Automating Post-mortem Debugging Analysis in Node.js

by

Anil Hitang

Bachelor of Computer Applications, Pokhara University, 2014

A MASTER'S REPORT SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Masters of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor:	Kenneth Kent, PhD Computer Science
Examining Board:	Gerhard Dueck, PhD Computer Science Panos Patros, PhD Computer Science

This report is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

June, 2021

Abstract

Post-mortem debugging involves analyzing a raw memory dump of either a portion of memory or the whole memory of the system at an instance in time. The process of post-mortem debugging is complex and time consuming, as it requires many operations in order to be executed. Automating post-mortem debugging can potentially improve its effectiveness in localizing software faults or bugs. The aim of this study was to implement an automated post-mortem debugging solution for Node.js from the information available in the bug reports. The automated solution was aimed to automatically process the bug reports and generate more concrete results in order to be used in troubleshooting software faults such as memory-leak problems and optimize memory usage in Node.js applications. Overall, the implementation of an automated solution was only partially achieved in this study.

Dedication

I'd like to dedicate this report to those who have stood by me through my difficult times.

Acknowledgements

This research was conducted within the Centre for Advanced Studies–Atlantic, Faculty of Computer Science, University of New Brunswick. The authors are grateful for the colleagues and facilities of CAS–Atlantic in supporting our research. The authors would like to acknowledge the funding support of the Natural Sciences and Engineering Research Council of Canada (NSERC), 501197-16. Furthermore, we would also like to thank the New Brunswick Innovation Foundation for contributing to this project.

Table of Contents

List of Tables

List of Figures

List of Symbols, Nomenclature or Abbreviations

LLDB	A low-level debugger component of the LLVM project.
LLNode	A plugin for the LLDB debugger.
N-API	A toolkit introduced in Node 8.0. 0 that acts as an intermediary between C/C++ code and the Node JavaScript engine.
ELF	Executable and Linkable Format is a common standard file format for executable files, object code, shared libraries, and core dumps.
DWARF	A debugging file format used by many compilers and debuggers to support source-level debugging.

Chapter 1

Introduction

In recent years, software engineers have put their knowledge and efforts to produce software that runs smoothly without any intervention or bugs. Regardless of best efforts and practices, there will always be some unhandled issues. Such issues might arise at some point in time when the application is used by real users under real circumstances. At the point when this occurs, it is important to quickly understand such failure. The process can be termed as fault localization in debugging [31]. The identified faults have to be fixed and the solutions have to be deployed to prevent other issues. Post-mortem debugging provides an alternative to the iterative traditional debugging processes. Traditional debugging approaches require developers to repeatedly set breakpoints and rerun the program. The dynamic nature of JavaScript and extensive use of callbacks/promises in Node.js can make this process difficult and time-consuming [26]. Post-mortem debugging is debugging the program after it has already crashed. It involves analyzing a memory dump of a portion of the memory of the system at an instance of time [2]. Normally, this memory snapshot is taken when the system has crashed or has terminated abnormally. Memory snapshots will be different from each other, due to the way that the operating system manages its memory. This report will focus on Linux post-mortem debugging specifically. Linux

memory dumps are usually referred to as core dumps or core files [25]. Currently, the concept of post-mortem debugging is popular in areas such as operating systems and their native execution environments, but it is less explored in dynamic environments such as JavaScript and Node.js. Postmortem analysis was arguably less critical for such an environment in the past because crashes are less important in these environments: most end-user applications save work often anyway, and after a crash, the operating system or browser will also restart the application. However, these crashes still reflect user experience disturbances, and postmortem debugging is the only hope of understanding such failures [26]. The method of post-mortem debugging is time-consuming since it necessitates many file operations. Post-mortem debugging currently takes a significant amount of time due to the numerous variables that must be considered in order to effectively interpret the data produced during a system crash. The objective of this study is to provide a solution for automating post-mortem debugging in Node.js which automatically parses bug reports (in the form of core dumps) and capture useful information for debugging the application. The study is focused around the development of a debugging module for Node.js, which is aimed at providing in-depth heap memory analysis, parsing the core dumps. In other words, generate heap snapshots from linux core dumps, which could be useful in fault localization and improving post-mortem debugging experience in Node.js. Furthermore, cloud-based development frameworks based on Node.js are becoming more common. One of the most challenging aspects of cloud-based development platforms is ensuring that developers can debug their code efficiently. However, when developing a remote service, the debugging client and the server running the actual code are isolated, revealing several issues that would not be visible in a typical debugging scenario. Generally, when debugging such issues in remote systems, developers dig into the error logs to do a postmortem analysis, which can be a needle in a haystack problem [29]. The approach described in this study, to automatically parse

bug reports and perform post-mortem analysis, has the potential to help with remote debugging issues, since the debugging method is based on the existing post-mortem approach, but with an automated process. Additionally, in the field of DevOps', one of guiding the principles is to automate everything [18]. It pervades the continuous delivery pipeline, from the developer's workspace to efficiently managing applications and systems in development. Debugging in a live environment can be difficult for operational factors, software such as ltrace and SystemTap/DTrace are commonly used to investigate problems. If more visibility into what is in memory is required, a core dump could be generated without killing a running process, which can then be used for further analysis.

Chapter 2

Background and Related Work

Since the beginning of software development, much effort has been made to prevent and reduce the impact of software faults. This chapter reviews the basic terminologies that are utilized in this report to develop a debugging module for Node.js.

2.1 Post-mortem Debugging in Dynamic Environments

Post-mortem debugging is a technique that refers to the concept of entering into the debug mode after the application has already crashed. Unlike traditional debugging, in this approach there is no setting up of breakpoints involved, which would make it quick to inspect the stack traces and errors. Post-mortem debugging requires a core dump file and a Node.js executable from the time of the application crash [2]. Whereas the post-mortem analysis techniques are well developed and widely used in areas such as operating systems and their native execution environments, post-mortem analysis is a comparatively less explored topic in the realm of dynamic environments such as JavaScript. More significantly, dynamic languages such as Node.js are growing in popularity as building blocks for larger distributed systems, where

minor software faults, if neglected, could lead to critical faults later. As a result, as with operating systems and core services, it is important to fully understand each failure to achieve the desired reliability levels of such fundamental software. Providing a post-mortem facility for dynamic environments, however, is not easy. While native programs can leverage operating-system support for core dumps, dynamic languages must present a post-mortem state using the same higher-level abstractions with which their developers are familiar.

2.2 Node.js

Node.js is a JavaScript-based platform built on Google Chrome's JavaScript V8 Engine [10]. It is one of the widely accepted technologies for real-world web application development because of its performance, scalability and cross-platform development nature. Node.js has been a part of many modern web applications such as video streaming sites, single-page applications, and other web applications. It is built on top of Libuv [3], which is a support library with the implementation of event-loop, asynchronous sockets, DNS resolution, file system operations, etc., all of which combined, makes Node.js a great candidate for data-intensive and real-time applications that run on a distributed platform.

2.3 Core Dumps

A core dump is a file containing the exact state of an application's memory at the time of an application crash. A typical core dump file may include information such as: the processor's state, processor's register contents, memory management information, program counters, stack pointers and other OS-level information. Core dump files must adhere to a common structure because they contain more information than the plain memory of the target process. In particular, they comply to the specification of

ELF Header
Program header table
Segment 1
Segment 2
...
Section header table optional

Table 2.1: The ELF object file format in the execution view is described in the ELF specification [4]

.

the Executable and Linking Format (ELF specification) [4]. The ELF specification defines three different types of object files: relocatable files, executable files, and shared object files [4]:

- Relocatable files contain information that can be used to create executable files or shared object files.
- Executable files contain information about the program to be executed, in particular its memory segments and the starting address.
- Shared object files can be used among multiple executable files. They contain both code and data.

