regular text-based transmission, the transmitted data has no structure. Hence, the system needs to create a structured data format to simplify the communication process. The test system uses a JSON format to transmit the result data from the shell program to the controller program. By using a JSON format, the system can utilize a readily available JSON library to generate and process JSON data. It cuts the development time and significantly reduces the complexity of the program.

Listing 4.47. JSON structure for the result data

```
[ /* Array of test iteration */
  {
    "FunctionName": "func0", /* Invoked function*/
    "Args": [ /* Arguments in int64 value */
      "2203442879026261627",
      "2785472889303938557"
    ],
    "Elapsed": "98759", /* Elapsed time */
    "Success": false, /* Successful function call (no trap) */
    "MemoryDiff": { /* Difference in memory */
      /* "memory_index" : [bytes_before, bytes_after] */
      "512294": [159, 171],
      /* ... */
    }
    "GlobalDiff": { /* Difference in global */
      /* "global_name" : ["int64_before", "int64_after"] */
      "global1": ["159", "171"],
      /* ... */
    }
  }, /* ... */
]
```

Listing 4.47 presents the JSON schema for the result data. The root of the data is an array of test iteration. For every iteration, it specifies the called function, the specified argument, elapsed time, success state, and the difference of memory and global. The function argument is specified as an array of 64-bit integer, to maintain the argument bit state. It is to avoid rounding errors from floating-point representation. The memory difference is specified in key-value pairs. The key is the memory index, and the value is the byte value before and after the execution. The global variable difference is also specified similarly.

However, the value uses a 64-bit integer instead of a byte value.

4.8.2 Shell Spawner

The control program is responsible for launching the shell program for every test iteration. The procedure follows the basic POSIX system to launch a child process. The controller program forks itself to create a new process, then calls the exec function to launch the shell executable. The control program also passes the test commands through the exec function.

Before the exec function is called, the control program must configure the file descriptor (FD) for the communication between the control program and the shell program. The control program prepares a new FD instead of using `STDOUT` descriptor. It is to avoid conflicting with the JS engine that uses `STDOUT` to print error messages. The control program calls `pipe` function to prepare a new FD. The new FD from the `pipe` function is copied to the agreed FD value in the child process. Listing 4.48 shows the listing for spawning the child shell process, and Listing 4.49 shows the implementation to access the FD pipe in the shell program.

The spawn function returns the process ID (PID) and the FD. The PID is crucial as it allows the control program to track and finalize the child process accordingly. We must avoid spawning zombie processes while executing the experiment. Therefore, the process must be closed accordingly after each iteration completes. It also allows the control program to kill a shell process that does not terminate.

The generated Wasm program is not guaranteed to terminate. It is possible for the random program to enter an infinite loop, which the control program must take over and terminate manually. The control program waits for a specific duration before forcefully terminates the Wasm program. In this way, the experiment is guaranteed to always terminate for any test case input.

The control program implements this logic by incorporating a two-level thread. The first level thread is to parallelize multiple SUT implementation. Each JS engine has its separate shell program. The control program must parallelize the test sequence at this level to allow the test to execute parallelly. The first-level thread is responsible for spawning the child process and the second-level thread, which handles interprocess communication.

The second-level thread performs the interprocess communication by reading the output from the child process. The I/O operation is handled in a separate thread to allow non-blocking I/O operation and allow the first-level thread to monitor the state of the child process. The control program uses this programming design due to the unavailability of asynchronous I/O operation in the C++ standard library.

Figure 4.4 presents the swimlane diagram for the positive logic of a terminating test case. Figure 4.5, on the other hand, shows the logic of a non-terminating

Listing 4.48. Spawning the child shell process (`runner-coordinator.cpp`)

```
auto SpawnTester(std::string const& path, std::string const& input_wasm,
↪  std::string const& mem_path, std::string const& arg_seed) ->
↪  std::tuple<pid_t, int> {
  pid_t pid;

  // Prepare pipe
  int fd[2];
  pipe(fd);

  // Split
  pid = fork();

  if(pid == 0) {
    // Child process
    close(fd[0]); // Close STDIN

    // Copy to the common file descriptor
    dup2(fd[1], COMMON_FILE_DESCRIPTOR);

    // Redirect stdout and stderr to /dev/null so it is not
    // going to be printed in the controller program
    int stdnull = open("/dev/null", O_RDONLY);
    dup2(stdnull, STDOUT_FILENO);
    dup2(stdnull, STDERR_FILENO);

    // Close unused file descriptor
    close(fd[1]);
    close(stdnull);

    // Execute the shell program and supply the argument
    execl(path.c_str(), path.c_str(),
                        "-mode", "single",
                        "-input", input_wasm.c_str(),
                        "-memory", mem_path.c_str(),
                        "-arg-seed", arg_seed.c_str(),
                        (char*)0);
    std::abort(); // Error, not going to reach here if execl succeed
  } else {
    // Parent process
    close(fd[1]); // Close write
    return { pid, fd[0] }; // Return process id and pipe fileno
  }
}
```

test case, in which the control thread must take over the control and forcefully terminate the child process. The process is repeated for every test iteration.

4.8.3 Database Design

The control program handles the persistence of the result data. The program stores the result data from every test iteration in non-volatile storage to allow result analysis. The persistence uses a Relational Database, which provides a well-established framework to store and operate structured data.

In order to minimize the system requirement, the test system uses an in-process database management system (DBMS) with the SQLite library. The

Listing 4.49. Using the file descriptor in the shell program (`runner-common.cpp`)

```
std::ostream* output = &std::cout; // Default fallback to STDOUT

// Check if the FD is opened by the parent process
auto flag = fcntl(COMMON_FILE_DESCRIPTOR, F_GETFD);

// Wrap in C++ STL stream
__gnu_cxx::stdio_filebuf<char> filebuf_out(COMMON_FILE_DESCRIPTOR,
↪  std::ios::out);
std::ostream os(&filebuf_out);

// Use the specific FD if it is available
if(flag >= 0) {
  output = &os;
}
```

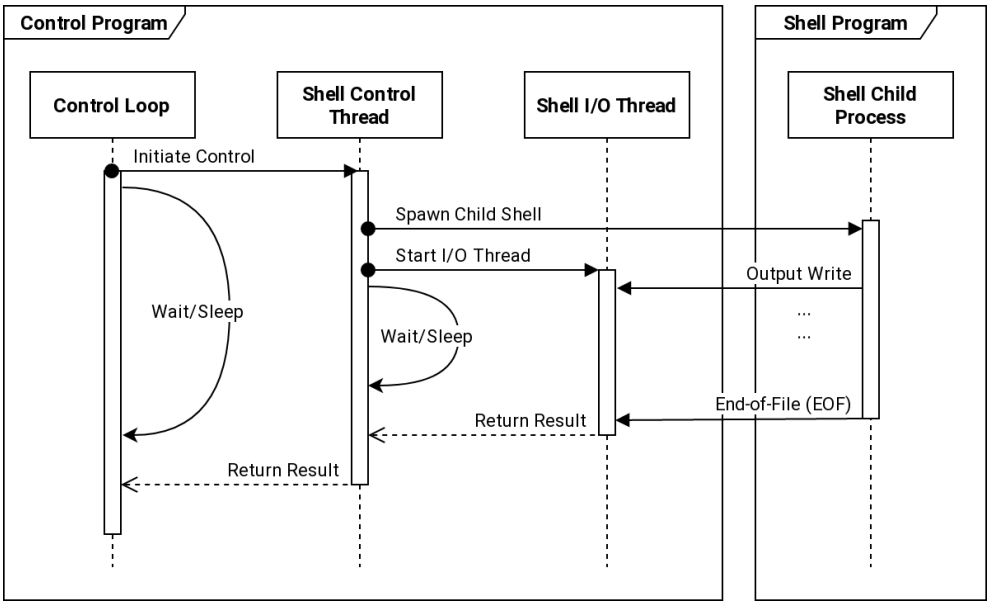

Figure 4.4. Swimlane diagram for control thread logic

SQLite allows a lightweight DBMS to be embedded into a program without requiring a standalone process to manage the database. It also provides an extensive SQL processing that is satisfactory for this experiment. The system also uses a C++ Object-Relational Mapping (ORM) library to support the database programming. This experiment uses a Quince-Lib library, which provides an adequate and intuitive API to operate an SQLite database in C++ programs.

The database is designed to persist the result data obtained from the test procedure. It stores the configuration of the test, which includes the seed used to generate the test case. The database design is normalized into separate entities to minimize the redundancy and duplication of the data. This approach reduces the disk space required to store the result data, which can grow to hundreds of megabytes after executing the test multiple times. Figure 4.6 shows the database design used in the testing system.

From the database design, we can create a SQL query to search the memory

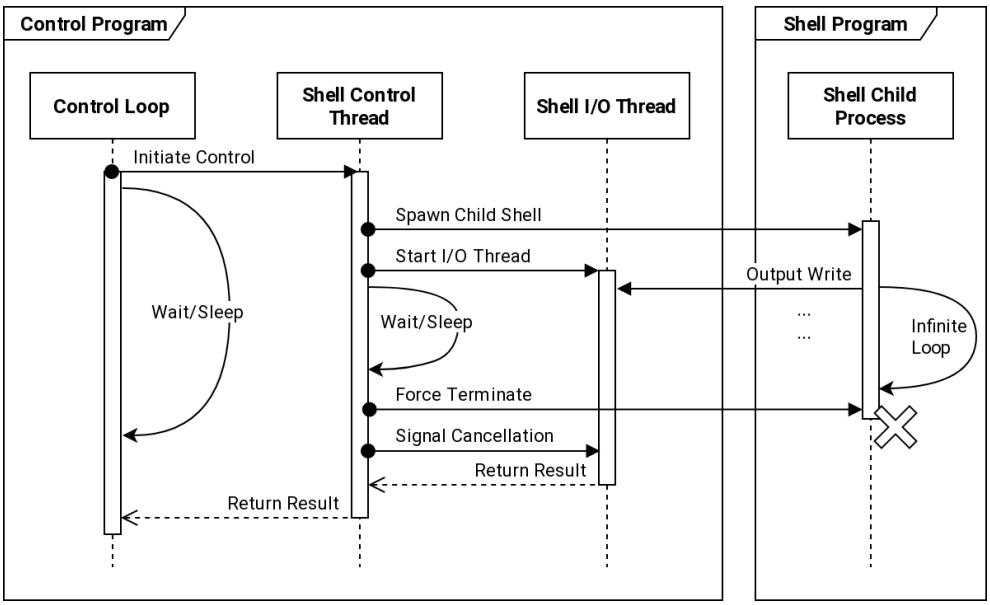

Figure 4.5. Swimlane diagram for control thread logic in case of non-terminating test

difference between two implementations that occurs in the experiment. Listing 4.50 shows the query to create the intermediate view to obtain this information. The `memorydiff_with_impl` view joins the memory differences in the experiment with the information of the SUT, i.e., the JS engine. The `memorydiff_flat_match` joins the result for each JS engine into a single table. Finally, the `memorydiff_search` view obtains the differences between both implementations. This same query is also used to obtain the global variable difference since it has the same entity structure.

