Demystifying React Native Android Apps for Static Analysis

YONGHUI LIU, Monash University, Australia
XIAO CHEN, University of Newcastle, Australia
PEI LIU, CSIRO's Data61, Australia
JORDAN SAMHI, CISPA, Germany
JOHN GRUNDY, Monash University, Australia
CHUNYANG CHEN, Technical University of Munich, Germany
LI LI[*], Beihang University, China

React Native, an open-source framework, simplifies cross-platform app development by allowing JavaScript-side code to interact with native-side code. Previous studies disregarded React Native, resulting in insufficient static analysis of React Native app code. This study initiates the investigation of challenges when statically analyzing React Native apps. We propose ReuNify to improve Soot-based static analysis coverage for JavaScript-side and native-side code. ReuNify converts Hermes bytecode to Soot's intermediate representation. Hermes bytecode, compiled from JavaScript code and integrated into React Native apps, possesses a unique syntax that eludes current JavaScript analyzers. Additionally, we investigate opcode distribution and conduct in-depth analyses of the usage of opcode between popular apps and malware. We also propose a benchmark consisting of 97 control-flow-related cases to validate the control-flow recovery of the generated intermediate representation. Furthermore, we model the cross-language communication mechanisms of React Native to expand the static analysis coverage for native-side code. Our evaluation demonstrates that ReuNify enables an average increase of 84% in reached nodes within the call graph and further identifies an average of two additional privacy leaks in taint analysis. In summary, this paper demonstrates that ReuNify significantly improves the static analysis for the React Native Android apps.

CCS Concepts: • **Software and its engineering → Abstraction, modeling and modularity**.

Additional Key Words and Phrases: Android, React Native, Mobile App, Static Analysis

ACM Reference Format:
Yonghui Liu, Xiao Chen, Pei Liu, Jordan Samhi, John Grundy, Chunyang Chen, and Li Li. 2024. Demystifying React Native Android Apps for Static Analysis. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (November 2024), 33 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Mobile apps have become the primary source of digital consumption, with a growing number of users relying on apps for various purposes such as shopping, entertainment, and communication. As a result, businesses are investing heavily in mobile app development to reach their

[*] Corresponding author.

Authors' addresses: Yonghui Liu, Yonghui.Liu@monash.edu, Monash University, Melbourne, Australia; Xiao Chen, Xiao.Chen@newcastle.edu.au, University of Newcastle, Newcastle, Australia; Pei Liu, CSIRO's Data61, Melbounre, Australia, Pei.Liu@data61.csiro.au; Jordan Samhi, CISPA, Germany, jordan.samhi@cispa.de; John Grundy, John.Grundy@monash.edu, Monash University, Melbounre, Australia; Chunyang Chen, chun-yang.chen@tum.de, Technical University of Munich, Munich, Germany; Li Li, Beihang University, Beijing, China, lilicoding@ieee.org.

1049-331X/2024/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

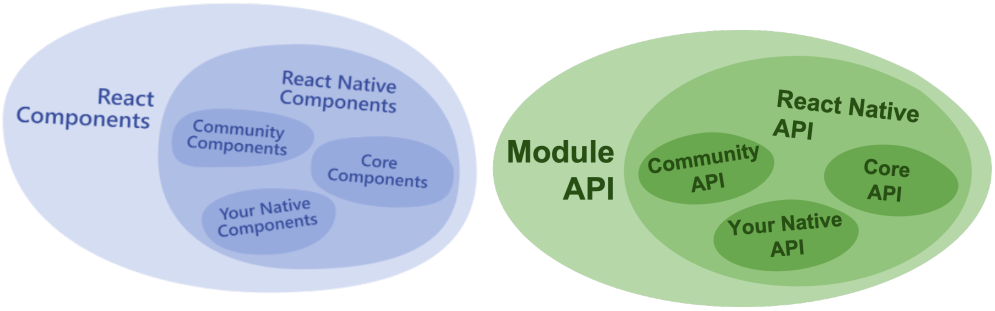
target audience and remain competitive in the market. Many companies are facing the challenge of needing to build mobile apps for multiple platforms, specifically for both Android and iOS. This cross-platform mobile app development has gained popularity due to its consistency across platforms, cost-effectiveness, time efficiency, wide audience reach, and easier maintenance [36].

Nowadays, React Native (used in Facebook, Shopify, Skype, etc.) and Flutter (used in Google Ads, Reflectly, Alibaba, etc.) have become the two most popular frameworks for cross-platform mobile app development [37]. Each of these cross-platform solutions has its own capabilities and strengths [57]. React Native, an open-source framework, gained popularity since its 2015 launch by combining traditional mobile development with Node.js-based flexibility. The core idea of React Native is to empower cross-platform JavaScript APIs to invoke platform-specific functions involving invoking Objective-C/Swift or Java/Kotlin functions to utilize iOS and Android components. This feature sets it apart from other cross-platform mobile application development technologies which often end up rendering web-based views. With React Native, developers can create a shared codebase in JavaScript that works on both Android and iOS. This is achieved by providing a set of cross-platform APIs and Components that conceal platform-specific native code and abstract the differences between platforms. React Native is flexible and can be used in existing Android and iOS projects or to create a new app from scratch [92].

The stats from AppBrain [3] report that among the top 500 Android apps in the US, 14.85% of installed apps are built with React Native. In fact, in the category of top 500 US Android apps, React Native is the third most popular framework, right after Kotlin and Android Architecture Components. While the use of the React Native framework can streamline the app development process, it also introduces new challenges for app analysis, particularly in terms of static analysis. The main difficulty with static analysis on React Native apps is their use of multiple programming languages with varying semantics, along with the complex mechanisms inherent in the React Native framework. React Native apps now package Hermes bytecode instead of traditional JavaScript code. This transition poses a further challenge for existing static analyzers, as they cannot interpret Hermes bytecode. These factors can make it very challenging to thoroughly analyze and fully comprehend the app's codebase.

In the last decade, Android app analysis has been a prominent research theme in software engineering. Static analysis techniques have been implemented by many approaches and tools for bug detection, security property checking, malware detection, and empirical studies. Unfortunately, as far as we know, there are no existing techniques or tools for analyzing apps developed with React Native. Approaches used by the current state-of-the-art app analysis tools, which were intended for traditional Android apps, are not sufficient for efficiently covering the executable code in React Native apps [57]. This is due to the complexity of the underlying mechanism of the React Native framework and the adoption of Hermes bytecode in these apps. In light of these challenges, we explore a new research direction to enable static analysis of the whole program of React Native Android apps.

We propose ReuNify, aiming to fill the gap in the whole-app analysis [83]. ReuNify extracts and unifyies artefacts from both the Java and JavaScript sides of React Native Android Apps into Jimple [111], the intermediate representation in Soot. To the best of our knowledge, ReuNify is the first static analyser for React Native Android apps[79]. By transforming JavaScript-side code into Jimple, ReuNify provides the opportunity for several analyses (e.g., call graph analysis, control flow graph and taint flow analysis) in the literature to readily account for JavaScript code. By modelling React Native mechanism, ReuNify increases the coverage of Java-side code analysis. ReuNify is thus a multi-step static analysis approach that we implement as a framework to enable the whole-programme analysis for React Native Android Apps. This paper substantially extends our earlier conference paper on ReuNify [83], providing much more technical details, additional



(a) React Component

(b) Module API

Fig. 1. The structure of React Component and Module API in the developer's JavaScript side

evaluation, and novel benchmarking for future React Native Android Apps analysis tools. This research makes the following key contributions:

- We provide the first systematic categorization of the challenges for static analysis on React Native Android apps. We propose the *first* effective techniques to facilitate static analysis on React Native Android apps;
- We propose ReuNify, a novel approach to build a unified model for React Native Android Apps code. We have implemented ReuNify that generates the Jimple code from both native-side code and JavaScript-side code, which facilitates the whole program analysis on the React Native Android app package.
- We investigate the prevalence of Hermes Opcodes in real-world apps and also propose a benchmark to assess the control-flow sensitivity of the Jimple code generated from Hermes bytecode.
- We demonstrate that ReuNify can significantly enhance the static analysis coverage of React Native Android Apps' native side code. Running ReuNify on real-world malware and popular apps, the size of callgraph for native side code is significantly enhanced. Using ReuNify in conjunction with FlowDroid can reveal previously unseen sensitive leaks.
- We release our open-source prototype ReuNify and all artifacts used in our study at:

https://github.com/DannyGooo/ReuNify

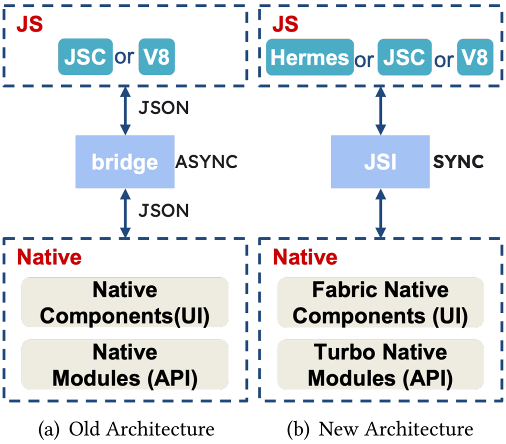
The remainder of this paper is organized as follows. We outline the key motivation for this work in Section 2, and Section 3 presents challenges for static analysis on React Native Android apps. Section 4 presents key aspects of our approach. Section 5 presents our studied datasets, our experimental setup, and our experimental results. Section 6 presents the discussion for our research. Section 7 discusses key related work, and Section 8 summarises this paper.

2 BACKGROUND AND MOTIVATION

2.1 Background

2.1.1 React Native. React Native, developed by Meta, is an open-source framework that simplifies mobile app development. It provides a unified platform for iOS and Android app development without relying on WebView [34], setting it apart from competitors like Ionic [14] and Cordova [9]. React Native is based on the popular *React* framework [24], which is a Node.js-based JavaScript library used for creating web user interfaces. Further, it features cross-language communication between JavaScript and the native side, blending native app performance with web development's flexibility and efficiency.

When developing a React Native application, the features (i.e., React's declarative UI paradigm and JavaScript.) of React are used to organize reusable and nestable *React Components* [1] for building the mobile user interface. Various business logic and *Module APIs* can be further used inside the state or lifecycle [19, 35] of those *React Components* to attain the desired features and functionalities. Figure 1 further categorizes *React Components* and *Module APIs* based on the entity responsible for maintaining them. React Native is equipped with pre-existing *core Components* and *core APIs* that are readily available for use [92]. In addition, the React Native team offers documentation on how to encapsulate native-side functionality for JavaScript-side code [1, 21]. The React Native ecosystem has been enriched by a diverse range of third-party libraries that are actively maintained by the community [26]. These libraries play a significant role in enhancing the overall robustness of the platform.



(a) Old Architecture (b) New Architecture

Fig. 2. Cross-Language Communication Mechanism in React Native

2.1.2 Native-side code in React Native Apps.

A React Native app provides access to native-side code that is not inherently available in JavaScript. The React Native team has published guides for encapsulating native-side features in both the Old Architecture [1, 21] and the New Architecture [22]. While React Native doesn't expect this feature to be part of the usual development process, It is essential that it exists in case developers want to use code in Objective-C, Swift, Java, or C++. The *bridge*/*JSI* can expose instances of Java/Objective-C/C++ (native) classes to JavaScript (JS) as JS objects, facilitating cross-language communication inside React Native apps. As demonstrated in Figure 2(a), *bridge* [38] was used to facilitate the exchange of information between JavaScript and native side code in the old architecture of React Native. *Bridge* allowed JavaScript to interact with the platform-specific *Native Components* and *Native Modules* for building mobile apps. However, this architecture suffers from issues such as asynchronous behavior, single-threading, and extra overheads (JSON format) that impact performance and flexibility [38].

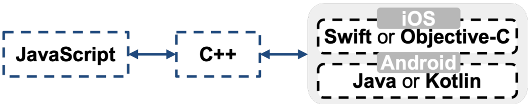


Fig. 3. React Native's Cross-Language Implementation in Developer's side

To address these issues, the new architecture of React Native adopts the *JavaScript Interface* (*JSI*) [18], as shown in Figure 2(b). The *JSI* allows a JavaScript object to hold a reference to a C++ object and vice versa, enabling synchronous execution, concurrency, lower overhead, code sharing, and type safety [38]. This approach provides several advantages over the old architecture and serves as the foundation of the cross-language communication mechanism for new architecture. With *JSI*, developers can use *Turbo Native Modules* and *Fabric Native Components* to achieve high-performance mobile applications on both iOS and Android platforms.

In both the New and Old Architecture of React Native, the cross-language communication mechanism is built using C++. Furthermore, React Native exhibits compatibility with three distinct JavaScript engines, namely Hermes [13], JSC [16], and V8 [32], all of which are constructed using C-based programming languages. Those engines enable the execution of JavaScript code, while the *bridge* and *JSI* serve as channels for the injection of variables, functions, and declaration of global elements, therefore, augmenting the pre-existing JavaScript code. React Native can leverage this capability to enable communication between JavaScript and the C/C++ world. The C/C++ code can further communicate with platform-specific native code. On the iOS side, Objective-C is capable of communication with C/C++ programming languages because it extends the C language. On the Android side, the Java Native Interface (JNI) is used to establish contact with the C/C++ environment. As shown in Figure 3, JavaScript code can communicate with those platform-specific native codes through C++. The native-side feature can be encapsulated from C++ code or platform-specific code. This study only focuses on the Dalvik bytecode generated from Java or Kotlin source code, which gets investigated by the majority of Android static analyzers [79].

2.1.3 JavaScript-side code in React Native Apps. React Native enables developers to build the app's logic and user interface in JavaScript. When one React Native project gets built as a mobile app, the JavaScript code within the React Native project undergoes bundling using *Metro*[20], a JavaScript bundler. This bundler accepts options and an entry file, and in return, it produces a JavaScript file that includes all the JavaScript files. The bundled JavaScript file would be further compiled into bytecode, with Hermes selected as the JavaScript Engine. Once the app launches, the code in the bundled file is loaded and further executed by the JavaScript engine.

Since version 0.70.0 (September 2022), the default JavaScript engine in React Native has been changed from JavaScriptCore (JSC) [16] to Hermes Engine [13]. Before that, Hermes Engine was introduced to React Native Android and React Native iOS since version 0.60.4 and version 0.64.0 as an optional engine, respectively. The legacy JavaScript Engine parses all JavaScript codes using just-in-time (JIT) compilation. With the inclusion of Hermes engine, JavaScript source code would be compiled to bytecode ahead of time (AOT), which saves the interpreter from having to perform this expensive step during app startup and also contributes to a smaller app bundle size. However, the use of the Hermes engine in React Native can make static analysis much more challenging. The generated Hermes bytecode is not as easily readable or accessible as the JavaScript code, which makes the current state-of-the-art tools designed for JavaScript [68, 69, 97] useless in front of Hermes bytecode. Additionally, current state-of-the-art Android static code analysis approaches [49, 74, 79, 101] overlook the apps developed with React Native.