Core dump files do not need an explicit type declaration because their data fits with the definition of executable files. The primary purpose of both executable files and core dump files is to create a memory image. With executable files, a new memory image is created, whereas with core dump files, the memory image of a process at a certain state is restored. There are two relevant views for showing the contents of an ELF file: the execution view and the linking view. While the execution view provides information relevant when executing a file, the linking view focuses on information used by linkers. Table 2.1 shows the execution view for ELF files. The linking view is irrelevant for the scope of this report [4].

An ELF header resides at the beginning of the file and contains a "route map" defining the file's organization. An overview of the ELF header is described in

Subsection 5.2.1. Data in an ELF file can be stored either in sections or in segments. While sections contain data that is used during linking, segments contain data that is used at runtime. Sections and segments are defined in the section header table and in the program header table, respectively. As segments hold information that is required to create the memory image, the program header table plays an important role in analyzing core dumps. Linking information is irrelevant for core dump files. Thus, they usually do not contain any sections.

2.4 Heap Dump

All non-primitive data types are stored in the heap memory and stay there until garbage collection removes those that it determines to be garbage. A heap dump is a snapshot of the current heap memory, which contains a snapshot of all live objects that are present in the heap memory. It contains all the internal, user-defined variables and allocations [1]. The structure of the heap snapshot we generate in this report is below [9].


```
{"snapshot": {
    "meta": {
      "node_fields": ["type","name","id","self_size", "
         edge_count","trace_node_id"],
      "node_types": [["hidden","array","string","object","code
         ","closure","regexp","number","native","synthetic","
         concatenated string","sliced string"],
      "string","number", "number","number","number","number"],
      "edge_fields": ["type","name_or_index", "to_node"],
      "edge_types": [
        ["context","element","property","internal","hidden","
           shortcut","weak"],"string_or_number","node"],
      "trace_function_info_fields": ["function_id","name","
         script_name","script_id","line", "column"],
      "trace_node_fields": ["id","function_info_index","count
         ","size", "children"],
      "sample_fields": ["timestamp_us","last_assigned_id"]},
      "node_count": 35968,
      "edge_count": 158089,
      "trace_function_count": 0},
      "nodes": [9, 1, 1, 0, 4, 0,
      ....
      9, 1158, 2274944298, 0, 1, 0],
      "edges":[1, 1, 6,
      ....
```

```
    3, 178, 215796],
    "trace_function_infos":[],
    "trace_tree":[],
    "samples":[],
    "strings":["<dummy>",
      "",
    ....
  "bound_argument_2",
  "get listening"]}
```


2.5 LLVM, LLBD, LLNode and N-API

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. A low-level debugger (LLDB) is the default debugger component of the LLVM project [11]. It is built as a set of reusable components which extensively use existing libraries from LLVM, such as the Clang expression parser and the LLVM disassembler. The llnode is a plugin that adds the ability to inspect JavaScript stack frames, objects, source code and more to the standard C/C++ debugging facilities when working with Node.js processes or core dumps in LLDB [12]. The Node-API is an API (Application Programming Interface) for developing native Addons. It is maintained as part of Node.js and is independent of the underlying JavaScript runtime (for example, V8). APIs exposed by Node-API are generally used to create and manipulate JavaScript values [13].

2.6 Automated Debugging Solutions

Software testing and debugging is an important topic, almost half of the resources allocated to any software project are used for software testing [23]. There are considerable numbers of studies conducted on debugging, but very few on post-mortem debugging. Moreover, the field of automated debugging is even less explored.

Parnin and Orso [27] investigated the benefits of automated debugging tools and techniques for developers through a set of human studies. The results produced

through the experiments showed positive results. The authors concluded that the programmers who debug with the assitance of automated debugging tools and techniques locate bugs faster. Addtionally, authors also believed that the effectiveness of an automated technique is also affected by the increase in the complexity of the debugging task. The study showed both positive and negetive results [27].

Umukoro and Okordudu [30] provided an overview of automated debugging systems. The authors discussed existing automated debugging solutions and their benefits and limitations.

Lwakatare et al. [22] mention debugging taking a significant amount of human resources and automated debugging could greatly improve the workload of the software. The authors studied the benefits of automated debugging of embedded systems. The authors also agreed that embedded systems require special knowledge, they run on specialized hardware and have specific limitations, which limits the availability of commonly available debugging tools. So, this fact makes the use of existing debugging information more reasonable and cost effective to designing a high-level custom tailored debugging solution.

Kuzara et al. [21] agrees with these facts and suggests if there is an already setup debugging environment, there are a number of options for adding extra debug information with minimal processing and development cost while improving the efficiency of debugging.

Hoyecki [19] and Huselius et al. [20] mention that post-mortem debugging provided good, quick accurate information of the base system and its running processes compared to various available tools and techniques.

2.7 Summary

Debugging is a time-consuming and challenging task. Researchers have invested a significant amount of time developing automated techniques and tools to assist with various debugging tasks. Despite their possible usefulness, most of these techniques have yet to be proven in practice [27]. Despite our best efforts, the related work section of this study was unable to locate any research work that is attempting to put together an artifact of extremely similar functionality and end goal. Previous studies that have been identified either use automated post-mortem debugging on embedded systems or multi-processing environments, and different language runtime environments. Currently, there are no studies on automated post-mortem debugging techniques in Node.js. So, in this study, we have focused on the importance and implementation of an automated solution for post-mortem debugging in Node.js.

Chapter 3

Research Problems and Method

The aim of this study is to develop a solution for automating post-mortem debugging in Node.js, that is superior to the existing solutions and methods. This chapter will talk about the research problem and method, by defining research questions.

3.1 Research Questions

In this report, we seek to address two research questions that would provide insight into post-mortem debugging in Node.js. The research questions (RQ) have been formulated as follows:

- **RQ1**: Why is an automated solution for post-mortem debugging in Node.js necessary?
- **RQ2**: How can an automated solution for post-mortem debugging in Node.js be implemented?

By answering **RQ1** and **RQ2**, we could gain more insight on post-mortem debugging scenarios in Node.js.

3.2 Generating Core Dump from Test Code

The code below is to run Monte Carlo simulations inside docker containers using WebWorker threads [24]. The code shows a "Segmentation Fault" error occasionally. This was the starting point for this study and the first research question: Why is an automated solution for post-mortem debugging in Node.js necessary?

To further study the cause of the occasionally recurring segmentation fault, we recorded the dump for the error for the module with one worker and 2,000,000 points. The following syntax records the core dump inside the working directory.

