POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

# BN-Hermes: a decompilation and analysis plugin for React Native's Hermes bytecode

Tesi di Laurea Magistrale in
Computer Science and Engineering -
Ingegneria Informatica

Author: **Cristian Falvo**

Student ID: 964797
Advisor: Prof. Stefano Zanero
Academic Year: 2022-23

# Acknowledgements

Ringrazio in primis il mio relatore, Professor Stefano Zanero, per la sua disponibilità e per avermi dato la possibilità di svolgere il tirocinio da cui questo progetto di tesi è nato.

Ringrazio tutto il team di Secure Network, per il supporto professionale ed umano ricevuto in questi mesi di tirocinio; voglio in particolare ringraziare Eros, Jacopo e Marcello per avermi proposto questo progetto e per il sostegno tecnico e soprattutto morale che ho ricevuto durante tutto il suo sviluppo.

Voglio ringraziare mia mamma, mio papà e mia sorella, per essermi stati vicini sempre, anche nei momenti difficili di questo percorso universitario e anche se spesso distanti, per avermi sempre mostrato il lato positivo delle cose, e di ricordarmi costantemente il mio valore come persona.

Ringrazio i miei nonni, per l'amore incondizionato che mi hanno sempre dimostrato.

Ringrazio tutti i miei amici per il tempo passato insieme, fisicamente e a distanza, e per il supporto reciproco nelle difficoltà del percorso universitario. Senza di loro molti momenti, soprattutto durante il periodo di pandemia, sarebbero stati estremamente difficili.

# Abstract

Mobile applications are one of the most widespread classes of software currently in use, and their development is one of the main focuses of the software development industry. Due to this importance, the analysis of mobile applications to assess their security - including reverse engineering and static analysis - is a key activity in the computer security field, with many techniques, tools and services being developed for this purpose. Such tools often fall short in analyzing applications written using modern frameworks.
A framework that poses specific challenges to analysis tools is React Native, a popular cross-platform application development framework based on JavaScript. Indeed, the default engine for React Native, Hermes, uses a custom bytecode that must be properly understood and processed by analysis tools. The few currently available tools to analyze Hermes bytecode are lacking, as they mainly perform disassembly of the bytecode without reconstructing the high-level semantics of the code.
In this thesis, we design and develop a solution to analyze Hermes bytecode of React Native applications. Our solution is built as a plugin for a popular reverse engineering platform, Binary Ninja, leveraging the provided API to represent the Hermes bytecode and architecture and ultimately allowing to transform the bytecode into a human-understandable format to aid analysis. We validate the capability of our solution on a set of representative React Native applications showing that it is able to analyze them and reconstruct the code semantics.

**Keywords:** Reverse engineering, Hermes engine, React Native, disassembly, binary analysis

# Abstract in lingua italiana

Le applicazioni *mobile* sono una delle forme di software più comuni a livello mondiale, ed Il loro sviluppo è una delle attività principali dell'industria di sviluppo software. A causa della loro importanza, l'analisi di applicazioni *mobile* al fine di valutare la loro sicurezza - che include le tecniche di *reverse engineering* e analisi statica - è un'attività chiave nel settore della sicurezza informatica, con una varietà di tecniche, strumenti e servizi che sono sviluppati a questo fine. Tali strumenti spesso si rivelano insufficienti nell'analizzare applicazioni sviluppate con framework moderni.
Un framework che rappresenta una sfida per gli attuali strumenti di analisi è React Native, un diffuso framework per lo sviluppo di applicazioni multipiattaforma basato su JavaScript. In particolare, l'interprete di default per React Native, Hermes, usa un *bytecode* specifico che deve essere compreso e processato dagli strumenti di analisi. I pochi strumenti attualmente disponibili sono insufficienti allo scopo, in quanto effettuano quasi esclusivamente *disassembly* del *bytecode* senza ricostruire i costrutti semantici di più alto livello del codice sorgente.
In questa tesi progettiamo e sviluppiamo una soluzione per analizzare il *bytecode* Hermes di applicazioni sviluppate in React Native. La nostra soluzione è costituita da un *plugin* per una popolare piattaforma di *reverse engineering*, Binary Ninja, utilizzando le *API* che essa fornisce per rappresentare il *bytecode* e l'architettura di Hermes, con il risultato ultimo di rappresentare il *bytecode* in una forma più comprensibile per supportare le attività di analisi. Abbiamo poi valutato le capacità della nostra soluzione con una selezione rappresentativa di applicazioni in React Native, mostrando la capacità di analizzarle e di ricostruirne la semantica a livello di codice.

**Parole chiave:** Reverse engineering, interprete Hermes, React Native, disassembly, analisi di binari

# Contents

# 1 | Introduction

Mobile applications are arguably the main way people from all over the world access digital services. Their extremely fast diffusion over the last fifteen years has sparked the interest in their development on all levels, from personal to enterprise, and many technologies were created to support developers in their work. Nowadays there exist many frameworks that allow development of applications; some of them, usually referred to as *cross-platform*, allow to write code that can then be executed on multiple kinds of devices (mainly Android and iOS smartphones) with minimal or no adjustments needed. Most of these projects are open-source and community developed, usually with a relevant support of large tech companies who leverage these technologies in their products.

One of such development frameworks is React Native [9], an open-source framework that transposes React, a popular JavaScript framework used to create the front-end of web applications, to mobile development. To do so, it leverages a combination of the original library and the "native" functionalities of the target platform. It is currently one of the most popular frameworks for app development on Android and iOS, being natively cross-platform, and has been expanded to support other platforms, including systems based on Windows and MacOS and specialized environments such as AndroidTV [22]. Thanks to the popularity of both JavaScript as a programming language [21] and React as a framework for web application development [32], React Native reached a 32% share for usage among cross-platform mobile application frameworks, according to JetBrains statistics [26].

This increase of popularity of usage and production of mobile applications also meant an elevated interest in their security, similarly to what happened with the diffusion of the World Wide Web and web-based services. This meant that security analysis of mobile applications became, and currently is, one of the main activities in the field of security assessment. Although some widely used tools and techniques used in "regular" application analysis (such as traffic analysis) can be applied to mobile apps as well, there was and still is a need for tools that tackle some specific analysis. In the case of React Native, some other challenges arise from its cross-platform nature and its implementation.

First of all, when testing any mobile application, *not* having access to the source code is

the norm; this is commonplace in all the software industry as source code is commonly covered by copyright, and is part of the original work of the developers. This is a first challenge in application analysis since the source code would be a prime source of information for analysis, and it obviously applies to mobile applications as well.
Secondly, there is a substantial difference between and native (or platform-specific) and cross-platform applications, from a development and therefore also an analysis standpoint. Native applications leverage heavily the target platform's specific properties, mainly the corresponding SDK, are written in different languages based on the platform, and the resulting packages and binaries also differ. This means that developing the "same" native application for different platforms implies doing it separately for each supported platform. Cross-platform development instead is based on developing the application once and independently from the target platform, and delegating the execution on the various target platforms to the underlying framework; this also allows freedom in the choice of the programming language, as seen for React Native and its usage of JavaScript, a language originally intended for web applications. These differences highlight the fact that different analysis techniques could be needed, especially when analyzing the applications' internal components.
For React Native specifically, mobile application analysis had similarities with web application analysis, since the JavaScript engine (the component that executes JavaScript code) had always been JavaScriptCore, the engine originally developed for iOS' browser, Safari, and now used in projects based on the popular WebKit browser engine. This meant that the application bundle contained the JavaScript source code, potentially minified or webpacked, and it could be analyzed similarly to web-based JavaScript applications.
This changed when a new JavaScript engine, Hermes, was introduced as an optional choice, and recently became the default option for new projects. Instead of the JavaScript source code, applications bundled with Hermes are compiled ahead of time and the bundle contains raw instructions for the Hermes engine. This means that the previous analysis techniques are no longer usable in this case; therefore a need for tools that tackle Hermes' new bundle format emerged.
The first tool that was immediately accessible was `hbcdump` [5], a disassembler provided as a part of the Hermes project itself; being a disassembler, what it provides is a conversion from raw bytes to the corresponding opcodes' names, but missing structure. The first community-developed tool, `hbctool`[12], came out in 2021; it consists of a wrapper for `hbcdump`, providing a bit more readability to the same result in the form of disassembled instructions. Only in January 2023 a security researcher published `hermes-dec`, a tool that produces code with a syntax closer to that of JavaScript, but still lacking some important aspects such as control flow.

The focus of this thesis is the development of an analysis tool for Hermes bytecode, that aims to solve the shortcomings of the previous works, by providing a higher level of abstraction than disassembly and obtaining code with full control flow structures such as conditionals and loops. This was obtained by leveraging an existing reverse engineering platform, Binary Ninja [3]. This platform provides an API that gives access to many of its internal components, allowing in this case to develop a representation of the Hermes architecture and of the binary file itself, and leveraging the core analysis functionalities to lift human-readable code from the disassembled instructions. The final product is therefore a plugin for Binary Ninja, which provides the functionalities that perform disassembly of the bytecode and subsequent translation to Binary Ninja's internal intermediate representation language; this last step is often referred to as "lifting" since it brings low-level instructions to a higher level of abstraction. What this plugin is set to provide is an analysis of application bundles compiled for Hermes, structuring the internal data structures and providing Hermes' custom bytecode to Binary Ninja for decompilation.

# 2 | Background

## 2.1. React Native

React Native [9] is an open-source JavaScript application framework developed by Meta and based on React, which is one of the most popular JavaScript libraries for development of the front-end of web applications. It provides an ecosystem to develop mobile applications on the front-end side, reusing many constructs of its web-focused counterpart. The main features making it among the most used frameworks for this purpose are it being platform independent, in the sense that applications developed in React Native can be then built for multiple platforms, such as Android, iOS and AndroidTV, with minimal or no modifications required, and the similarity with the already popular React framework.

### 2.1.1. Core features

React Native inherits some key features from React. In particular the two main ones are JSX and components.
JSX [28] is an extension of JavaScript syntax (the acronym stands for JavaScript Syntax eXtension), based on markup languages like HTML and XML, and is used to create so-called *React elements*. These elements are similar in syntax and structure to HTML tags, but the key aspect is that they are valid JavaScript expressions and can contain JavaScript code to access contextual variables. In fact, JSX is a syntactic help to write structures intuitively with respect to how they are then rendered in the application; implementation wise, they are actually calls to dedicated React directives, the main one being `React.createElement`. The conversion from JSX to function calls is delegated to a dedicated code transformation toolchain, Babel [2]. The specific of the conversion from JSX to JavaScript is described in Babel's documentation for its React extension [7].
Components can be considered the building blocks of React projects; they are JavaScript functions that take a set of properties and return a React element, usually containing some of the properties. The key aspect is that components can be used within one another, as components can be used as React element types. For example, if a `Person` component

returns a React element containing some text and the `name` property, we can then create a React element with type `Person` and `name="Andrea"`; this will make use of the `Person` component, and our React element will contain the text with the `name` property set to `Andrea`.