2.1.4 Soot. The development of Soot originated from a Java compiler testbed project initiated at McGill University in the year 2000. Over time, the research community has shown a growing interest in static code analysis across various applications. Consequently, Soot has come to be the preferred program analysis framework for both Java and Android. As shown by the proceedings of international conferences, multiple prototypes rely on Soot to perform their individual static analysis. Soot has been the most popular program analysis framework for Android apps [79]. The continued relevance of soot could be attributed to its notable qualities, notably its widely used intermediate representations (IR).

Soot has supported Java source code, Java bytecode, Dalvik Bytecode, and CIL (Common Intermediate Language) bytecode for program analysis. Soot provides the corresponding front end to transform those input codes into its main intermediate representation (IR), Jimple [111]. Jimple is a stack-less, three-address representation which features only 15 instructions. It uses explicit control flow without nesting, i.e., solely through conditional or unconditional gotos. Any method code can be viewed as a graph of Jimple statements associated with a list of Jimple local variables, which enables the creation of simple control flow graphs (CFGs). Furthermore, Soot offers multiple algorithms for constructing call graphs. To enable a more complete static analysis on React Native Android app, we propose a new front end to transform Hermes bytecode into Jimple to make those JavaScript-side codes analyzable inside the Soot framework. In addition, we explore the cross-language mechanism of React Native to increase the reachability of the call graph derived from Dalvik bytecode.

2.2 Motivating Example

The React Native framework's complex mechanism conceals a significant portion of the executable code of Android apps built with it from state-of-the-art static analysis tools [49, 77, 100]. With one analysis for the React Native Android Apps, *Skype*, *com.skype.raider* [29], we make the case that React Native mechanism should be considered in static analysis approaches.

Skype is a popular app for real-time video calls, with more than one billion installations. This app is developed with React Native framework. Considering the cross-language communication mechanism in React Native, we discuss both the JavaScript side code and Java side code. In the example, we've sourced version 8.83.0.411 of the Skype app from APKMirror[1].

JavaScript Side: The app, *Skype*, incorporate version 89 of the Hermes engine, and stores the JavaScript-side code as Hermes bytecode. This bytecode can be decompiled into a textual disassembly file containing 3,589,897 lines of text and a file size of 119 megabytes. The disassembly file contains 87,400 methods and 3,016,341 lines of opcode statements. We propose a new front-end system to convert Hermes bytecode into Jimple. This transformation will make Hermes bytecode analyzable within the Soot framework.

Java Side: We generated a callgraph of the app, *Skype*, using FlowDroid for taint flow analysis. The callgraph consisted of 5,169 nodes and 18,282 edges, and no privacy leaks were detected. After examining the call graph, it was found that the Java methods exposed through the Native Module API (135 Modules, 724 methods) and React Native Components (106 Components, 813 methods) were not captured in the callgraph. Upon these methods, the call graph expanded considerably to include 13,629 nodes and 51,395 edges, and three privacy leaks were identified.

This paper presents a novel strategy to address the challenge of the hidden executable code in React Native Android apps, which has been a gap in the current research. The aim of this paper is to enable whole program analysis for React Native Android app.

3 CHALLENGES AND ILLUSTRATION

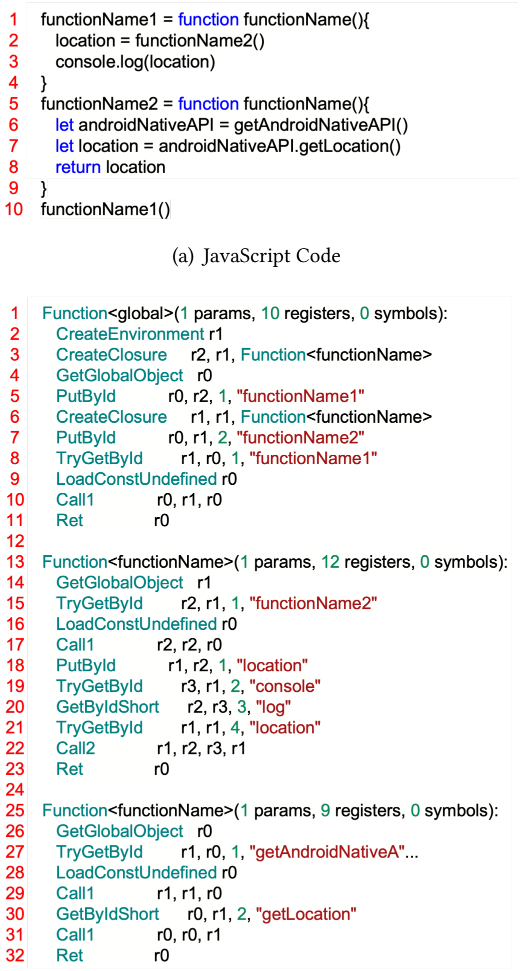
In this section, we discuss the challenges related to code coverage during static analysis for React Native Android apps, focusing on JavaScript-side and platform-specific code (Java/Kotlin). Although React Native apps involve C++ code for the framework, including the JavaScript engine and cross-language mechanism, development documents for creating apps with C++ are rare. Developers typically use JavaScript and Java/Kotlin, which are the focus of this work. We began by disassembling Hermes bytecode for manual analysis and exploring the challenges associated with Dalvik bytecode when using existing static analysis tools.

The process of classifying these challenges into distinct categories involved iterative discussions among the authors. While it's worth noting that this methodology doesn't offer a formal assurance against the possibility of uncovering more challenges in the future, we are confident that the importance of these identified challenges presents obstacles to the static analysis of React Native Android apps.

[1] https://www.apkmirror.com/apk/skype/skype-skype/skype-skype-8-83-0-411-release/

3.1 Challenges related to the Hermes bytecode

The execution of the JavaScript code in Figure 4 would leak the "*location*" returned from the Android native-side API to the *console.log()*. However, understanding its corresponding Hermes bytecode proves to be challenging. Binary code analysis inherently presents difficulties [91] due to the complex representation of the compiled code, which hinders proper investigation [72]. Furthermore, binary code analysis not only inherits most of the challenges associated with its source code analysis but also introduces new obstacles resulting from optimization techniques applied during the compilation process. As a bytecode variation within the specific JavaScript VM (Hermes engine in this work), Hermes bytecode naturally inherits most of the well-known challenges related to JavaScript source code analysis. In this section, we delve into the obstacles that hinder the static analysis of Hermes bytecode. These challenges are associated with implementing automated static analysis for large-scale Hermes bytecode.



```
functionName1 = function functionName(){
  location = functionName2()
  console.log(location)
}
functionName2 = function functionName(){
  let androidNativeAPI = getAndroidNativeAPI()
  let location = androidNativeAPI.getLocation()
  return location
}
functionName1()
```

(a) JavaScript Code

```
Function<global>(1 params, 10 registers, 0 symbols):
  CreateEnvironment r1
  CreateClosure     r2, r1, Function<functionName>
  GetGlobalObject  r0
  PutById          r0, r2, 1, "functionName1"
  CreateClosure     r1, r1, Function<functionName>
  PutById          r0, r1, 2, "functionName2"
  TryGetById        r1, r0, 1, "functionName1"
  LoadConstUndefined r0
  Call1            r0, r1, r0
  Ret              r0

Function<functionName>(1 params, 12 registers, 0 symbols):
  GetGlobalObject  r1
  TryGetById        r2, r1, 1, "functionName2"
  LoadConstUndefined r0
  Call1            r2, r2, r0
  PutById          r1, r2, 1, "location"
  TryGetById        r3, r1, 2, "console"
  GetByIdShort      r2, r3, 3, "log"
  TryGetById        r1, r1, 4, "location"
  Call2            r1, r2, r3, r1
  Ret              r0

Function<functionName>(1 params, 9 registers, 0 symbols):
  GetGlobalObject  r0
  TryGetById        r1, r0, 1, "getAndroidNativeA"...
  LoadConstUndefined r0
  Call1            r1, r1, r0
  GetByIdShort      r0, r1, 2, "getLocation"
  Call1            r0, r0, r1
  Ret              r0
```

(b) Hermes bytecode

Fig. 4. SootClass and their relationship in Soot

Challenge 1: Framework-Engine Synchronization. The tight coupling between React Native framework versions and their corresponding Hermes engine versions creates challenges for static analysis. Each React Native version requires a specific Hermes engine, and any mismatch can lead to compatibility issues and unexpected behavior. During the building process, JavaScript code must be compiled with the exact Hermes engine version that corresponds to the React Native framework version in use to ensure proper functionality. This, in turn, necessitates the version-specific decompiler for the targeted version of the React Native app. By November 2023, over 40 Hermes engine versions had been released to sync with React Native framework updates. The evolution of Hermes bytecode (e.g., new or altered instruction format in bytecode), reflected in the change of textual disassembly from its official disassembler, *hbcdump*, across versions, necessitates that static analyzers remain adaptable and informed to interpret the bytecode. The scant documentation on Hermes bytecode disassembly hinders static analyzer developers from staying informed with modifications. Consequently, designing, developing, and validating one static analyzer for various versions of Hermes bytecode becomes a demanding, resource-heavy task necessitating adaptability to version-specific intricacies.

Challenge 2: Limitation in Offical Decompiler. Hermes engine provides a built-in disassembler called *hbcdump*, but it has limitations. Firstly, it cannot display the complete value of a string in its textual disassembly if the string's length exceeds a certain limit. For example, in Figure 4(a), the string "*getAndroidNativeAPI*" in the JavaScript code at line 6 cannot be fully represented in the textual disassembly generated by *hbcdump* at line 27 of Figure 4(b). This incomplete representation of the string not only impacts the string analysis but also has implications for the property names of objects associated with analysis for heap objects. Additionally, it lacks clear function identifiers in function declaration instructions in textual disassembly. In JavaScript, developers can define functions with the same name, including duplicate or anonymous functions. As shown in lines 1 and 5 of Figure 4(a), two separate functions with the same function name, "*functonName*", are defined without any parameters. In the Hermes bytecode, the "*CreateClosure*" opcode is utilized to declare a function in a register, programmatically, facilitating the function declaration in the runtime environment. As demonstrated in lines 3 and 6 of Figure 4(b), functions with the same name, *"functonName"*, are stored in registers *r2* and *r1*, respectively. Two code blocks with the same function name,*"functonName"* are defined in lines 13 and 25, respectively. The use of unclear function identifiers at function declaration in textual disassembly, along with the separation of function declaration and its definition, creates difficulties in relating function declaration and function definition. Overall, the incomplete display of string values and unclear function identifiers in function declaration instructions significantly hinder the analysis of Hermes bytecode.

Challenge 3: Interprocedural Analysis. In Hermes bytecode, functions are treated as *First-Class Objects* similar to their handling in JavaScript, enabling them to be passed as arguments, returned from functions, and stored in registers. Function invocations in Hermes bytecode are made through functions assigned to registers, utilizing an *Indirect Function Call* for all invocation instructions. As shown in line 22 in Figure 4(b), *"Call2 r1, r2, r3, r1"*, invokes the function store at register *r2* with arguments (i.e., *r3* and *r1*), and finally the return value would be stored at and overwrite the value at *r1*. The type of each register in Hermes bytecode is not stable, as indicated by the multiple instructions (at lines 14, 21, and 22) assigning different values to the host register *r1* demonstrated in the second function definition at Figure 4(b). This lack of stable types renders type-based call graph algorithms, like RTA or XTA, inapplicable to Hermes bytecode. Moreover, Hermes can execute functions from external APIs, like "*console.log*", which adds to the uncertainty since these functions are outside the bytecode file. In React Native, interfaces provided by the C++ and device sides (Objective-C/Swift for iOS and Java/Kotlin for Android) introduce further uncertainty. Analyzing Hermes bytecode alone doesn't reliably predict calls to external APIs without considering the host environment interactions. The distinctive syntax and optimizations present in Hermes bytecode, combined with the dynamic characteristics inherited from JavaScript, make interprocedural analysis more challenging, marking it as an area for future study.

3.2 Challenges Related to Dalvik bytecode

This section examines the challenges impeding inter-procedural static analysis of Dalvik bytecode, focusing on the shortcomings of state-of-the-art call graph algorithms in capturing the developer's Java/Kotlin code implementation for React Native apps. These obstacles stem from the design of call graph construction algorithms specifically tailored for Android's Dalvik bytecode.

Challenge 4: Cross-Language Mechanism The cross-language communication mechanism in React Native enables JavaScript code to access native functionality. Regardless of the architecture (Old or New), developers encapsulate native functions as JavaScript interfaces, allowing runtime registration and interaction with JavaScript code. To accurately predict an app's control flow, static analyses must consider React Native's cross-language mechanisms. Developing sound analysis for mobile apps proves non-trivial and requires specialized algorithms, particularly for apps that integrate complex frameworks like Android and the not-yet well-explored React Native framework across at least three programming languages (as depicted in Figure 3).

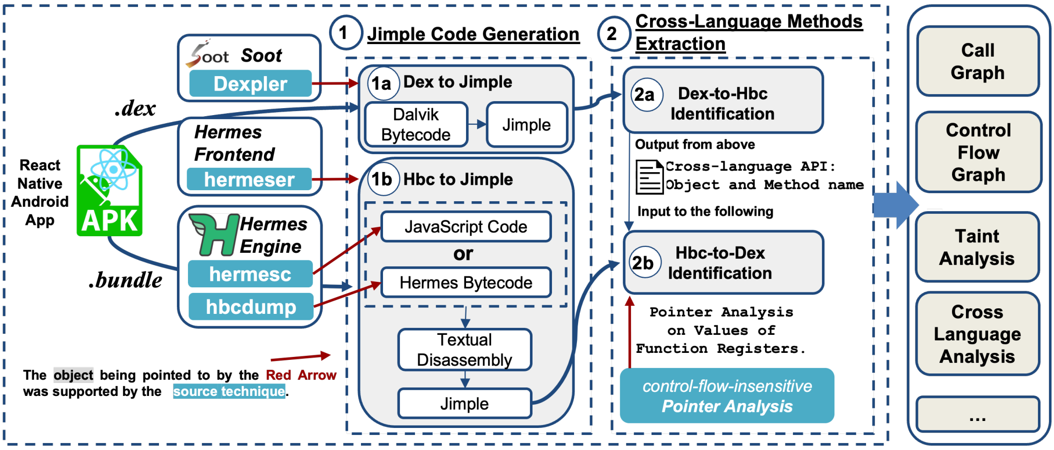


Fig. 5. Overview of ReuNify.