In the terminal, we ran the Monte Carlo Simulation:

```
node --abort-on-uncaught-exception app.js 1 2000000\\
```

The source code for Monte Carlo Simulation (app.js):

```
console.time("Webworker-threads App");
console.log("#WorkerNum:",process.argv[2]," Points: ",process.
   argv[3]);
console.log("Main Process",process.pid);
function run(cb){
 const Worker = require('webworker-threads').Worker;
 var numWorkers = process.argv[2];
 var points = process.argv[3];
 var allDone = 0;
 var aggregate = 0;

 function fn$(){
   var estimatePI;
   estimatePI = function(n){
     var i = n;
     var inside = 0;
     while (i-- > 0) {
         var x = Math.random();
         var y = Math.random();
         if ((x * x) + (y * y) <= 1) {
             inside++;
         }
     }
     return inside / n * 4;
   };
   return this.onmessage = function(arg$){
     var data;
     data = arg$.data;
```

```
    postMessage(estimatePI(data));
    self.close();
  };
};
function fn1$(arg$){
  var data;
  data = arg$.data;
  aggregate+=data;
  if (++allDone == numWorkers){
    var pi = aggregate/numWorkers;
    console.timeEnd("workers");
    return cb(pi);
  }
};
console.time("workers");
var worker;
for (var i = 0; i < numWorkers; ++i) {
  worker = new Worker(fn$);
  worker.onmessage = fn1$;
  worker.postMessage(points);
}

};

run(function(data){
 console.timeEnd("Webworker-threads App");
 console.log("PI:",data);
 console.log( JSON.stringify(Date.now() )+' PI: ', data);
 const used = process.memoryUsage().heapUsed / 1024 / 1024;
 console.log(`The script uses approximately ${used} MB`);
});
```

Initially the issue was analyzed with tools such as GDB and Valgrind, observing the stack traces generated by such tools. Later, it was determined that a more rigorous investigation was necessary. For investigating Node.js core problems, LLDB with LLNode was more fitting.

3.3 Analyzing Linux Core Files

This section discusses core dumps in more depth and what kind of useful knowledge can be derived from program analysis, especially for debugging Node.js applications. Firstly, we provide a general overview of the ELF file format, which is used to store machine code and other related program data on Linux, and which will be addressed later.

3.3.1 ELF

ELF stands for Executable and Linking Format, which is a file format used on Linux and other Unix-like operating systems [17]. ELF can be used for:

- Object files (used during compilation and linking)
- Executable files
- Shared libraries
- Core dumps

The ELF format specifies a mechanism for the compiler to put debugging information into the executable in a format called DWARF, which can be used by the debugger to restore information like the variable names, data-structures and source code, etc. [17]. However, this study does not deeply explore ELF, nor the life cycle of a linux program nor how the operating system generates core dumps internally. Most of the low-level executions are handled with LLDB and llnode.

3.3.2 Debugging Information

In order to comprehend issues or bugs during the development phase, developers quite often follow a naive method to insert statements into the program source code that prints out the values of the suspicious variables. The more advanced approach is to run the program in a debugger, that allows developers to interrupt the program execution at the exact location in the code and inspect the values of arbitrary variables [17]. The debugger enables developers to investigate the source code in a more intuitive manner, abstracting machine level details. To make this possible, the debugger requires some additional information that relates the machine code to the source code. This debugging information can be generated by compilers alongside the program executable. In particular, debugging data makes it possible to

map code addresses to lines in source code files, obtain the value of named variables, provide information about complex data types and analyze stacktraces, etc.

3.3.3 DWARF

DWARF (Debug With Arbitrary Record Format) is a format for debugging information for ELF files. The ELF format specifies a mechanism for the compiler to put debugging information into the executable in a format, called DWARF, which can be used by the debugger to restore information like the variable names, data-structures and source code, etc. [17]. DWARF debugging information can be held in designated sections of the ELF file of a program or a shared library, or kept in a separate file, which is also an ELF file, but does not contain the machine code and usual accompanying metadata. Such information consists of several parts, such as Debugging Information Entry and unwind tables. Typical developers do not need it since the files can be very large. For the purpose of this report, we have used LLDB and llnode debugger to parse the debugging information.

3.4 Examining the Core Dump

A core dump contains the contents of the process address space in the form of contiguous memory segments together with some additional machine states. The analysis of core dumps generally relies on two types of data: (1) data that is directly retrieved from various parts of the core dump file and (2) data that can only be retrieved by inspecting the memory of the targeted process. The former can be read from the core dump file, whereas, the latter can only be extracted after the memory image is retrieved, in addition to the core dump file requiring all loading of binary files and information.

3.4.1 Post-mortem Diagnostics

Post-mortem diagnostics is basically a technical terminology for debugging core dumps created during a crash of a running process. Depending on the configuration of the system, a dump capturing the state of the crashed process is written to the disk before the process terminates. Some linux servers might be configured to generate core dumps, which are like snapshots of the memory space of the crashed process and other useful data. In the Windows operating system this format is called mini dump format. There are also other formats that can be configured by the user, for example, a node core module that is integrated into the Node.js core can generate the summary with Node.js specific information when a Node.js process crashes [14]. Figure 5.1 depicts the post-mortem diagnostic procedure. In this section, we will focus on core dumps and specifically linux core dumps.

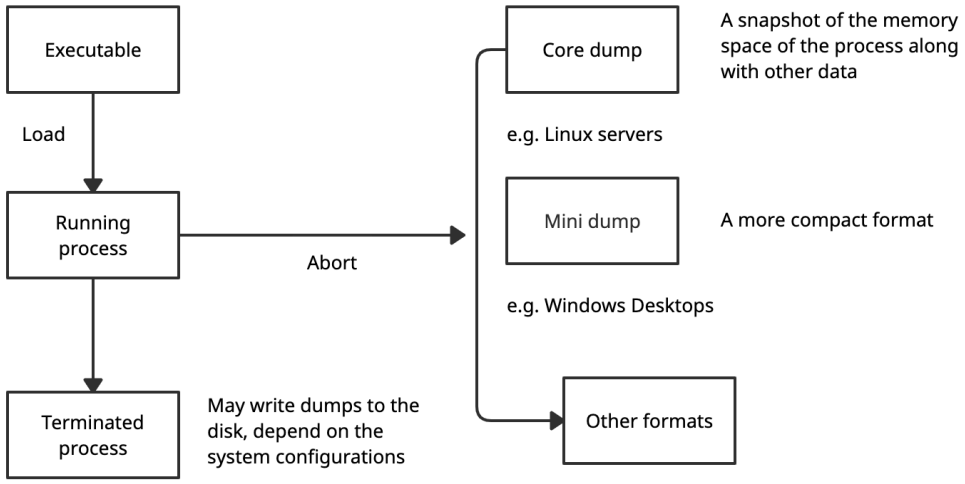

Figure 3.1: Post-mortem diagnostics [14]

As discussed above, linux executable binaries are usually encoded in the ELF format. The ELF format specifies a way for the compiler to put debugging information (DWARF format) into certain sections inside the executable files [17].

3.4.2 Recovering States from Dynamic Language VMs

Debuggers like GDB or LLDB can map the debugging information in the executable onto the process snapshot from the core dump to reconstruct the state of the process during the crash. For programs written in the static languages like C/C++, we can use LLDB to learn about things like function calls or the state of the variable during the crash. However, programs written in dynamic languages run by a language virtual machine may be harder to debug with such native debuggers because the debugging information like the type information is dynamically generated during the runtime. In that case, this dynamically generated debugging information is in the core dump instead of the executable file. Also, the debugging information might vary depending on the implementation of the virtual machine [14].

