Static Analyses of Interlanguage Interoperations

by

Jyoti Prakash

A dissertation submitted to the
Faculty of Computer Science and Mathematics
in partial fulfillment of requirements for the degree of
Doctor of Natural Science (Dr. rer. nat.)

Advisor: Prof. Dr-Ing. Christian Hammer

Chair of Software Engineering I
Faculty of Computer Science and Mathematics
University of Passau

Passau, October 2023

Advisor: Prof. Dr-Ing. Christian Hammer
Reviewers: (1) Prof. Dr-Ing. Christian Hammer
(2) Prof. Dr. Peter Thiemann

Acknowledgements

This thesis would not have been possible without the support of my family, friends, and colleagues.

Thank you to my PhD advisor, Prof. Dr.-Ing. Christian Hammer, without whom this thesis would not have been possible. It has been a long journey in three universities since I first met him in Saarbrücken. He introduced me to the topic of program analysis and offered me the opportunity to write a master's thesis and have an exciting roller-coaster ride of a Ph.D. I am grateful to him for having faith in me and my work and for his help in navigating bureaucratic hurdles.

I am also grateful to Prof. Dr. Peter Thiemann for agreeing to be a reviewer for the thesis.

With Abhishek, I enjoyed the PhD time inside and outside of work. We faced the trials and tribulations of research in many of our joint projects. His suggestions have significantly improved my writing and presentation skills. I was pleased to meet Adebayo when he joined the Group and amazed to see how his expressions could speak a thousand lines. Discussions with Max when we were TAing for the course, Static Program Analysis strengthened my understanding of the subject. Last but not least, I am also thankful to my colleagues at Potsdam and Passau for maintaining a cordial and relaxed work atmosphere.

Outside of work, a huge thanks to all my friends from various stages of my life. Your presence has made adjusting to the new environment easy at every stage of my life. I have always enjoyed and cherished your company.

I cannot thank my parents and parents-in-law enough for supporting my goals. Thank you to my wife for trusting me and being patient while working on this thesis. I am grateful to my sister, who has always helped me on my bad days and always has been a pillar of strength. Thank you to my brother-in-law, who is also on his way to a PhD, for being a high achiever. I am thankful to my cousins, who ensured I had a great time during my vacations. I had to miss many important events of family and friends while working on this thesis; I owe a debt of gratitude to them for putting up with my absence.

I am also indebted to my high school Computer Science teacher for introducing me to

programming and Java. At last, I humbly dedicate this thesis to my parents. I can never imagine making the kind of sacrifices they have made for their children's education.

Jyoti Prakash
Passau, October 2023

Dissemination and Contribution Statement

Excerpts from the following works feature in this thesis:

(P1) Tiwari, A., Prakash, J., Groß, S., and Hammer, C. (2019). LUDroid: A Large Scale Analysis of Android – Web Hybridization. In 2019 IEEE 19th International Working Conference on Source Code Analysis and Manipulation (SCAM), Cleveland, OH, USA, 2019 pp. 256-267. https://doi.org/10.1109/SCAM.2019.00036

- Contributions: The author conducted an empirical study of Javascript programming patterns on the benign apps. Abhishek Tiwari implemented the technique to extract the WebView information, source code, and URL snippets. Sascha Groß conducted the evaluation on URLs in benign dataset. All of the authors participated in writing the paper.
- Parts from this work appears in Chapter 3.

(P2) Tiwari, A., Prakash, J., Groß, S., and Hammer, C. (2020). A Large Scale Analysis of Android — Web Hybridization. In Journal of Systems and Software, Volume 170, 2020, 110775, ISSN 0164-1212, https://doi.org/10.1016/j.jss.2020.110775.

- Contributions: This paper is a longer version of (P1) and subsumes its findings. Here, The author extended the empirical study of Javascript programming patterns in both benign apps and malware apps. In addition, the author performed the study of URLs in malware apps. Abhishek Tiwari implemented the technique to extract the WebView information, source code, and URL snippets. Abhishek, Christian, and Jyoti participated in writing the paper.
- Parts from this work appears in Chapter 3.

(P3) Prakash, J.*, Tiwari, A.*, Hammer, C. (2023). Demand-driven Information Flow Analysis of WebView in Android Hybrid Apps. *34th IEEE International Symposium on Software Reliability Engineering*, to appear.

- Contributions: The author and Abhishek Tiwari jointly developed, implemented, and evaluated the idea. Christian Hammer contributed to the discussions and writing of the paper.
- Parts from this work appears in Chapter 4.

(P4) Prakash, J., Tiwari, A., Hammer, C. (2023). Modular Analysis of Polyglot Interoperations via Summary Specialization. *Submitted for review to a major conference in Programming Languages.* A pre-print is available on https://arxiv.org/abs/2305.03916.

- Contributions: The author developed, implemented, and evaluated the idea. Abhishek Tiwari and Christian Hammer contributed to the discussions and writing of the paper.
- Parts from this appears in Chapter 5.

These published works by the author are not part of the thesis.