Challenge 5: Framework Transition. React Native is transitioning from its legacy Architecture to the New Architecture, altering how developers create Native API and UI Components. *Native Module* and *Native Components* are the established technologies utilized in the legacy architecture, but they will be deprecated once the New Architecture stabilizes. The New Architecture introduces *Turbo Native Modules* and *Fabric Native Components* as replacements [21]. The transition from Old Architecture to New Architecture is one continuing process across multiple versions. The first major milestone of the New Architecture was the introduction of the JSI in React Native 0.59, released in March 2019 [27]. The migration to the New Architecture is still ongoing (April 2024). The transition process would have implications on the implementation for developers, which would pose challenges for algorithms designed for the identification of those Native APIs and UI Components.

4 APPROACH

To facilitate the automatic analysis of the React Native Android Apps, a prototype named ReuNify has been developed. This tool addresses the previously mentioned challenges and aims to improve static analysis capabilities for React Native Android apps. As depicted in Figure 5, ReuNify contains two key modules including ***(1) Jimple Code Generation*** and ***(2) Cross-Language Methods Extraction***.

(1) Jimple Code Generation: This module proposes solutions for challenges associated with the Hermes Bytecode (Challenges 1, 2, and 3). It facilitate the transformation of both Dalvik-side and JavaScript-side code in React Native Android apps into the unified intermediate representation known as Jimple. This conversion facilitates a wide range of automated analyses.

(2) Cross-Language Methods Extraction: This module analyzes the Jimple code produced by the first module to model cross-language communication in React Native Android apps. It tackles challenges in Dalvik bytecode analysis (Challenges 4 and 5) with the aim of modeling Dalvik bytecode invocations from the JavaScript side, thereby improving code coverage. The module achieves this by automatically identifying customized native-side functionality interfaces.

These modules work together to provide a comprehensive solution for analyzing the complex, multi-language structure of React Native Android applications.

4.1 Jimple Code Generation

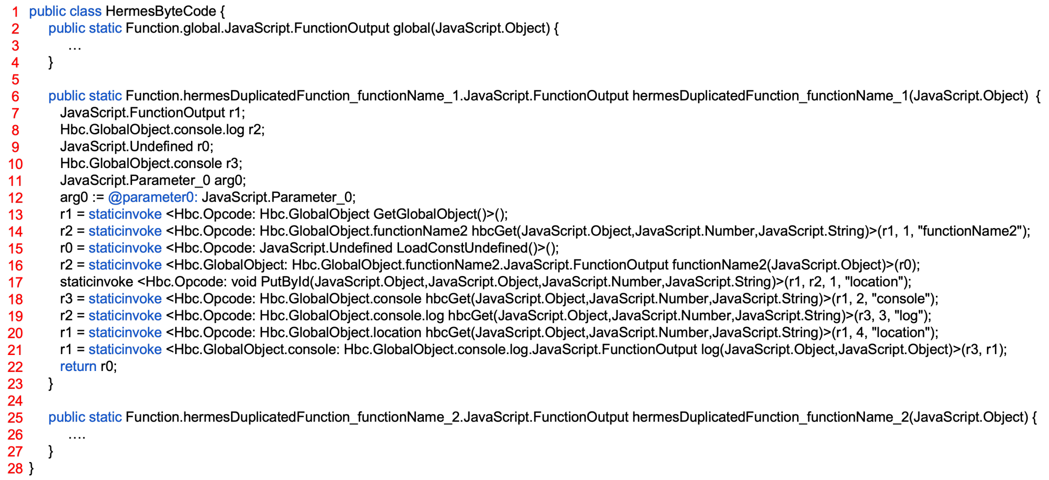
This module serves as the front end for disassembling React Native Android Apps, aiming to create a unified intermediate representation (IR) for both Java-side and JavaScript-side code. As discussed in section 3.1 regarding challenges 1, 2, and 3, the main obstacle in this unification process arises from React Native's implementation of Hermes bytecode. To address these challenges, we developed hermesr, a front-end tool that automates the decompilation of various Hermes bytecode versions using custom-built decompilers. The module then transforms the decompiled Hermes bytecode into Jimple format, facilitating intra-procedural analysis. Our proposed Hermes frontend, *hermesr*, works in conjunction with Dexpler to automate the generation of Jimple code from input directories of React Native Android Apps.

This module leverages a *divide-and-conquer* strategy to facilitate the construction of unified intermediate representation for the Java-side code and JavaScript-side code in React Native Android apps. As shown in **1a** of Figure 5, the Dalvik bytecode files within the React Native Android app can be transformed into Jimple by *Dexpler*[51] that is a front-end dealing with Dalvik bytecode, and has been integrated into Soot as one module. In the implementation in sub-step **1b** of Figure 5, either JavaScript code or Hermes bytecode can be represented into Jimple. The normal JavaScript code can be compiled and represented as textual disassemblies by *hermesc* (i.e. the Hermes compiler), while the Hermes bytecode can be decompiled and represented as textual disassemblies by *hbcdump* (i.e., Hermes bytecode disassembler). Both *hermesc* and *hbcdump* can be built from the Hermes engine project [12]. Additionally, we implement a parser inside *hermeser* that would be used to parse and transform the textual disassemblies into Jimple code.

In a typical analysis case, Soot is launched by specifying the target directory as a parameter. This directory contains the program (one *.apk* in this example) for analysis. First, the *main()* method of the Main class is executed. It calls *Scene.loadNecessaryClasses()*, where Soot locates the specified source code files (*.bundle* file for JS-side React-Native code in this example) from the input *.apk* file by *SourceLocator.v().getClassesUnder(path)*. Second, *HbcClassSource*, is implemented as a module inside Soot framework to create a *SootClass* from the corresponding disassembled Hermes bytecode. When the resolver has a reference to a *ClassSource* (*HbcClassSource*), it calls resolve() on it. *SootMethod*s are then created, and *MethodSource*s (containing the information from the function in textual disassembly) are distributed for each *SootMethod*. When a function of textual disassembly is stored into *MethodSource*, its opcode instructions are organized into blocks that can link to each other through the control flow. During the solving of each *SootMethod*, the Jimple statements would be created from Hermes opcode instructions within all blocks, then the *jump* between each block can be connected. So that the generated Jimple code keeps the same control flow with the Hermes bytecode.

ReuNify first generates disassembly texts from the located either normal JavaScript or Hermes bytecode. Subsequent transformations are based on these disassembly texts. The transformation from one representation to another is inherently complex because it requires an understanding of the semantics of both representations. The bytecode file contains the bytecode functions, along with essential metadata and auxiliary data sections, which are imperative for the successful execution in runtime. The file format is described in the header file BytecodeFileFormat.h [5] at Hermes open source project. It consists of many parts, including the *FILE HEADER*, *FUNCTION HEADER TABLE*, *STRING TABLE&STORAGE*, and *FUNCTION BYTECODES*.

The transformation procedure involves extracting the section for *FUNCTION BYTECODES*. Following this, the string from the *FUNCTION HEADER TABLE* and the *STRING TALBE&STORAGE* will be correlated in order to fully reconstruct the whole string used in the *FUNCTION BYTECODES* section. Furthermore, we continue to partition the code block for each function serialized in



```
public class HermesByteCode {
    public static Function.global.JavaScript.FunctionOutput global(JavaScript.Object) {
        ...
    }

    public static Function.hermesDuplicatedFunction_functionName_1.JavaScript.FunctionOutput hermesDuplicatedFunction_functionName_1(JavaScript.Object) {
      JavaScript.FunctionOutput r1;
      Hbc.GlobalObject.console.log r2;
      JavaScript.Undefined r0;
      Hbc.GlobalObject.console r3;
      JavaScript.Parameter_0 arg0;
      arg0 := @parameter0: JavaScript.Parameter_0;
      r1 = staticinvoke <Hbc.Opcode: Hbc.GlobalObject GetGlobalObject()>();
      r2 = staticinvoke <Hbc.Opcode: Hbc.GlobalObject.functionName2 hbcGet(JavaScript.Object,JavaScript.Number,JavaScript.String)>(r1, 1, "functionName2");
      r0 = staticinvoke <Hbc.Opcode: JavaScript.Undefined LoadConstUndefined()>();
      r2 = staticinvoke <Hbc.GlobalObject: Hbc.GlobalObject.functionName2.JavaScript.FunctionOutput functionName2(JavaScript.Object)>(r0);
      staticinvoke <Hbc.Opcode: void PutById(JavaScript.Object,JavaScript.Object,JavaScript.Number,JavaScript.String)>(r1, r2, 1, "location");
      r3 = staticinvoke <Hbc.Opcode: Hbc.GlobalObject.console hbcGet(JavaScript.Object,JavaScript.Number,JavaScript.String)>(r1, 2, "console");
      r2 = staticinvoke <Hbc.Opcode: Hbc.GlobalObject.console.log hbcGet(JavaScript.Object,JavaScript.Number,JavaScript.String)>(r3, 3, "log");
      r1 = staticinvoke <Hbc.Opcode: Hbc.GlobalObject.location hbcGet(JavaScript.Object,JavaScript.Number,JavaScript.String)>(r1, 4, "location");
      r1 = staticinvoke <Hbc.GlobalObject.console: Hbc.GlobalObject.console.log.JavaScript.FunctionOutput log(JavaScript.Object,JavaScript.Object)>(r3, r1);
      return r0;
    }

    public static Function.hermesDuplicatedFunction_functionName_2.JavaScript.FunctionOutput hermesDuplicatedFunction_functionName_2(JavaScript.Object) {
        ....
    }
}
```

Fig. 6. Jimple Code generated from the JavaScript code or Hermes Bytecode in Figure 4(a).

the *FUNCTION BYTECODES* section. Subsequently, the bytecode instructions pertaining to each individual bytecode function would be converted into Jimple statements. In order to ensure the parser's version awareness, we have accounted for a total of 207 opcode types spanning over 39 distinct versions of the Hermes engine. This comprehensive coverage is achieved by conducting a thorough review of the version indicator file [7] and opcode list file [6] from the open-source project, as documented in their git history. We would open source *hermeser* and continue to maintain it. As shown in Fig 4(b), one instruction of Hermes bytecode is composed of one or more operands and one opcode. Hermes bytecode is a register-based bytecode, and it uses registers as operands for opcode instructions. Hermes bytecode adopts variable-length instructions. Each operand to a bytecode instruction has a fixed type and width, defined by the opcode. Fixed-type/width instructions enable the efficient decode process in the interpreter, which also contributes to an efficient implementation for our parser. A full list of Hermes bytecode opcodes can be found in BytecodeList.def at the Hermes open source project. [6]

In the transformation process, each Hermes opcode instruction is then mapped to its associated Jimple statements. The majority of Hermes opcode instructions are transformed into the corresponding static invoke statement, *staticinvoke*, on one callee function. Those callee functions are declared with class *Hbc.Opcode*, with the same method name as the corresponding Hermes opcode name, which preserves the original semantics of the program. However, there are opcodes related to conditional or unconditional gotos, which need to be transformed into corresponding goto statements in Jimple to keep control flow syntactically and semantically. The Hermes opcode instructions are arranged into blocks, and these blocks can be interconnected through control flow statements. As the MethodSource is processed, Jimple instructions are generated for each of the blocks, and the connections between each block's jumps are established. This ensures that the resulting Jimple code maintains the same control flow as the original Hermes bytecode.

To ensure *hermeser* is capable of analyzing different versions of React Native apps, we invested significant engineering resources in a thorough examination and further modification of all 38 versions of the source code of the Hermes engine disassembler. Specifically, we delve into the source code of *hbcdump*, to eliminate the effects of the length restriction of the string display for all 38 Linux-based versions of *hbcdump*, and further, we build our customized *hbcdump* tool. This endeavor was crucial for guaranteeing the comprehensive presentation of string values in textual disassembly. Without this improvement, the engine would default to displaying partial values when the string length surpassed a particular threshold, potentially impacting the subsequent static analysis process.

Apart from the challenge posed by partial string values, another perplexing aspect of the program representation in the textual disassembly of Hermes bytecode is the duplication of function names. As illustrated in lines 3 and 6 of Figure 4(b), the same function name, *"functionName"*, was employed to register two distinct functions defined in lines 13 and 25, rendering it impossible to link the function name to its corresponding function code block. To tackle this issue, we take a proactive approach during the parsing phase of hermeser by recording all function names. These duplicated function names are then renamed following a specific strategy, where the new name comprises *"hermesDuplicatedFunction"* followed by the original function name and its index (i.e., the order of appearance) combined with an underscore (*"hermesDuplicatedFunction_thefunctionName_index"*). For instance, the duplicated name "functionName" would be renamed as *"hermesDuplicatedFunction_functionName_0"* and *"hermesDuplicatedFunction_functionName_1"*, respectively. Ensuring a unique function name for each function code block is vital for effective function identification, especially in the context of the *CreateClosure* opcode for interprocedural analysis.

To grab inter-procedural behavior, a control-flow-insensitive strategy was employed to generate Jimple statements from Hermes opcode instructions, with opcode registers translated into Jimple local variables. These variables' types are dynamically assigned and modified as each Hermes instruction is processed. For instance, in Figure 6, the return value type evolves from *Hbc.GlobalObject* to *Hbc.GlobalObject.console.log* between lines 13, 18, and 19, leading to a method invocation on line 21. This approach enables the consistent inference of methods like console.log(). Typically, Hermes opcodes are converted into Jimple's *staticinvoke* statements using *Hbc.Opcode* for the class name, with the opcode value as the method name, and return types are dynamically tracked. As the transformation propagates, the return type for subsequent statements is determined by the opcode's semantic significance and the associated register's type. For example, the opcodes "*TryGetById*" and "*GetByIdShort*" depicted in lines 19 to 20 of Figure 4(b), which serve to retrieve a single value from an object using a property name. This operation leads to a dynamic adjustment of return types, as exemplified in lines 18 to 19 of Figure 6, where the types evolve into *Hbc.GlobalObject.console* and *Hbc.GlobalObject.console.log*, respectively. In addition to addressing the load-related opcode mentioned above, we also abstract the function initialization opcode to enable some level of inter-procedural analysis. However, it's important to note that achieving a sound call graph construction within Hermes bytecode is a complex task, and it falls outside the scope of this current work. We plan to introduce improved point-analysis techniques in our future research efforts to address this challenge.