## 2.2. The Hermes engine

Hermes is a JavaScript engine optimized for React Native. Its main feature is the fact that it performs Ahead-Of-Time (AOT) compilation, producing a custom bytecode which is bundled in the application and then executed by a dedicated virtual machine. This differs from other JavaScript engines, which usually either act as interpreters or perform Just-In-Time (JIT) compilation, producing instructions native to the platform they are being executed on, which the underlying system then executes immediately. This solution has a relevant impact on application performance, since no compilation is required at runtime, even though some performance is lost by having to run the precompiled bytecode in a virtual machine instead of the native architecture. The usage of AOT compilation has another effect: the content of the application bundle is now a binary file, instead of JavaScript code, which makes analysis as it was performed before impossible.
The following provides an overview of the properties of the Hermes architecture that are relevant for the project's implementation.

### 2.2.1. Versions

The Hermes engine was first introduced as an optional engine for React Native in release 0.60.1 [27]; this meant that developers could decide to use it as the JavaScript engine for their application instead of JavaScriptCore. The development of Hermes and React Native continued on parallel paths, with each having its own versions, until React Native version 0.70, when Hermes became its default engine [29].

### 2.2.2. Architecture

Hermes is a virtual machine based on a register stack; this means that the number of registers is infinite, and memory operations correspond to "allocating" new registers on top of the register stack. This makes Hermes more similar to a pure stack machine, where "registers" are just fixed-size memory areas managed on the stack, and just conveniently named with an ever-increasing index. Moreover, Hermes' register stack is unique to each function, meaning that the values in "registers" set by a given function are not seen by the

function's caller or callees. These "registers" have a fixed size (64 bits) and they contain either primitive types (integer, float...) or pointers to strings and objects managed by the garbage collector. This is one clear difference between Hermes and classic computing architectures, based on a fixed and shared set of registers, and a stack in memory.

### 2.2.3. Garbage collector

The garbage collector is a component that is present in all JavaScript engines. JavaScript is an object-oriented language, with inheritance based on a prototype system, and objects are data structures based on key-value pairs that can be edited, added and removed at runtime; moreover, other important inner structures of JavaScript, such as lexical environments, are dynamically handled as well. The garbage collector, therefore, performs all the memory management required to handle these dynamic structures, by allocating, moving and deleting objects and data structures in an efficient manner, and gives access to the "live" data structures to the core architecture. The way garbage collection is performed is not considered in this work, since static analysis (as decompilation is) only targets the bytecode without executing it, while the garbage collector only operates at runtime.

### 2.2.4. Application Binary Interface

The instruction structure for the Hermes bytecode [17] is rather simple, but it is counterbalanced by a large number of different opcodes, many of which are overloaded versions of the same functionality that have been separated for performance reasons; for example, there exist two unconditional jump instructions, `Jump` and `JmpLong`, that differ in the size of the offset parameter, and the "long" version is only used when the offset overflows the size expected by the "short" version. All opcodes have a fixed number of parameters, each with a fixed type; types are predefined and they map directly to C++ types; they are fundamentally either register indexes, signed and unsigned integers, and doubles; addresses are treated as 32-bit integers, and are always offsets relative to the instruction pointer. Accesses to tables, such as the string table or the object tables, are done by index using an unsigned integer as parameter type.
Since Hermes is a JavaScript engine, there are many peculiar opcodes, that perform operations that would be very high level, or not implemented at all, for regular architectures; for example, there are dedicated options for get/set/delete operations on JavaScript objects, as well as an opcode that implements the `switch` construct.

### 2.2.5. Calling convention

Another relevant aspect of the ABI is the calling convention: since registers and stack coincide, argument passing in functions behaves differently for Hermes compared to classic computing architectures and calling conventions. Firstly, as the register stack is separate for each function, every function sees only the registers used by itself. Secondly, there are no fixed registers that are supposed to contain function arguments in call instructions; instead, there are two ways the registers are passed to the call instructions: some calling opcodes expect the registers containing function arguments as parameters, while others will have a parameter representing the number of registers to be passed as function arguments, and they are the `n` top registers in the register stack. For optimization, the first type of call works with function having up to four arguments, while the second is used for more than those. It is important to observe that, in JavaScript, the number of arguments passed to a function and the number of arguments that the called function expects are not necessarily static nor have to be equal, as per the ECMAScript standard; Hermes, as other JavaScript engines, solves these differences at runtime.

### 2.2.6. File format

The Hermes binary file has a rather simple high level structure, while the individual sections contain data in different formats, chosen for both reduction of file size and ease of compilation. From a high level point of view, the structure is the following:

- File header: contains information to identify the file, its version, and global metadata regarding the file, its sections and their size.
- Function header table: a list of all function headers, which consist of metadata such as function offset and length in bytes within the file, name and number of parameters.
- String table and storage: a list of metadata entries, each representing a string used in the binary, and the raw string bytes.
- Object, BigInt and regex tables: a list of metadata entries and values for, respectively, objects initialized from static values, static BigInt values, and regular expression objects with their dedicated bytecode. For BigInt specifically, they were added in bytecode version 90 to support the recently added BigInt construct in JavaScript [14].
- CommonJS and source tables: tables of entries that map, respectively, CommonJS

modules to their imported functions, and functions to their source code (which requires dedicated directives to be populated); in the tests it has never appeared to have any content, most probably because modules are resolved and organized beforehand in a single bundle; this is performed by React Native's bundling module, Metro [8]. Moreover, having function code in a released app is of no use and can even be an unnecessary information disclosure.

- Function bytecode: the raw bytecode of the functions, along with their exception handling table and other metadata.
- Debug info: Metadata tables and raw bytes of the debug information.

The internal structure of each field will not be described in detail here, but there are a few solutions worth mentioning.
Since the most populated metadata tables in the binary are those referring to functions and strings, they both have been optimized in a similar way: for most instances, a "small" version of the headers is used, to use as little space as possible; when, rarely, the instance would overflow the size of the existing fields, a larger "overflowed" entry is used.
The other main optimization to reduce binary size is the packing of both strings and object literals (static values used to initialize non-empty objects): the final blob of data is obtained by overlapping, when possible, the single instances (strings/object contents) instead of simply concatenating them; each instance is then represented by a metadata entry composed of its starting offset in the blob and its length.

## 2.3. Binary Ninja

Binary Ninja [3] is a reverse engineering platform developed by Vector35, which provides disassembly, decompilation and other binary analysis tools. The choice of this tool at the start of the development of this project was due to the availability of a well tested and rich API and the fact that the result is interactive and actively analyzable through an existing interface. These factors made this the chosen platform to work on, rather than using the few other alternatives (Ghidra being the main one) or developing a standalone decompiler from scratch, which could have allowed for a more architecture-specific implementation, but would have lost much in terms of user experience in the use of the final tool.
More details on how Binary Ninja was used are provided in Chapter 3.

## 2.4. Current solutions

Since the choice of making Hermes the default engine for React Native is rather recent, the amount and level of advancement of existing tools is limited.

### 2.4.1. hbcdump

`Hbcdump` is the disassembly utility that is part of the Hermes repository, along with other utilities such as the compiler, a debugger, the virtual machine executable itself, and some testing tools. Since it is a disassembly tool, it only allows the conversion of the raw bytes to a human-readable form of the opcodes, along with some the static data available in the binary. Its main drawback, clearly, is the fact that it is just a disassembly tool: its output are the raw instructions, in terms of registers, constant values and references to other parts of code in the form of data offsets or, more often, indexes of elements in tables.
What this tool lacks is most of the features of human-readable source code: variables usage, a representation of control flow, and readable function calls. Moreover, applications are always compiled with all the imported and required code they use (just like a statically compiled executable), even the simplest functions can contain several thousand functions, making manual analysis with such a basic tool as this even more cumbersome. Lastly, every version of the Hermes bytecode releases with its own version of hbcdump, which requires retrieving the correct version of the disassembler based on the bytecode version.

### 2.4.2. hbctool

`Hbctool` is the first community-developed tool that attempts to improve on `hbcdump`. It was presented in a blog post [12] by security researcher `bongtrop` in 2021, where the tool is showcased to disassemble and patch a dummy application. It fundamentally consists of both a disassembler and an assembler, which also provides extraction of static objects and string data; the availability of an assembler is relevant for patching of the binary, as it is shown in the small experiment described in the post above. The disassembler does provide an added layer of readability to the result by adding the typing to every instruction's parameters.
It still however manifests the same shortcomings as `hbcdump`, that is, it is still not a decompiler, and therefore lacks the same higher level analysis aspects. It does, however, support various bytecode versions, thanks to the contributions of other developers to the tool's Github project [20].

### 2.4.3. hermes-dec

On January 9th, 2023, researcher Marin Moulinier published a post on P1 Security's blog, announcing the release of `hermes-dec`, a disassembler and decompiler for the Hermes bytecode [31]. In the current form, it consists of a set of tools: a parser of the binary file, a disassembler and a decompiler that produces a pseudocode with a JavaScript-like syntax. This clearly sets it above the previous two described tools as it is the only available tool that performs some form of decompilation, and also supports all currently available versions of the bytecode, down to the early pre-release versions. It does, however, have its shortcomings, mainly the usage of registers instead of variables, and the fact that control flow and conditional instructions are not fully reconstructed; moreover, it produces a single output file, which can become large and hard to manage because of the large amount of static code that is compiled with every React Native application.

## 2.5. Goals and challenges

The objective of this project is to provide a decompilation tool for the Hermes bytecode that expands on the existing approaches by obtaining language features such as control flow and function calls; moreover, use the functionalities that the Binary Ninja platform provides to bundle the analysis results in an interface that resembles the ones available for well-studied architectures, such as x86 and ARM. Although Binary Ninja provides a rich API to implement plugins, the platform's main targets for analysis are usually binary files compiled for "regular" computing architectures, which are implemented physically and normally share some common properties, like the existence of a limited set of shared registers and a stack allocated in memory. Hermes, on the other hand, is an engine that is based on a virtual machine, which executes the compiled bytecode, and happens to not match these expected properties fully.

The main challenge in the development of this project is therefore gaining an understanding of Hermes' bytecode and finding a suitable representation of the architecture that can be mapped effectively onto Binary Ninja's API. The focus of these challenges are the aspects of Hermes' architecture that deviate the most from classical architectures: the register stack, the calling convention and some opcodes in the bytecode that perform high level operations.