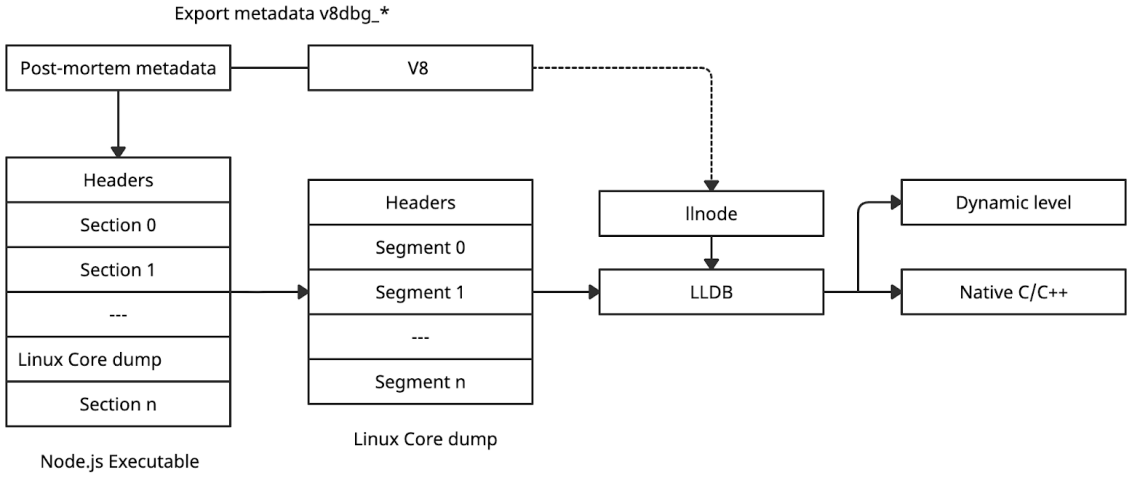

Figure 3.2: Analysis of Linux Core dump [28], [14]

To assist LLDB with the object model of a dynamic language runtime like V8, requires additional debugging information available in the executable. So in V8, the layout of the JavaScript values and frames are defined by a set of static offsets and constants. As shown in Figure 5.2, these values are exported by V8 into the Node.js executables known as post-mortem metadata, they are the symbols prefixed with `v8dbg_*` [8]. Using the post-mortem metadata exported by the V8, llnode is able

to understand the V8 object model and reconstruct JavaScript states such as objects in the heap.

3.4.3 Reconstruct JavaScript values from the Core Dump

In V8, the heap memory is organized in pages, in each page the memory is organized into words and they are aligned [16]. In a 64-bit machine [14], the words are in 8 bytes, so given the arbitrary address of a word in memory, the last bit of the word can be examined first (Figure 5.3). If the last bit is 0, then it is a Small Integer (SMI), where the first 32 bits are useful to store signed integers and their values can be displayed by interpreting those 32 bits. Most of the time, SMIs are just primitive data types pointed to other values.

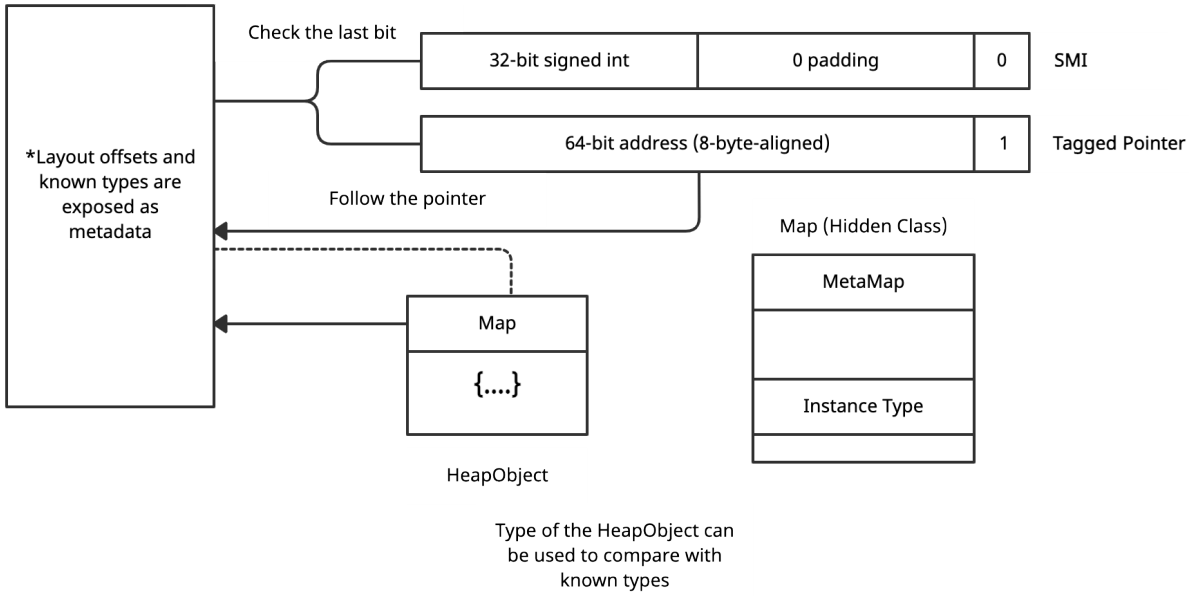

Figure 3.3: Reconstruct JS Values from Raw Memory [14]

In the context of V8 [14], if the last bit of the word is 1 it generally represents a pointer that could contain an address of another block in the memory. V8 aligns the memory blocks in eight bytes, so the addresses are multiple of eight which means the last three bits of these are all zeros. To interpret the word as an address, the first 61

bits could be padded with three zeros and interpreted as tagged pointers. Tagged pointers can be used to look for another block in the core dump. In V8, the tagged pointers point to a block of memory representing a heap object and the first word of the heap object block is a pointer to a map, also known as the header class that contains meta information of the object. The map pointer in the heap object again can be followed to read another block of memory that represents the map, so here map is also a type of heap object called meta map. The field at a given offset in the map of the heap object can be read, which contains the information about the type of the heap object instance [14]. The type information can be compared with the known types/constants available as the post-mortem metadata to obtain further information regarding interpretation of this object.

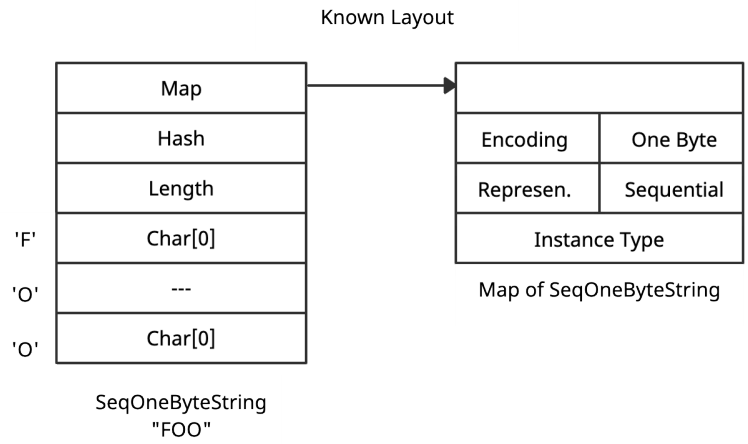

Figure 3.4: Known Layout for heap object type in V8 [14]

In V8 [14] [16], there are many different types of heap objects, some of them have a well- known layout which can be interpreted using information present in the metadata eg. ASCII strings in V8. However, there are also some values with flexible structure that are dynamically generated during the runtime. Most of the JavaScript objects in V8 of type JS objects have flexible structure and there are multiple ways to store the properties of these objects (Figure 5.4).