This module addresses key challenges associated with the Hermes engine, specifically focusing on automated analysis of different bytecode versions. By leveraging custom decompilers, we enable automatic decompilation of various Hermes bytecode iterations. The module then converts this bytecode into Jimple format, facilitating intra-procedural analysis. To capture inter-procedural behavior, we implemented a straightforward control-flow insensitive approach during the transformation process. While this module successfully tackles Challenge 1 and Challenge 2, the development of a robust framework for inter-procedural analysis (Challenge 3) remains an open problem. We have identified this as a promising avenue for future research, acknowledging the non-trivial nature of this undertaking.

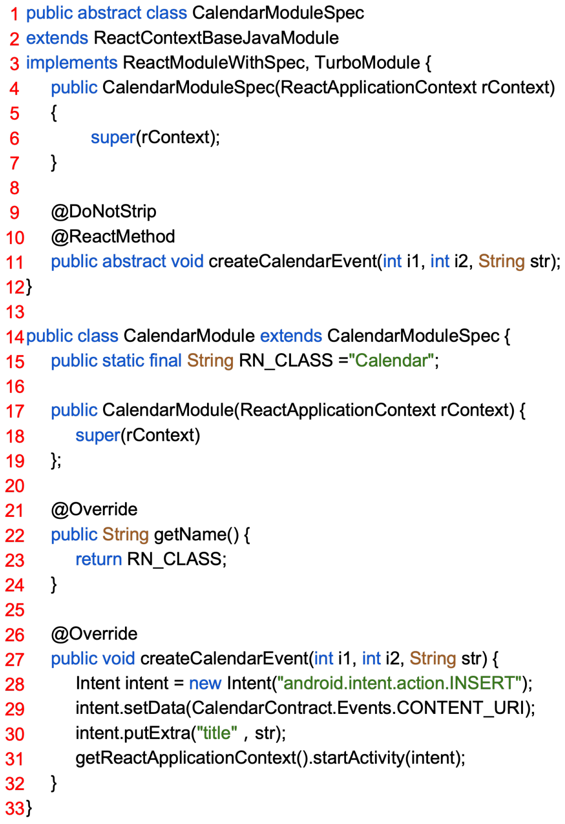
4.2 Cross-Language Methods Extraction

This module analyzes the Jimple code produced by the first module to properly model cross-language invocations of Dalvik side from the JavaScript side. As outlined in section 3.2 concerning challenges 4 and 5, React Native apps allow JavaScript code to access native functionalities through Dalvik-side function calls. However, existing static analyzers inadequately model these Dalvik-side code invocations. Moreover, cross-language invocations lack inherent connections between sides. This module addresses these issues by identifying and extracting these cross-language invocations, a crucial step towards comprehensive whole-program analysis.

Step 2a: Dalvik-to-Hermes Invocation Extraction. As mentioned in Section 2, *React Native* is gradually replacing the legacy Architecture with the New Architecture. The implementation for developers to create Native API and Native Components is also changed with the update of Architecture. *Native Module* and *Native Components* are the established technologies utilized in the legacy architecture. They will be deprecated in the future once the New Architecture becomes stable. The New Architecture uses *Turbo Native Module* and *Fabric Native Components* to achieve similar results [21]. In this case, we take *Turbo Native Module* as one example to explain step **2a** of ReuNify. (The intuition is that the identification process for *Native Module* is similar and easier than the *Turbo Native Module*.)

This step is performed over 5 sub-steps: ❶ Analyzing class hierarchy to record classes that extend *ReactContextBaseJavaModule* and also implement both *ReactModuleWithSpec* and *TurboModule* as shown in the line 2 and line 3 of Figure 7. ❷ Records the name for the method with *@ReactMethod* annotation. (as shown in the line 10 of Figure 7) ❸ Track class hierarchy to detect the classes that extend the classes recorded in sub-step 1 (as shown in the line 14 of Figure 7). ❹ Go through the methods in the class recorded in sub-step 3, and retrieve out the methods that overwrite the methods recorded in sub-step 2 (as shown in the line 27 of Figure 7). ❺ Retrieve the method with the sub-signature, *java.lang.String getName()* (as shown in line 22 of Figure 7), and further extract the return value of this method (e.g., the return value is *Calendar* at line 15 of Figure 7) as the Module API name.

The aforementioned procedure showcases ReuNify's approach for the Dalvik-to-Hermes Identification within the New Architecture (i.e., *Turbo Native Module*). Implementing the Dalvik-to-Hermes Identification within the Old Architecture (i.e., Native Module) is less challenging than the aforementioned process. A process resembling sub-step ❶ is essential to discover the class that encapsulates the *Native Module*, where



```
1  public abstract class CalendarModuleSpec
2  extends ReactContextBaseJavaModule
3  implements ReactModuleWithSpec, TurboModule {
4      public CalendarModuleSpec(ReactApplicationContext rContext)
5      {
6          super(rContext);
7      }
8
9      @DoNotStrip
10     @ReactMethod
11     public abstract void createCalendarEvent(int i1, int i2, String str);
12 }
13
14 public class CalendarModule extends CalendarModuleSpec {
15     public static final String RN_CLASS ="Calendar";
16
17     public CalendarModule(ReactApplicationContext rContext) {
18         super(rContext)
19     };
20
21     @Override
22     public String getName() {
23         return RN_CLASS;
24     }
25
26     @Override
27     public void createCalendarEvent(int i1, int i2, String str) {
28         Intent intent = new Intent("android.intent.action.INSERT");
29         intent.setData(CalendarContract.Events.CONTENT_URI);
30         intent.putExtra("title" , str);
31         getReactApplicationContext().startActivity(intent);
32     }
33 }
```

Fig. 7. Module API registration example in New Architecture of React Native.

methods annotated with *@ReactMethod* would be considered as Module API methods. Subsequently, sub-step ❺ can be carried out to determine the Module API name in the same class. A similar approach can be used to identify cross-language communication on the Dalvik side for *Native Components* and *Fabric Native Components*, by tracking the method annotated with either *@ReactProp* or *@ReactPropGroup*. However, due to space limitations, we cannot provide all the technical details here. For a more comprehensive understanding, please refer to ReuNify's open-source project.

Step 2b: Hermes-to-Dalvik Invocation Extraction. The Module API name and methods name retrieved from the Step (2a) would be used as the identifier for the cross-language invocation on the JavaScript side. Compared to Java code analysis, pointer analysis is more challenging in Hermes bytecode due to the language's dynamic feature, as Hermes bytecode is compiled from JavaScript. This means that register values are not determined until runtime, which potentially leads to instability of the value in function invocation's callee registers and complicates analysis. To address this, we use a control-flow-insensitive technique to track the value stored in a register (variable). As seen in Figure 6, from lines 13, 18, and 19, the value type on the left-hand side changed from *Hbc.GlobalObject* to *Hbc.GlobalObject.console.log*.

In the process of Hermes-to-Jimple transformation, all the registers that are used as callee of function invocations are recorded. The Module API names and method names retrieved from step (2a) are used as a filter to detect the Hermes-to-Dalvik invocation. To implement the cross-language invocation for the Java-side code from the JavaScript side, one object name will be used on the JavaScript side to access the object that is exposed from the Java-side code. In the example in Figure 7, the value, *Calendar*, which is retrieved by the sub-step ❺ at Step (2a), is the object name exposed to the JavaScript side code. To access the method wrapped into the exposed cross-language object, the method name would be used to retrieve the value (Java-side function) stored into key-value pair. The method name, *createCalendarEvent*, will be used to refer the function at line 27 at Figure 7. To implement invocation at *hbc*, the Hermes opcode instruction for function invocation is used with the callee register. By matching callee register values to Module API and function names, potential Hermes-to-Dalvik invocations can be identified. The effectiveness of this analysis hinges on accurate pointer analysis of the callee registers in function invocations. However, Hermes' use of *First-class Objects* for all functions complicates static analysis of program behavior. This complexity is especially pronounced in sophisticated frameworks like React Native, and is further amplified when JavaScript bundlers are involved.

This module tackles key challenges in Dalvik bytecode analysis, with a focus on improving code coverage through automated detection of native-side functionality interfaces. By addressing Challenge 4 and Challenge 5, we lay the groundwork for more comprehensive analysis of Android applications. The module's approach necessitates pointer analysis on Hermes bytecode to enable effective cross-language analysis. While our current implementation provides a solid foundation for these tasks, we recognize the need for further advancements. In particular, our future research will concentrate on developing sophisticated inter-procedural analysis techniques for Hermes bytecode, aiming to deepen our understanding of complex application behaviors and interactions.

5 EVALUATION

In this section, we commence by undertaking a preliminary study to explore the extent of React Native's utilization. Subsequently, we delve into the following research questions to gauge the significance of our contributions:

- **RQ1:** What insights can be gained from profiling Hermes opcode usage in real-world apps? This research question presents the first comprehensive analysis of Hermes bytecode opcode usage in real-world applications. By examining actual opcode patterns, it aims to illuminate the intricacies of Hermes bytecode analysis, providing readers with a deeper understanding of the challenges inherent in static analysis of Hermes bytecode. This pioneering study not only sheds light on current complexities but also serves to inform and guide future research directions in the field of Hermes bytecode analysis.

- **RQ2:** To what extent does the Jimple code generated by *hermeser* maintain the original bytecode's control flow? This RQ introduces one benchmark to evaluate the control-flow sensitivity of the transformation from Hermes bytecode to Jimple. This RQ showcases the capability of the generated Jimple for intra-procedural analysis, highlighting its potential for various intra-procedural static analysis tasks.
- **RQ3:** How well does ReuNify enhance Soot-based static analysis on React Native Android Apps? This RQ investigates the extent to which ReuNify can facilitate static analysis for both JavaScript-side and Dalvik-side code in real-world React Native Android apps, focusing on achieving sound code coverage. This RQ aims to demonstrate ReuNify's ability to provide comprehensive analysis across the two sides of the React Native framework.
- **RQ4:** Can ReuNify reveal previously unreachable sensitive data leaks in React Native Android Apps? The purpose of this RQ is to assess ReuNify's ability to support downstream analysis, particularly taint analysis, when applied to real-world React Native Android applications. By investigating ReuNify's performance in practical scenarios, this RQ seeks to validate its extensibility and scalability, demonstrating its potential to enhance the effectiveness of static analysis techniques for React Native apps.

We ran all of our experiments on a Linux server with Intel (R) Core (TM) i9-9920X CPU @ 3.50GHz and 64 GB RAM.

5.1 Preliminary Study

We first conducted a preliminary study to explore the utilization of the React Native framework across a spectrum of Android apps, encompassing both popular and potentially malicious applications.

Dataset: To create a dataset of popular Android apps, we began by gathering a list of 15,854 Android apps from ANDROIDRANK [2]. This list included the top 500 apps for each of the 32 app categories available on Google Play. We then downloaded the latest version of 14,874 out of 15,854 of these apps from AndroZoo [44]. The remaining 980 apps were not available for download.

In addition, we obtained a dataset of 60,618 malware apps from VirusShare [33], which included Android malware apps collected by VirusShare in 2022. We also gathered 67,135 malicious apps from AndroZoo. We consider an app to be malicious if at least 10 antivirus engines in VirusTotal have flagged it.

Study Design: The React Native framework is developed using multiple programming languages, including Java, C++, JavaScript, Objective-C, and others [25]. The framework code is typically included in the release build to ensure proper app functionality. To gauge the extent of React Native framework adoption in Android apps, we conducted a preliminary study in which we examined the APK file of each app for the presence of the Java package, *com.facebook.react.* It is noteworthy that code obfuscation will not affect this package name [23]. In order to understand the usage scenario of the Hermes engine, it is necessary to identify the specific file that ends with the extension ".bundle" inside the resource directory of the unzipped Android Package ("*index.android.bundle*" is the default file name

Table 1. JavaScript-Code format in most popular apps and malware apps

Category	Hermes Bytecode	JavaScript	**Total**
Popular	494	574	1 068
Malware	28	413	441
Total	522	987	1 509

for React Native JavaScript-side code.). The *file* command [10] in the Linux operating system can be used to ascertain the specific file type. If the file is in Hermes bytecode, the program will display the file type and the version of the Hermes engine.

Results: As shown in Table 1, our empirical study indicates that **1,068 apps, accounting for 7.2% of those 14,874 most popular apps collected from AndroZoo**, were developed using the React Native framework. Of these React Native Android apps, 494 (46.3%) utilized the Hermes engine as the JavaScript runtime and compiled the JavaScript into Hermes bytecode. In contrast, among the 60,618 malware collected from VirusShare, **there were 441 apps developed with the React Native framework**. Out of these 441 React Native Android malware apps, only 28 of them used the Hermes engine.

> Within the selection of the 14,874 most popular Android applications, approximately 7.2% have been created using the React Native framework. The presence of malware has extended to encompass React Native applications as well. Furthermore, the employment of the Hermes engine exhibits lower frequency among malware apps in comparison to its prevalence within popular applications.

5.2 RQ1: What insights can be gained from profiling Hermes opcode usage in real-world apps?

This research question examines the usage patterns of Hermes opcodes in real-world React Native apps, initiating the first study of its low-level behavior. By analyzing opcode frequency and patterns across a range of applications, including popular apps and malware, the study aims to reveal common programming patterns and features. It addresses the gap between theoretical definition and practical implementation of Hermes opcode, with a focus on improving static analysis techniques. The findings will guide the development of more efficient and accurate static analyzers and establish a foundation for understanding how React Native code works in low-level operations. This investigation sets the stage for future advancements in Hermes bytecode feature modeling, providing valuable insights for both developers and researchers in the field.

Experimental setup: In order to evaluate the use cases of opcodes in real-world apps, we collect the opcodes that appeared in both the most popular apps and malware. Table 2 displays opcode types that have a frequency exceeding 1% for both popular apps and malware. Specifically, we decompile those 1,068 most popular React Native Android apps, as well as 441 React Native malware instances, where we further locate the bundle files. The bundle file containing JavaScript source code or Hermes bytecode can be converted into textual disassembly, allowing for the extraction of Opcode instructions.

Findings: JavaScript-side bundle files were successfully identified in 337 out of 441 malware apps and 956 out of 1,068 of the most popular apps. Nevertheless, because of the customized naming strategy and dynamic distribution of bundle files, 104 instances of malware and 112 popular apps were unable to locate their JavaScript-side code through the "*.bundle*" file. In the future, conducting a more comprehensive analysis of the React Native app build pipeline will be essential to understand the potential effects of employing a versatile technique on loading JavaScript-side content into React Native's runtime.