# 3 | Methodology

## 3.1. Overview

The objective of our project is developing an analysis and decompilation tool for the Hermes bytecode; we chose to use Binary Ninja as support for our work, providing the core analysis functionalities we need, and the form of the resulting work is therefore a plugin for Binary Ninja that provides a representation of the bytecode files and converts bytecode instructions to Binary Ninja's representation language. Our plugin performs three tasks, which are executed in the order they are presented, and each one provides the following with the necessary data:

- Parsing of the bytecode file, which extracts the bytecode instructions and the static data from it; in particular, the data structures are parsed into equivalent structured objects in our plugin, for ease of access, and they are also remapped to Binary Ninja's representation of the binary file, to allow their visualization in the GUI.
- Disassembly of the bytecode instructions, which transforms the raw instruction bytes into a structured format that separates and organizes the opcode and the parameters; these are also used in a component inside Binary Ninja to provide a textual representation of the bytecode instructions and a partial reconstruction of the control flow.
- "Lifting", which in this context means "translating" and "abstracting", the disassembled bytecode to the lowest level of Binary Ninja's internal intermediate representation languages. This transforms the disassembled bytecode into an assembly-like language, which Binary Ninja then analyzes and translates to other higher level representation languages.

The starting point for the project was a two-part blog post and guide by Binary Ninja [11], which goes over the basics to implement a new `Architecture` component and its correspondent `BinaryView`; this project obviously went above what is explained there, but it represented a foundation for it. Lastly, it should be mentioned that Binary Ninja's

API is available for the Python, C++ and Rust programming languages, and plugins are therefore developed with these same languages; the Python and C++ ones are the most complete and better documented ones, and equivalent in terms of available functionality. We decided to develop our plugin in Python, simply because it was the most familiar of the two languages. Since the source code of the whole project is rather large, and a significant portion of it is structure definition and static data, most of it is not featured in this thesis; snippets of code have been referenced where deemed useful.

## 3.2. Implementation

### 3.2.1. Structure of the plugin

The plugin we developed can be divided in four main components; these are based on the guide to the development of architecture plugins in Binary Ninja's blog [11], with a collection of other classes and helper functions that provide other necessary functionalities.
The `BinaryView` class is one of the two core components required in Binary Ninja architecture plugins. It fundamentally consists of a wrapper class for the binary file to be analyzed. It provides read and, to a lesser extent, write accesses, with an interface similar to a file descriptor. It is also the component in which several structures available in the file itself are defined, like file segments and sections, static data structures and types, strings and symbols. All these structures, once mapped, are then rendered in Binary Ninja's GUI.
The `Architecture` class is the other core component of architecture plugins. It consists of a model of the target computing architecture, and requires the definition of some of its key properties, like the list of registers and the default address size. Most importantly, here is where the core functionalities of textual representation of the disassembled instructions, control flow reconstruction and lifting to the intermediate representation language need to be implemented.
A set of classes and functions constitutes the parsing component of the plugin; this was developed from scratch and was based off of the documentation and source code of the Hermes project.
Finally, a disassembler is a prerequisite for the core functionalities of the `Architecture` component, and especially for the lifting aspect; our implementation is described in Subsection 3.2.3.

### 3.2.2. File parsing

The parsing of the whole file meant understanding the data structures in it; the main source of information were source code files [16] [18]. Mapping the data structures defined here to Python classes was rather straightforward. The only exceptions were some bitmask-based flags and non byte-aligned compact data structures, and the arguably inconsistent insertion of padding (as data in the Hermes binary is supposed to be 4-byte padded, but most sections pad internally and some others don't). At this point, it was convenient to organize the representation of the file in the `BinaryView` class. With the parsing of the file completed, we could construct the file's sections and segments, in a way that was based off of the structure mentioned in Section 2.2.6, and map static data structures, symbols and function references on the file's representation in the plugin. In addition to the existing sections of the file, two more were added by writing at the end of the binary file, for specific purposes:

- an artificial string section, where all string entries were rewritten in null-terminated form, instead of the packed and overlapping format available in the file; this improved the readability of code at the various representation levels. This choice of implementation was made because, in Binary Ninja's intermediate languages (which decompile to a language similar to C), strings are treated as character pointers, and therefore length is inferred by stopping at the first null termination character. In their overlapping and packed form that the Hermes file format uses, the strings are not terminated, as they are extracted using both their offset in the main string and their length from their metadata; this difference meant that pointing to just the beginning of the string showed also characters after the expected end of the string, reducing readability.
  We also decided to not generate this section and the strings within it when the overall string data is excessively large, to limit memory usage, as this section more than doubles the total length of strings within the plugin. When this happens, the plugin reverts to using the string offsets from the packed string format, which as mentioned reduces readability.

- an external symbol section, where a set of functions and constants were added as `external` symbols to artificially map some high level instruction opcodes and functions; this is explained in more detail in Section 3.2.11 and Section 3.2.12.

The result of this first phase of analysis is a `File` object, which contains all the data structures available in the binary file, properly parsed into appropriate Python objects; at this point the main instance of the `BinaryView` class is also initialized with the ap-

propriate references and data structures. The parsing part that produces the `File` object can actually be used separately from the rest of the plugin. An example of one of the data structures, representing the `global` function of the binary later used for testing in Section 4.1, is shown in Listing 3.1.

### 3.2.3. Disassembly

Disassembly is the first step to obtain some form of human readable result from the raw bytecode. The objective is to obtain the same kind of output as the other available disassemblers we mentioned in Section 2.4, and mapping it to the provided API to leverage the platform's functionalities, mainly the interaction with address text tokens and some form of control flow graph (which is not necessarily complete at this level, because of indirect jumps and calls).

The `Architecture` component of the plugin expects the implementation of two fundamental functions: `get_instruction_info` and `get_instruction_text`. The first takes an offset in the bytecode and bytecode data as input, and expects the function to create branches, which are the main component of the control flow at the assembly level; of course, only instructions that implement jumps in the code (conditionals, function calls, return) will have branches associated with them. The second function takes the same input but is supposed to return a list of strings, referred to as text tokens, which are the textual, human readable representation of a disassembled instruction.

We decided to implement a disassembler from scratch, instead of leveraging the existing ones; this choice was due to the fact that the official disassembler, `hbcdump`, only works for the bytecode version it was built for, and therefore supporting multiple version meant packaging multiple version of the executable and choosing the right one to execute at runtime; the other existing tool, `hbctool` [12], at the time this choice was made did not support versions of the bytecode newer than version 74, while we were working on bytecode version 90. Moreover, implementing the disassembler from scratch meant that we could create some custom classes and structures to map functions, which we could then access conveniently in other parts of the plugin.

The core source of information for the disassembly is a C++ definition file [17] that lists all opcodes, the number and type of their parameters, along with relevant documentation regarding how each instruction maps to JavaScript instructions. The way the bytecode is unserialized into instructions was obtained by analyzing the implementation of the `hbcdump` tool described previously, and these are the relevant findings, which are quite standard for a bytecode:

**Listing 3.1:** The `SmallFunctionHeader` object for the `global` function of a simple Hermes bytecode file.

```
Function(
    small_header=SmallFunctionHeader(
        offset=348,
        paramCount=1,
        bytecodeSizeInBytes=285,
        functionNameIndex=0,
        infoOffset=652,
        frameSize=17,
        environmentSize=0,
        hiReadCacheIndex=9,
        hiWriteCacheIndex=5,
        flags=FunctionHeaderFlag(
            flags=b'\x1a',
            address=143,
            prohibitInvoke=<Prohibits.ProhibitNone: 2>,
            strictMode=False,
            hasExceptionHandler=True,
            hasDebugInfo=True,
            overflowed=False
        ),
        address=128,
        size=16,
        type_name='small_function_header'
    ),
    overflowed_header=None,
    code=b'3\x04\x00[...]\x00Z\x00',
    name='global',
    jump_table_addr=None,
    jump_table=None,
    exception_handler_table=ExceptionHandlerTable(
        header=1,
        exception_handlers=[
            ExceptionHandlerEntry(
                start=231,
                end=258,
                target=260,
                address=656
            )
        ],
        address=652
    )
)
```

- each instruction begins with one byte representing the index of the opcode in the aforementioned list;
- each opcode has a fixed list of parameters, each with its own type;
- parameter types are mapped to C++ types in the same definition files;

We parsed this list of opcode definitions to a Python list, since instructions are implicitly associated to their index in the listing; this is observed in the serialized bytecode because the first byte of each instruction is the numerical index of the opcode in the list as a 1-byte integer. Parameters are then listed in the order and size as described in the definition file, so all was needed was to model the few possible parameter types, and make the disassembler match the parameter list from the opcode with the given bytes.

### 3.2.4. Architecture details

Before tackling the lifting aspect, it is necessary to describe some details regarding Hermes' architecture, and how they have been mapped to the `Architecture` component of the plugin.

#### Registers

As we described in Section 2.2.2, Hermes' register stack effectively behaves as a stack, but the Binary Ninja API expects the architecture definition to match the paradigm of a set of shared registers plus a stack in memory. Implementing the whole architecture as a pure stack machine, having only the basic stack pointer registers and treating registers as memory areas would have been possible, but we chose against this solution as it would have probably complicated lifting and would not have used the whole set of tools available for architecture design in the API. We therefore opted for a large set of registers, 500 initially but then increased to 1,000 since some binaries we tested used more than 500 registers in a single function. These registers are shared, since it's the only way Binary Ninja treats registers at the architecture level; we therefore needed to find a solution to emulate the fact that these registers are independent; this was tackled in the implementation of the calling convention, as explained later.
We then made the decision to use the stack modeled in Binary Ninja's `Architecture` only to handle function calls; this required to add two more registers to handle stack allocation with function calls, `sp` and `bp`, stack and base pointer respectively. We then needed to add one last helper register, `rr`, standing for return register, because of the way function returns work for Hermes: the `Ret` instruction makes the function return the

content of a register, which is then set to the awaiting register in the `call` instruction in the caller; since the register stacks are separate, we needed to store the return value, perform the operations necessary to keep the registers in the state they are expected to be after exiting the function, and then set the awaiting register to the stored value. The operations required to handle the registers in the way the architecture expects it are described in Section 3.2.10.

### Environments

Environment records are an internal component of JavaScript that Hermes manages through dedicated opcodes, so it is worth giving a brief definition and description.
As defined in the ECMAScript specification [24], environment records are internal structures implemented by engines to manage identifiers in code blocks. In JavaScript, environments refer to and manage a block of code with the same scope; the documentation mentions, as examples for code blocks, function definitions, the catch part of a try-catch construct, and in general code that is enclosed in curly brackets. Environments can be nested, and the implementation for such nesting is simple: in addition to managing identifiers, every environment keeps a reference to its "outer" environment, which is simply the environment of the code block containing it (for example, a catch block within a function definition). For the outermost environment, this reference is simply `null`.
The way Hermes implements environments is a simple dynamic list-like data structure; every time a new identifier is introduced in a code block a new entry in the environment is added. Hermes uses a series of opcodes to access and manage environments, and environment references are important for function calls.
Our idea to implement environments in our plugin was to use the representation of the stack in Binary Ninja as the equivalent of environments, and making accesses to environment-defined variables correspond to memory accesses on the stack. This makes sense since scope in languages like C is handled by managing the stack. Unfortunately we had to abandon this idea, for two reasons: first of all, the JavaScript subset that Hermes supports only implements function-level scoping; this means that variables defined inside a given code block are available outside the same block, but within the same function; moreover, in principle, entries can be added to environments different from the innermost one; for our implementation, this would mean inserting data in stack frames different from the one in use, which is not possible on a regular stack. The way we ended up managing instructions that access environments is described in Section 3.2.11, but the main point is that there was some inevitable loss of information.