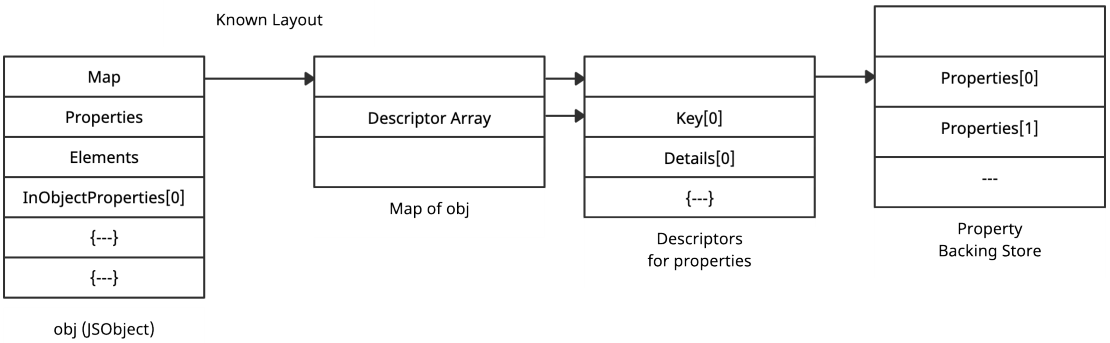

Figure 3.5: Object Representation in V8 [14] [16]

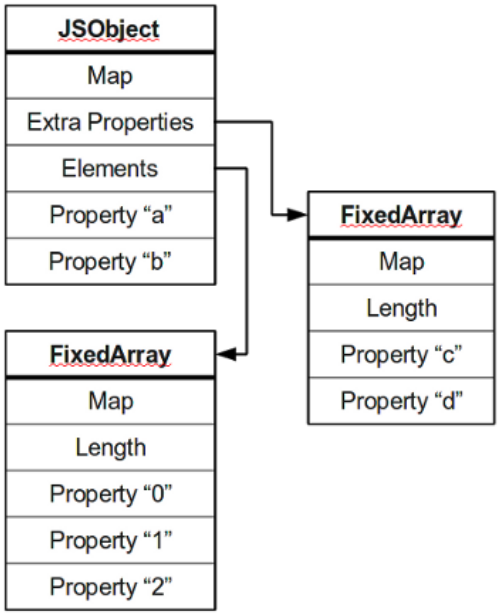

Figure 3.6: Simple Diagram of Object Representation in V8 [16]

In V8 [14] [16], JS objects contain a hidden class called Map and two pointers, one for named Properties and another for Elements, that are basically properties with indexed keys (Figure 5.5). To interpret the named properties, information in the Map of the object can be used, which contains a pointer to the descriptor array, which then contains key names and details of each property as shown in Figure 5.5. Furthermore, for fast property access, objects in V8 have an InObjectProperties attribute that

stores a limited number of properties directly in the object after the two pointers i.e., Properties and Elements. InObjectProperties are also available in the Descriptor array. The contents of Elements can be viewed through the Property Backing Store pointed to the Elements pointer in the JS object (Figure 5.6). The objects that contain too many properties can be put into a flexible property dictionary instead of a set of fixed descriptors.

3.4.4 Call Stack Unwinding

Native debuggers like LLDB can unwind the native call stack by looking at register values and unwind conventions encoded in the executable in the form of debugging information, interpreting the debugging information and restoring the frames in the call stack [14] [7]. LLDB is able to restore the symbols like function names, local variables and arguments from the C/C++ stack frames natively by using the debugging information in the executable file, but by the C/C++ compiler. Whereas, in the case of dynamic languages such as JavaScript, LLDB is unable to interpret JavaScript values from the debugging information. The debugger is only able to unwind the frames and see their locations, but it is unable to interpret JavaScript values.

In this case [14], an LLDB plugin known as llnode is useful to interpret the debugging information specific to JavaScript values. Most of the types of frames in V8 are usually formatted with a fixed header that contains a pointer to a JS function object. Therefore, since the pointer to a stack frame, llnode, first attempts to locate the JS function pointer and known offsets, then attempts to interpret the object to which it refers, the JS function object. The JS function object contains a pointer to a SharedFunctionInfo object, which points to the Script object (Figure 5.8). The information regarding the JavaScript function called in the frame including the source code of the function, the line number of the function statement, etc. can be retrieved

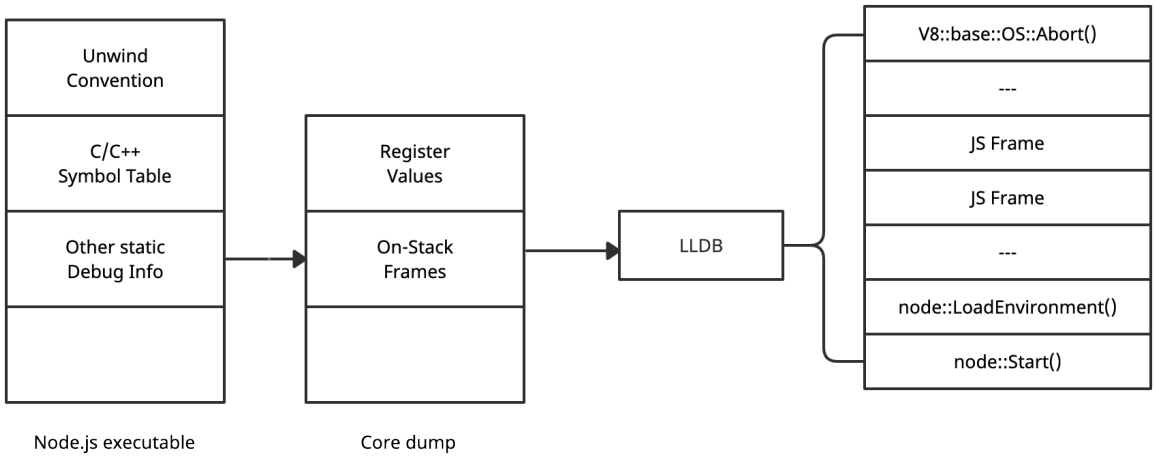

Figure 3.7: Unwinding the Native Stack: LLDB [16]

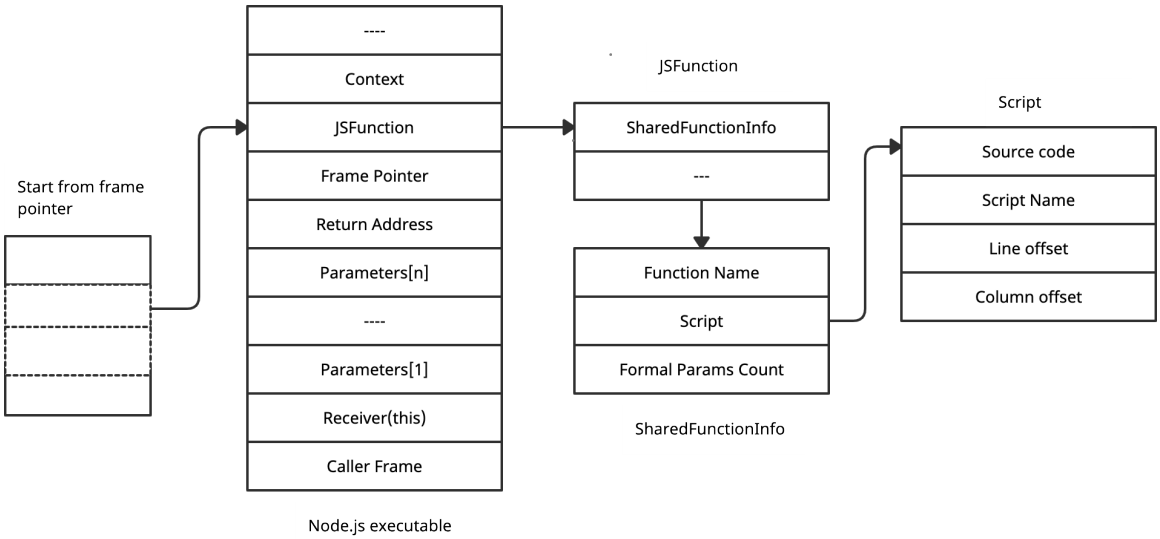

Figure 3.8: Unwinding the JS Values and Symbols: LLDB and lldb [16]

using the Script object. The SharedInfoObject also contains the number of formal parameters in the function as an attribute shown in Figure 5.8. V8 lays out the arguments matching the formal parameters as well as the receiver i.e. this object sequentially at another offset from the frame pointer. V8 sets the arguments that match both the formal parameters and the receiver, i.e., this object, sequentially at another offset from the frame pointer [14].