Listing 4.50. SQL to query the memory difference

```
CREATE VIEW "memorydiff_with_impl" AS
SELECT m1.id AS id, functioncall_id, implementation_id, m1.'index', 'before',
↪  'after'
FROM testcases t, function_call fc, testcase_call tc, memorydiff_call m1
WHERE t.id = tc.testcase_id
AND fc.id = tc.functioncall_id
AND tc.id = m1.testcasecall_id;

CREATE VIEW "memorydiff_flat_match" AS
SELECT m1.functioncall_id, m1.'index', m1.'before' AS v8_before, m1.'after'
↪  AS v8_after, m2.'before' AS sm_before, m2.'after' AS sm_after, m1.id,
↪  m2.id
FROM memorydiff_with_impl m1, memorydiff_with_impl m2
WHERE m1.implementation_id = 1 AND m2.implementation_id = 2
AND (m1.functioncall_id = m2.functioncall_id AND m1.'index' = m2.'index');

CREATE VIEW "memorydiff_search" AS
SELECT * FROM memorydiff_flat_match
WHERE v8_before != sm_before OR v8_after != sm_after;
```

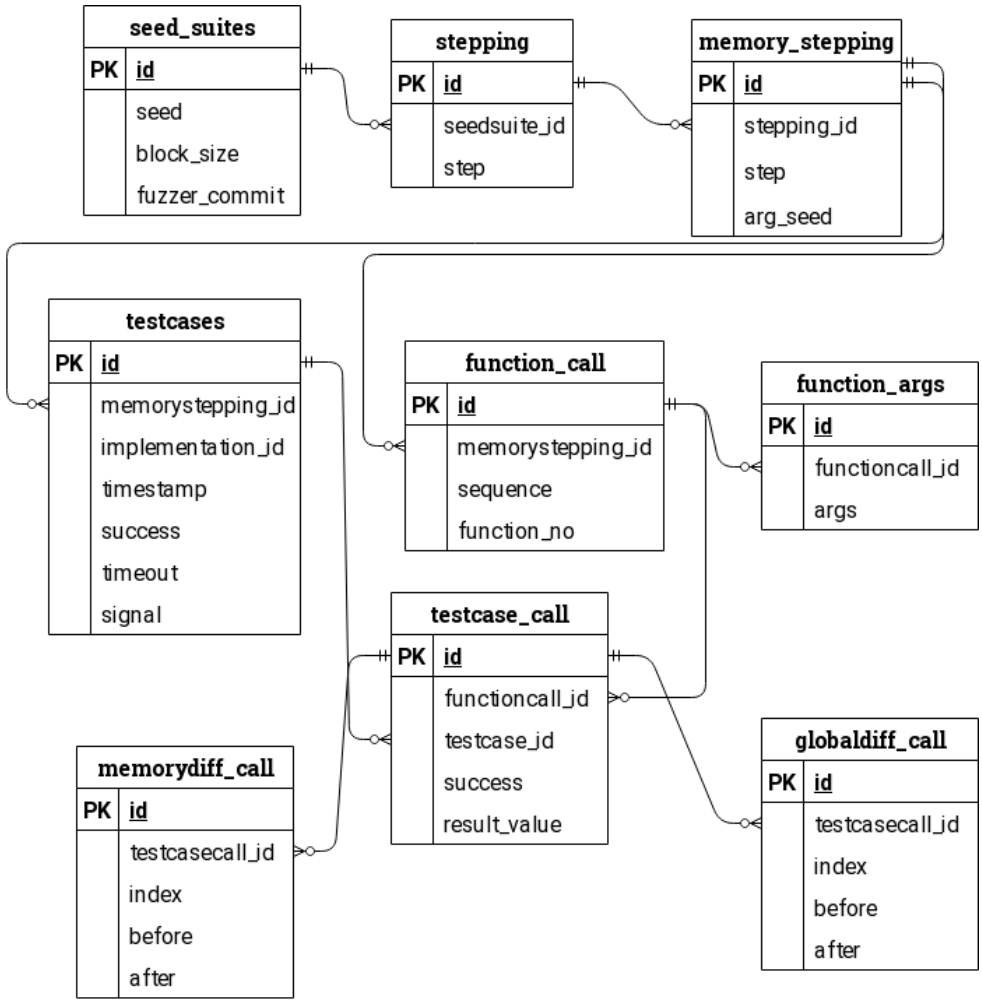

Figure 4.6. Database design used in the experiment

Chapter 5

Analysis

> *"In testing terms, the exploratory tester learns about the system under test, and uses this new knowledge to explore the system more deeply with more focused tests. Because exploratory testing is also highly creative, it is difficult to describe the process precisely. It clearly depends on the attitude and motivation of the tester, but it also depends on the nature of the system under test and on the priorities of the system stakeholders."*
>
> *"Exploratory Testing from Software Testing: A Craftman's Approach"*
> *Paul C. Jorgensen* [30]

This section describes the result of conducting the experiment using the developed test system. Section 5.1 briefly describes the execution environment and the test result. Section 5.2 presents the analysis of prepared test cases in order to understand the differences between both JS engine in generating the machine instruction. This analysis gives a background knowledge before analyzing the identified cases from the differential testing via static and dynamic analysis process. Finally, Section 5.3 presents the identified difference between the JS engines and how to reproduce it in the actual browser execution.

5.1 Running the Experiment

5.1.1 Experiment Environment

The experiment is conducted in a Intel-based x64 system. The machine uses six cores and 12 threads Intel Core i7-5820K processor, with stock speed 3.30 GHz overclocked to 4.40 GHz. The experiment sources, including the JS engines, are compiled with Clang/LLVM version 10, running on OpenSuse Linux with kernel version 5.7.

It is important to understand that due to the use of the random generator in this experiment, the test case presented in this experiment may not be regenerated in different machines. The generator program described in Section 4.6 relies on

several mechanisms which may be optimized by compilers, such as inlining function call and template instantiation. These optimizations may change the code path of the program generation, which ultimately affects the final generated program. For addressing this limitation, the source code repository for this thesis includes every identified test case that is discussed in this thesis.

5.1.2 Experiment Result

The experiment generates over 400.000 random Wasm program. The program size ranges from 1KB to 6KB. The program consists of one to four different functions, which produces around 1KB to 2KB machine instruction per function. From these 400.000 random programs, seven cases are identified that produce different behavior in both JS engines.

5.2 Differentiating Wasm Instruction

This section presents the differences between JS engine implementation by using a crafted test case. This experiment aims to identify the fundamental behavior of the JS engine in compiling the Wasm program. It is more difficult to identify these basic behaviors by directly using a randomly generated Wasm program. Therefore, these experiments use specifically crafted test cases that trigger the basic behavior, such as calling convention, register scheduling, and instruction selection.

This experiment also aims to provide a clear picture of the JS engines' internal implementation, which is mostly lacking in documentation. The disassembly program shown in this section is manually analyzed and properly labeled to help the reader understands the assembly program easily. However, only selected Wasm instructions presented here, which only to give a rough picture of how the JS engine generates the machine instruction.

5.2.1 Calling Convetion

This comparison is to differentiate the calling convention used in the Wasm implementation. We are interested in the JS engine mechanism in Wasm function compilation. As the engine generates new machine instruction based on the loaded program by its JIT or AOT compiler, it must follow a specific convention for the engine to execute the generated instruction.

The comparison uses a predefined Wasm module program, where we obtain the resulting machine instruction through our instrumented API. We can then compare the generated machine instruction to spot the basic differences between the engine implementation. Both engines lack of complete documentation of the internal calling convention. Hence, we can use this experiment to identify the internal mechanism of the JS engine.

The first experiment is identifying the convention for supplying simple integral arguments. We prepared Wasm functions, which accepts one to seven integer arguments, assuming that at some point, the compiler spills arguments on the stack. Listing 5.1 shows an example of the Wasm program.

Listing 5.1. Example of Wasm program to examine the calling convention (`calling-convention.wat`)

```
(module
  (func (export "parami1")
    (param i32) (result i32)
      local.get 0
      local.get 0
      i32.add)
  (func (export "parami7")
    (param i32 i32 i32 i32 i32 i32 i32) (result i32)
      local.get 0
      local.get 1
      local.get 2
      local.get 3
      local.get 4
      local.get 5
      local.get 6
      i32.add
      i32.add
      i32.add
      i32.add
      i32.add
      i32.add))
```

By inspecting the compiled machine instruction, we can obtain the subroutine structure of the compiled function. We can also infer the calling convention used by the engine for the compiled Wasm program. Table 5.1 compares the subroutine structure between the SpiderMonkey and V8 engines. SpiderMonkey generates a function prologue that stores the `r14` and `rbp` register. On the other hand, V8 only stores the rbp register in its prologue.

Table 5.1. Comparison of the compiled Wasm subroutine for each engines

SpiderMonkey	**V8**
push r14	**push** rbp
push rbp	**mov** rbp,rsp
mov rbp,rsp	**push** 0xa
...	**push** rsi
pop rbp	...
pop r14	**mov** rsp,rbp
ret	**pop** rbp
	ret 0x10

By examining the differences between functions that have different parameter counts, we can obtain the calling convention used by the JS engine. Figure 5.1 shows the calling convention comparison between the engines. SpiderMonkey uses the System V AMD64 ABI calling convention that Linux/POSIX environment uses. Instead of using a well-known calling convention, V8 uses its own calling convention specification.