### 3.2.5. Lifting

As lifting is the main challenge of the whole project, it is useful to analyze the interface that the Binary Ninja API provides, how this differs from the properties of the subset of JavaScript that React Native represents and that Hermes is expected to execute, and evaluate possible solutions where the architecture's properties do not have an equivalent in the API.

The fundamental concept regarding lifting is the series of representation languages [1] that Binary Ninja uses to perform its analysis and abstract code to different levels, from an equivalent of assembly to pseudocode with a C-like syntax. The word "lifting" is used for this very reason: we are taking some low-level, hard to understand bytecode, and bringing it to a "higher" level of abstraction, represented by a higher level language, and more human-readable. The most relevant of these representation languages is the one that architecture plugins are expected to produce: `LLIL`, which stands for Low Level Intermediate Language, and is the lowest of the languages Binary Ninja uses to perform its analyses, and it resembles an assembly-style language. Lifting the disassembled code to this language mean giving semantics to the disassembly: it means, for example, that a set of raw bytes is disassembled to a `call` instruction at the assembly level (which is just a text-based, 1:1 matching representation of the raw bytes), but to allow Binary Ninja to understand that the instruction `call`s another function it must be told what that instruction does in a way it understands. The way semantics is provided is emitting instructions in LLIL starting from the disassembly; Binary Ninja understands the semantics of LLIL, and can perform its subsequent analyses, like creation and management of variables and creation of complex control flow instructions like loops. These analyses emit code in the higher level internal languages of Binary Ninja, `MLIL` and `HLIL` (which are the *Medium* and *High* level counterparts to LLIL); the first of the two converts register usage to variables, and analyzes types; the second instead performs code and variable optimizations, and creates the aforementioned higher level control flow instructions. Our job was therefore to map each disassembly instruction to one or more LLIL instructions, to allow Binary Ninja to provide the higher levels of analysis.

LLIL's instruction structure, as Binary Ninja's other intermediate representation languages, is tree-based; the basic component of each instruction is an `expression`; each instruction is itself an expression, and each expression can have zero or more subexpressions, which are expressions themselves, based on its type; a graphical representation of a simple Hermes instruction, `Add r3, r1, r0` (which adds the content of register `r1` and register `r0` and sets `r3` to the result) lifted to LLIL as `r3 = r1 + r0`, is shown in Figure 3.1. The architecture class requires a function for lifting, `get_instruction_low_level_il`, and

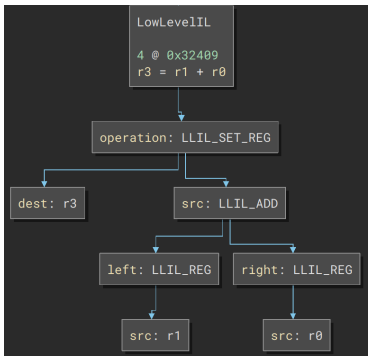


Figure 3.1: Graphical expression tree for the Hermes instruction `Add r3, r1, r0`.

provides a `LowLevelILFunction` object, which is a container to which the emitted LLIL instructions are appended that also has some additional properties that can be useful in lifting.
Since Hermes' instruction set is quite large (there exist 204 different opcodes in bytecode version 90; other architectures have fewer opcodes but more flexible parameter handling), we developed a strategy to split them in an effective and reasonable way. The main objective was to group the opcodes that result homogeneous in their lifting; for example, all arithmetic instructions take two register operands as input, and set the return register to the result of the operation. Moreover, as previously stated, many opcodes are overloads of one another for efficiently handling the smallest parameter size possible; for example, many opcodes have a counterpart with the `Long` or `LongIndex` suffixes that supports table indexes up to 32 bits instead of the regular 8, and other similar solutions for different opcodes exist. The implementation choices are described in the following sections.

### 3.2.6. Unary and binary operations

The first and simplest set of opcodes to lift contains those that perform simple logical, arithmetic and bit-wise operations, operating on registers and setting the result of the operation to a register. These can be separated in two subgroups:

- "unary" operations, that operate on the content of one register and return to another register; for the Hermes architecture, these consist of `Mov` and negation (logic, bit-wise and algebraic) opcodes. These posed no challenge, as they can be mapped directly to LLIL instructions; the only notable detail is that LLIL does not distinguish between bit-wise and logic negation, while JavaScript distinguishes these two for non-boolean and boolean values.
- "binary" operations, that operate on two registers and return to another register;

these are the equality and inequality opcodes, and the algebraic and bit-wise operations (like bit-wise `xor`). It should be noted that JavaScript has two forms of equality, "lax" and "strict", which use different operators in the syntax, for example, for equality, `==` and `===` respectively. The difference is that the first will attempt to convert variables to the same type before comparison if necessary, while the other will not; this difference is due to the weak typing of JavaScript and does not have an equivalent set of comparison instructions in Binary Ninja's LLIL, therefore the two are lifted to the same LLIL comparison instruction. It should also be noted that equality and inequality instructions are treated this way since they return the boolean value of the comparison expression to the return register of the instruction, as the Hermes architecture does not use flags for conditional instructions.

Similarly, Hermes has dedicated opcodes for the algebraic operations when both operands are numbers (so both a generic `Add` and a numbers-only `AddN` opcodes exist), since the algebraic operations can also be applied to non-numeric types in JavaScript. The simplest example of this is string concatenation, where an instruction like `var c = "a" + "b"` is compiled to Hermes' `Add` instruction, since it's an "addition" in JavaScript terms, and the expected value of the `c` variable is indeed `ab`.

Since the Binary Ninja LLIL is too low-level to represent string concatenation and other similar behaviors effectively, which would require to infer the type of the registers' contents (that are dynamic and in general only available at runtime), these instructions are lifted to their most correspondent instruction disregarding this and other type-related behaviors (for example, the division between two integers results in a floating point number), as the result of analysis keeps the semantics of the instructions in the sense given by JavaScript.

### 3.2.7. Loads

These are instructions that load constants to registers; since JavaScript separates between integers and booleans as types, both `LoadConstFalse` and `LoadConstZero` exist and have different results; these are not available in LLIL, as its C-like typing treats zero integers as false and non-zero integers as true, and loads of the two boolean values are treated as loading integer zero for false and integer one for true. For the loading of constant strings, we load the address of the null-terminated string when available, or the one that points to the packed string data format, as discussed in Section 3.2.2.

### 3.2.8. Jumps

These are the main instructions that change the flow; Hermes is very verbose in this and there are many different conditional jump instructions, some of which might seem to overlap; for example, both `JNotLess` and `JGreaterEq` exist, although usually they have the same semantic values. However, as described in the definition file, there are reasons for this choice:

- Fall-through optimization: conditions can be logically inverted to make the first of the two jump destinations the following instruction; this also has to do with how jump instructions are executed by the virtual machine and how it falls through when executing the conditional instruction, and is an optimization performed at compile time.
- `NaN` (Not a Number): in JavaScript there exists a global property, `NaN`, as defined by the IEEE 754 standard for floating point numbers [10]; all equality and inequality comparisons with it are false; this means that a condition such as `!(x < NaN)`, which compiles to a `JNotLess` instruction, is interpreted as `not (x < NaN)`, since `x < NaN` is false, the whole expression is true; instead `x >= NaN`, which compiles to a `JGreaterEq` instruction, evaluates as false.

Since `NaN` is very rarely used as a static property and is usually the result of erroneous computations, we decided to not enforce these logical differences; it is however possible to observe them at the disassembly level, since the underlying opcodes remain different. The syntax that results from the analysis is also very similar to the corresponding one in JavaScript, as the difference exists in Hermes' implementation of the conditional jumps, and not their syntax in the source code that the decompilation aims to emulate.
Lastly, conditional jump instructions also have their numerical-only versions, that are used for efficiency when the compiler knows the registers being compared only contain numerical values, just as the arithmetic operations mentioned above.

#### Switch

The `SwitchImm` opcode in Hermes is quite unusual, since the switch construct is a common language feature that is compiled to simpler comparison and jump instructions. The way it's implemented uses jump tables, mapped immediately after the end of the bytecode of a function that contains an instruction with this opcode. The instruction therefore takes an integer as input, potentially corresponding to an element in the jump table, a minimum and a maximum integer and a target for the default jump; so if the input is between the

minimum and maximum integers, the entry in the jump table with that index is used to jump, otherwise the default jump is taken. The usage of the minimum and maximum integers is used to allow the creation of a single jump table for each function, and using a subset of it (limited by the minimum and maximum indexes) to access the part relevant to a specific `SwitchImm` instruction; this can also provide some added efficiency since different `SwitchImm` instructions can reuse parts of the same table.

### 3.2.9. Exception handling

These opcodes handle the `try-catch-finally` construct of JavaScript. This is how the compiler handles them: every function that has a `try-catch` statement has an associated exception table, which associates the set of instructions in the `try` block with the address of the corresponding `catch` statement; the `finally` code is just added as a second association of the same set of instructions with a different catch statement.
For reasons relative to how functions are called (see Section 3.2.10), in most situations a `try-catch` statement cannot catch exceptions thrown by functions called in the `try` block, but only the `throw` instructions present in the block itself. This fact and the impossibility to tell Binary Ninja to analyze the `finally` blocks automatically are the main reasons why the code coverage of the analysis performed is not 100%; testing, however, showed that the uncovered sections of code are extremely rare, and Binary Ninja can force their analysis if needed.

### 3.2.10. Calling convention and calls

Hermes' calling convention has been mentioned in Section 2.2.5, but we are going to expand on it and describe how it has been mapped to Binary Ninja API endpoints handling calling conventions.

#### Closures

Before referring to the opcodes that perform the calls themselves, it is necessary to define the concept of closure. In Hermes' context, a closure is the combination of a function reference and a reference to an environment record, as described in Section 3.2.4 Hermes has four opcode dedicated to creating closures, `CreateClosure` and `CreateAsyncClosure` and their `LongIndex` versions; `async` versions also exist, but asynchronous functionalities are still a work in progress in the Hermes project and in fact the corresponding opcodes have not appeared in any of the analyzed binaries. To select which function the closure contains, functions are accessed by index in the function table, which is the in-memory

mapping of functions based on the function metadata entries in the file header, as described in Section 3.2.2. Closures are then used as parameters in instructions exactly as they were a function reference; most importantly, they are used in function calls.