3.4.5 Heap Analysis

Languages like Java, support garbage collection, which frees programmers from manually dealing with memory allocations. Heap analysis allows programmers to access memory and information about every object and their references or obtain a complete snapshot of memory to analyze later in more detail. Most modern debuggers allow programmers to examine a particular variable or object by stopping a program's execution, but unfortunately lack in providing coherent ways to examine objects and their relationship within the heap memory. The most common approach to examine memory is through heap dumps instead of monitoring the execution of the program. Heap analysis can be used as a way to find the cause of a fault without source code analysis [15]. Previously, many researchers investigating fault localization have invested a considerable amount of effort in developing different tools and techniques based on heap memory analysis [31].

3.5 Description of the Code

The automated solution to convert core dump into heap snapshot is currently an underdeveloped prototype, which is entirely built on top of LLNode and LLDB. As our approach towards automating post-mortem debugging in Node.js, the module processes core dumps as a means of bug reporting, and performs a complete automated heap analysis to provide a constricted report in the form of heap snapshots. Our approach was to extend LLNode, since LLNode is able to understand the V8 object model and reconstruct JavaScript states and objects in the heap. To implement the module, we started the development on top of LLNode's existing code base. The command "createsnapshot" executes our module within the LLNode command line interface (Figure 5.9).

The HeapSnapshotJSONSerializer is the main implemented class, which contains all

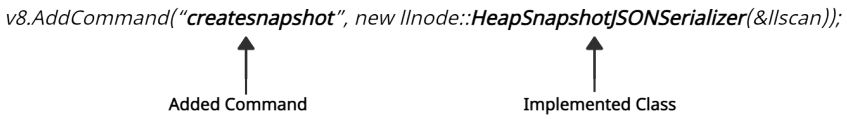

Figure 3.9: Implementation details

the methods required for the module to run. The full source code is publicly available as a forked, open-source project hosted on GitHub [6].

There are the following C++ classes:

1. **HeapSnapshotJSONSerializer** class that includes all methods responsible to serialize data necessary to create a heap snapshot. The list of methods implemented in this class are as follows:

   - RecordEntry: Go through all the V8 object instances in the core dump and handle them according to their type and add the objects to the graph structure. The function utilizes LLNode methods such as GetMapsToInstances and GetInstances to extract object instances from raw memory addresses.
   - GetInstanceType: The function takes the 64-bit word argument and scans for different object types in V8 mapped as an enum. The most object types it scanned by the method are as follows:

     kHidden = 0, kArray = 1, kString = 2, kObject = 3, kCode = 4, kClosure = 5, kRegExp = 6, kHeapNumber = 7, kNative = 8, kSynthetic = 9, kConsString = 10, kSlicedString = 11, kSymbol = 12, kBigInt = 13, kInvalid = -1
   - GetStringId: Returns the id of the object instances.
   - GetChildrenCount: Returns the number of elements and properties of V8 objects.

- ImplementSnapshot: Creates a structure for the heap snapshot.
- SnapshotSerializer: Creates static content of the heap snapshot structure.
- SerializeNodes: Serializes nodes data.
- SerializeNode: Outputs specific nodes to the heap snapshot structure.
- GetNodeSelfSize: Returns the size of the node i.e., V8 object.
- SerializeEdges: Serializes edges data.
- SerializeEdge: Outputs specific edge to the heap snapshot structure.
- SerializeStrings: Serializes all the string data in the heap memory.
- SerializeString: Outputs specific strings on the heap snapshot structure.

2. **HeapGraphNode** class that includes attributes necessary to construct the nodes in the graph structure [9] Eg.:
   - node_id: represents id of the V8 heap object.
   - name: represents name of the V8 heap object.
   - address: memory address of the V8 heap object.
   - size: represents size of the node or the V8 heap object.
   - children: represents the number of edges for individual nodes.
   - trace_node_id: unknown
   - type: represents all the V8 heap object types, such as kHidden, kArray, kString, kObject, kCode, kClosure, kRegExp, kNative, kSynthetic, kConsString, kSlicedString, kSymbol, kBigInt, and kInvalid.
3. **HeapGraphEdge** class which includes attributes necessary to construct the edges in the graph structure [9] Eg.:
   - to_node_id: represents connecting to node.

- name_or_index: represents the name or index of the edge between two nodes.
- from_node_id: represents connecting from node.
- to_address: memory address of the connected to node.
- from_address: memory address of the connected from node.

Chapter 4

Results

4.1 Generated Heap Snapshot

This chapter aims to discuss the results of the study. The objective of the study was to provide an automatic solution to process bug reports, a core dump in this case, and create a heap snaphot from it, which consists of more specific details. The module developed in this study produces a heap snapshot from core dump in order to be used in troubleshooting software faults, such as memory leak problems, optimizing memory usage and improving the overall post-mortem debugging experience in Node.js. The entire contents of the acquired heap snapshot is listed in Figure 6.1. The acquired heap snapshot is at a preliminary stage and misses information on some detailed contents such as retained heap. Retained heap is the amount of memory that will be freed when the particular object is garbage collected. This information is extremely useful to troubleshoot memory-leak problems and optimize memory usage in Node.js applications. Unfortunately, the obtained snapshot is also missing information such as identifying the root elements in the memory. The distance of an object from the root object determines how long it takes to load and process.

```
{"snapshot":{"meta":{"node_fields":["type","name","id","self_size",
"edge_count","trace_node_id"],"node_types":[["hidden","array","string",
"object","code","closure","regexp","number","native","synthetic",
"concatenated string","sliced string"],"string","number","number",
"number","number","number"],"edge_fields":["type","name_or_index",
"to_node"],"edge_types":[["context","element","property","internal",
"hidden","shortcut","weak"],"string_or_number","node"],
"trace_function_info_fields":["function_id","name",
"script_name","script_id","line","column"],"trace_node_fields":
["id","function_info_index","count","size",
"children"],"sample_fields":["timestamp_us","last_assigned_id"]},
"node_count":12313,"edge_count":8353,
"trace_function_count":0},
"nodes":[9,1,1,0,0,0
,9,2,3,0,0,0
,1,3,5,32,1,0
,1,3,7,32,1,0
,1,3,9,32,13,0
.....
.....
"edges":[1,4,18
,1,4,18
,1,4,18
,1,4,18
.....
.....
],
"trace_function_infos":[],
"trace_tree":[],
"samples":[],
"strings":["<dummy>",
"",
"(GC roots)",
"(Array)",
"<non-string>",
"Host",
"localhost:1337",
"User-Agent",
"curl/7.58.0",
"Accept",
"*/*",
"isArrayBuffer",
"isArrayBufferView",
.....
.....
```