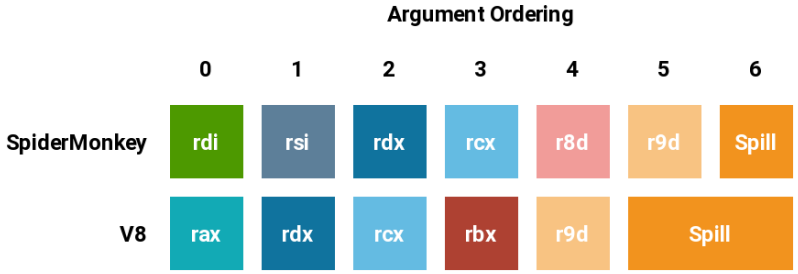

Figure 5.1. Argument order in the register for the function call

In addition, the V8 subroutine call is using the callee clean-up convention. It is shown in the generated instruction for a function that takes more than five parameters. The function epilogue uses a return instruction with the offset value to move the stack pointer, which equals to the number of spilled argument on the stack. Since the SpiderMonkey adheres to the AMD64 ABI calling convention, the generated instruction uses the caller clean-up convention. For the return value, both uses `rax` register.

Figure 5.2 shows the stack layout of the compiled Wasm subroutine used in both engines. From the stack examination, we found that despite the SpiderMonkey subroutine prolog stores the previous `rbp` register. The caller of the Wasm instruction from the JS engine realm does not use a base pointer, at least on the point when the execution enters the Wasm realm. On the other hand, V8 engine utilizes the base pointer so that it is possible to trace the trail of the stack frame.

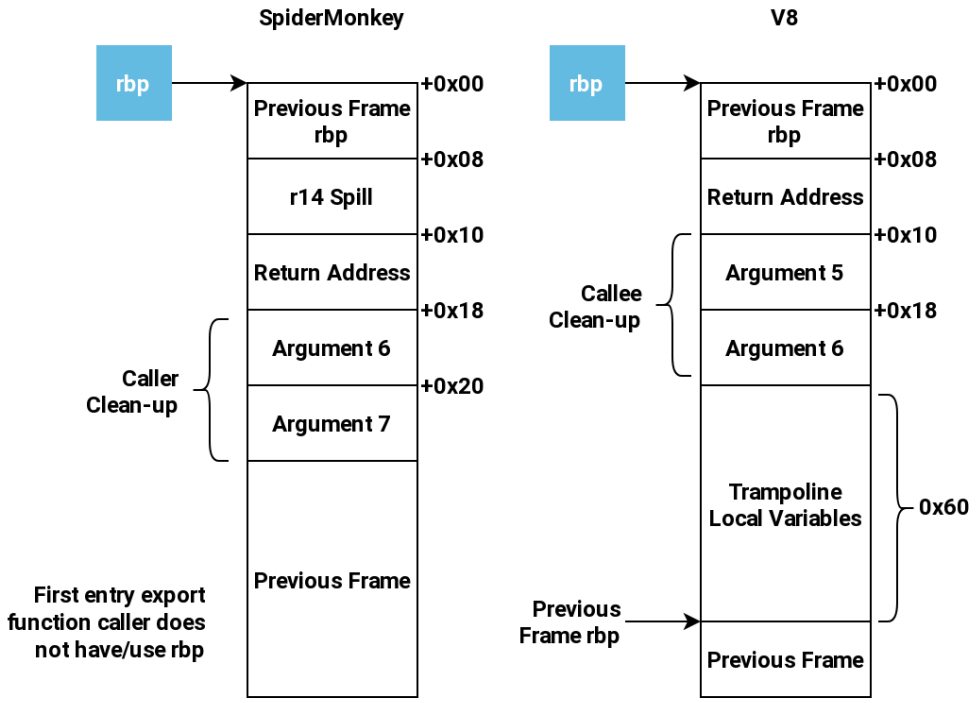

Figure 5.2. Stack frame of the called Wasm function

Based on this inspection, both engines do not seem to implement a stack

guard for the compiled program. It is possible since the Wasm architecture does not allow direct operation with the stack, such as allocating a buffer. Therefore, the risk of corrupting the main call stack from the Wasm program is minimal.

5.2.2 Arithmetic Instructions

This comparison is to differentiate the instruction choice between JS engines for arithmetic operations. Although it might seem trivial, the JS engines may have their own instruction selection to generate the final machine instruction.

Table 5.2 presents the comparison of the addition, subtraction, and multiplication operation between two engines. The arithmetic instruction is enclosed in a simple Wasm function with two integer parameters. Therefore, the register selection refers to the calling convention described in Section 5.2.1. The resulting arithmetic instruction is the same for both 32-bit and 64-bit integers, which differs in the choice of the operand size. For the 64-bit counterpart, it results in a slightly larger code since it has to specify the 64-bit integer name, e.g., `rax` instead of `eax`.

The SpiderMonkey seems to generate unnecessary register copy, which makes it looks inefficient compared to V8. Although the copy instruction is necessary to store the value in the designated return value register (`eax`), it can be simplified without an intermediate register (`ecx`). The operand should be copied directly from `edi`, which is the first parameter, to `eax`.

Table 5.2. Comparison of integer arithmetic instruction in Wasm

SpiderMonkey	**V8**
`(i32.add (local.get 0)(local.get 1))`	

`mov ecx,edi`
`mov eax,ecx`
`add eax,esi`
	`add eax,edx`

`(i32.sub (local.get 0)(local.get 1))`	

`mov ecx,edi`
`mov eax,ecx`
`sub eax,esi`
	`sub eax,edx`

`(i32.mul (local.get 0)(local.get 1))`	

`mov ecx,edi`
`mov eax,ecx`
`mul eax,esi`
	`mul eax,edx`

Table 5.3 shows the division instruction in Wasm. Wasm provides two division instruction flavors: unsigned and signed. It is expected as the majority of hardware differentiates these two instructions. Both implementations perform divisor checking prior to executing the instruction. It needs to raise the exception trap for the divide-by-zero condition. In a native program, this exception is raised trough the operating system exception, which is propagated via an interrupt.

Typically, unhandled exception in a native program results in a program crash. Since Wasm emulates the machine inside the JS engine, this exception control flow

must not happen, and the engine must handle the condition properly. Without this mechanism, the divide-by-zero exception passes through the interrupt channel, which eventually crashes the entire JS engine process.

Table 5.3. Comparison of integer division instruction in Wasm

SpiderMonkey	**V8**
`(i32.div_u (local.get 0)(local.get 1))`	

`mov rax,rdi`
`test esi,esi`
`jne not_zero`
`ud2`
`not_zero:`
`xor edx,edx`
`div esi`
	`mov rbx,rdx`
	`cmp ebx,0x0`
	`je div_by_zero`
	`xor edx,edx`
	`div ebx`
	`...`
	`div_by_zero:`
	`call TrapDivByZero`

`(i32.div_s (local.get 0)(local.get 1))`	

`mov rax,rdi`
`test esi,esi`
`jne not_zero`
`ud2`
`not_zero:`
`cmp edi,0x80000000`
`jne divide`
`cmp esi,0xffffffff`
`jne divide`
`ud2`
`divide:`
`mov rax,rdi`
`cdq`
`idiv esi`
	`mov rbx,rdx`
	`cmp ebx,0x0`
	`je div_by_zero`
	`cmp ebx,0xffffffff`
	`je negative`
	`divide:`
	`cdq`
	`idiv ebx`
	`...`
	`negative:`
	`cmp eax,0x80000000`
	`je overflow`
	`jmp divide`
	`div_by_zero:`
	`call TrapDivByZero`
	`overflow:`
	`call TrapDivUnrepresentable`

Both engines have a different mechanism to raise a trap. In SpiderMonkey, it relies on `ud2` instruction. It is an "undefined instruction," provided explicitly by the x86 architecture, to trigger the undefined opcode exception. The operating system, in this case, POSIX, raises SIGILL illegal for the instruction. SpiderMonkey catches this signal and transfer the control to the `WasmTrapHandler` function to handle the trap [`WasmSignalHandler.cpp`:926].

The V8 engine handles this trap condition by calling the respective subroutine for the trap type. The subroutine itself does not return to the caller and return the control back to the JS realm. Since V8 differentiates every trap type, it can provide the trap information to the engine with appropriate error code and message. It is not the case for the SpiderMonkey since the hardware exception erases the semantic of the cause of the trap.

For the signed division instruction, both engines perform an additional check of dividing maximum negative value (-2^{32} in 32-bit) with -1. Since this division overflows, it yields a trap since the division result is not representable in 32-bit signed integer, which has a maximum value of $2^{32} - 1$.

For the unsigned remainder instruction, both engines implement the same instruction as in unsigned division instruction. The only addition is to copy the remainder result to the target return value register. We can assume that this copy operation is elided by the JS engine when the result of the operation is chained to other instructions.

Table 5.4. Comparison of integer remainder instruction in Wasm

SpiderMonkey	**V8**
`(i32.rem_u (local.get 0)(local.get 1))`	

`mov rax,rdi`
`test esi,esi`
`jne not_zero`
`ud2`
`not_zero:`
`xor edx,edx`
`div esi`
`mov eax,edx`
	`mov rbx,rdx`
	`cmp ebx,0x0`
	`je TrapDivByZero`
	`xor edx,edx`
	`div ebx`
	`mov rax,rdx`
	`...`
	`div_by_zero:`
	`call TrapDivByZero`

`(i32.rem_s (local.get 0)(local.get 1))`	

`mov rax,rdi`
`test esi,esi`
`jne not_zero`
`ud2`
`not_zero:`
`test edi,edi`
`js check_signed`
`mov rdx,rsi`
`sub edx,0x1`
`test esi,edx`
`jne divide_unsigned`
`and edx,edi`
`jmp return`
`divide_unsigned:`
`xor edx,edx`
`idiv esi`
`jmp return`
`check_signed:`
`cmp edi,0x80000000`
`je is_signed`
`divide_signed:`
`cdq`
`idiv esi`
`return:`
`mov eax,edx`
`...`
`is_signed:`
`cmp esi,0xffffffff`
`jne divide_signed`
`xor edx,edx`
`jmp return`
	`mov rbx,rdx`
	`cmp ebx,0x0`
	`je div_by_zero`
	`cmp ebx,0xffffffff`
	`je zero_rem`
	`cdq`
	`idiv ebx`
	`return:`
	`mov rax,rdx`
	`...`
	`zero_rem:`
	`xor edx,edx`
	`jmp return`
	`div_by_zero:`
	`call TrapDivByZero`

The situation differs in the signed remainder instruction. SpiderMonkey implements a highly optimized operation to avoid unnecessary use of the division instruction. It checks if the divisor is 1 or -1, which directly returns zero without executing the division. V8 also performs a similar check, but it only works for -1

value. The comparison is shown in Table 5.4.