### Call opcodes

Hermes has ten opcodes that perform function calls; all have two properties in common:

- they all return to a register, meaning they all have a register to write the result of the call in; this differs from the `return` instruction of other ABIs, where the return value is put in one or more registers, which are shared between the caller and the callee. As already stated, the Hermes architecture doesn't share registers, as it treats them as stack locations. Also, in JavaScript there is no `void` type, and `undefined` is used in its place in most cases; so if a function does not return anything, Hermes' compiler will make the callee return `undefined` and not use or overwrite the return register in the caller. A natural consequence of this, since the register stack is not shared between functions and every functions sees only the registers it works on, is that the callee will return the content of one of its registers to the caller, which will put it in one of its, possibly a different one; "different" here means that it is a different entity, and more often than not it is actually a register with a different index with respect to the register stack of each function. To handle this peculiar behavior, a dedicated register is added to the architecture implemented in the plugin, called `rr`, which stands for "return register"; `rr` is set to the value of the return register of the callee, kept across the change of context, and the caller's return register is set to the value of `rr`.
- They all have at least one parameter: in JavaScript, all functions are properties of objects, since functions are objects and all objects are properties of an object, be it the global object or another in the object hierarchy; for this reason, all functions can operate on the `this` object reference, which in Hermes is passed as the first parameter in the function call; for functions that do not need to operate on `this`, `undefined` is passed as first parameter instead.

Of the ten calling opcodes, two of them, `CallDirect` and `CallDirectLongIndex`, take the index of the function to call as input (that is the reason why they are named `Direct`), while all the others take a register containing the closure of the function to call, which contains the reference to the function. The direct call opcodes appear to never be used in the binaries analyzed.
Another property to analyze is the way parameters are passed to the function: of the ten

function calling opcodes, four of them, `Call1`, `Call2`, `Call3` and `Call4`, have the function parameters as instruction parameters, in the number found in their name; for example, the signature of the `Call3` opcode is `Call3(Reg8, Reg8, Reg8, Reg8, Reg8)`; the first register is the return register, the second contains the closure of the function to call, as described above, and the remaining three are the three parameters of the function being called (the first of which, as stated, contains the reference to `this`). This is an optimization based on the fact that many function calls have a low number of parameters, and this way to pass parameters is faster for the architecture. The other six opcodes have at least these three parameters: the return register, the register containing the closure, and a positive number `n`; this number is the number of parameters to pass to the function being called, and they are expected to be the `n` top registers on the register stack; for example, if an opcode of this kind has `n=6`, and the top register on the register stack is `r21`, then the function call will take registers from `r16` (which contains the reference to `this`) to `r21` as the parameters to be passed.

### Load opcodes and the `arguments` property

Hermes has a few simple opcodes to load function parameters to registers: `LoadParam`, `LoadThisNS`, `ReifyArguments` and `GetArgumentsPropByVal`; the first accesses the n-th parameter of the function, usually with n>=1 since the first function parameter is the `this` reference; `LoadThisNS` does exactly this, and as documented it is equivalent to `LoadParam(0)` and `CoerceThisNS`, which transforms the reference to `this` to an object. The `arguments` property is a property of functions that allows access to the parameters the function was called with, treated as an array. This array does not necessarily correspond with the function signature, as JavaScript does not make any assumption on signature and input matching. `ReifyArguments` transforms the function parameters to an array object, while `GetArgumentsPropByVal` accesses the n-th element of the reified `arguments` array.

### Implementation

Given all the above properties, it is clear that some of these behaviors are different from more common architectures, regarding both parameter passing and the usage of closures. Mapping these properties to the API required some workarounds and also tolerating some loss of information (mainly regarding environments, as explained in Section 3.2.4 and Section 3.2.11).
Regarding closures, we decided to lift them as the setting of the return register's value to the address of the function. This, as we just stated, means losing the information regarding

the environment paired with the function, but this information would end up not being usable anyway. Regarding the calling itself, we needed to choose a calling convention to apply to Binary Ninja's LLIL to obtain a behavior that emulates the weird way functions are called and also the independence of the register stacks between caller and callee. To do this, two separate parts of the API need to be leveraged: the LLIL lifting part, which we were already working on, to emit the function prologues and epilogues, and handle parameter passing; the calling convention definition [4], which is unfortunately not very extended and effectively undocumented; to understand the way it works, other publicly available Binary Ninja plugins were taken as example. The choice made for the calling convention was to reimplement the well known `cdecl` calling convention. Since Hermes does not have a stack in the classical sense, and all operations are register based, we decided to use Binary Ninja's stack representation to manage the passing of parameters for function calls, and that's what `cdecl` does. Therefore, when a function call instruction is encountered in lifting, the following actions are performed in the caller:

- parameters are pushed on the stack; for the instructions that pass function parameters directly as instruction parameters, these are already available; for the others, which have the number of function parameters as instruction parameters, the register on top of the stack needs to be computed; this is done by recursively exploring the basic blocks in the function, from the current instruction upwards in the control flow, and keeping track of the maximum register index encountered (the code for this code exploration function is found in Appendix A.1); performance wise, this is the single most costly operation in all analysis. Once the top `n` registers are found, their values can be pushed on the stack.
- The target of the call is computed: if the call is direct, as described before, the function index is resolved to the corresponding address, and that is treated as the call destination; otherwise the register containing the closure (which is a parameter of the instruction) is used as the call destination, which becomes an indirect call.
- the `call` LLIL instruction is emitted; it simply pushes the instruction pointer on the stack and jumps to the given destination.
- the return register of the call instruction is set to the content of the `rr` register, which contains the return value of the function (as, in an ideal execution, the called function would have terminated within the previous `call`) .
- the stack pointer register is adjusted, as per the `cdecl` convention, to remove the function parameters from the stack.

In the callee, all that is required is to set the base and stack pointer as required by `cdecl`: in the prologue the base pointer, `bp`, is pushed on the stack and then set to the value of the stack pointer, `sp`; in the epilogue (when the function returns), the stack pointer is set to the base pointer and the base pointer is popped from the stack.
Technically, the callee is also where all the registers should be pushed to the stack to be stored until the function returns, when they are popped back from the stack; this is the way we can emulate the fact that the register stack is separate for each function. Practically, since decompilers do not execute the code, it is enough to specify that all the registers are callee saved in the calling convention configuration to make the Binary Ninja internals not persist register values across function calls. If we were to actually insert the necessary push and pop instructions, two solutions are possible:

- we push all registers, disregarding the fact that almost all of them are not used; in fact, the optimization that uses 1-byte register indexes and can extend to 4-byte indexes only if necessary is based on experimentation by the developers, who observed that functions very rarely use more than 255 registers; therefore pushing all 1,000 registers would add 2,000 instructions to the LLIL code for every function analyzed (1,000 for pushing in the prologue and 1,000 for popping in the epilogue); given that even the simplest application binary has over 4,000 functions, as seen in Section 4.2, that adds literal millions of instructions, which take a large amount of memory during analysis and slow it considerably.
- we only push the "active" registers, that means, all registers up to the highest one on the stack that is in use; this requires to use the aforementioned recursive exploration of basic blocks for every function call; even with the optimization that has been implemented, which is to cache the highest register used in a fully analyzed basic block , this would add a lot of analysis time, since it is still the most costly operation to perform during analysis time wise by almost an order of magnitude compared to all other instructions.

It can be observed that, if we modeled the architecture as a pure stack machine, this problem would not have arisen, since the status of the stack would have been an already available information.
Regarding the `arguments` array, `ReifyArguments` just returns the content of the base pointer register, which allows treating the function parameters as an array starting there, accessed by pointer (in a C-like way, and definitely not as a JavaScript object). The `GetArgumentsPropByVal` instruction, instead, is equivalent to accessing the n-th parameter; this makes it almost identical to a normal load, and this is acceptable for this plugin, as reifying the parameters in the JavaScript sense is only possible at runtime. This is also

why one of the opcodes that operate on the reified `arguments` array, `GetArgumentsLength`, is not implemented in the same way, because the length of the array is only available at runtime.

### 3.2.11. High level opcodes

These are opcodes that cannot be effectively mapped to LLIL instructions; this set of instructions contains instructions that perform high-level operations that exist in JavaScript but would be either extremely complex, long or not implementable at all in LLIL; most of them are not implementable because of the dynamic nature of JavaScript data structures. These opcodes can be divided in the following groups.

#### Object-related opcodes

These are all the opcodes that create and access dynamic JavaScript objects. As already mentioned, JavaScript variables are either of primitive type (boolean, integer, string, and a few others), or objects. JavaScript objects are structured as key-value pairs, they are dynamic, meaning keys can be added and removed, and values can be changed both in value and type. To handle objects, Hermes provides opcodes that perform the main operations on objects: `Get`, which returns a property's value, `Put`, which sets it, and `Delete`, which deletes it; these can be accessed both by `Id` (the key in the key-value pairs) and by `Val`, which can be considered an index-based list access. There are also other opcodes that perform some more high level operations, like creating the objects themselves and adding properties (these opcodes have the `Own` keyword in them to distinguish "own" properties from inherited properties), or even doing it passing a *getter* and a *setter* function for an added property.
Other miscellaneous opcodes in this category are those that operate on JavaScript's `globalObject` [25] and some that handle object creation (like `CreateThis`, which takes an object as prototype and a function as the constructor, and produces an empty object that has to be initialized with a `Construct` instruction).

#### Array-related opcodes

In JavaScript, arrays are objects (as all non-primitive types) with special properties that allow indexing and index-based access. They are however dynamic, elements can be added and removed, and can contain entities of different types.

### Environment-related opcodes

Environment records have been described in Section 3.2.4; Hermes has a few opcodes that manage environments: `CreateEnvironment` creates an environment for the current scope, and returns a reference to it to a register; `GetEnvironment(n)` returns the environment `n` levels up the environment hierarchy, with `n=0` being the current one; then there are a set of `Load` and `Store` instructions to manage the environments' contents (accessed by index).

### Miscellaneous opcodes

There are quite a few opcodes that don't have a specific categorization, and perform some high level or very JavaScript specific operations that are impossible to map effectively to LLIL instructions. Some examples include:

- boolean evaluations that check object type, like `InstanceOf`;
- opcodes that handle the property list of object, used for `for...of` loops;
- `DirectEval`, which performs strict-mode `eval`;
- Loading of JavaScript specific variables, like `LoadConstUndefined`, or objects like the global object or `this` for object methods;
- type conversions, like `AddEmptyString`, which converts non-string entities to string (the name comes from the fact that `x = "" + x` is a common way to perform such conversion).
- Creation and usage of generator functions, as described in Hermes' documentation and following ECMAScript's standard [13][19].