Figure 4.1: Generated Snapshot Output from Core Dump

4.2 Parsed Output of the Heap Snapshot

The parsed result of the obtained snapshot file is provided in order to depict the contents of the snapshots organization internally. Snapshot parsing was done with a module named "heapsnapshot-parser" [5]. The contents are arranged in the form of a graph with nodes and edges. Nodes represent a number of objects and are labelled using the name of the constructor function that was used to build them. Edges are the properties of the objects and are labelled using the names of properties [1].

```
Node {
  type: 'object',
  name: 'WriteStream',
  id: 24897,
  self_size: 104,
  edge_count: 20,
  trace_node_id: 0,
  references:
   [
       Edge {
           type: 'property',
           name_or_index: 'finalCalled',
           name: 'finalCalled',
           toNodeIndex: 1832,
           toNode: [Node],
           fromNode: [Circular] },
        ............
        ............
        ............
Node {...}
Node {...}
...........
...........
```

4.3 Heap Snapshot Imported in Chrome DevTools

The generated heap snapshot can be imported to Chrome DevTools Heap Profiler for examination. The Heap Profiler displays the contents of the file in an organized structure. Figure 6.2 contains an example where the column Constructor shows individual nodes, the Distance column normally shows the number of property references on the shortest retaining path from the root node, since the information captured is still incomplete, the Distance column is empty. The Objects Count shows the

number of objects associated with individual nodes. The Shallow Size column shows the amount of memory that is held by the object itself. Additionally, the Retained Size column shows the memory that is freed once the object itself is deleted along with its dependent objects that were unreachable from the root object [1]. Unfortunately, the heap snapshot parser is currently unable to collect all reachable objects from the roots, as the paths from the root cannot be traced at this time, that is, to monitor all live objects in the heap after garbage collection. Since the roots are not available, the distance of an object from the root object and also how long it takes to load and process, cannot be determined at this stage. More research is needed to fully comprehend this topic. Therefore, the data in the Retained Size column is inaccurate.

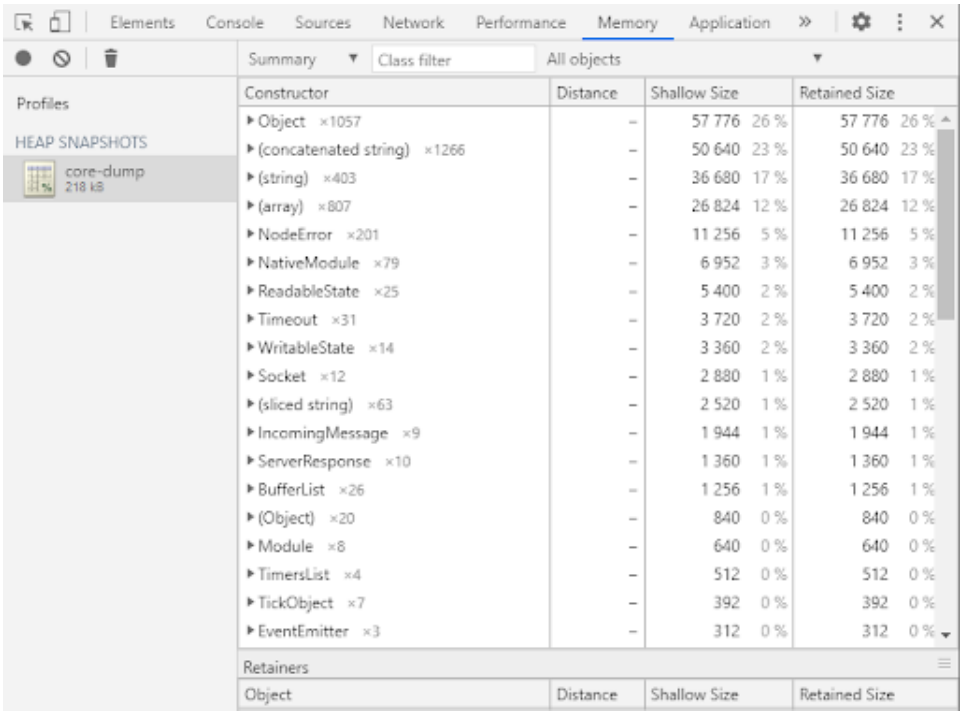

Figure 4.2: Importing Heap Snapshot in Chrome DevTools

4.4 Data Collection Statistics

Figure 6.3 shows the level of progress in parsing the core dump files and retrieving information to construct the heap snapshot. As the development of the module progressed, more data was added to the snapshot. Figure 6.3 shows the number of heap snapshots generated as output of automatically parsing a single core dump. Each snapshot in the figure is based on the amount of data extracted from the core dump over time, with the progress in the parser. The snapshots contain information, such as a list of object types with their properties that reference an Array, String, Number, or regular expressions, and a count of references to a group of objects through function closures, etc. Furthermore, everything related to compiled code is also included in the heap snapshot.

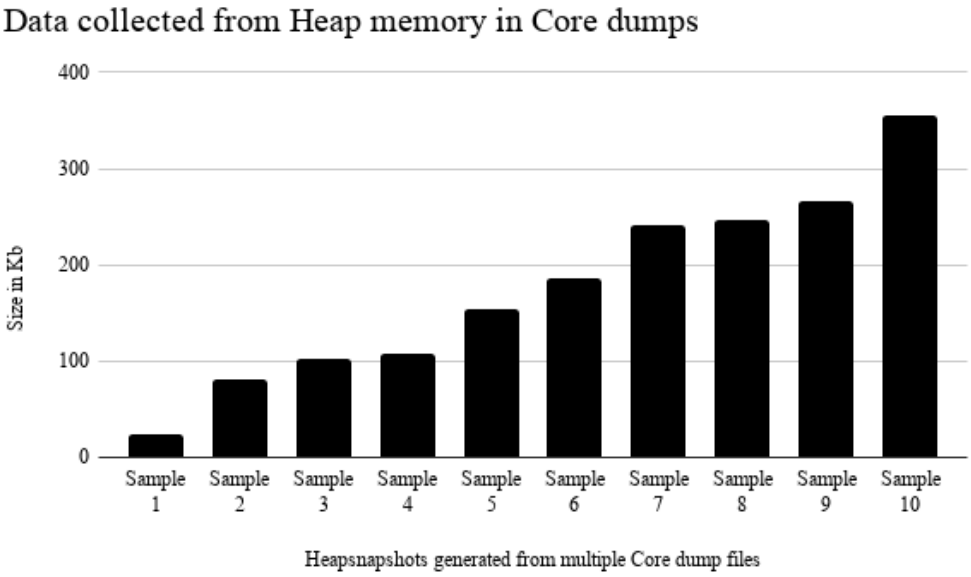

Figure 4.3: Data retrieve from Core Dumps

Chapter 5

Conclusions and Future Work

In this section, we seek to address two research questions that would provide insight into post-mortem debugging in Node.js. Discuss the limitations of the automated solution established in this report, as well as potential research directions.

5.1 Summary

In this section, we discuss the identified research questions(RQ) in this study in more detail. The research questions (RQ) that have been identified in this study, have been formulation as shown below:

- **RQ1** Why is an automated solution for post-mortem debugging in Node.js necessary?
- **RQ2** How can an automated solution for post-mortem debugging in Node.js be implemented?

With respect to **RQ1**, the purpose of this research question was to investigate, why an automated solution for post-mortem debugging in Node.js was necessary. Post-mortem debugging is generally more popular in areas such as operating systems and their native execution environments, but less in dynamic environments and languages

such as JavaScript and Node.js. Although, post-mortem debugging is currently being used in Node.js as a debugging technique it does have complexities. Developers have to manually inspect core dump files using debuggers such as LLDB and LLNode. The size of core dump files is typically very large and contains the full image of the application's address space, include code, stack and heap. Debugging with such an enormous amount of data is often very challenging. Furthermore, debugger interfaces such as LLDB and LLNode can be difficult to use when debugging complex software faults.