Table 5.5 shows the comparison of floating-point operation for the 32-bit floating-point type. The 64-bit variants only differ on the selected instruction, which uses the double-precision variants instead of the single-precision one. V8 and SpiderMonkey use different instruction encoding for floating-point instruction. V8 uses a three-operands floating-point instruction, while SpiderMonkey uses the two-operands variants. The slight difference appears in the floating-point division operator, in which the V8 emits a copy instruction from the result register to itself.

Table 5.5. Comparison of floating-point instruction in Wasm

SpiderMonkey	**V8**
`(f32.add (local.get 0)(local.get 1))`	
`addss xmm0,xmm1`	`vaddss xmm1,xmm1,xmm2`
`(f32.sub (local.get 0)(local.get 1))`	
`subss xmm0,xmm1`	`vsubss xmm1,xmm1,xmm2`
`(f32.mul (local.get 0)(local.get 1))`	
`mulss xmm0,xmm1`	`vmulss xmm1,xmm1,xmm2`
`(f32.div (local.get 0)(local.get 1))`	
`divss xmm0,xmm1`	`vdivss xmm1,xmm1,xmm2` `vmovaps xmm1,xmm1`

5.2.3 Control Structure Instructions

This experiment compares the basic control structure in Wasm. This experiment uses the example program previously described in section 2.2.4. This section presents four examples, which include basic block, conditional block, do-while equivalent block, and while with break instruction.

Basic Block

This test case uses the example code presented in Listing 2.7. This test case should demonstrate the behavior of the simplest control flow block, which involves a conditional and a jump. Table 5.6 presents the comparison of the resulting machine instruction between the engines.

From the comparison, it is apparent that the SpiderMonkey engine uses a very rigorous optimization mechanism. It avoids the division instruction altogether to compute the remainder value for the conditional logic. Instead of using division instruction, which tends to be costly, it uses a combination of multiplication, bit-shift, and addition.

The resulting instruction from the V8 also tries to do optimization, similar to the remainder instruction presented in Table 5.4. However, this optimization leads to a dead code path, because the divisor, which is stored in `ecx`, is a constant. Therefore, the control never takes the jump instruction to the label. The V8, nevertheless, optimizes the multiplication by 3 with a lea instruction, which

Table 5.6. Comparison of basic control structure in Wasm

SpiderMonkey	V8

`mov ebx,edi`
`mov ecx,ebx`
`mov eax,0x66666667`
`imul ecx`
`sar edx,1`
`mov eax,ecx`
`sar eax,0x1f`
`sub edx,eax`
`imul eax,edx,0xfffffffb`
`add eax,ecx`
`test eax,eax`
`je exit`
`mov eax,ebx`
`imul eax,eax,0x14`
`mov ebx,eax`
`exit:`
`mov eax,ebx`
`imul eax,eax,0x3`
	`mov rbx,rax`
	`mov ecx,0x5`
	`cmp ecx,0xffffffff`
	`je zero`
	`mov rax,rbx`
	`cdq`
	`idiv ecx`
	`cond:`
	`cmp edx,0x0`
	`je exit`
	`imul ebx,ebx,0x14`
	`exit:`
	`lea eax,[rbx+rbx*2]`
	`...`
	`zero:`
	`xor edx,edx`
	`jmp cond`

works perfectly for this case.

Apart from the optimization of the remainder instruction, both engines implement the basic jump instruction in a similar fashion. We can expect this as the test case program has an elementary control structure without complex control flow.

Simple Conditional

This test case uses the simple conditional example from Listing 2.8. Despite using a different control structure instruction, the resulting instruction is equivalent to the instruction with basic jump instruction presented in Table 5.6. Therefore, we can assume that internally, the simple conditional instruction translates similarly to the basic block, which results in the same final instruction.

Do While

This test case uses the do-while example from Listing 2.10. As explained before, this type of loop checks the condition at the end of the block. Hence, this loop is the simplest form of an iteration in an assembly program, which only requires a single backward jump to form the loop. Table 5.7 compares the resulting machine instruction.

We can observe that V8 puts a stack-guard mechanism in each of the loop iterations. The stack-guard mechanism calls the internal V8 engine implementation named `Runtime_WasmStackGuard` [`runtime-wasm.cc`:173]. This function resides in the JS engine runtime, and the Wasm code uses a trampoline to transfer the control from Wasm realm to the runtime.

The SpiderMonkey compiled instruction also seems to perform a similar action. It performs some checking at the beginning of the loop. In case the check fails, the execution triggers a trap. The conditional instruction checks the data pointed by r14 register. Based on the SpiderMonkey source for the

Table 5.7. Comparison of do-while structure in Wasm

SpiderMonkey	V8

`mov eax,0x3`
`loop_block_check:`
`cmp DWORD PTR [r14+0x38],0x0`
`je loop_block`
`ud2`
`loop_block:`
`imul eax,esi`
`mov ecx,edi`
`add ecx,0x1`
`cmp edi,0x64`
`jge loop_exit`
`mov edi,ecx`
`jmp loop_block_check`
`loop_exit:`
`imul eax,ecx`
	`mov ebx,0x3`
	`jmp check_stack_guard`
	`nop WORD PTR [rax+rax*1+0x0]`
	`xchg ax,ax`
	`loop_block:`
	`mov rbx,rdi`
	`mov rax,rcx`
	`check_stack_guard:`
	`mov rcx,QWORD PTR [rsi+0x23]`
	`cmp rsp,QWORD PTR [rcx]`
	`jbe do_stack_guard`
	`loop_code_start:`
	`lea ecx,[rax+0x1]`
	`mov rdi,rdx`
	`imul edi,ebx`
	`cmp eax,0x64`
	`jl loop_block`
	`imul edi,ecx`
	`...`
	`do_stack_guard:`
	`mov QWORD PTR [rbp-0x18],rax`
	`mov QWORD PTR [rbp-0x20],rdx`
	`mov QWORD PTR [rbp-0x28],rbx`
	`call WasmStackGuard`
	`mov rax,QWORD PTR [rbp-0x18]`
	`mov rdx,QWORD PTR [rbp-0x20]`
	`mov rbx,QWORD PTR [rbp-0x28]`
	`mov rsi,QWORD PTR [rbp-0x10]`
	`jmp loop_code_start`

x64 assembler, SpiderMonkey reserves `r14` to store TLS data for Wasm function [`Assembler-x64.h`:223].

The TLS data, which is represented by `js::wasm::TlsData` struct [`WasmTypes.h`:2832], contains several important data for the current Wasm thread. The check refers to the offset `0x38`, which is the `interrupt` field. Therefore, we can assume that the SpiderMonkey changes this interrupt flag in case of a detected fault, and halt the Wasm execution by trigerring a trap.

While-Loop with Break

This test case uses a more advanced construct of the loop, as presented in Listing 2.14. This type of loop checks the condition at the beginning of the loop. It also has a condition that breaks the loop in the middle. Table 5.8 presents the comparison from both engines.

Similar to the do-while loop construct, both engines install integrity checking code in each loop iteration. However, the compiled function in the V8 has a more complex control flow graph compared to the SpiderMonkey. This characteristic may indicate that SpiderMonkey generates a more efficient and compact machine instruction compared to V8.

SpiderMonkey control flow is more faithful to the Wasm program counterpart.

Table 5.8. Comparison of do-while with break in Wasm

SpiderMonkey

```
  mov  ebx,edi
  cmp  edi,0x64
  jle  loop_init
  mov  ecx,0x3
  mov  ebx,edi
  jmp  loop_exit
loop_init:
  mov  eax,edi
  add  edi,0xffffffff
  mov  edx,esi
  mov  ebx,edi
  imul edx,eax
  add  edx,0xffffff9c
  mov  ecx,0x3
loop_block_check:
  cmp  DWORD PTR [r14+0x38],0x0
  je   loop_entry
  ud2
loop_block:
  add  edx,esi
  test edx,edx
  je   loop_end
  imul ecx,esi
  mov  eax,ebx
  add  eax,0x1
  cmp  eax,0x64
  jge  loop_end
  mov  ebx,eax
  jmp  loop_block_check
loop_end:
  mov  eax,ebx
  add  eax,0x2
  mov  ebx,eax
loop_exit:
  mov  eax,ebx
  imul eax,ecx
```

V8

```
  sub  rsp,0x20
  cmp  eax,0x64
  jg   skip_loop
  mov  rbx,rdx
  imul ebx,eax
  sub  ebx,0x64
  lea  ecx,[rax-0x1]
  mov  edi,0x3
  jmp  check_stack_guard
  nop  WORD PTR [rax+rax*1+0x0]
loop_reenter:
  mov  rcx,rdi
  mov  rdi,r8
check_stack_guard:
  mov  r8,QWORD PTR [rsi+0x23]
  cmp  rsp,QWORD PTR [r8]
  jbe  do_stack_guard
loop_code_start:
  add  ebx,edx
  je   breaking_loop
  mov  r8,rdx
  imul r8d,edi
  lea  edi,[rcx+0x1]
  cmp  edi,0x64
  jl   loop_block
  mov  rdi,r8
breaking_loop:
  lea  eax,[rcx+0x2]
  jmp  loop_exit
skip_loop:
  mov  edi,0x3
loop_exit:
  imul edi,eax
  mov  rax,rdi
  ...
do_stack_guard:
  mov  QWORD PTR [rbp-0x18],rcx
  mov  QWORD PTR [rbp-0x20],rdi
  mov  QWORD PTR [rbp-0x28],rdx
  mov  QWORD PTR [rbp-0x30],rbx
  call WasmStackGuard
  mov  rcx,QWORD PTR [rbp-0x18]
  mov  rdi,QWORD PTR [rbp-0x20]
  mov  rdx,QWORD PTR [rbp-0x28]
  mov  rbx,QWORD PTR [rbp-0x30]
  mov  rsi,QWORD PTR [rbp-0x10]
  jmp  loop_code_start
```

The original Wasm program has seven basic blocks, which is exactly equivalent with the SpiderMonkey generated program. The V8 compiled instruction uses more branching, thus creating more basic blocks. Although the control flow is equivalent, the additional basic block seems to be introduced due to the register scheduling in the generated instruction.