### The solution

The way all these opcodes have been implemented in the plugin is by defining some external functions and constants, which correspond to these opcodes, and lifting these instructions is done by calling them with the instruction parameters as function parameters; the calling convention used is the one described in Section 3.2.10, with the only difference that `this` is not passed as the instructions would not have access to it anyway. There are four exceptions to this: `LoadConstUndefined`, `LoadConstNull`, `CoerceThisNS` and `GetGlobalObject`, as Hermes uses these to load the `undefined`, `null` constants, the reference to `this` and the `globalObject` reference respectively; they have been mapped to external constants instead, and the opcodes are lifted to accesses to them. The idea

behind this choice is that most of these instructions handle objects that are created on a dedicated (and garbage collected) heap; in many programming languages there exist library functions that provide these functionalities, like `malloc` from the C standard library. Binary Ninja's plugin for binaries compiled from C code maps such external functions as external function symbols, extending the file representation to include an artificial section where these functions are mapped, to then refer to them in the binary when they are called. This solution is also therefore a practical workaround for some other opcodes that operate on too high of a level to be effectively mapped to LLIL instructions, like the `GetArgumentsLength` opcode we described in Section 3.2.10. This solution also allows to regroup different opcodes that have been separated for optimization; for example, the `GetByIdShort`, `GetById` and `GetByIdLong` opcodes can be mapped to the same external function, `get`.
The table in Appendix B.1 contains all the opcodes that have been managed in the way described above with their signature, and the artificial functions that have been created to represent them.

### 3.2.12. Builtin functions

Hermes handles some JavaScript functions and object properties through some functions that are built into the engine and made callable through the `CallBuiltin` opcode. These are both standard-defined functions accessible to the user, like the mathematical functions built into the `Math` object, or internal functions that perform operations on objects that are either too infrequent or complex to create a dedicated opcode for. Based on their definitions in the source code, they are actually divided into categories, `private`, `default` and `JS` builtins; the main difference is that `private` ones are accessed the same way in all cases (like `HermesBuiltin.arraySpread`, which performs array spreading for function calls [23]), the `default` ones are accessed via builtins only if the binary was compiled with an option that enforces static builtin functions, and lastly the `JS` ones are called by creating a closure for them with `GetBuiltinClosure` and then calling it with a regular function call. This is documented in the source code file where the builtins are defined [15]; it is also interesting that these functions are accessed by index of definition in this file.
The implementation for these functions was very similar to the one described in Section 3.2.11, by creating an additional section at the end of the file to map them and treating them as external functions. Their signature was based off of the functions' implementation in the Hermes engine and on the ECMAScript documentation for those that are defined there, like the mathematical functions.

### 3.2.13. Miscellaneous

Only three opcodes are left out of the above categorization:

- `Unreachable`, which in the analyzed version of the bytecode corresponds to byte `0x00`; we decided to lift it to `nop`, even though it is not supposed to appear, and trying to execute it stops the execution of the whole application; it is however not expected to ever appear, for the reason we just mentioned.
- `Inc` and `Dec` opcodes, that are an optimization to have the `++` and `-` operators available separately from addition and subtraction, since they are very frequently used. They are simply lifted as additions and subtractions with operand 1.

# 4 | Evaluation

The result of the project described in the previous chapters is a plugin for the Binary Ninja binary analysis platform; this is, by nature, a tool that aids in the analysis of binaries by providing a higher level representation of the binary, in the form of a more human-friendly language. To evaluate the quality of our work, we show the analysis results of our plugin for some representative JavaScript and React Native applications; in most cases, it is impossible to show the complete analysis result as it would be impossibly long to list, so instead we provide meaningful snippets and a summary of the result. We chose to also provide the original React Native source code for the parts that are shown, to compare it directly to the analysis result.

## 4.1. A minimal JavaScript code snippet

The first and simplest test case is the bytecode compiled from a small JavaScript program. This example exists mainly to showcase how simple constructs are represented using the plugin. Listing 4.1 shows the program's source code, which uses some common flow control methods such as conditionals and loops and shows the basic use of JavaScript objects.

**Listing 4.1:** The analyzed JavaScript code snippet.

```
function sum(arg1, arg2, arg3) {
    return arg1 + arg2 + arg3;
}

print(sum(1,2,3));

var obj = {
    "first":"no",
    "second":3,
    "fifth":{"a":1},
    "sixth":[0,0,1]
};
print(obj["fifth"]);
```

```
var a = "yes";

if(sum(obj["sixth"]) == 1){
    print(a);
} else {
    print(obj["first"]);
}

var base = 0;
for (i of [0,1,2,3]){
    base += i;
}
print(base);
```

### 4.1.1. Results

Figure 4.1 shows the result of the analysis performed with our plugin on the main function, in the form of Binary Ninja's highest level intermediate representation language (HLIL), and using the graph view that the platform provides. It should be noted that Binary Ninja analyzes each function separately, and therefore the `sum` function is not visible in the provided view.

The results clearly show how the control flow is reconstructed correctly, as well as most variable names and values; the only instruction that is substantially different is the creation of a static object. This has already been discussed in Section 3.2.11, because of the difficulty of managing dynamic objects on the stack.

Listing C.1 shows the output of the `hermes-dec` decompiler for the same bytecode file. By comparison, the main aspect that is missing is the reconstruction of control flow; the current solution the tool uses is a loop with no end condition, with an inner switch on a variable that represents the instruction offset of jump targets in conditional instructions. This is a simple solution that uses information directly available in the conditional instructions, but it is not as readable as regular control flow expressions. Another observation that can be made is the fact that our result is shorter in overall length, since Binary Ninja's lifting to HLIL groups instructions and function calls in a way that's quite natural for code understanding, while `hermes-dec` decompiles most bytecode instructions one by one. The one evident advantage that `hermes-dec` has over our plugin is the fact that it reconstructs object access and creation syntax to JavaScript's original one. This


Figure 4.1: The analysis result in the highest intermediate language (HLIL).

is possible because `hermes-dec` decompiles to a text file, so the issue of using any syntax is a matter of text output formatting. For our plugin, every instruction is represented through Binary Ninja's API, and especially for the HLIL output, is bound to use Binary Ninja's C-like output syntax.

## 4.2. A baseline React Native application

The next test subject is a true React Native application: following React Native's documentation and using Android Studio, we built the release bundle for the "Hello World" example available in the documentation tutorial section [30]. This application is extremely simple, consisting only of a main component, containing a single view element that shows

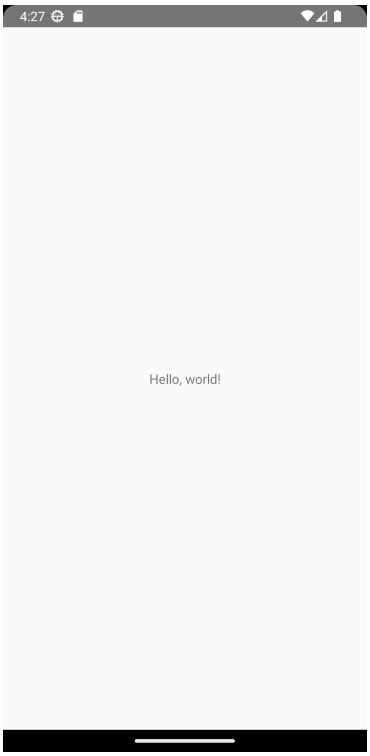


Figure 4.2: Screenshot of the application from the Android Studio emulator.

the classic example text. The main and only view of the application can be seen in Figure 4.2, while the source code of the main React Native file of the application is provided in Listing 4.2. This example is mainly meant to show how React Native components are compiled for Hermes, and additionally observe the large amount of code that is compiled statically with every React Native project.

**Listing 4.2:** The code of the main component of the "Hello World" application.

```
import React from 'react';
import {Text, View} from 'react-native';

const HelloWorldApp = () => {
  return (
    <View
      style={{
        flex: 1,
        justifyContent: 'center',
        alignItems: 'center',
      }}>
      <Text>Hello, world!</Text>
    </View>
```

```
  );
};
export default HelloWorldApp;
```

## Results

Listing 4.3 shows the bytecode file's header, as parsed by our plugin; from there it can be seen that this simple application is composed of 4,170 functions, contains 5,402 strings for a total size of over 77KB, and the overall size of the file is about 718KB. For added context, the whole `apk` file was about 19MB in size.
Our following analysis later revealed that there only actually exist 3,622 functions, as there are some function entries that actually refer to the same function in the bytecode. This example shows that even with a minimal application a lot of additional functions and strings are contained in the file. By looking up the signature of some of the functions that appear in the file, they can be matched with functions that are part of the code bases for React and React Native; since the `import` directives for these two modules can be found in the source code, this is further confirmed. There is also another set of functions that is not from the React and React Native code bases, and it can be traced back to the Metro bundler [8], which is a module that resolves imports in React Native projects and groups the required sources into a single bundle. It is therefore the component that is responsible for the large size of the file.
Listing 4.4 shows the decompilation result of the `HelloWorldApp` component. It clearly illustrates how components and especially JSX elements are compiled for Hermes: as mentioned in Section 2.1.1, JSX elements are converted into method calls by the Babel syntax transformation toolchain [2], returning an element object; for the component, since it is effectively a function, the whole listing is the analysis result of the component itself, and the return value is the aforementioned element object.

**Listing 4.3:** File header metadata for the application as parsed by the plugin.

```
BytecodeFileHeader(
    magic=b'\xc6\x1f\xbc\x03\xc1\x03\x19\x1f',
    version=90,
    sourceHash=b'\x01\xb3\x1c[...]=\x1b\xd1',
    fileLength=735380,
    globalCodeIndex=0,
    functionCount=4170,
    stringKindCount=3,
    identifierCount=2972,
```

```
    stringCount=5402,
    overflowStringCount=13,
    stringStorageSize=79530,
    bigIntCount=0,
    bigIntStorageSize=0,
    regExpCount=45,
    regExpStorageSize=5829,
    arrayBufferSize=7918,
    objKeyBufferSize=1897,
    objValueBufferSize=1330,
    segmentID=0,
    cjsModuleCount=0,
    functionSourceCount=7,
    debugInfoOffset=735340,
    options=BytecodeOptions(
        flags=0,
        address=108,
        type_name='bytecode_options'
    ),
    padding=b'\x00\x00\x00[...]\x00\x00\x00',
    address=0,
    size=128,
    type_name='file_header'
)
```

**Listing 4.4:** Binary Ninja's analysis result of the `HelloWorldApp` component.

```
HelloWorldApp:
    int64_t rr_17 = getEnv(level: 0)
    int64_t rr = load(env: rr_17, index: 0)
    int64_t rr_1 = load(env: rr_17, index: 1)
    void* const r3 = undefined
    int64_t rr_4 = get(rr(r3, get(object: rr_1, index: 3)), "jsx")
    int64_t rr_5 = load(env: rr_17, index: 2)
    int64_t rr_6 = get(rr_5, "View")
    int64_t rr_9 = get(rr(r3, get(object: rr_1, index: 3)), "jsx")
    int64_t rr_10 = get(rr_5, "Text")
    int64_t rr_11 = newObject()
    putown(object: rr_11, "Hello, world!", property: "children")
    int64_t rr_12 = rr_9(r3, rr_10, rr_11)
    int64_t rr_13 = newObject()
    putown(
        object: rr_13,
        newObject(size: 3, n_elems: 3, key_index: 0x75f, value_index: 0x199),
        property: "style"
    )
```

```
    putown(object: rr_13, rr_12, property: "children")
    return rr_4(r3, rr_6, rr_13)
}
```

## 4.3. A full-sized application

Lastly, we decided to test the plugin on a full-fledged application; the main objective for this test was performance evaluation against larger application bundles. For this, we chose a moderately popular open source application, Joplin [6], which is a cross-platform note-taking app. Its source code is available on GitHub, so we could still compare our analysis results with the original source. To check that the analysis still worked correctly under the heavier analysis load, we reported a function that performs an update on the state of the application, specifically handling multiple selection of note identifiers. Its source code is provided in Listing 4.5.