Our approach deals with these complexities and creates a minified version of the core dump in the form of heap snapshots. Currently, there are ways to manually generate heap snapshots of a running process, however, taking a heap snapshot is costly. Generally, taking a heap snapshot requires memory twice the size of the heap at the time of the snapshot and the CPU is held until the snapshot is written; if the heap size is larger it takes more time. Additionally, it is not possible to get a heap snapshot on a process that is no longer running. Our approach automates the process of capturing a heap snapshot from the existing bug report in the form of a core dump.

With respect to **RQ2**, the purpose of this research question was to figure out how an automated solution for post-mortem debugging in Node.js could be implemented. The specific implementation details are in Chapter 3 Research Problems and Methodology. It can be observed that the implementation required maximum dependency on V8 specifications. The low-level aspects, such as analyzing the debugging information in core dump file, were all handled by LLDB and the performed post-mortem diagnostics to recover JavaScript values from the core dump were performed using LLNode.

5.2 Limitations and Future Work

The main objective of this research was to provide a solution for automating post-mortem debugging in Node.js. The automated solution was aimed to automatically process core dumps and generate more concrete results in the form of heap snapshots, which would be an improvement over the existing solutions and workings. The objective of the research was only partly accomplished. The solution implemented is generic, and the results obtained are still in the preliminary stage of development.The V8 JavaScript engine is constantly evolving, which makes the implementation even more challenging to keep things up to date with the V8 specifications. The current automated parsing of the core dump is unable to fully construct, the complete working heap snapshot. The study was only able to set up a mechanism to parse the core dump and capture V8 objects and their properties. One of the future research directions could be, solving the issue of constructing the roots of all live objects, which will allow navigating through the heap graph. Furthermore, this could solve the issue of data inaccuracy in the Retained Size column in the Chrome DevTools, shown in Figure 4.2. Other improvements of the developed module could be to make it compatible with other Node.js memory profiling tools, such as NSolid, which provides some benefits over Chrome DevTools in a production environment.

The automation of post-mortem debugging could potentially be fully integrated into the debugging practices in Node.js. The benefits of automation, as well as a direct comparison to other similar solutions, can be explored further.

Bibliography

[1] *Chrome devtools*, https://developers.google.com/web/tools/chrome-devtools, Last accessed 2021-01-22.

[2] *Definition, Post-mortem Debugging*, https://almarklein.org/pm-debugging.html, Last accessed 2021-03-21.

[3] *Design overview — libuv documentation*, http://docs.libuv.org/en/v1.x/design.html, Last accessed 2021-03-22.

[4] *Executable and linkable format*, http://www.skyfree.org/linux/references/, Last accessed 2021-03-22.

[5] *Heap snapshot parser module*, https://github.com/jwalton/node-heapsnapshot-parser, Last Accessed: March 02, 2021.

[6] *Hitang,A 2021*, https://github.com/anil9t5/llnode/tree/create-heapsnapshot, Source Code (Forked from LLNode).

[7] *The lldb debugger. (n.d.).*, https://lldb.llvm.org/, Retrieved January 15, 2021.

[8] *Postmortem support*, https://github.com/nodejs/node/blob/master/doc/guides/node-postmortem-support.md, Last Accessed January 07, 2021.

[9] *V8 heap snapshot schema*, https://gist.github.com/mmarchini/, Last Accessed: March 02, 2021.

[10] *V8 JavaScript engine*, https://v8.dev/, Last accessed 2021-03-22.

[11] *What is lldb?*, https://lldb.llvm.org/, Last Accessed: March 09, 2021.

[12] *What is llnode?*, https://github.com/nodejs/llnode, Last Accessed: March 09, 2021.

[13] *What is n-api?*, https://nodejs.org/api/n-api.html, Last Accessed: March 12, 2021.

[14] Joyee Cheung, *Bringing JavaScript back to life (node+js interactive 2018)*, https://www.slideshare.net/igalia/bringing-javascript-back-to-life-nodejs-interactive-2018, Last accessed 2021-02-27.

[15] Ki-Yong Choi and Jung-Won Lee, *Fault localization by comparing memory updates between unit and integration testing of automotive software in an hardware-in-the-loop environment*, Applied Sciences (2018), 2260, 8.

[16] J. Conrod, *A tour of v8: Garbage collection*, http://www.jayconrod.com/posts/55/a-tour-of-v8-garbage-collection, Retrieved January 14, 2021.

[17] Michael J Eager, *Introduction to the dwarf debugging format*, https://studylib.net/doc/14136149/introduction-to-the-dwarf-debugging-format-michael-j.-eag, Last accessed Apr. 2012.

[18] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano, *Devops*, IEEE Software **33** (2016), no. 3, 94–100.

[19] R. (2009) Hoyecki, *Multiprocessor debugging challenges*, http://signal-processing.mil-embedded.com/articles/multiprocessor-debugging-challenges/, Last accessed 2021-02-17.

[20] J. Huselius, Daniel Sundmark, and Henrik Thane, *Starting conditions for post-mortem debugging using deterministic replay of real-time systems*, 08 2003, pp. 177– 184.

[21] Blasciak A. J. Parets G. S. Kuzara, E. J., 1995, U.S. Patent No. 5,450,586. Washington, DC: U.S. Patent and Trademark Office.

[22] L. E. Lwakatare, T. Karvonen, T. Sauvola, P. Kuvaja, H. H. Olsson, J. Bosch, and M. Oivo, *Towards devops in the embedded systems domain: Why is it so hard?*, 2016 49th Hawaii International Conference on System Sciences (HICSS), 2016, pp. 5437–5446.

[23] Glenford J. Myers, Corey Sandler, and Tom Badgett, *The art of software testing*, 3rd ed., Wiley Publishing, 2011.

[24] Audreyt. (n.d.), *Webworker threads*, https://github.com/audreyt/node-webworker-threads, Retrieved March 14, 2021.

[25] Aleksandar Nedelchev, *Automating linux post-mortem debugging*, http://jultika.oulu.fi/files/nbnfioulu-201806022428.pdf, 2018.

[26] David Pacheco, *Postmortem debugging in dynamic environments*, Commun. ACM (2011), 44–51, 54.

[27] Chris Parnin and Alessandro Orso, *Are automated debugging techniques actually helping programmers?*, Proceedings of the 2011 International Symposium on Software Testing and Analysis (New York, NY, USA), ISSTA '11, Association for Computing Machinery, 2011, p. 199–209.

[28] H. A. Qadeer, *Porting lldb*, https://llvm.org/devmtg/2015-02/slides/abid-lldb.pdf, Retrieved January 07, 2021.

[29] M Subhi Sheikh Quroush and Tolga Ovatman, *Debugging remote services developed on the cloud.*, CLOSER, 2018, pp. 426–431.

[30] Anthony Umukoro and Joseph Okorodudu, *Automated debugging system*, (2019).

[31] W. Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa, *A survey on software fault localization*, IEEE Transactions on Software Engineering (2016), 1–1, 42.

Vita

Candidate's full name: Anil Hitang
University attended (with dates and degrees obtained):
Crimson College of Technology, Affiliated to Pokhara University, Nepal, Bachelor of Computer Applications, 2014
Publications: N/A
Conference Presentations: N/A