5.2.4 Memory Instruction

This test case compares the memory operation in the Wasm program. The test program includes a single memory load, store, and a combination of both. The test program also experiments with the offset specifier in the Wasm instruction. Table 5.11 presents a comparison of the resulting memory load operation.

Table 5.9. Comparison of memory load operation in Wasm

SpiderMonkey	**V8**
`((i64.load offset=0 (local.get 0))`	

`mov eax,DWORD PTR [r15+rdi*1]`	`mov rbx,QWORD PTR [rsi+0xb]`
	`mov rdx,QWORD PTR [rsi+0x13]`
	`sub rdx,0x3`
	`mov ecx,eax`
	`cmp rcx,rdx`
	`jae out_of_bound`
	`mov eax,DWORD PTR [rbx+rcx*1]`
	`...`
	`out_of_bound:`
	`call WasmTrapMemOutOfBounds`

`((i64.load offset=1024 (local.get 0))`	

`mov rax,QWORD PTR [r15+rdi*1+0x400]`	`mov rbx,QWORD PTR [rsi+0xb]`
	`mov rdx,QWORD PTR [rsi+0x13]`
	`sub rdx,0x407`
	`mov ecx,eax`
	`cmp rcx,rdx`
	`jae out_of_bound`
	`mov rax,QWORD PTR [rcx+rbx*1+0x400]`
	`...`
	`out_of_bound:`
	`call WasmTrapMemOutOfBounds`

V8 accesses Wasm linear memory from a pointer that is stored in the `rsi` register. `rsi` register itself contains a pointer to the `Context` object. V8 sets the register in the trampoline function before jumping to the Wasm realm, as indicated in the `Generate_JSEntryTrampolineHelper` function. The offset itself points to the `memory_start` and `memory_size` field in `WasmInstanceObject` field. The `memory_start` pointer stores the address to the linear memory, while `memory_size` is used to perform access bound checking.

From the listing, it is apparent that V8 generates a bound-checking instruction in the compiled instruction. SpiderMonkey, however, relies on the hardware interrupt again to detect the invalid memory access. SpiderMonkey mapped a memory region that is used for Wasm linear memory region. The size of the region is equal to the current size of the Wasm memory. SpiderMonkey stores the start address of the Wasm memory in `r15` register.

SpiderMonkey takes advantage of 64-bit address space to seal the border between Wasm memory and the rest of memory. Since Wasm address space is limited to 32-bit, a Wasm program cannot access the process memory beyond

the 32-bit limit. The actual address of the Wasm memory also cannot underflow because the negative value in the index is not sign-extended in the final address computation. With these properties, SpiderMonkey can rely on the segmentation fault signal from the operating system to identify invalid memory access and handle the error accordingly.

Table 5.10. Comparison of memory store operation in Wasm

SpiderMonkey	**V8**
((i64.store offset=0 (local.get 0)(local.get 1))	

`mov QWORD PTR [r15+rdi*1], rsi`	`mov rbx,QWORD PTR [rsi+0xb]`
	`mov rcx,QWORD PTR [rsi+0x13]`
	`sub rcx,0x7`
	`mov eax,eax`
	`cmp rax,rcx`
	`jae WasmTrapMemOutOfBounds`
	`mov QWORD PTR [rbx+rax*1],rdx`
	`...`
	`out_of_bound:`
	`call WasmTrapMemOutOfBounds`

((i64.store offset=0 (local.get 0)(i64.add (i64.load offset=0 (local.get 0))(local.get 1)))	

`mov rax,QWORD PTR [r15+rdi*1]`
`add rax,rsi`
`mov QWORD PTR [r15+rdi*1],rax`
	`mov rbx,QWORD PTR [rsi+0xb]`
	`mov rcx,QWORD PTR [rsi+0x13]`
	`sub rcx,0x7`
	`mov eax,eax`
	`cmp rax,rcx`
	`setb cl`
	`movzx ecx,cl`
	`cmp ecx,0x0`
	`je out_of_bound`
	`mov rsi,QWORD PTR [rbx+rax*1]`
	`cmp ecx,0x0`
	`je out_of_bound`
	`add rdx,rsi`
	`mov QWORD PTR [rbx+rax*1],rdx`
	`...`
	`out_of_bound:`
	`call WasmTrapMemOutOfBounds`

Table 5.11 presents the comparison of memory store operation and chaining memory load and store operation. Both engines use a separate memory load and store operation independently. Instead of emitting add instruction with a memory operand, both engines load the memory and store it in the intermediate register before performing the addition.

5.2.5 Global Variable Instruction

This test case examines the JS engine compilation for the global variable storage. The test uses the Wasm program in Listing 5.2. It examines basic get and set operation on global variables. Table 1 presents the result of the compilation.

Listing 5.2. Example of Wasm program to examine the global variable (`calling-convention.wat`)

```
(module
  (import "" "g_i32" (global $g_i32 (mut i32)))
  (import "" "g_i32_2" (global $g_i32_2 (mut i32)))
  (import "" "g_i32_3" (global $g_i32_3 (mut i32)))
  (import "" "g_i64" (global $g_i64 (mut i64)))
  ...
)
```

Table 5.11. Comparison of global variable operation in Wasm

SpiderMonkey	V8
`(global.set $g_i32 (local.get 0))`	

`mov rax,QWORD PTR [r14+0x60]`
`mov DWORD PTR [rax],edi`
	`mov rbx,QWORD PTR [rsi+0x57]`
	`mov rbx,QWORD PTR [rbx]`
	`mov DWORD PTR [rbx],eax`

`(global.set $g_i32 (i32.add (global.get $g_i32_2)(global.get $g_i32_3)))`	

`mov rax,QWORD PTR [r14+0x68]`
`mov eax,DWORD PTR [rax]`
`mov rcx,QWORD PTR [r14+0x70]`
`mov ecx,DWORD PTR [rcx]`
`add eax,ecx`
`mov rcx,QWORD PTR [r14+0x60]`
`mov DWORD PTR [rcx],eax`
	`mov rax,QWORD PTR [rsi+0x57]`
	`mov rbx,QWORD PTR [rax+0x8]`
	`mov ebx,DWORD PTR [rbx]`
	`mov rdx,QWORD PTR [rax+0x10]`
	`mov edx,DWORD PTR [rdx]`
	`mov rax,QWORD PTR [rax]`
	`add ebx,edx`
	`mov DWORD PTR [rax],ebx`

Both engines use the same approach for implementing Wasm global variables. The global variable is stored in some memory location, which is accessible through a pointer table. The table entry is ordered based on the index of the global variable. The JS engine computes the offset to the table to retrieve the address of the global variable storage and uses basic mov instruction to retrieve or store values.

Since the implementation uses a pointer table, accessing global variables, require two indirections: resolving the pointer table, and access the actual global variable storage. The only difference between the engines is the location to store the pointer table. In V8, it is stored in the `WasmInstanceObject` that is pointed at `rsi` register. While in SpiderMonkey, it uses a dedicated `r14` register.

5.3 Investigating Differences Found

5.3.1 Sample Description

The experiment observed a different behavior in the experiment with the configuration, as described in Table 5.12. The module has three different functions, where the function 0 triggers the difference. The differences occurred five times in the same memory index, in which the difference can be observed in the very

first iteration of the fuzzing loop.

Table 5.12. Sample description for Sample Case #1

Seed	**Step**	**Block Size**	**Arg Seed**	**Memory**
227760203	4719	8192	2883752934	rand3.mem
Signature				
`(func $func0 (param $p0 f64)(param $p1 i32)(result i32))`				
`(func $func1 (param $p0 f32 f32 i32 i32 f64 f32 f64 f64 i64 i64 f64 i64 i64 f32 f64))`				
`(func $func2 (param f64, f32, f32, i32, i64, f32, f32, f32, f32, f64, i32, i32)(result f64))`				

5.3.2 Dynamic Analysis of the Module

We used dynamic analysis to identify the point in which the Wasm program modifies the memory value. We put a watchpoint on the Wasm memory address, which stops after the program modifies a specific location. From this watchpoint, we identified the instruction that changes the memory value, as presented in Table 5.13.

Table 5.13. The problematic instruction for Case #1

SpiderMonkey	**V8**
`movss dword ptr[r15+rax+16A38h],xmm1`	`vmovssdword ptr[rbx+rdi+16A38h],xmm7`

The instruction stores a value in a floating-point register to the memory. The instruction itself uses an offset value, which can give a hint of the Wasm instruction that triggers the difference. The offset is `0x16A38`, which is 92728. Therefore, we need to find the floating-point store instruction with that offset on the original Wasm program. Listing 5.3 presents the snippet of the function that leads to the said instruction. Fortunately, the identified instruction is not far from the function entry point, which can help to trace the instruction evaluation.

Listing 5.3. Snippet of the Wasm that reach to the differing behavior

```
(module
  (import "" "global10" (global $.global10 (mut f32)))
  (func $func0 (param $p0 f64) (param $p1 i32) (result i32)
    global.get $.global10
    f64.promote_f32
    i32.trunc_f64_u
    i32.const 1310720
    i32.rem_u
    f32.const 0x1.68p-144 (;=6.30584e-44;)
    i32.const 0
    i32.const 0
    br_if 0 (;@0;)
    drop
    f32.const 0x0p+0 (;=0;)
    i32.const 0
    i32.const 1310720
    i32.rem_u
    i64.const 0
    i64.store8 offset=235
    f32.sub
    i32.const 0
    i32.const 0
    i32.eq
    memory.grow
    i32.const 1310720
    i32.rem_u
    i32.load8_s offset=645820
    f32.reinterpret_i32
    f32.trunc
    f32.max
    f32.store offset=92728 align=2
    ...
  ))
```

5.3.3 Static Analysis of the Module

From the identified instruction, we can perform static analysis on the Wasm program. The static analysis allows us to trace the logical structure of the program to reach the remarked instruction. Wasm semantic structure, which resembles an expression tree, also enables the static analysis to describe the code path easily. Figure 5.3 presents the expression tree of the Wasm program that reaches to the memory write instruction that triggers the difference.

From the expression tree, we can observe that the value stored in the store instruction originates from the floating-point max instruction. The max instruction itself selects the maximum value between a constant and a value loaded from memory. The program loads 8-bit value from memory, and bit-cast the value to the floating-point type.