The average number of opcode instructions gathered from the most popular apps (533,740) is roughly ten times greater than that observed in malware cases (51,864). The observation that the number of JavaScript-side codes in malware cases is often smaller than in popular applications might be attributed to the fact that successful apps tend to provide a comprehensive set of functionalities and features to accommodate a diversified user population. The incorporation of these functionalities

Table 2. Distribution of Hermes opcodes for opcodes occurring in more than 1% of both popular apps and malware.

Popular Apps		Malware Apps	
Opcode	**Frequency**	**Opcode Type**	**Frequency**
GetByIdShort	8.98%	MovLong	8.85%
LoadFromEnvironment	7.15%	LoadFromEnvironment	7.31%
Call2	5.84%	Mov	7.28%
GetById	5.14%	GetByIdShort	6.77%
Mov	3.89%	StoreToEnvironment	4.68%
LoadParam	3.55%	LoadParam	4.49%
Ret	3.50%	CreateClosure	4.21%
PutNewOwnByIdShort	3.46%	Call2	3.90%
StoreToEnvironment	3.30%	GetById	3.80%
GetByVal	3.08%	TryGetById	3.48%
CreateClosure	2.84%	Call	3.22%
LoadConstUndefined	2.77%	Ret	2.96%
PutById	2.68%	LoadConstUndefined	2.57%
LoadConstString	2.68%	LoadConstUInt8	2.38%
NewObject	2.61%	GetByVal	2.33%
GetEnvironment	2.50%	LoadConstString	2.02%
LoadConstUInt8	2.40%	PutById	2.00%
PutOwnByIndex	2.27%	LoadConstInt	1.99%
PutNewOwnById	2.14%	GetEnvironment	1.95%
MovLong	2.12%	PutNewOwnByIdShort	1.93%
TryGetById	2.10%	NewObject	1.51%
Call3	1.90%	Call3	1.25%
JmpFalse	1.24%	PutNewOwnById	1.24%
Call4	1.23%	NewArrayWithBuffer	1.21%
JmpTrue	1.21%	CreateEnvironment	1.15%
Total Above	80.57%	**Total** Above	84.45%
Others	19.43%	**Others**	15.55%

necessitates a greater amount of code implementation, resulting in an expanded codebase. There are a total of 207 opcodes present in 39 distinct versions of Hermes engines, but our examination of these Hermes opcodes in real-world applications indicates that there are 169 opcodes in popular apps and 142 opcodes in instances of malware. We further compiled a list of opcode types in Table 2, including those with a frequency exceeding 1%. Among these, the top 25 opcodes with a frequency exceeding 1% collectively account for 80.57% of opcodes in popular apps and 84.45% of opcodes in malware instances. In the following, we discuss the usage of these opcodes in real-world apps.

As shown in Table 2, Hermes bytecode relies heavily on opcodes for manipulating heap objects, such as loading values (*GetByIdShort*, *GetById*, *GetByVal*, *TryGetById*), assigning values (*PutNewOwnByIdShort*, *PutById*, *PutNewOwnById*), and initializing new objects (*NewObject*). In an analysis of popular apps, these opcodes accounted for 27.51% of all opcode occurrences, while in malware apps, they made up 23.06%. The high frequency of opcodes related to loading, assigning, and initializing heap objects, which account for around one-quarter of all Hermes opcode usage in both popular apps and malware apps, underscores the critical role that heap object manipulation plays in the Hermes bytecode program. In addition to the heap object, there are instruction opcodes, such as PutOwnByIndex and NewArrayWithBuffer, that are used to manipulate the content of arrays. The widespread use of opcodes for manipulating objects or arrays highlights the significance of developing abstraction techniques for these manipulations to enhance downstream analyses and optimizations on the Hermes bytecode program.

The opcode *LoadFromEnvironment* ranks as the second most frequently employed opcode in both popular apps and malware. Its primary function is to retrieve values from the closure environment, a process akin to fetching values from the lexical scope in JavaScript. The Hermes engine provides four base opcodes for handling value manipulation within a specific environment, which include *LoadFromEnvironment*, *StoreToEnvironment*, *GetEnvironment*, and *CreateEnvironment*. These four opcodes of environment-value manipulation play an important role within the Hermes opcode, accounting for 13.88% and 15.43% of all opcode occurrences in popular apps and malware, respectively. There are five additional opcodes specifically designed for addressing case scenarios, such as the retrieval or storage of non-pointer values within an environment, which collectively make up less than 1% of the total opcode occurrences. As the prevalent usage for closure environment object manipulation, particular techniques are needed to abstract the manipulation on the Cloure Environment Object to facilitate reliable analysis.

Table 2 shows that Hermes bytecode includes opcodes for calling functions with different numbers of arguments: *Call2*, *Call3*, *Call4*, and *Call*. The numeric suffix denotes the number of arguments passed per invocation, except for Call, where the last argument specifies the number of arguments. These invocation opcodes represent 8.97% of opcode usage in popular apps and 8.37% in malware, with *Call2* being the most frequent. Hermes bytecode invocations have one additional argument compared to their JavaScript counterparts, suggesting that most real-world app function invocations use a single argument. All invocations across Hermes bytecode call the function stored in registers. The first-class nature of functions in Hermes bytecode enables the use of indirect function calls. Within Hermes bytecode, each of its call site statements occurs via indirect function calls, posing significant challenges for interprocedural analysis because the targeted function is determined only during runtime. It results in sophisticated analysis techniques to be used to accurately model and reason about the behavior of the program in the use of indirect functions.

The opcodes "*Mov*" and "*MovLong*" were the first and third frequently used opcodes in malware apps, accounting for 16.13% of all opcodes. In contrast, these opcodes were much less prevalent in popular apps, where they constituted only 6.01% of the total opcodes. They are used for variable assignments and data transfers within a computer's memory or registers. The usage of these opcodes could potentially indicate the use of *Runtime Polymorphism* [93], a technique that enables code to dynamically adapt and change its behavior during execution, making it more difficult to detect and analyze. It adds further complexity by demanding the analysis of polymorphism.

When comparing the frequency of opcode types between popular apps and malware instances, a noticeable trend emerges. The varying frequencies of opcode usage often reflect distinct code combinations that are closely tied to the app's business logic, which can be viewed as the smells of a particular category of apps. Although the difference in opcode frequencies between popular apps and malware can be considered a potential character of malware, it should not be used in isolation. Instead, it is crucial to combine this feature with other indicators to effectively identify malicious applications. To further advance the understanding of malware characteristics, a comprehensive and systematic study is needed. Such research would greatly benefit the cybersecurity community by providing valuable insights into the telltale signs, or "*smells*," of malware, enabling more accurate detection and prevention strategies.

Techniques like points-to analysis, shape analysis, and abstract interpretation can create accurate and efficient object abstractions, which can be used by static analysis tools and optimizers to reason about program properties, detect issues, and apply targeted optimizations. This is particularly important for the Hermes bytecode program, given its prevalent use of opcodes for object and array manipulation. While differing opcode frequencies between popular apps and malware can be a distinguishing feature for identifying malicious code, it should be used in combination with other relevant features. A comprehensive and systematic analysis of React Native malware features is crucial for future research, providing insights into the unique characteristics and behaviors of malicious code within the React Native apps, ultimately contributing to the development of more effective malware detection techniques.

> **Answer to RQ1:** Hermes bytecode comprises 207 opcodes, with 169 used in popular apps and 142 in malware. Frequent use of opcodes for object, array, and closure manipulation, along with indirect function calls, necessitates pointer analysis. Opcode usage frequency can prioritize technique development. Differences in opcode frequencies between popular apps and malware suggest variations in their business logic.

5.3 RQ2: To what extent does the Jimple code generated by hermeser maintain the original bytecode's control flow?

⊛ = true, × = not generated, ○ = false

Statement.	JS_No.	HBC	Jimple	JS_No.	HBC	Jimple	JS_No.	HBC	Jimple	JS_No.	HBC	Jimple
						Control flow						
return	00	•	⊛	01	•	⊛	02	•	⊛			
break	00	•	⊛	01	•	⊛	02	•	⊛	03	•	⊛
continue	00	•	⊛	01	•	⊛	02	•	⊛			
throw	00	•	⊛	01	•	⊛	02	•	⊛	03	×	
if...else	00	•	⊛	01	•	⊛	02	•	⊛	03	•	⊛
switch	00	•	⊛	01	•	⊛	02	•	⊛	03	•	⊛
	04	•	⊛	05	•	⊛	06	•	⊛	07	•	⊛
try...catch	00	•	⊛	01	•	⊛	02	•	⊛	03	•	⊛
	04	•	⊛	05	•	⊛	06	•	⊛	07	•	⊛
						Iterations						
do...while	00	•	⊛	01	•	⊛	02	•	⊛			
for	00	•	⊛	01	•	⊛	02	•	⊛	03	•	⊛
	04	•	⊛	05	•	⊛	06	•	⊛	07	•	⊛
	08	•	⊛	09	•	⊛	10	•	⊛	11	•	⊛
	12	•	⊛	13	•	⊛	14	•	⊛	15	•	⊛
for...in	00	•	⊛	01	•	⊛	02	•	⊛	03	•	⊛
	04	•	⊛	05	•	⊛	06	•	⊛	07	•	⊛
	08	•	⊛									
for...of	00	•	⊛	01	•	⊛	02	•	⊛	03	•	⊛
	04	•	⊛	05	•	⊛	06	•	⊛	07	•	⊛
	08	•	⊛	09	•	⊛	10	•	⊛	11	•	⊛
	12	•	⊛	13	•	⊛						
for await...of	00	×		01	×		02	×		03	×	
	04	×		05	×		06	×				
while	00	•	⊛	01	•	⊛	02	•	⊛	03	•	⊛
	04	•	⊛	06	•	⊛	05	•	⊛			
						Complex						
Control flow with Iterations	00	•	⊛	01	•	⊛	02	•	⊛	03	•	⊛
	04	•	⊛	05	•	⊛	06	•	⊛	07	•	⊛

Table 3. Results of control flow sensitivity evaluation on HermesControlFlow bench.

Our primary aim is to verify the control-flow sensitivity of the generated Jimple code, rather than validate the full semantic content of the original source. This approach aligns with Intermediate

Representations (IRs) in static analysis, which preserve essential semantics to facilitate easier automatic analysis instead of directly exposing the full semantic meaning [58, 62, 111]. As detailed in Section 4.1, our proposed abstraction specifically transforms control flow-related opcodes, while preserving other opcode names and parameters. This method allows us to grasp the execution's control flow and enables future abstraction of other opcodes for specific analysis needs. IRs balance critical program information with a simplified, standardized format for analysis and optimization. While semantic preservation can vary based on IR format and analysis goals, IRs maintain the core logic and structure necessary for meaningful analysis. We provide a more detailed discussion of our IR generation in Section 6.1. By verifying control-flow sensitivity, we establish a foundation for future work involving control flow graphs, Single Static Assignment, and downstream techniques such as intra-procedural taint analysis.

Benchmark construction: The benchmark involves 97 code samples involving both base cases originating from JavaScript statements and complex cases. We first collected a set of 89 JavaScript code examples, which were developed based on 13 essential JavaScript statements. These statements were sourced from the Mozilla Developer Network (MDN) documentation on statements and declarations in the JavaScript language. [40] In the JavaScript documentation on the MDN, statements and declarations are grouped into five primary categories: *control flow*, *declaring variables*, *functions and classes*, *iterations*, and *others*. For the purpose of this benchmark, we particularly focus on the statements that impact the intraprocedural analysis breaking the top-to-bottom execution sequence. We specifically collected 89 code examples from the "*Try it*" and "*Examples*" sub-sections for each of 13 statements belonging to the *control flow* and *iterations* categories. The "*Try it*" section in each statement's documentation is designed to provide an interactive code execution environment, helping readers grasp the functionality of the corresponding statements. The "*Examples*" section, on the other hand, emphasizes the diverse applications and use cases of each statement. This benchmark includes JavaScript code, the corresponding Hermes bytecode, and the generated Jimple code. The textual disassembly is also provided to aid in comprehending the transformation process from JavaScript to Hermes bytecode and subsequently to Jimple. It is important to note that the code examples for the 13 essential JavaScript statements often include a combination of multiple statement types, rather than solely focusing on the specific statement itself. Furthermore, to enhance the evaluation of complex intra-procedural behavior, we included an additional set of 8 code samples that specifically target scenarios involving a combination of control flow and iteration statements. These complex scenarios enable a more thorough examination of how our approach handles intricate intra-procedural patterns that arise from the interplay between different statement types.

Findings: Our experiments involved a total of 97 code samples, and the results are summarized in TableTable 3. In instances where a red cross (×) is indicated, the Hermes engine encountered difficulties in generating the corresponding Hermes bytecode. Consequently, Hermeser was unable to produce the equivalent Jimple code for these cases. Upon closer inspection, it was discovered that all these code samples (7 samples of *for await...of* statement and 1 sample of *throw* statement) contained the "*await*" statement. It is crucial to note that the Hermes engine lacks built-in support for the "*await*" statement. To retain the functionality of "*await*", developers must preprocess their code using a tool like Babel before compiling with the Hermes engine [39]. However, the code transformed by Babel would include a substantial amount of older JavaScript versions of code to replicate the functionality of the "*await*" statement. For instance, code sample 00 in the for await...of statement consists of 28 nodes in its Abstract Syntax Tree (AST), whereas the code generated by Babel contains 3,240 nodes for the corresponding AST, which is excessively large for manual evaluation. The automatic evaluation of generated Jimple code, particularly for large programs, is left as future work.

Table 4. Average number of Volume of code

Category	# apps	Soot without ReuNify		Soot with ReuNify		Difference	
		# Methods	# LOC	# Methods	# LOC	# Added Methods	# Added LOC
Popular	494	132 093	1 697 294	162 195	2 879 754	30 102 (+22.79%)	1 182 460 (+69.67%)
RN Toy App	1	43 005	476 859	47 209	632 880	4 204(+9.78%)	156 021 (+32.72%)

For all other test cases, the generated Jimple code successfully maintains the same level of control flow sensitivity as its original JavaScript counterparts. While the generated Jimple code demonstrates 100% accuracy on the given Hermes bytecode, proving that this accuracy holds true across all real-world scenarios is a challenging task. Nevertheless, based on the evaluated samples, we assert that the Hermeser tool is effective in handling control-flow analysis. To further validate the control flow semantic meaning of the generated Jimple code, future work could explore the application of JavaScript compiler fuzzing techniques. However, designing an automatic verification process to assess control flow sensitivity at the Intermediate Language level is a non-trivial undertaking.

Answer to RQ2: To evaluate the control flow sensitivity of the Jimple code produced by *hermeser*, we developed the *HermesControlFlowBench* benchmark, which includes all essential JavaScript statements that disrupt the top-to-bottom execution order. The results demonstrate that the Jimple code generated by *hermeser* maintains the same level of control flow sensitivity as the original code.