**Listing 4.5:** The original source code of the function.

```
function updateSelectedNotesFromExistingNotes(draft: Draft<State>) {
        const newSelectedNoteIds = [];
        for (const selectedNoteId of draft.selectedNoteIds) {
                for (const n of draft.notes) {
                        if (n.id === selectedNoteId) {
                                newSelectedNoteIds.push(n.id);
                        }
                }
        }
        if (
        JSON.stringify(draft.selectedNoteIds) === JSON.stringify(newSelectedNoteIds)
    ) return;
        draft.selectedNoteIds = newSelectedNoteIds;
}
```

### Results

Dimension wise, the bytecode file of this application was over 13MB in size (the whole app was 28MB); this proved challenging for Binary Ninja and our plugin, which took over 37 minutes to analyze almost fully. "Almost" is due to the fact that Binary Ninja has a limit on the size of functions it analyzes automatically; two functions (global and another unnamed one) were not analyzed for this reason; Binary Ninja can obviously analyze them if asked manually.

The analysis result for the function we chose is provided in Listing 4.6, this time in pseudo-C form; this consists of a form very similar to the HLIL we used before, but has a closer resemblance to actual C syntax. Regarding analysis, the plugin still performed

it properly, reconstructing the control flow and object accesses coherently with respect to the original source code. One detail that can be observed is the fact that all strings in this analysis snippet are not terminated correctly, often containing additional characters at the end. This is the effect of our solution for large amounts of string data mentioned in Section 3.2.2, which uses the non null-terminated version of strings to reduce memory usage when analyzing large applications.
We could also observe that a common optimization for JavaScript code, minification, is not applied; minifying code consists in replacing function names with strings as short as possible, often one character long, to reduce overall file size. Its absence is advantageous for reverse engineering purposes, as function names are kept and they can help understanding what functions do from their name, but it obviously implies a larger file size, as the strings containing the names of the functions are longer.

**Listing 4.6:** Binary Ninja's analysis result of a function in pseudo-C format.

```
void* const updateSelectedNotesFromExistingNotes(int64_t arg1)

{
    void* const r0 = undefined;
    undefined;
    undefined;
    int64_t rr = newArray(0);
    int64_t rr_1 = get(arg1, "selectedNoteIdsDupliceert de not\dots");
    int64_t rr_2 = iteratorBegin(rr_1);
    while (true)
    {
        int64_t rr_3 = next(rr_2, rr_1);
        if (rr_2 == r0)
        {
            break;
        }
        int64_t rr_4 = get(arg1, "notes\n\t\t\t\t\tWHERE\n\t\t\t\t\dots");
        int64_t rr_5 = iteratorBegin(rr_4);
        while (true)
        {
            int64_t rr_6 = next(rr_5, rr_4);
            if (rr_5 == r0)
            {
                break;
            }
            if (get(rr_6, "idth: 1em;\n\t\t\t\theight: 1e\dots") == rr_3)
            {
                get(rr, "push-notification-ios/push-notif\dots")(
                    rr, get(rr_6, "idth: 1em;\n\t\t\t\theight: 1e\dots")
                );
            }
        }
    }
    void* const r3_1 = globalObject;
    int64_t rr_11 = get(r3_1, "JSON file: ADD COLUMN user_updat\dots");
    int64_t rr_14 = get(rr_11, "stringify]:-([?&])_=[^&]*.jsEngi\dots")(
```

```
        rr_11, get(arg1, "selectedNoteIdsDupliceert de not\dots")
    );
    int64_t rr_15 = get(r3_1, "JSON file: ADD COLUMN user_updat\dots");
    if (rr_14 == get(rr_15, "stringify]:-([?&])_=[^&]*.jsEngi\dots")(rr_15, rr))
    {
        return r0;
    }
    put(arg1, rr, "selectedNoteIdsDupliceert de not\dots");
    return r0;
}
```

# 5 | Conclusions and future work

This work had the objective of analyzing the Hermes engine's architecture and bytecode and produce a tool that could support mobile application analysis. Although the platform we used as support had some limitations in relation to our needs, and Hermes has some internal behaviors that are inherently hard to model, our result is a step forward in the effort of reverse engineering of the Hermes architecture, and an useful aid for analysis of React Native applications.

The first and most evident limitation in the work presented so far is the platform it was built on, Binary Ninja; although it is an excellent tool for reverse engineering purposes, it falls short in the representation of structures such as those encountered analyzing the Hermes engine and its bytecode. This is however reasonable, as Binary Ninja's focus is mostly on binaries for physical architectures, which feature the common register-memory stack combination, and modeling something like a virtual machine that makes extensive use of heap-managed data structures is understandably not the perfect fit, despite the workarounds and abstractions that we put in place.

Another quite evident limitation of our plugin is performance: as we described in Chapter 4, the analysis of large applications can take several minutes and it rarely ended up not being possible at all, because of the excessive memory usage and possible crashes that can happen when handling very large application bundles. The causes for this last problem can be summarized in two related issues:

- the size of the application bundles: React Native uses Metro [8] as its bundling tool; what this does is creating a large, single file from the tree of requirements that forms when an application is developed, including all the corresponding code that is required; this means a lot of code from libraries is included in the bundle. This has been described in the test provided in Section 4.2, where an extremely basic React Native application ended up having over 3,500 different functions, most of which are part of the React Native libraries, Metro bundling libraries and more. It is easy to imagine that importing additional libraries for added functionalities can add up to a very large amount of static code and data that is usually of very little interest for application analysis compared to the code sections unique to the app

being analyzed.

- Plugin implementation efficiency: the way our plugin maps bytecode files in memory, and the subsequent additional memory usage of Binary Ninja's internal functionalities, make a way larger usage of memory than the size of the binary being analyzed. This is because almost all data, be it static data or code, is represented in memory at least twice, as the raw binary file itself and as multiple representations (for example, all the metadata structures are obviously in the raw file as raw bytes, but they also exist as separate variables in memory once the parser has handled them).

There are improvements that can be made in the future regarding the aforementioned shortcomings:

- Reducing redundancy between Binary Ninja's data structures and the ones our plugin creates could reduce memory usage and analysis time;
- the addition of new functionalities and support for more general constructs in the Binary Ninja API would allow some of the analysis that were worked around in our plugin to be implemented fully, providing a substantially more accurate result. In particular, "workflows" are a new analysis automation tool that has very recently been released in its primordial form; this tool could potentially allow modifying the result of the current analysis at the higher level intermediate languages, for example rewriting our "workaround" function calls with a more JavaScript-like syntax.

Looking at more structural changes, recreating our plugin in C++ could provide performance benefits, since Python's ease of use is counterbalanced by its poorer performance in computation-intensive tasks. Moreover, in that case, a lot of code could be taken directly from Hermes' source code, since the project is developed in C++ as well; we chose Python for our project mainly because it was the one we had by far the most familiarity with development-wise. Another possible project-wide rework could be implementing the architecture as a pure stack machine, as mentioned in Chapter 3; this would pose a new set of challenges, but it might benefit the overall result as it would model the architecture more closely to its actual form.

# Bibliography

[1] Binary Ninja intermediate languages. URL https://docs.binary.ninja/dev/bnil-overview.html.

[2] Babel toolchain website. URL https://babeljs.io/.

[3] Binary Ninja platform website. URL https://binary.ninja.

[4] Binary Ninja calling convention API module. URL https://api.binary.ninja/binaryninja.callingconvention-module.html.

[5] GitHub directory for hbcdump tool. URL https://github.com/facebook/hermes/tree/c2cd9e385a922f486a55e6bd70db2032f78379ed/tools/hbcdump.

[6] Joplin application website. URL https://joplinapp.org/.

[7] JSX conversion plugin in Babel. URL https://babeljs.io/docs/babel-plugin-transform-react-jsx.

[8] The Metro bundler website. URL https://facebook.github.io/metro/.

[9] React Native project website. URL https://reactnative.dev.

[10] IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. doi: 10.1109/IEEESTD.2019.8766229.

[11] Binary Ninja architecture plugin guide, January 2020. URL https://binary.ninja/2020/01/08/guide-to-architecture-plugins-part1.html.

[12] hbctool blog post, 2021. URL https://suam.wtf/posts/react-native-application-static-analysis-en/.

[13] Generators in Hermes' IR documentation, 2021. URL https://hermesengine.dev/docs/ir#generator-overview.

[14] Bigint object in ECMAScript specification, 2022. URL https://262.ecma-international.org/#sec-bigint-objects.

[15] Hermes builtins, 2022. URL https://github.com/facebook/hermes/blob/0763eee4bf461df30ffefe0180d09835bb6bb58d/include/hermes/FrontEndDefs/Builtins.def. Source code.

[16] Hermes bytecode file format, 2022. URL https://github.com/facebook/hermes/blob/hermes-2022-11-03-RNv0.71.0-85613e1f9d1216f2cce7e54604be46057092939d/include/hermes/BCGen/HBC/BytecodeFileFormat.h. Source code.

[17] Hermes bytecode opcodes definitions, 2022. URL https://github.com/facebook/hermes/blob/hermes-2022-11-03-RNv0.71.0-85613e1f9d1216f2cce7e54604be46057092939d/include/hermes/BCGen/HBC/BytecodeList.def. Source code.

[18] Hermes bytecode stream generation, 2022. URL https://github.com/facebook/hermes/blob/hermes-2022-11-03-RNv0.71.0-85613e1f9d1216f2cce7e54604be46057092939d/lib/BCGen/HBC/BytecodeStream.cpp. Source code.

[19] Generator object in ECMAScript specification, June 2022. URL https://262.ecma-international.org/#sec-generator-objects.