On a side note, the remainder operation that produces the index for the store operation is the instruction generated by our modified fuzzer generator. This instruction is used to set the upper bound of the memory access, which prevents

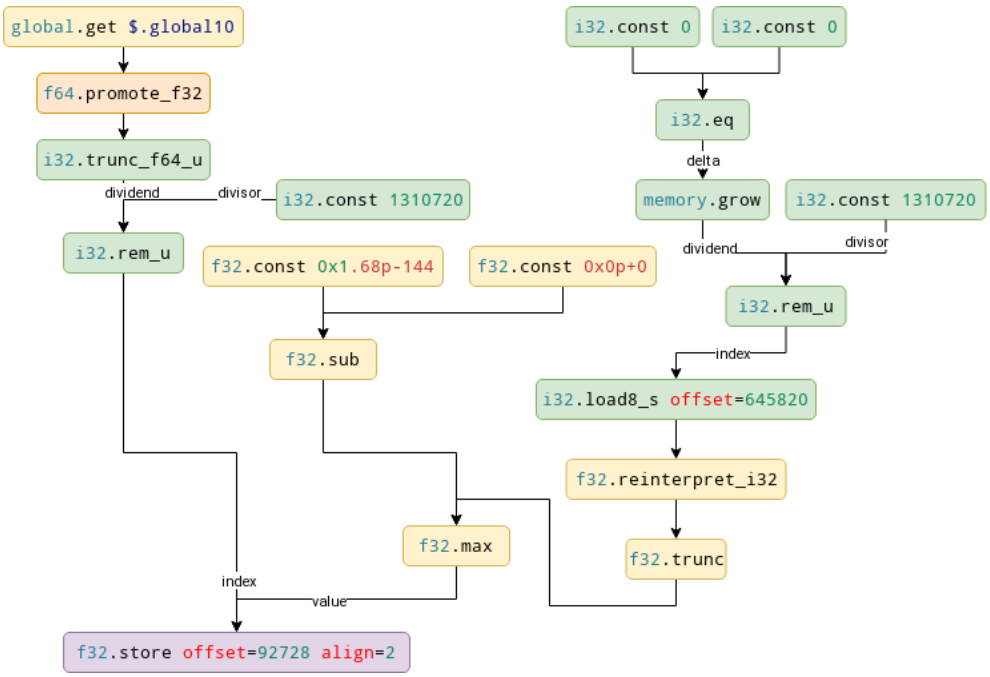

Figure 5.3. Expression tree to the store instruction

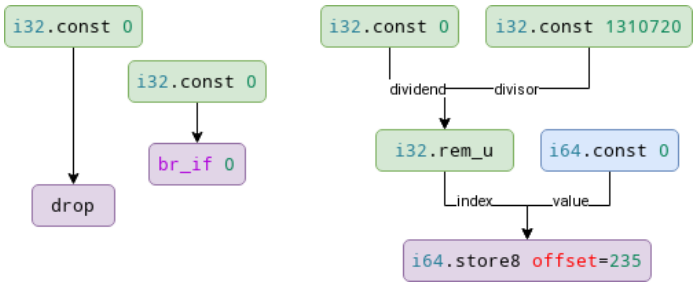

Figure 5.4. Other terminating expression that does not affect the store instruction

the Wasm program from unnecessarily terminates due to invalid memory access for every generated program. The other expressions presented in Figure 2 do not participate in forming the final value for the floating-point store instruction.

By revisiting the dynamic analysis, we found that the load 8-byte value produces `0xe3` value. Since the load instruction uses the signed variant, the final 32-bit value is `0xffffffe3`, because of the sign-extension operation. This value is reinterpreted directly as a single-precision floating-point value. The `0xffffffe3` is translated to a Not-a-Number (NaN) value in floating-point. Therefore, the subsequent max instruction also yields a NaN value.

The differences occur in how both engines treat NaN value in the f32.max instruction. While SpiderMonkey directly uses the NaN value from the previous instruction, V8 generates a canonical Quiet-NaN value, which is `0xffc00000`. Therefore, we can observe the difference between SpiderMonkey and V8 on the final value that is written to memory.

Listing 5.14 compares the generated machine instruction that performs this operation. Both engine checks whether the comparison is valid through the ucomiss instruction. In V8, when the comparison detects an unordered comparison, which is indicated by the positive flag[9], the control jumps to an instruction that overwrites the floating-point register to the canonical NaN value. V8 uses divide-by-zero to generate its canonical NaN value. This behavior is not present in the SpiderMonkey, where the program uses the NaN value obtained from the memory load.

Table 5.14. Handling of NaN value in the f32.max instruction (Note: Some V8 codes are redacted since it is related to the memory bound checking)

SpiderMonkey	V8

`movsx eax,byte ptr[r15+rax +9DABCh]`
`movd xmm0,eax`
`roundss xmm0,xmm0,3`
`movss xmm1,[rsp+24h]`
`subss xmm1,[rsp+1Ch]`
`subss xmm0,[rsp+1Ch]`
`ucomiss xmm1, xmm0`
`jnz compute_max`
`jp max_is_unordered`
`andps xmm1,xmm0`
`jmp end_max_inst`
`max_is_unordered:`
`ucomiss xmm1,xmm1`
`jp end_max_inst`
`compute_max:`
`maxss xmm1,xmm0`
`end_max_inst:`
`mov eax,[rsp+18h+var_8]`
`movss dword ptr[r15+rax+16 A38h],xmm1`
	`movsx r12d,byte ptr[r12+rdi +9DABCh]`
	`...`
	`vmovd xmm0,r12d`
	`...`
	`vroundss xmm0,xmm0,xmm0,0Bh`
	`...`
	`vucomiss xmm7, xmm0`
	`jp gen_canoncal_nan`
	`ja short xmm7_above`
	`jb short xmm7_below`
	`vmovmskps r10d, xmm7`
	`test r10b, 1`
	`jz short xmm7_above`
	`xmm7_below:`
	`vmovss xmm7,xmm7,xmm0`
	`xmm7_above:`
	`...`
	`vmovss dword ptr [rbx+rdi+16 A38h], xmm7`
	`...`
	`gen_canoncal_nan:`
	`vxorps xmm7,xmm7,xmm7`
	`vdivss xmm7,xmm7,xmm7`
	`jmp xmm7_above`

5.3.4 Other Sample Cases

the experiment found six other test cases. All those cases are similar in nature with the sample case presented in this section. The floating-point memory write causes all differences in those cases. The value written by the instruction originates from invalid NaN values loaded either from random parameter value or random memory value.

Since the root-cause is equals, this thesis is not going to discuss further the rest of the identified cases.

[9]The ucomiss instruction sets the positive, zero, and carry flags in the flag register if the floating-point operands is incomparable, i.e., unordered [27]. Therefore, the subsequent jp instruction is taken when any of the operand is NaN.

5.3.5 Demonstrating the Difference in Actual Browser

From this behavior, we can craft a simple Wasm program to trigger the difference. We can use this differing characteristic to identify the environment that is executing the Wasm module. Therefore, the Wasm module can become aware of its execution environment without requiring any JavaScript API.

We only need to follow the expression tree previously presented. We create a function that accepts a 32-bit integer value and also returns a 32-bit integer value. The function performs the floating-point reinterpretation that leads to differing NaN behavior. The function then reinterprets back the floating-point to the integer type and returns to the caller.

The caller of the function supplies an integer value that leads to non-canonical NaN when it is reinterpreted to a floating-point type. It then compares the result of the function, in which a differing value indicates that the V8 engine executes the module. Listing 5.4 presents the Wasm code, and Listing 5.5 presents the JavaScript binding for the Wasm program.

Listing 5.4. Wasm code that is aware of its executing environment

```
(module
  (memory 1)
  (func (export "am_i_spidermonkey") (result i32)
    i32.const 100
    i32.const 0
    i32.const -29
    call $i_am_different
    i32.const -29
    i32.eq
    select
  )
  (func $i_am_different (export "i_am_different") (param $input i32)
      (result i32)
    local.get $input
    f32.reinterpret_i32
    f32.const 0x1.68p-144
    f32.max
    i32.reinterpret_f32
  )
  (export "string" (memory 0))
  (data $d0 (i32.const 0)
    "Boo!␣The␣SpiderMonkey␣is␣not␣here.␣Poor␣you!\00")
  (data $d1 (i32.const 100)
    "Yes!!␣I␣am␣a␣proud␣SpiderMonkey!\00"))
```

This differing behavior can be demonstrated in Google Chrome version 83 and Firefox version 77. Figure 5.5 presents the result when we execute the Wasm program from the web browser.

Listing 5.5. JS code binding for the Wasm program in Listing 5.4

```
WebAssembly.compileStreaming(fetch("different.wasm"))
.then((x) => {
    return WebAssembly.instantiate(x)
}).then((x) => {
  var text = "";
  var c_str_ptr = x.exports.am_i_spidermonkey();

  // Print the C string
  const buffer = new Uint8Array(x.exports.string.buffer)
  while(buffer[c_str_ptr] != 0) {
    text += String.fromCharCode(buffer[c_str_ptr++]);
  }

  var h1 = document.createElement("h1");
  var node = document.createTextNode(text);
  h1.appendChild(node);
  document.body.appendChild(h1);
});
```