5.4 RQ3: How well does ReuNify enhance Soot-based static analysis on React Native Android Apps?

Our objective with this RQ is to understand how ReuNify enhances the static analysis on React Native Android Apps in both JavaScript-side code and Dalvik-side code.

Experimental setup: We evaluate ReuNify on those 494 Hermes engine-enabled apps out of the 1,068 most popular React Native Android Apps from two perspectives: ❶ the number of generated Jimple Code, ❷ the number of identified Dalvik-to-Hermes invocation. Since the implementation of Hermes engine impacts the volume of code implemented in React Native framework in Android apps [11], we focused our analysis on popular apps that adopted the Hermes engine. This approach was selected to provide a fair and unbiased comparison. Moreover, given that the Hermes engine has been widely adopted as the primary engine for React Native, our research results provide significant contributions to the existing knowledge on the present status of React Native Android applications using the Hermes engine.

To further assess the practicality of ReuNify, we utilized FlowDroid to generate callgraphs for 1,068 popular React Native apps and 441 React Native malware apps, and compared the ❸ size of the callgraphs before and after integrating ReuNify. Because there are extra JavaScript-side code bases counted with ReuNify, including the JavaScript-side callgraph with ReuNify in the comparison would be unfair to the callgraph without ReuNify. A wide range of React Native applications (containing both popular apps and malicious apps) are involved to further demonstrate ReuNify's efficacy. A diverse set (including both popular apps and malware apps) of React Native apps can further prove the effectiveness of ReuNify.

Volumn of Jimple Code: The quantity and quality of static analysis results produced by Soot's framework are heavily reliant on the availability of Jimple code. With ReuNify's *hermeser* integrated into Soot framework, an additional *class* for Hermes bytecode is created. This class comprises an average of 30,102 SootMethods with 1,182,460 lines of Jimple code. According to Table 4, **with the**

augmentation of ReuNify's *hermeser*, there are 70% more Jimple statements generated compared with 1,697,294 lines of code generated by Soot.

ReuNify successfully generated the additional Jimple statement for 452 out of 494 apps. The unsuccessful cases were due to the customizable nature of the bundle name [4], which made it difficult to locate the JavaScript-side bundle file. To improve the reliability of the analysis, future work should focus on developing more robust techniques for locating bundle files. For the apps with located bundle files, all of them were successfully transformed into Jimple code from Hermes bytecode. Moreover, all *SootMethods* generated by ReuNify's *hermeser* passed Soot's body validation (*<soot.jimple.JimpleBody: void validate()>* [17] in Soot), indicating that the generated Jimple code is valid in the Soot framework. This allows for additional Soot-based analysis on Hermes bytecode.

Table 5. Average number of Native Module API and Native Component UI

Category	Apps	Native			
		Module API	Module API Methods	Component UI	Component UI Methods
Popular	494	92	532	55	489
React Native Toy App	1	51	213	22	365

Number of Hermes-to-Dalvik invocation: React Native enables accessing methods on the Java side from the JavaScript side. As shown in Table 5, React Native apps have an average of 93 Native Module APIs, which contain 569 methods accessible to JavaScript code, and 52 Native React Components comprising 477 methods for setting UI attributes. As shown in Figure 1, the Native Module APIs and Components can be sourced from the React Native framework, third-party libraries, or the developer's own implementation. To determine the extent of Hermes-to-Dalvik invocations coming from sources beyond the Core Module APIs and Core Components, we build a Toy app from the project (React Native CLI Quickstart [28]) in React Native version 0.71. This Toy app only includes the Core Module APIs and Core Components without any developer's code or third-party library. According to Table 5, **the most popular React Native Android apps have over twice the number of Native Module API methods (532 methods) compared to the React Native Toy app (213 methods)** using React Native version 0.71. With the use of Native Module API and Native Components, more powerful functionalities (in terms of performance and access to system resources) can be exposed to the JavaScript side. It is customary to involve extra Native Module APIs and Native Components while developing a React Native Android app.

Size of Callgraph: In static analysis models, callgraph is a crucial component as it offers a complete perspective of the program's behaviour. To evaluate the effectiveness of ReuNify in generating callgraphs, we compared the size of callgraphs produced by FlowDroid with and without the augmentation of ReuNify, for both popular and malicious React Native Android Apps. Out of the 1,068 most popular React Native apps and 441 React Native malware apps, callgraphs get generated successfully on 1,007 and 421 apps respectively, with or without the use of REUNIFY. Nonetheless, in some cases, due to time limitations or obfuscation techniques, Callgraph failed to be generated on 61 popular apps and 20 malware apps.

Table 6. Average numbers of nodes and edges before and after ReuNify on 1,007 most popular apps and 421 malware apps

Category	# apps	without ReuNify		with ReuNify		Difference	
		# Nodes	# Edges	# Nodes	# Edges	# Added Nodes	# Added Edges
Popular Apps	1 007	9 206	70 344	16 940	102 830	7 734 (+84.01%)	32 486 (+46.18%)
Malware Apps	421	6 465	36 572	9 824	48 460	3 359 (+51.96%)	11 888 (+32.51%)

We first report the average number of nodes (i.e., the number of methods) and edges (i.e., the number of potential invocations) in the callgraphs obtained before and after having applied ReuNify.

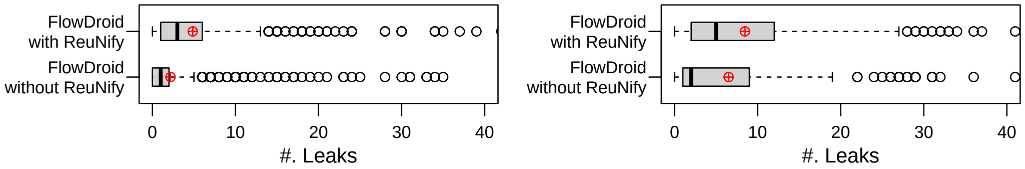
The call-graph augmentations introduced by ReuNify can be seen in Table 6, where the number of apps affected by the changes is represented by the # apps column. We observe that all apps' callgraphs are enlarged by the use of ReuNify (1,007 and 421 for popular and malware apps, respectively). Additionally, we notice that the number of nodes and edges uncovered with ReuNify is higher for popular apps than for malware apps: 7,734 vs 3,359 on average per app for nodes and 32,486 vs 11,888 for edges. This highlights that **traditional static analyzers that do not consider the executable code in React Native apps miss a substantial number of nodes and edges** in their call graphs.

By considering the mechanism of React Native, ReuNify can identify previously unreachable Java methods that are now reachable. The number of such previously unreachable methods is highly correlated with the number of Hermes-to-Dalvik invocations. The discovery of newly reachable nodes is significant because it allows static analyzers to avoid treating them as dead code.

> **Answer to RQ3:** Soot tends to miss a significant portion of executable code when analyzing React Native Android apps. However, by converting Hermes bytecode to Jimple, there is a 70% increase in the number of lines of Jimple code in Soot. Taking into account the React Native mechanism on the Java side, popular apps experience an increase of approximately 84% in new nodes for callgraph, while malware apps experience an increase of around 52% in nodes for callgraph.

5.5 RQ4: How effective is ReuNify in finding sensitive data leaks in React Native Android Apps?

In this research question, we demonstrate the capability of ReuNify in finding potential privacy leaks in real-world React Native Android apps.



(a) Popular Apps

(b) Malware Apps

Fig. 8. Distribution of the number of leaks detected by FlowDroid with and without ReuNify

Experimental setup: In order to evaluate the effectiveness of ReuNify in finding privacy leaks, we conducted experiments on both popular apps and malware to demonstrate its effectiveness. Specifically, we tested ReuNify on 1,068 of the most popular React Native Android apps, as well as 441 React Native malware instances detected in the year 2022 and sourced from VirusShare[33]. In order to ensure a fair comparison, we utilized the default sources and sinks provided by FlowDroid. However, it should be noted that ReuNify supports custom sources and sinks tailored to specific needs and interests, such as those pertaining to JavaScript. Sources and sinks in the context of privacy leaks refer to the entry and exit points in an app's code where data can enter and leave the system. FlowDroid is capable of identifying data flows from sensitive sources to potentially unsafe sinks. It is important to keep in mind that dataflow analysis can be both time and memory intensive, and therefore, for each app, we set a maximum time limit of 30 minutes for FlowDroid to complete its analysis.

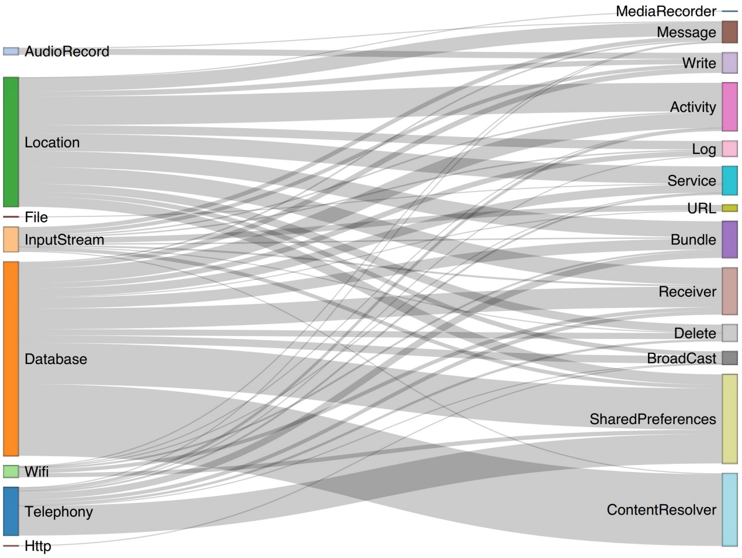


Fig. 9. Sankey diagram of all newly detected privacy leaks.

Findings: FlowDroid was executed successfully on 1,007 out of the 1,068 most popular apps and on 421 out of the 441 malware apps, with or without ReuNify augmentation. However, due to time constraints or obfuscation techniques, FlowDroid failed to run on 61 of the most popular apps and 20 of the malware apps. **In total, applying ReuNify resulted in the detection of 2,690 (4,892 – 2,202) additional privacy leaks for popular apps and 827 (3,576 – 2,749) additional privacy leaks for malware apps, respectively.** The average number of leaks is indicated by the ⊕ labels in Figure 8(a) and Figure 8(b), respectively. On average, Figure 8(a) indicates that by incorporating ReuNify, an extra 2 privacy leaks (totaling 4 leaks) were identified in popular apps compared to only running FlowDroid, which could detect only 2 leaks. Similarly, as shown in Figure 8(b), with the augmentation of ReuNify, an additional 2 privacy leaks were detected on average, making a total of 8 leaks, compared to only running FlowDroid (i.e., 6 leaks) for those malware apps. It is not surprising that more leaks are detected from malware apps than popular benign apps, as the number of leaks is highly reflective of potential issues in an app.

Types of newly detected privacy leaks: After identifying privacy leaks additionally discovered by ReuNify, we further categorize the sources and sinks according to SuSi's classification [31] to facilitate understanding of each privacy leak. For any sources or sinks that were not classified, we manually assigned categories based on the functionality of their classes and methods. Among them, the most common sink type was the *Replace* sink, represented by the method *<java.lang.String: java.lang.String replace(java.lang.CharSequence,java.lang.CharSequence)>*. The method, *replace*, is frequently used to substitute a particular sequence of characters in a string with another sequence of characters. However, if sensitive data (e.g., user credentials) is included in either the original or replacement character sequences, this information can be inadvertently leaked. We found that for both popular and malware apps, the most common type of leaked information was data stored in the database. The second most common type of leaked information for popular apps was Wi-Fi-related information including Service Set Identifier (SSID) and MacAddress. For malware apps, the second most common leaked type of source information was telephony information, including Device Id, Line1Number (phone number of the device's SIM card), subscriber ID, and SimSerialNumber. For both popular and malware apps, more than 98% of sources that leaked from the method, *replace*, come from the top two most common sources as described above.

To better comprehend and visualize additional privacy leaks discovered by ReuNify, we have created a Sankey diagram (Figure 9) that includes newly detected leaks for both popular and malware apps while excluding the leaks with *replace* as sinks. It can be observed from Figure 9 that the primary sources of privacy leaks are Database, Location, and Telephony. The sensitive information is predominantly leaked to SharedPreferences, ContentResolver, and Activity. In fact, our analysis shows that the use of ReuNify resulted in a significant increase in the number of detected sensitive data leaks for both popular Android apps and malware.

Answer to RQ4: ReuNify is effective for identifying data leaks that were previously unseen. Specifically, on average, 2 additional potential leaks can be detected in both popular apps and malware.

6 DISCUSSIONS

6.1 Limitations

Our approach is a step towards realizing the ambition of full code unification for Android static analysis. Our current prototype of ReuNify, despite promising performances, presents a few limitations:

Firstly, ReuNify's implementation depends on existing tools to extract call graphs for native code and identify mutual invocations between Hermes bytecode and native code. Consequently, the limitations of these tools are inherited by ReuNify. These limitations include the difficulty in accurately determining the boundaries of native functions and the unsoundness in app modeling with FlowDroid caused by reflective calls, multi-threading, and dynamic loading.

Secondly, our prototype currently relies on the suffix of the default name of the JavaScript side code, which is non-scalable in some cases. Therefore, the JavaScript side code of some React Native apps was not located due to the unknown customized loading strategy adopted by the app developers. In the future, conducting a more comprehensive analysis of the React Native app build pipeline will be essential to understand the potential effects of employing a versatile technique on loading JavaScript-side content into React Native's runtime.

Thirdly, ReuNify does not support analysis for C++ code in React Native Android apps. Both *bridge* and *JavaScriptInterface* modules in old and new React Native frameworks serve for cross-language communication, which necessitates the implementation of C++ code to achieve the encapsulation of the platform-specific native code. This would impact the thoroughness of our analysis of code for React Native applications. However, the usage of JSI and C++ is still experimental, and the implementation of C++ is being gradually automated by the Codegen module [8] in the React Native framework. Since C++ code is not as prevalent as Java code at present, ReuNify plans to gradually incorporate support for C++ code in the future.