[20] Hbctool GitHub repository, 2022. URL https://github.com/bongtrop/hbctool.

[21] JavaScript usage statistics, 2023. URL https://w3techs.com/technologies/details/cp-javascript/.

[22] React Native out of tree platforms, 2023. URL https://reactnative.dev/docs/out-of-tree-platforms.

[23] ECMA. Array spread operator syntax in ECMAScript specification, June 2022. URL https://262.ecma-international.org/#prod-SpreadElement.

[24] ECMA. Environment records in ECMAScript specification, June 2022. URL https://262.ecma-international.org/#sec-environment-records.

[25] ECMA. The global object in ECMAScript specification, June 2022. URL https://262.ecma-international.org/#sec-global-object.

[26] JetBrains. Mobile applications development statistics, 2023. URL https://www.jetbrains.com/lp/devecosystem-2022/miscellaneous/.

[27] Meta. React Native release 0.66.1, 2019. URL https://github.com/facebook/react-native/releases/tag/v0.60.1.

[28] Meta. JSX specification, August 2022. URL https://facebook.github.io/jsx/.

[29] Meta. React Native release 0.70, 2022. URL https://reactnative.dev/blog/2022/09/05/version-070.

[30] Meta. "Hello World" example in React Native documentation tutorial, March 2023. URL https://reactnative.dev/docs/tutorial#hello-world.

[31] M. Moulinier. Releasing hermes-dec, an open-source disassembler and decompiler for the React Native Hermes bytecode. Technical report, January 2023. URL https://labs.p1sec.com/2023/01/09/releasing-hermes-dec-an-open-source-disassembler-and-decompiler-for-the-react-native-hermes-bytecode/.

[32] StackOverflow. Trends for the main front-end development frameworks, 2023. URL https://insights.stackoverflow.com/trends?tags=reactjs%2Cvue.js%2Cangular%2Csvelte%2Cangularjs%2Cvuejs3.

# A | Source code

This section contains the Python code used to recover the register with the highest index used within a function, up to the given instruction address. This is necessary to recover function parameters for one of Hermes' two calling conventions, as described in Section 3.2.10.

**Listing A.1:** Python code to retrieve the highest index register used in a function.

```
from binaryninja import Function, BasicBlock, InstructionTextTokenType
from .logger import hermes_logger

highest_reg_cache = {}

# return the highest register index in @fun up to @addr;
def find_highest_used_reg(fun: Function, addr: int) -> int:
    cur_bb: BasicBlock = fun.get_basic_block_at(addr)
    fun_key = fun
    if fun_key not in highest_reg_cache:
        highest_reg_cache[fun_key] = {}
    highest_reg = explore_bb(
        cur_bb, addr - cur_bb.start,
        highest_reg_cache[fun_key]
    )
    return highest_reg

# returns the highest register index allocated in the basic block
def explore_bb(
        bb: BasicBlock,
        end: Optional[int] = None,
        visited: Dict = {}
    ) -> Optional[int]:
    max_reg = None
    passed_size = 0
```

```
    if not end:
        end = bb.end
    for instr in bb:
        if len(instr[0])>=3:
            try:
                # text tokens are "<opcode_name>"," ","<register>"
                token = instr[0][2]
            except Exception as e:
                hermes_logger.log_error(instr)
            if token.type == InstructionTextTokenType.RegisterToken:
                id = int(token.text[1:])
                if max_reg is None or id>max_reg:
                    max_reg = id
        passed_size += instr[1]
        if passed_size >= end:
            break

    if end == bb.end and bb:
        # the whole basic block has been checked
        # cache the highest reg index
        visited[bb.start] = max_reg
    for prev_bb in bb.strict_dominators:
        if prev_bb.start in visited:
            new_m = visited[prev_bb.start]
        else:
            new_m = explore_bb(prev_bb, None, visited)
        if max_reg:
            if new_m and new_m>max_reg:
                max_reg = new_m
        else:
            if new_m:
                max_reg = new_m
    return max_reg
```

# B | High level opcodes

This section shows the high level opcodes described in Section 3.2.11 and the signatures of the artificial functions and constants created to represent them.

| **Hermes opcode** | **artificial function/constant signature** |
|---|---|
| NewObjectWithBuffer | void* newObject(size, n_elems, keys, values) |
| NewObjectWithBufferLong | void* newObject(size, n_elems, keys, values) |
| NewObject | void* newObject(arg) |
| NewObjectWithParent | void* newObject(parent) |
| TypeOf | void* typeOf(arg) |
| InstanceOf | void* instanceOf(arg1, arg2) |
| IsIn | void* isIn(arg1, arg2) |
| GetEnvironment | void* getEnv(level) |
| StoreToEnvironment | void storeToEnv(env, index, value) |
| StoreToEnvironmentL | void storeToEnv(env, index, value) |
| StoreNPToEnvironment | void storeToEnv(env, index, value_non_ptr) |
| StoreNPToEnvironmentL | void storeToEnv(env, index, value_non_ptr) |
| LoadFromEnvironment | void* load(env, index) |
| LoadFromEnvironmentL | void* load(env, index) |
| GetGlobalObject | const globalObject |
| GetNewTarget | void* getNewTarget() |
| CreateEnvironment | void* createEnv() |
| DeclareGlobalVar | void declareGlobal(name) |
| GetByIdShort | void* get(object, property) |
| GetById | void* get(object, property) |
| GetByIdLong | void* get(object, property) |
| TryGetById | void* get(object, property) |
| TryGetByIdLong | void* get(object, property) |
| PutById | void put(object, value, property) |
| PutByIdLong | void put(object, value, property) |
| TryPutById | void put(object, value, property) |

| Hermes opcode | artificial function/constant signature |
|---|---|
| TryPutByIdLong | void put(object, value, property) |
| PutNewOwnByIdShort | void putown(object, value, property) |
| PutNewOwnById | void putown(object, value, property) |
| PutNewOwnByIdLong | void putown(object, value, property) |
| PutNewOwnNEById | void putown(object, value_non_enum, property) |
| PutNewOwnNEByIdLong | void putown(object, value_non_enum, property) |
| PutOwnByIndex | void putown(object, value, index) |
| PutOwnByIndexL | void putown(object, value, index) |
| PutOwnByVal | void putown(object, value, property, is_enum) |
| DelById | void* del(object, property) |
| DelByIdLong | void* del(object, property) |
| GetByVal | void* get(object, index) |
| PutByVal | void put(object, value, index) |
| DelByVal | void* del(object, index) |
| PutOwnGetterSetterByVal | void put(object, property, getter, setter, is_enum) |
| GetPNameList | void* getPropList(object, index, prop_size) |
| GetNextPName | void* getNextProp(props, object, index, prop_size) |
| Catch | void* catch(exception) |
| DirectEval | void* directEval(code) |
| ThrowIfEmpty | void* throwIfEmpty(arg) |
| Debugger | void debugger() |
| AsyncBreakCheck | void asyncBreakCheck() |
| ProfilePoint | void profilePoint() |
| CreateThis | void* createThis(prototype, constructor) |
| SelectObject | void* selectObj(this, constructor_ret) |
| LoadConstUndefined | const undefined |
| LoadConstNull | const null |
| CoerceThisNS | const this |
| ToNumber | void* toNumber(arg) |
| ToNumeric | void* ToNumeric(arg) |
| ToInt32 | void* toInt32(arg) |
| AddEmptyString | void* toString(arg) |
| GetArgumentsLength | void* argumentsLength(arg) |
| CreateRegExp | void* createRegExp(pattern, flags, bytecode_idx) |
| StartGenerator | void startGenerator() |
| ResumeGenerator | void* resumeGenerator(do_return) |

| Hermes opcode | artificial function/constant signature |
|---|---|
| CompleteGenerator | void completeGenerator() |
| IteratorBegin | void* iteratorBegin(source) |
| IteratorNext | void* next(iterator_or_index, source_or_next) |
| IteratorClose | void iteratorClose(iterator_or_index, ignore_exc) |
| NewArrayWithBuffer | void* newArray(size, n_elems, index) |
| NewArrayWithBufferLong | void* newArray(size, n_elems, index) |
| NewArray | void* newArray(size) |

# C | Hermes-dec examples

This section contains the analysis result of the `hermes-dec` tool [31] for a simple JavaScript application.

**Listing C.1:** The result of decompilation using the `hermes-dec` tool.

```
_fun0: for(var _fun0_ip = 0; ; ) switch(_fun0_ip) {
case 0:
    obj = undefined;
    a = undefined;
    base = undefined;
    sum = undefined;
    r2 = undefined;
    r0 = global;
    r1 = function(a0, a1, a2) { // Original name: sum, environment: r1
        r1 = a0;
        r0 = a1;
        r1 = r1 + r0;
        r0 = a2;
        r0 = r1 + r0;
        return r0;
    };
    r0['sum'] = r1;
    r4 = r0.print;
    r6 = r0.sum;
    r3 = 1;
    r5 = 2;
    r1 = 3;
    r1 = r6.bind(r2)(r3, r5, r1);
    r1 = r4.bind(r2)(r1);
    r4 = {};
    r4['a'] = r3;
    r1 = {'first': 'no', 'second': 3};
    r1['fifth'] = r4;
    r4 = [0, 0, 1];
    r1['sixth'] = r4;
    r0['obj'] = r1;
```

```
    r4 = r0.print;
    r1 = r0.obj;
    r1 = r1.fifth;
    r1 = r4.bind(r2)(r1);
    r1 = 'yes';
    r0['a'] = r1;
    r4 = r0.sum;
    r1 = r0.obj;
    r1 = r1.sixth;
    r1 = r4.bind(r2)(r1);
    if(!(r1 != r3)) { _fun0_ip = 185; continue _fun0 }
case 162:
    r3 = r0.print;
    r1 = r0.obj;
    r1 = r1.first;
    r1 = r3.bind(r2)(r1);
    _fun0_ip = 201; continue _fun0;
case 185:
    r3 = r0.print;
    r1 = r0.a;
    r1 = r3.bind(r2)(r1);
case 201:
    r1 = 0;
    r0['base'] = r1;
    r1 = [0, 1, 2, 3];
    r3 = r1[Symbol.iterator];
    r1 = r3().next;
case 220:
    r4 = r1().value;
    r5 = r3;
    if(!(r5 !== r2)) { _fun0_ip = 267; continue _fun0 }
case 231: // try_start_0
    r0['i'] = r4;
    r5 = r0.base;
    r4 = r0.i;
    r4 = r5 + r4;
    r0['base'] = r4;
case 258: // try_end0
    _fun0_ip = 220; continue _fun0;
case 260: // catch_target0
    CatchBlockStart(arg_register=1);
    r3.return();
    throw r1;
case 267:
```

```
    r1 = r0.print;
    r0 = r0.base;
    r0 = r1.bind(r2)(r0);
    return r0;
}
```

# List of Figures

# Listings