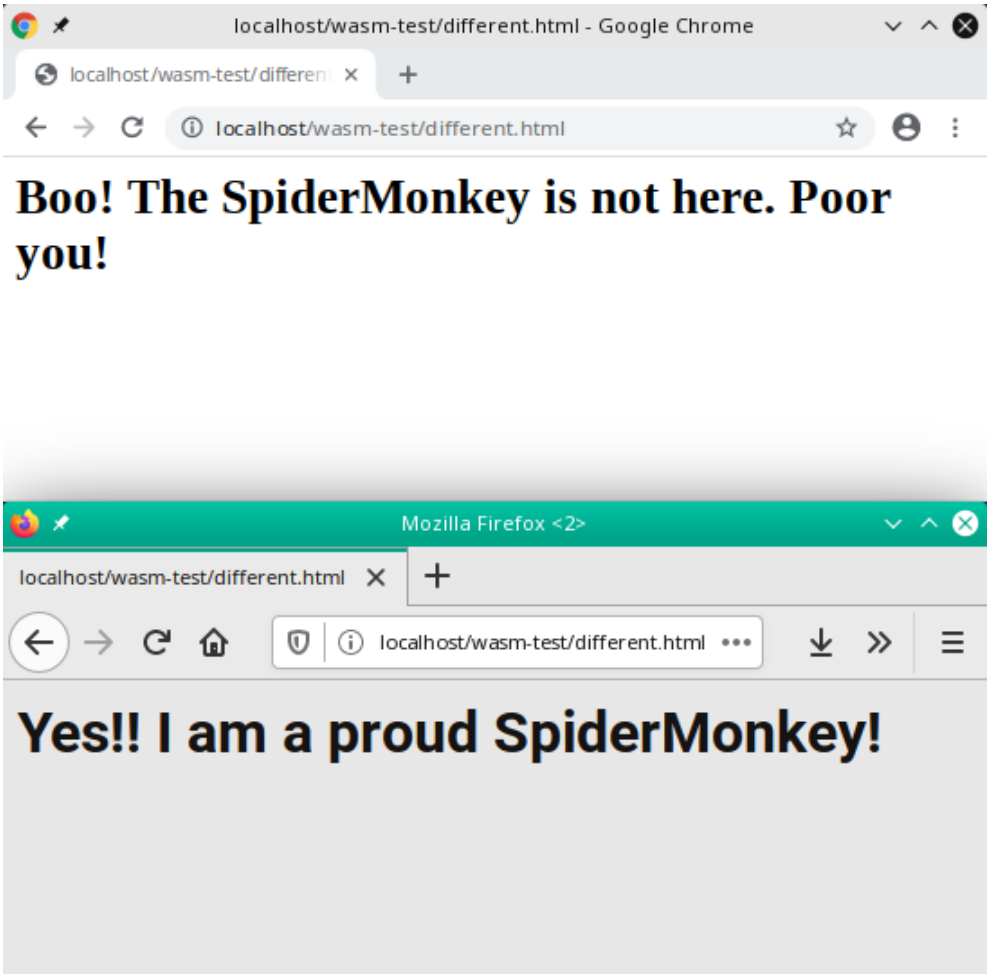

Figure 5.5. Demonstrating the difference in the recent browser version

Chapter 6

Conclusion

> *"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."*
>
> *Edsger Dijkstra, 1976* [47]

This chapter concludes the thesis by presenting the conclusion from the experiment. Section 6.1 presents the finding from the experiment, as well as the issues encountered during the experiment. Section 6.2 describes the author's contribution to the open-source project during the experiment development. Finally, Section 6.3 proposes ideas to follow-up beyond this thesis research.

6.1 Experiment Result

6.1.1 Findings

After conducting the differential fuzzing experiment on the Wasm and the JS engines, we came into several conclusions concerning the Wasm implementation.

1. Wasm is a strict specification that includes a structure validation of its program. The Wasm validation rules verify the Wasm module to ensures it is well-formed [22]. The JS engine strictly enforces this validation rule, which in turn leaves no room for invalid Wasm program.
2. Wasm specification also clearly defines the program instruction semantics [22]. The specification explicitly defines the behavior of every Wasm instruction, which the JS engines follow. Therefore, we can expect that a proper implementor of Wasm specification must adhere to this semantic behavior. In other words, a Wasm program must be executed equally between different Wasm host environment.
3. However, Wasm specification also stated that several numerical instructions are non-deterministic, for example, different NaN values [22]. This non-deterministic behavior is demonstrated through the experiment in this

thesis. Both JS engines discussed in this thesis employs a different approach to determine the outcome of those non-deterministic numerical instructions.

4. The non-deterministic nature of NaN representation is specified in the IEEE 754-2019 Standard for Floating-Point Arithmetic [3]. This standard defines a NaN representation to not only a single value constant, but a range of values. Both JS engine deals this case differently and triggers the differences in the execution result.
5. This experiment confirms that both JS engines produce equal semantic behavior, which we can safely assume that they adhere to the Wasm specification. This experiment can be a mechanism to verify the regression of the Wasm development to ensure conformity to the Wasm specification.

6.1.2 Experiment Issues and Possible Solution

During the experiment, we encountered several issues and challenges that can be addressed to improve the experiment. This section presents several issues and the possible solution to address it.

Reducing Collected Data

The experiment system collected the complete result of the test case execution because the comparison is performed at the end of the test. It created a massive unnecessary data that became garbage because only a few relevant information that can be gathered from the data.

We can reduce the collected data by performing the comparison right after the test case is executed. By using this approach, we can only collect and store relevant data, which is a test case result that is different between the two engines.

Meaningless Test Case

The test case generator selects the generated instruction through a random process similar to rolling a dice. It selects the next instruction without considering any context. It includes when the generator tries to generate control flow blocks.

The generator in most of the observed cases fails to generate a program with complex control flows. Many randomly generated program with loop block does not actually perform a loop due to no proper backward jump in the loop block (See section 2.2.4). It can be either no branch instruction or a branch instruction without an appropriate control loop variable. In some cases, the randomly generated program produces infinite loop or, worse, infinite recursion.

Another problem with the test case is that it produces multiple zero constants in the generated program. The generator terminates the generation process when it runs out of random values to obtain. The termination is marked by generating a constant zero instruction to complete the generation production. Since the generation algorithm uses a tree-based recursion process, the constant zero instruction is used as the base case of the recursion, which also ensures the

generated Wasm program remains valid.

This limitation can be addressed by improving the test case generation. Section 6.3.1 proposes some improvements to the test case generation to produce a more meaningful test case for the fuzz-testing.

Inconclusive Result on Comparing the Elapsed Time

The test case system tries to measure the elapsed time during the Wasm execution. However, the result indicates a difference in the order of a hundred milliseconds, which yields an inconclusive result. Other overheads may be present within the invocation function that is used to call the Wasm function.

Although it is interesting to measure the performance of the generated program between the two engines, the instrumentation proposed in this thesis cannot produce an accurate result to observe their performance. A more thorough investigation of its internal code is required in order to perform an apple-to-apple comparison of the generated code performance. Due to this issue, the elapsed time result is excluded from this final thesis report.

6.2 Contribution to Open Source Project

6.2.1 V8 Commit: a40f30a

During the experiment, the writer found a bug in the random Wasm generator in the V8 fuzzer components. The bug is caused by a boolean specialization of a templated function. The template function calls `memcpy` function from random bytes to the target return value. In the case of the boolean return value, it is effectively similar to assigning `bool` variable via `reinterpret_cast` to **`unsigned char`**. It is an undefined behavior in C++, which triggers various runtime errors. It occurs especially when the compiler compiles the code in full optimization. One of the errors is inconsistent randomization in two different randomization executions.

> **6.9.1 Fundamental types [basic.fundamental]**
>
> Footnote:
>
> 50) Using a **`bool`** value in ways described by this document as "undefined", such as by examining the value of an uninitialized automatic object, might cause it to behave as if it is neither **`true`** nor **`false`**.
>
> *ISO Standard - Programming Languages — C++ (2017)* [28]

The code is fixed by adding template specialization for boolean. By specializing it, the compiler is forced to explicitly convert the expression into a boolean value. It eliminates the error caused by the undefined behavior.

The fix is subsequently removed when the boolean specialization is no longer

Listing 6.1. Original program that causes undefined behavior

```
class DataRange {
  template <typename T, size_t max_bytes = sizeof(T)>
  T get() {
    STATIC_ASSERT(max_bytes <= sizeof(T));
    const size_t num_bytes = std::min(max_bytes, data_.size());
    T result = T();
    memcpy(&result, data_.begin(), num_bytes);
    data_ += num_bytes;
    return result;
  }
};
```

Listing 6.2. Fix for bool data type

```
template <>
bool DataRange::get<bool>() {
  return get<uint8_t>() % 2 == 0;
}
```

required. As of May 2020, the fuzzer generator is extended to also support multiple return values. This enhancement erased the random boolean generation, which was used to determine the function return. However, a static assertion is added to ensure the function is not called with a boolean type argument.

6.2.2 V8 Commit: 2d9313e

The test case generator for the V8 fuzz testing is actively developed. In the recent update, there was a small bug in the test case generator which prevents the test case to be correctly generated. The bug was due to a redundant call to the generator function. This unnecessary call does not write to the output Wasm module, but it consumes the random data. The fix is simply by removing the redundant function call.

6.3 Further Works and Improvement

6.3.1 Improving the Test System

This section proposes some improvements to the test system after evaluating the experiment process conducted for this thesis work.

Improving Test Case Generator

As described in Section 6.1.2, the experiment suffers the limitation from the test case generator that produces meaningless or simplistic test cases. This issue can be addressed by improving the test case generator.

The first aspect that can be improved is to generate a more balanced test case using breadth-first based algorithm. Instead of recursive depth-first based algorithm, breadth-first based algorithm allows the expression tree to be more balanced between its neighboring sub-expression. It may prevent excessive constant-

zero instruction that terminates the expression caused by running out of random sequences.

The challenge with this approach is that designing a breadth-first based algorithm is more difficult compared to depth-first based one. The recursion model used in the depth-first based simplifies the generation logic since the Wasm instruction itself is formed as a tree-like structure. It may require a complete rewrite of the Wasm generator to employ this approach.

Another improvement in the test case generation is to generate more contextual control-flow instructions. The generator needs to be able to generate a proper loop-control variable and use it to produce loop-variant value. This mechanism can emulate several common control structures, such as buffer writing, iterative computation, and complex branching.

Introducing Oracle

The test system can be improved for more general-purpose testing, such as validating the correctness of the JS engine implementation of the Wasm specification. The test system can introduce an oracle to verify the correctness of the executed Wasm program.

Several Wasm simulator exists that can be used to execute Wasm outside of the JS engine. The WebAssembly Binary Tools (WABT) provides several toolkits, including a Wasm Interpreter. This tool can execute the Wasm module and produce the trace result of the execution. The feasibility of using this tool out-of-the-box, however, requires further investigation.

Use High-level Languages

The test system can also benefit by using a randomly generated Wasm that is produced from a high-level language. The random generator generates a high-level language program, such as in C, C++, or Rust, which then compiled to Wasm using the toolchain.

This approach has been used to test compiler correctness [49]. The JS engine is no different from a regular high-level language compiler, so it is possible to employ this approach on the testing. It may also introduce program idioms specific to a high-level language, which may increase the complexity of the test case—for example, pointers, static data, and internal data structure generated by the compiler.