Fourthly, a significant limitation arises from the dynamic and complex language features of Hermes bytecode, which limits the effectiveness of advanced interprocedural analyses, such as points-to analysis, on the generated Jimple code. Our findings from RQ1 revealed that each Hermes bytecode opcode carries unique semantic meaning. Fully capturing and transforming these opcode semantics at the IR level is a complex undertaking beyond our current scope, as IR is primarily designed to facilitate easier automatic analysis rather than directly expose full semantic meaning [58, 62, 111]. In this study, ReuNify focused on interpreting Hermes opcodes related to conditional branching and function declaration during transformation, and the control-flow sensitivity was verified in RQ2. This method allowed us to apply advanced static analysis techniques, such as

Single Static Assignment (SSA), Control Flow Graph (CFG), and intra-procedural taint analysis to the generated Jimple code. These techniques facilitate tracking information flow and observing the program behavior. Other Hermes opcodes were transformed into Jimple statements, preserving their original names and parameters. This serves as a foundation for future research to further explore their syntax and semantics within the Jimple representation. Future work aims to enhance the abstract interpretation of Hermes opcodes into Jimple, facilitating more sophisticated static analysis techniques and points-to analysis.

6.2 Threats to Validity

Evaluation of Intra-procedural analysis on Hermes bytecode. To verify the correctness of the results, we perform a manual verification of the semantic meaning of the branching transformation. To achieve this, we design the second research question (RQ2) in Section 5.3 by introducing a benchmark comprising code snippets of branching-related JavaScript statements sourced from MDN documentation and curated complex cases. While Hermeser demonstrates complete accuracy across the benchmarks, guaranteeing 100% control flow sensitivity in all scenarios remains a challenge due to the countless possible combinations of control flow statements. Future studies could investigate the use of JavaScript compiler fuzzing techniques to assess the control flow semantic meaning of the generated Jimple code [63]. However, developing an automated verification process for control flow sensitivity at the Intermediate Language level is a complex and non-trivial task [117].

Manual Checking. To verify the use of the native side functionality, we manually checked fifty Android React Native apps. For Dalvik-to-Hermes links, as the symbols were always available for the apps we checked (since native methods were pragmatically registered), we were able to confirm the correctness of those links in the callgraph generated by ReuNify. We reverse-engineered these apps through Java bytecode decompilers (Jadx [15]) and were able to reach the same conclusions. Regarding Hermes-to-Dalvik links, the method names are represented as strings, which are not directly available in the native code. Therefore, we faced a challenge to check if the symbolic execution yielded correct links. However, we present a solution that utilizes a control-flow-insensitive technique to infer the type of the register value, which can identify some invocations for the Java-side code and recover build-in API methods (e.g., *console.log()*, *alert()*, *JSON.parse()*, etc.). Nonetheless, it remains a challenge to verify if the Hermes-to-Dalvik identification has yielded correct links. One possible way to verify this would be to execute the code section to trigger the native code and ensure that the correct information is yielded by Hermes-to-Dalvik identification. However, this is beyond the scope of this study. Therefore, we have made the hypothesis that the correct results are yielded from Hermes-to-Dalvik identification.

6.3 Future Work

Our research carries several implications and offers recommendations for future direction on React Native app analysis.

Support Whole Program Analysis. Expanding the analysis to support C++ code analysis for React Native applications is crucial for providing comprehensive and in-depth analysis capabilities. By incorporating C++ code analysis, the framework can offer a holistic view of the entire React Native application, considering the interactions and dependencies between JavaScript, Dalvik bytecode, and C++ components. This expansion allows for the detection of cross-language vulnerabilities and compatibility issues arising from the interplay between these layers. Whole-program analysis empowers developers to identify and mitigate security risks, optimize performance, and ensure the overall stability and reliability of the React Native application across all its components. This comprehensive approach to React Native application analysis ultimately leads to high-quality, secure, and performant applications while minimizing the risk of vulnerabilities and performance issues arising from the interaction between JavaScript, Dalvik bytecode, and C++ code.

Static Analysis on Hermes bytecode. The successful application of static analysis techniques to Dalvik bytecode in native Android apps highlights the importance of analyzing Hermes bytecode in React Native apps, despite the significant challenges it presents. Binary code analysis is inherently difficult due to the complexity of representing compiled code in a format suitable for thorough analysis [72, 91], and it introduces additional complexities stemming from optimization techniques applied during compilation. Hermes bytecode, being a variant within the Hermes JavaScript virtual machine, naturally exhibits many well-known challenges associated with JavaScript source code analysis, further compounded by the optimization techniques employed by the Hermes engine during compilation. However, the software engineering and programming language research communities' extensive investigation of JavaScript code analysis can serve as a valuable foundation for approaching Hermes bytecode analysis, with techniques specifically designed for analyzing JavaScript features being adapted and applied to inform the development of Hermes bytecode analysis methods. Additionally, the adoption of Jimple as an intermediate representation in other popular static analysis frameworks, such as Doop [53], Tai-e [109], Qilin [64], and SootUp [30], could extend their advantages to Hermes bytecode analysis.

Support React Native iOS application analysis. Expanding an analysis framework to include React Native iOS application analysis is essential for achieving Expanding an existing analysis framework to support React Native iOS application analysis is a crucial step towards providing comprehensive and platform-agnostic analysis capabilities. React Native's use of JavaScript and the same code format across iOS and Android apps motivates the design of platform-agnostic analysis for the JavaScript-side code. By extending the framework to encompass iOS-specific input, developers/analyzers can conduct analysis on React Native applications seamlessly across both platforms, identifying and mitigating platform-specific vulnerabilities, performance issues, and compatibility concerns. Integrating iOS analysis support streamlines the development workflow, reduces duplication of efforts, and promotes code reuse and maintainability, enabling organizations to efficiently detect and resolve issues, ensure adherence to best practices, and maintain a high standard of quality across their React Native applications on both iOS and Android.

Downstream analysis for React Native programs. Static analysis of Dalvik bytecode, designed for downstream analysis within Android apps, has proven effective in improving app quality by identifying various issues, such as privacy concerns [49], permission misuse [52], energy consumption problems [76], and compatibility issues [80]. With React Native's growing popularity, downstream static analysis has become crucial for ensuring the quality, reliability, security, and performance of React Native apps. The abstraction layer between JavaScript and native components introduces unique challenges and vulnerabilities that traditional static analysis techniques may not easily detect. Downstream analysis examines the compiled native code and its interactions with the platform, enabling analyzers/developers to identify security risks, performance bottlenecks, and compatibility issues specific to React Native applications. Analyzing Hermes bytecode could further include advanced Soot-based static analyses for logic bomb analysis [42] for the detection of security vulnerabilities[41], compatibility issues [80], and fault localization techniques [98]. However, addressing interprocedural analysis challenges related to heap objects and sophisticated pointer analysis techniques for Hermes bytecode remains a key area for exploration, which can also enhance optimization strategies within the Hermes engine.

7 RELATED WORK

Analysis of Multiple Languages in Android App. The research emphasis has been on analyzing languages used in Android Apps beyond just Java, and also on conducting cross-language analysis.

Lee et al. [74] analysed the inter-communication between Android Java and JavaScript and presented the framework, HybriDroid, to detect bugs and information leaks in hybrid apps. However, HybriDroid is Android version sensitive and only focuses on the bridge communication between Android Java and JavaScript (the other communication approach is callback communication). Alam et al. [41], in 2016, proposed DroidNative, which can perform Android malware detection considering both the bytecode and the native code. What's more, NDroid [99], TaintArt [106], and PolyCruise [81] were proposed for dynamic taint analysis so as to track sensitive information flows. JN-SAF and Jucify [101, 113] are also proposed as an inter-language static analysis framework to detect sensitive data leaks in Android apps. The Jimple statements produced by *Jucify* are insufficient and unable to capture the complete implementations of the native functions, which poses a challenge in commencing further research (inter-procedural analysis) on the native code for whole program analysis. All the aforementioned tools, however, are task-specific. They also, typically, perform their analyses separately for bytecode and native code, and later merge the outputs to present unified analysis results. In contrast, ReuNify is proposed to unify the representation before task-oriented analyses, which empowers popular analysis pipelines to be directly adopted on the output of ReuNify.

Android Dalvik Bytecode Analysis. Dalvik bytecode is compiled from either Java source code or Kotlin source code. In the past decade, static analysis of Android apps mostly targeted on those Dalvik bytecode. Li et al [79] provide a comprehensive survey of Android apps, focusing on static analysis approaches. Different static analysis approaches are utilized to detect compatibility issues [80, 88, 107, 108, 115] and other functional or non-functional faults [59, 66, 78, 82, 86, 114, 116] across different devices [84, 85]. Moreover, static analysis can be leveraged to collect information in apps towards improving dynamic testing approaches [70, 90, 105, 118], and contribute to learning-based approach [56, 65, 87, 88, 119]. The popular artifacts adopted by current researchers are MalloDroid by Fahl et al. [60], which detects improper use of transport layer security in apps; FlowDroid by Arzt et al. [49], which is able to find privacy leaks by inspecting illicit information flow; and IccTA by Li et al. [77], which extends FlowDroid by accounting for inter-component privacy leaks. Instead of focusing on Java-based Android apps analysis, our work has taken a step forward by proposing an approach to take an additional programming language, Hermes bytecode/JavaScript (used in React Native Android Apps), into consideration. We expect to provide the community with a readily usable framework, which enables researchers and practitioners to complete their analyses on React Native Android Apps.

JavaScript Program Analysis. JavaScript is traditionally used on client-side as the scripting language and has been studied [48, 54] long before the appearance of Node.js [95, 96]. Cross-site scripting (XSS) [94, 104, 112] and Cross-Site Script Inclusion attack (XSSI) [75] attacks are well studied on the client side. The dynamic and reflective nature of JavaScript poses challenges in building sound static analyses that can efficiently handle large real-world apps [45, 61, 67, 71, 95, 103]. TAJS [67] and JSAI [68] adopt abstract interpretation to analyze JavaScript programs for type inference. SAFE [73] and its follow-up work SAFEWAPI [50] covert JS to an Intermediate Representation (IR) for abstract interpretation. JavaScript call graph construction[55, 61, 96, 102] has been studied for a long time, which may use static [46], dynamic [110], or hybrid [47] analysis. For example, Nielsen et al. [96] scan Node.js application to construct modular (e.g., inter-file) call graph graph. Feldthaus et al. [61] design field-based flow analysis for constructing call graphs. Existing static call graph construction traditionally faces challenging issues for dynamic features, such as bracket syntax and Promise [43, 89]. Despite their usefulness for JavaScript code, these tools are not effective when applied to Hermes bytecode due to the unique syntax. In contrast, *Hermeser* is a parser designed for Hermes bytecode. By transforming Hermes bytecode into Jimple within Soot, ReuNify can be used for various control flow-based analyses.

8 CONCLUSION

As React Native gains popularity in Android app development, it's crucial to include it in static analysis. This study explores React Native's impact on Android app static analysis and introduces ReuNify, a tool that streamlines the process for both JavaScript-side and Dalvik-side code. ReuNify unify JavaScript-side code into the IR, *Jimple*, enabling static analysis on Hermes bytecode and Dalvik bytecode in the same static analysis framework, soot. The investigation of Hermes opcodes in real-world apps reveals usage patterns in popular apps and malware. The intra-procedural static analysis on Jimple code generated from Hermes bytecode is validated using the dedicated benchmark, "*HermesControlFlowBench*". ReuNify significantly increases the call graph size of Dalvik-side code in React Native Android apps, enhancing the Soot-based static analyzer's performance. Running FlowDroid with ReuNify uncovered an average of two additional privacy leakages in 1,007 popular React Native Android Apps, confirming the approach as a necessary improvement in the static analysis landscape for these apps.

Our research provides recommendations to improve future analysis of React Native applications. To begin, extending the scope of analysis to encompass the C++ code would be beneficial, given the frequent interaction between JavaScript and native C++ code. This broader coverage offers a more thorough understanding of the application's behavior. Additionally, enabling precise inter-procedural analysis on Hermes bytecode is crucial to understand complex interactions within React Native apps. Supporting the analysis of React Native iOS applications would further expand the coverage. Lastly, examining downstream analysis scenarios is another critical component to ensure the high quality and reliability of React Native applications. These enhancements would create a more robust framework for analyzing and improving React Native applications.

ACKNOWLEDGMENTS

Grundy is supported by ARC Laureate Fellowship FL190100035.

REFERENCES

[1] 2023. Android Native UI Components. https://reactnative.dev/docs/native-components-android
[2] 2023. AndroidRank. https://www.androidrank.org/
[3] 2023. AppBrain's statistics about SDKs for Android app frameworks. https://www.appbrain.com/stats/libraries/tag/app-framework/android-app-frameworks?list=top500
[4] 2023. bundleAssetName. https://reactnative.dev/docs/react-native-gradle-plugin#bundleassetname
[5] 2023. BytecodeFileFormat.h. https://github.com/facebook/hermes/blob/main/include/hermes/BCGen/HBC/BytecodeFileFormat.h
[6] 2023. BytecodeList.def. https://github.com/facebook/hermes/blob/main/include/hermes/BCGen/HBC/BytecodeList.def
[7] 2023. BytecodeVersion.h. https://github.com/facebook/hermes/blob/main/include/hermes/BCGen/HBC/BytecodeVersion.h
[8] 2023. Codegen. https://reactnative.dev/docs/next/the-new-architecture/pillars-codegen
[9] 2023. cordova. https://cordova.apache.org/
[10] 2023. file. https://www.ibm.com/docs/he/aix/7.2?topic=f-file-command
[11] 2023. Hermes: An open source JavaScript engine optimized for mobile apps, starting with React Native. https://engineering.fb.com/2019/07/12/android/hermes/
[12] 2023. Hermes engine project. https://github.com/facebook/hermes
[13] 2023. Hermes: JavaScript engine optimized for React Native. https://hermesengine.dev/
[14] 2023. ionic. https://ionicframework.com/
[15] 2023. JadX: Dex to Java decompiler. https://github.com/skylot/jadx
[16] 2023. JavaScriptCore. https://developer.apple.com/documentation/javascriptcore
[17] 2023. JimpleBody. https://www.sable.mcgill.ca/soot/doc/soot/jimple/JimpleBody.html
[18] 2023. JSI (JavaScript Interface). https://reactnative.dev/architecture/glossary#javascript-interfaces-jsi
[19] 2023. Lifecycle of Reactive Effects. https://react.dev/learn/lifecycle-of-reactive-effects

[20] 2023. Metro. https://facebook.github.io/metro/

[21] 2023. Native Modules Intro. https://reactnative.dev/docs/native-modules-intro

[22] 2023. New Architecture. https://reactnative.dev/docs/the-new-architecture/landing-page

[23] 2023. proguard-rules for React Native Android App. https://github.com/typeorm/react-native-example/blob/master/android/app/proguard-rules.pro

[24] 2023. React. https://react.dev/

[25] 2023. react-native: A framework for building native applications using React. https://github.com/facebook/react-native

[26] 2023. React Native Directory. https://reactnative.directory/

[27] 2023. React Native framework CHANGELOG. https://github.com/react-native-community/releases/blob/master/CHANGELOG.md

[28] 2023. Setting up the development environment. https://reactnative.dev/docs/environment-setup

[29] 2023. Skype. https://play.google.com/store/apps/details?id=com.skype.raider&hl=en&gl=US

[30] 2023. SootUp. https://soot-oss.github.io/SootUp/

[31] 2023. SuSi. https://github.com/secure-software-engineering/SuSi/tree/develop/SourceSinkLists/

[32] 2023. v8. https://github.com/v8/v8

[33] 2023. VirusShare. https://virusshare.com/

[34] 2023. WebView. https://developer.android.com/reference/android/webkit/WebView

[35] 2023. WebView. https://legacy.reactjs.org/docs/state-and-lifecycle.html

[36] 2023. What is cross-platform mobile development? https://kotlinlang.org/docs/cross-platform-mobile-development.html

[37] 2023. What is cross-platform mobile development? https://survey.stackoverflow.co/2022/#most-popular-technologies-misc-tech-prof

[38] 2023. Why a New Architecture. https://reactnative.dev/docs/the-new-architecture/why#old-architectures-issues

[39] 2024. hermes issues 1089. https://github.com/facebook/hermes/issues/1089

[40] 2024. mdn web docs. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements

[41] Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. 2017. DroidNative: Automating and optimizing detection of Android native code malware variants. computers & security 65 (2017), 230–246.