However, this approach relies on the correctness of the high-level language compiler to produce a correct Wasm program. Also, generating a test case in a high-level language can be more difficult compared to generating Wasm instructions. High-level languages have more complex semantics and language constructs compared to Wasm, which only has a simple expression tree structure.

6.3.2 Investigating Wasm Security Claims

Wasm is designed with security in mind to ensure the safety of its use in the web environment. It exercises various security measures, such as sandboxing and code-pointer abstraction [22]. This security claim is a potential area for further research to verify its effectiveness in preventing malicious intent via Wasm technology.

Currently, only a few research covers in this particular area. A recent paper authored by Daniel Lehmann et al. presented a possible vulnerability in the Wasm linear memory model when used through a memory-unsafe language such as C and C++ [32]. The paper argued that the linear memory model does not protect the program against certain types of buffer-overflow attacks. Although the attacker cannot maliciously directly modify the program control flow by, for example, overwriting function return address, an attacker can take several options to modify the program control flow. One of the options is hijacking control variables that are spilled on the linear memory.

6.3.3 Exploiting CPU Bugs

Another interesting subject to explore in Wasm technology is its feasibility to induce hardware CPU bugs. Recent CPU vulnerability in speculative execution, Spectre, allows a malicious program to observe the side effect of branch prediction that can leak private information [31]. Although this vulnerability is considered challenging to exploit, it is also difficult to address.

The original Spectre paper presented the vulnerability proof-of-concept by crafting a JavaScript program that maliciously trains the CPU branch predictor to make a wrong path [31]. It demonstrated that a high-level language used in the web environment is also vulnerable to this attack. Unlike JavaScript, Wasm is closer to a machine instruction has a more direct translation to the target machine. Therefore, Wasm is a more efficient and predictable approach to craft a Spectre gadget where the attacker can selectively craft a machine instruction through the respective Wasm instruction.

Bibliography

[1] WebAssembly. *WebAssembly*. URL: https://webassembly.org/.

[2] Marshaling details. *Microsoft Docs*, May 2018. Accessed: 18-Jun-2020. URL: https://docs.microsoft.com/en-us/windows/win32/com/marshaling-details.

[3] Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[4] Fundamental types. *cppreference.com*, May 2020. Accessed: 18-Jun-2020. URL: https://en.cppreference.com/w/cpp/language/types.

[5] Sebastian Anthony. Firefox sticks it to google with odinmonkey, which can boost javascript performance by 1000% or more. *ExtremeTech*, March 2013. URL: https://www.extremetech.com/computing/151403-firefox.

[6] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie. An empirical comparison of compiler testing techniques. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 180–190, 2016.

[7] Lin Clark. What makes webassembly fast? *Mozilla Hacks – the Web developer blog*, February 2017. URL: https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast/.

[8] Lin Clark. Making webassembly even faster: Firefox's new streaming and tiering compiler. *Mozilla Hacks – the Web developer blog*, January 2018. URL: https://hacks.mozilla.org/2018/01/making-webassembly-even-faster-firefoxs-new-streaming.

[9] Lin Clark. Standardizing wasi: A system interface to run webassembly outside the web. *Mozilla Hacks – the Web developer blog*, March 2019. URL: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/.

[10] TIS Committee. *Executable and Linking Format (ELF) Specification Version 1.2*. Tool Interface Standard (TIS), May 1995. URL: https://refspecs.linuxfoundation.org/elf/elf.pdf.

[11] P.J. Deitel and H.M. Deitel. *Java How to Program: Late Objects Version*. How to Program Series. Dietel/Pearson, 2015.

[12] Barry Doyle and Cristina Videira Lopes. Survey of technologies for web application development. *arXiv preprint arXiv:0801.2618*, 2008. URL: https://arxiv.org/abs/0801.2618.

[13] Sandra Loosemore et al. *The GNU C Library Reference Manual*. Free Software Foundation, Inc., 2020. URL: https://www.gnu.org/software/libc/manual/pdf/libc.pdf.

[14] D. Flanagan. *JavaScript: The Definitive Guide*. Definitive Guides. O'Reilly Media, Incorporated, 2011. URL: https://books.google.fr/books?id=4RChxt67lvwC.

[15] OpenJS Foundation. About node.js®. *Node.js*. URL: https://nodejs.org/en/about/.

[16] OpenJS Foundation. C++ addons. *Node.js v14.4.0 Documentation*. URL: https://nodejs.org/api/addons.html.

[17] The Mozilla Foundation. Javascript. *Mozilla.org*, August 2000. URL: https://web.archive.org/web/20000815053619/http://www.mozilla.org/js/.

[18] The Mozilla Foundation. Overview of the javascript c engine. *Mozilla.org*, August 2000. URL: https://web.archive.org/web/20000815211358/http://www.mozilla.org/js/spidermonkey/apidoc/jsguide.html.

[19] Ben Goodger. Welcome to chromium. *Chromium Blog*, September 2008. URL: https://blog.chromium.org/2008/09/welcome-to-chromium_02.html.

[20] Google. Design elements. *V8 JavaScript Engine - Google Code*, September 2008. URL: https://web.archive.org/web/20080912090906/http://code.google.com/apis/v8/design.html.

[21] D. Graham, E. Van Veenendaal, and I. Evans. *Foundations of Software Testing: ISTQB Certification*. Cengage Learning, 2008.

[22] WebAssembly Community Group. Webassembly core specification. *W3C*, Dec 2019. URL: https://www.w3.org/TR/wasm-core-1/.

[23] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. *SIGPLAN Not.*, 52(6):185–200, June 2017. doi:10.1145/3140587.3062363.

[24] M. Halvorson. *Microsoft Visual Basic 6.0 Professional Step by Step*. Step by Step Developer Series. Microsoft Press, 2003.

[25] Clemens Hammacher. Liftoff: a new baseline compiler for webassembly in v8. *V8 Developer Blog*, August 2018. URL: https://v8.dev/blog/liftoff.

[26] Adobe Inc. Adobe flash player eol general information page. *Adobe.com*, May 2020. URL: https://www.adobe.com/products/flashplayer/end-of-life.html#.

[27] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 2*, September 2016.

[28] ISO/IEC JTC 1/SC 22/WG 21. *Working Draft, Standard for Programming Language C++*, March 2017. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf.

[29] ISO/IEC JTC 1/SC 22/WG 21. *Programming Languages - C++*, March 2020. URL: https://isocpp.org/files/papers/N4860.pdf.

[30] P.C. Jorgensen. *Software Testing: A Craftsman's Approach, Second Edition*. Software engineering series. Taylor & Francis, 2002. URL: https://books.google.fr/books?id=Yph_AwAAQBAJ.

[31] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[32] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of webassembly.

[33] R. Lehtinen. *Computer Security Basics*. O'Reilly Series. O'Reilly Media, 2011.

[34] X.F. Li. *Advanced Design and Implementation of Virtual Machines*. CRC Press, 2016. URL: https://books.google.fr/books?id=jZG_DQAAQBAJ.

[35] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[36] Microsoft. Blazor: Build client web apps with C#. 2020. URL: https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor.

[37] P. Milbourne and D. Richardson. *Foundation ActionScript 3*. Expert's voice in Web development. Apress, 2014.

[38] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi, and Roglia Danilo Bruschi. A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *In Proceedings of the USENIX Workshop on Offensive Technologies (WOOT*, 2009.

[39] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann, Oct 2013.

[40] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface (Online Material Chapter 2.21 - Historical Perspective and Further Reading)*. Morgan Kaufmann, Oct 2013. URL: https://booksite.elsevier.com/9780124077263/downloads/historial%20perspectives/section_2.21.pdf.

[41] R.S. Pressman and B.R. Maxim. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education, 2019.

[42] Q-Success. Web technology survey. *W3Tech*, May 2020. URL: https://w3techs.com/technologies.

[43] C. Simmonds. *Mastering Embedded Linux Programming*. Packt Publishing, 2017. URL: https://books.google.fr/books?id=4Hc5DwAAQBAJ.

[44] Bjarne Stroustrup. *The C++ Programming Language Fourth Edition*. Pearson Education, 2013.

[45] H. Sutter. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. C++ in-depth series. Addison-Wesley, 1999.

[46] A. Takanen, J.D. Demott, and C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House information security and privacy series. Artech House, 2008. URL: https://books.google.fr/books?id=tMuAc_y9dFYC.

[47] M. Tedre. *The Development of Computer Science: A Sociocultural Perspective*. Dissertations / University of Joensuu, Computer Science. University of Joensuu, 2006. URL: https://books.google.fr/books?id=CEtclF-JMzAC.

[48] Seth Thompson. Experimental support for webassembly in v8. *V8 Developer Blog*, March 2016. URL: https://v8.dev/blog/webassembly-experimental.

[49] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011. doi:10.1145/1993316.1993532.

[50] Alon Zakai. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, page 301–312, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2048147.2048224.

[51] Alon Zakai. Why webassembly is faster than asm.js. *Mozilla Hacks – the Web developer blog*, Mar 2017. URL: https://hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js/.

Vita

Gilang Mentari Hamidy is an Indonesian born in 1989. He graduated with Cum Laude honor from Universitas Indonesia in 2011. He received an Erasmus Mundus Joint Master's Degree (EMJMD) scholarship to study a master's degree in Security and Cloud Computing (SECCLO). He conducted his study at Aalto University, Finland, and EURECOM, France, from 2018 to 2020. His main interests are programming languages, compilers, program analysis, and C++.

During his first year of study, he performed an industrial internship working in a static analysis in the compiler. In his internship, he extended an existing static analysis in the LLVM compiler. It aims to provide a suitable analysis required by hardware-based protection research. He also took research projects related to compilers and programming language. He developed a tool to extract and refactor kernel source code to support a memory forensic tool.

Before beginning his master's study, he worked for four years in a multinational consulting firm. He worked as a technology consultant, working in a system information project for a major energy industry in Indonesia. He moved to a technology research and development unit to pursue a career in the technical field, where he continued working for three years.

In the tech industry, he worked in mobile application development using the C++ programming language. He initiated and developed a C++ framework for the project. He also experimented using advanced C++ techniques such as template metaprogramming. From this extensive experience, he delivered a presentation at a C++ international conference in 2018.

He maintains a public blog that is accessible at **https://heliosky.com**. His public profile is also available online at **https://gilang.hamidy.net**.