[42] Marco Alecci, Jordan Samhi, Li Li, Tegawendé F Bissyandé, and Jacques Klein. 2024. Improving Logic Bomb Identification in Android Apps via Context-Aware Anomaly Detection. IEEE Transactions on Dependable and Secure Computing (2024).

[43] Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. 2018. Finding broken promises in asynchronous JavaScript programs. Proceedings of the ACM on Programming Languages 2, OOPSLA (2018), 1–26.

[44] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting millions of android apps for the research community. In Proceedings of the 13th international conference on mining software repositories. 468–471.

[45] Esben Andreasen and Anders Møller. 2014. Determinacy in static analysis for jQuery. In Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. 17–31.

[46] Gábor Antal, Péter Hegedus, Zoltán Tóth, Rudolf Ferenc, and Tibor Gyimóthy. 2018. Static javascript call graphs: A comparative study. In 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE, 177–186.

[47] Gábor Antal, Zoltán Tóth, Péter Hegedűs, and Rudolf Ferenc. 2021. Enhanced bug prediction in javascript programs with hybrid call-graph based invocation metrics. Technologies 9, 1 (2021), 3.

[48] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. 2011. A framework for automated testing of JavaScript web applications. In Proceedings of the 33rd International Conference on Software Engineering. 571–580.

[49] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Notices 49, 6 (2014), 259–269.

[50] SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. 2014. SAFEWAPI: Web API misuse detector for web applications. In Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering. 507–517.

[51] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. 2012. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis. 27–38.

[52] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2014. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android. IEEE Transactions on Software

Engineering 40, 6 (2014), 617–632.

[53] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications. 243–262.

[54] Yinzhi Cao, Vaibhav Rastogi, Zhichun Li, Yan Chen, and Alexander Moshchuk. 2013. Redefining web browser principals with a configurable origin policy. In 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 1–12.

[55] Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. 2022. Automatic root cause quantification for missing edges in javascript call graphs. In 36th European Conference on Object-Oriented Programming (ECOOP 2022). Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[56] Daihang Chen, Yonghui Liu, Mingyi Zhou, Yanjie Zhao, Haoyu Wang, Shuai Wang, Xiao Chen, Tegawendé F Bissyandé, Jacques Klein, and Li Li. 2024. LLM for Mobile: An Initial Roadmap. arXiv preprint arXiv:2407.06573 (2024).

[57] Haonan Chen, Daihang Chen, Yonghui Liu, Xiaoyu Sun, and Li Li. 2024. Are Your Android App Analyzers Still Relevant?. In Proceedings of the IEEE/ACM 11th International Conference on Mobile Software Engineering and Systems. 69–73.

[58] Cliff Click and Michael Paleczny. 1995. A simple graph-based intermediate representation. ACM Sigplan Notices 30, 3 (1995), 35–49.

[59] Luis Cruz, Rui Abreu, John Grundy, Li Li, and Xin Xia. 2019. Do energy-oriented changes hinder maintainability?. In 2019 IEEE International conference on software maintenance and evolution (ICSME). IEEE, 29–40.

[60] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In Proceedings of the 2012 ACM conference on Computer and communications security. 50–61.

[61] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient construction of approximate call graphs for JavaScript IDE services. In 2013 35th International Conference on Software Engineering (ICSE). IEEE, 752–761.

[62] James Gosling. 1995. Java intermediate bytecodes: ACM SIGPLAN workshop on intermediate representations (IR'95). In Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations. 111–118.

[63] Samuel Groß. 2018. Fuzzil: Coverage guided fuzzing for javascript engines. Department of Informatics, Karlsruhe Institute of Technology (2018).

[64] Dongjie He, Jingbo Lu, and Jingling Xue. 2022. Qilin: A new framework for supporting fine-grained context-sensitivity in Java pointer analysis. In 36th European Conference on Object-Oriented Programming (ECOOP 2022). Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[65] Haonan Hu, Yue Liu, Yanjie Zhao, Yonghui Liu, Xiaoyu Sun, Chakkrit Tantithamthavorn, and Li Li. 2023. Detecting Temporal Inconsistency in Biased Datasets for Android Malware Detection. In 2023 38th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW). IEEE, 17–23.

[66] Yangyu Hu, Haoyu Wang, Ren He, Li Li, Gareth Tyson, Ignacio Castro, Yao Guo, Lei Wu, and Guoai Xu. 2020. Mobile App Squatting. In The Web Conference 2020 (WWW 2020).

[67] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type analysis for JavaScript. In Static Analysis: 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings 16. Springer, 238–255.

[68] Vineeth Kashyap, Kyle Dewey, Ethan A Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A static analysis platform for JavaScript. In Proceedings of the 22nd ACM SIGSOFT international symposium on Foundations of Software Engineering. 121–132.

[69] Yoonseok Ko, Hongki Lee, Julian Dolby, and Sukyoung Ryu. 2015. Practically tunable static analysis framework for large-scale JavaScript applications (T). In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 541–551.

[70] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. 2018. Automated testing of android apps: A systematic literature review. IEEE Transactions on Reliability 68, 1 (2018), 45–66.

[71] Erik Krogh Kristensen and Anders Møller. 2019. Reasonably-most-general clients for JavaScript library analysis. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 83–93.

[72] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In International symposium on code generation and optimization, 2004. CGO 2004. IEEE, 75–86.

[73] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. 2012. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages. Citeseer, 96.

[74] Sungho Lee, Julian Dolby, and Sukyoung Ryu. 2016. HybriDroid: static analysis framework for Android hybrid applications. In Proceedings of the 31st IEEE/ACM international conference on automated software engineering. 250–261.

[75] Sebastian Lekies, Ben Stock, Martin Wentzel, and Martin Johns. 2015. The Unexpected Dangers of Dynamic {JavaScript}. In 24th USENIX Security Symposium (USENIX Security 15). 723–735.

[76] Ding Li, Shuai Hao, Jiaping Gui, and William GJ Halfond. 2014. An empirical study of the energy consumption of android applications. In 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, 121–130.

[77] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 1. IEEE, 280–291.

[78] Li Li, Tegawendé F Bissyandé, and Jacques Klein. 2019. Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark. IEEE Transactions on Software Engineering (TSE) (2019).

[79] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static analysis of android apps: A systematic literature review. Information and Software Technology 88 (2017), 67–95.

[80] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. 2018. Cid: Automating the detection of api-related compatibility issues in android apps. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. 153–163.

[81] Wen Li, Jiang Ming, Xiapu Luo, and Haipeng Cai. 2022. {PolyCruise}: A {Cross-Language} Dynamic Information Flow Analysis. In 31st USENIX Security Symposium (USENIX Security 22). 2513–2530.

[82] Pei Liu, Xiaoyu Sun, Yanjie Zhao, Yonghui Liu, John Grundy, and Li Li. 2022. A first look at CI/CD adoptions in open-source android apps. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. 1–6.

[83] Yonghui Liu, Xiao Chen, Pei Liu, John Grundy, Chunyang Chen, and Li Li. 2023. ReuNify: A Step Towards Whole Program Analysis for React Native Android Apps. In 2023 IEEE/ACM International Conference on Automated Software Engineering.

[84] Yonghui Liu, Xiao Chen, Yue Liu, Pingfan Kong, Tegawendé F Bissyande, Jacques Klein, Xiaoyu Sun, Chunyang Chen, and John Grundy. 2022. A Comparative Study of Smartphone and Smart TV Apps. arXiv preprint arXiv:2211.01752 (2022).

[85] Yonghui Liu, Li Li, Pingfan Kong, Xiaoyu Sun, and Tegawendé F Bissyandé. 2021. A first look at security risks of android tv apps. In 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW). IEEE, 59–64.

[86] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. 2022. Deep Learning for Android Malware Defenses: a Systematic Literature Review. ACM Computing Surveys (CSUR) (2022).

[87] Yue Liu, Chakkrit Tantithamthavorn, Yonghui Liu, and Li Li. 2024. On the reliability and explainability of language models for program generation. ACM Transactions on Software Engineering and Methodology 33, 5 (2024), 1–26.

[88] Yue Liu, Chakkrit Tantithamthavorn, Yonghui Liu, Patanamon Thongtanunam, and Li Li. 2022. Automatically Recommend Code Updates: Are We There Yet? ACM Transactions on Software Engineering and Methodology (2022).

[89] Magnus Madsen, Ondřej Lhoták, and Frank Tip. 2017. A model for reasoning about JavaScript promises. Proceedings of the ACM on Programming Languages 1, OOPSLA (2017), 1–24.

[90] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In Proceedings of the 25th international symposium on software testing and analysis. 94–105.

[91] Xiaozhu Meng and Barton P Miller. 2016. Binary code is not easy. In Proceedings of the 25th International Symposium on Software Testing and Analysis. 24–35.

[92] Developers Meta. 2023. Core Components and Native Components. https://reactnative.dev/docs/intro-react-native-components

[93] Robin Milner. 1978. A theory of type polymorphism in programming. Journal of computer and system sciences 17, 3 (1978), 348–375.

[94] Yacin Nadji, Prateek Saxena, and Dawn Song. 2009. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense.. In NDSS, Vol. 20.

[95] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. 2019. Nodest: feedback-driven static analysis of Node. js applications. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 455–465.

[96] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. 2021. Modular call graph construction for security scanning of Node. js applications. In Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. 29–41.

[97] Changhee Park and Sukyoung Ryu. 2015. Scalable and precise static analysis of JavaScript applications via loop-sensitivity. In 29th European Conference on Object-Oriented Programming (ECOOP 2015). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[98] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, 609–620.

[99] Chenxiong Qian, Xiapu Luo, Yuru Shao, and Alvin TS Chan. 2014. On tracking information flows through jni in android applications. In 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. IEEE, 180–191.

[100] Jordan Samhi, Alexandre Bartel, Tegawendé F Bissyandé, and Jacques Klein. 2021. Raicc: Revealing atypical inter-component communication in android apps. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 1398–1409.

[101] Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. 2022. JuCify: a step towards Android code unification for enhanced static analysis. In Proceedings of the 44th International Conference on Software Engineering. 1232–1244.

[102] Dominik Seifert, Michael Wan, Jane Hsu, and Benson Yeh. 2022. An asynchronous call graph for JavaScript. In Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice. 29–30.

[103] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of {ReDoS} Vulnerabilities in {JavaScript-based} Web Servers. In 27th USENIX Security Symposium (USENIX Security 18). 361–376.

[104] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. 2014. Precise client-side protection against {DOM-based}{Cross-Site} scripting. In 23rd USENIX Security Symposium (USENIX Security 14). 655–670.

[105] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. 245–256.

[106] Mingshen Sun, Tao Wei, and John CS Lui. 2016. Taintart: A practical multi-level information-flow tracking system for android runtime. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. 331–342.

[107] Xiaoyu Sun, Xiao Chen, Yonghui Liu, John Grundy, and Li Li. 2023. LazyCow: A Lightweight Crowdsourced Testing Tool for Taming Android Fragmentation. In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2127–2131.

[108] Xiaoyu Sun, Xiao Chen, Yonghui Liu, John Grundy, and Li Li. 2023. Taming android fragmentation through lightweight crowdsourced testing. IEEE Transactions on Software Engineering 49, 6 (2023), 3599–3615.

[109] Tian Tan and Yue Li. 2023. Tai-e: A Developer-Friendly Static Analysis Framework for Java by Harnessing the Good Designs of Classics. (2023).

[110] Tajkia Rahman Toma and Md Shariful Islam. 2014. An efficient mechanism of generating call graph for JavaScript using dynamic analysis in web application. In 2014 International Conference on Informatics, Electronics & Vision (ICIEV). IEEE, 1–6.

[111] Raja Vallee-Rai and Laurie J Hendren. 1998. Jimple: Simplifying Java bytecode for analyses and transformations. no (1998).

[112] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2007. Cross site scripting prevention with dynamic data tainting and static analysis.. In NDSS, Vol. 2007. 12.

[113] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. 2018. Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 1137–1150.

[114] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. 226–237.

[115] Lili Wei, Yepang Liu, Shing-Chi Cheung, Huaxun Huang, Xuan Lu, and Xuanzhe Liu. 2018. Understanding and detecting fragmentation-induced compatibility issues for android apps. IEEE Transactions on Software Engineering 46, 11 (2018), 1176–1199.

[116] Haowei Wu, Shengqian Yang, and Atanas Rountev. 2016. Static detection of energy defect patterns in android applications. In Proceedings of the 25th International Conference on Compiler Construction. 185–195.

[117] Zhiqiang Zang, Fu-Yao Yu, Aditya Thimmaiah, August Shi, and Milos Gligoric. 2024. Java JIT Testing with Template Extraction. arXiv preprint arXiv:2403.11281 (2024).

[118] Hailong Zhang, Haowei Wu, and Atanas Rountev. 2016. Automated test generation for detection of leaks in Android applications. In Proceedings of the 11th International Workshop on Automation of Software Test. 64–70.

[119] Yanjie Zhao, Pei Liu, Xiaoyu Sun, Yue Liu, YONGHUI LIU, John Grundy, and Li Li. [n. d.]. Autopatch: Learning to generate patches for automatically fixing compatibility issues in android apps. Available at SSRN 4254659 ([n. d.]).