(P5) Prakash, J., Tiwari, A., Hammer, C. (2021). Effects of Program Representation on Pointer Analyses — An Empirical Study. In: Guerra, E., Stoelinga, M. (eds) Fundamental Approaches to Software Engineering. FASE 2021. Lecture Notes in Computer Science, vol 12649. Springer, Cham. https://doi.org/10.1007/978-3-030-71500-7_12

(P6) Tiwari, A.*, Prakash, J.*, Rahimov, A., and Hammer, C. (2023), "Understanding the Impact of Fingerprinting in Android Hybrid Apps," 2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft), Melbourne, Australia, 2023, pp. 28-39, https://doi.org/10.1109/MOBILSoft59058.2023.00011. (Received the ACM SIGSOFT Distinguished Paper Award)

* indicates joint first authors

Abstract

Software Developers are moving towards a multilingual development where they combine two languages in a single application to harness the strengths of each language. For example, performance-critical components of a Java application can be implemented in C language. It provides flexibility, at the same time, it becomes difficult to statically analyze these applications. The amalgamation of two languages in a single application may introduce bugs ranging from type-mismatch to security vulnerabilities. Therefore, it is necessary to develop static analysis techniques to aid developers in multilingual development. In this thesis, we develop techniques to study and analyze these applications.

In the first part of the thesis, we study the prevalence of security and privacy vulnerabilities in hybrid apps. Hybrid apps are Android apps that combine both Java and Javascript components, where the Android part is secured (on the device), while the JavaScript part is exposed to web. Additionally, some of the Java functions are available to JavaScript component through an interface called as bridge interface. In the pursuit of the goal, we adopt a static backtracking of data dependencies to determine the flow of information from the android component to the web component. Our study revealed the potential sources of unsoundness in the existing static analyses. Static backtracing also induces imprecision in the analysis, i.e., there might be some flows that are not possible during runtime albeit are reported by the analysis. These were mitigated through a manual verification. This work reveals that the android-web hybridization can lead to (potential) vulnerabilities that might impact the confidentiality as well as the integrity properties of these apps.

From the communication patterns occurring in Android WebView, we noticed that its is feasible for an attacker to jeopardize the integrity of apps by corrupting some value, say an input on the web through bridge interfaces. Motivated by this, we define a information flow analysis of the bridge interfaces and the associated data flows in hybrid apps. In the first step, we propose a novel threat model where we model the attacker as someone who wants to influence the behavior of android app as an integrity violation. Based on this threat model, we then propose a demand-driven analysis technique to detect confidentiality and integrity violations. Our analysis leverages, a demand-driven technique, where it only analyzes the relevant part of app for the information flow analysis with the help of function summaries — escaping the need of a whole-program analysis.

In the second part of the thesis, we generalize the approach to static analysis of multilingual applications. To this end, we investigate into the question of combining existing single language analyses to analyze multilingual programs. To provide an affirmative answer, we define an analysis to leverage single language analyses for call-graph and pointer analysis of multilingual programs. Our analysis takes two existing unilingual analyses and analyzes the complete multilingual program. It uses a novel summary specialization technique that resolves the information flows at the bridge interfaces by utilizing independent pre-analyses (modulo foreign function interfaces) of each language component. We apply this technique to analyze Android-NDK and GraalVM Java-Python multilingual applications for generating call-graphs.

In summary, we have developed novel techniques for information flow and call-graph analysis for multilingual programs. With this, we motivate the need of static analyses for multilingual applications and its applications which includes, vulnerability detection, program understanding, amongst others.

Zussamenfassung

Softwareentwickler gehen immer mehr zu einer mehrsprachigen Entwicklung über, bei der sie zwei Sprachen in einer einzigen Anwendung kombinieren, um die Stärken der einzelnen Sprachen zu nutzen. So können beispielsweise leistungskritische Komponenten einer Java-Anwendung in der Sprache C implementiert werden. Das bietet Flexibilität, gleichzeitig wird es aber schwierig, diese Anwendungen statisch zu analysieren. Die Verschmelzung von zwei Sprachen in einer einzigen Anwendung kann zu Fehlern führen, die von Typ-Fehlern bis hin zu Sicherheitslücken reichen. Daher ist es notwendig, statische Analysetechniken zu entwickeln, um Entwickler bei der mehrsprachigen Entwicklung zu unterstützen. In dieser Arbeit entwickeln wir Techniken zur Untersuchung und Analyse dieser Anwendungen.

Im ersten Teil der Arbeit untersuchen wir das Vorkommen von Sicherheits- und Datenschutzschwachstellen in hybriden Apps. Hybride Apps sind Android-Apps, die sowohl Java- als auch Javascript-Komponenten kombinieren, wobei der Android-Teil (auf dem Gerät) gesichert ist, während der Javascript-Teil dem Web ausgesetzt ist. Außerdem sind einige der Java-Funktionen für die JavaScript-Komponente über eine Schnittstelle, die so genannte Bridge-Schnittstelle, verfügbar. Um dieses Ziel zu erreichen, verwenden wir eine statische Rückverfolgung der Datenabhängigkeiten, um den Informationsfluss von der Android-Komponente zur Webkomponente zu bestimmen. Unsere Studie hat die potenziellen Schwachstellen in den bestehenden statischen Analysen aufgedeckt. Die statische Rückverfolgung führt auch zu Ungenauigkeiten in der Analyse, d.h. es könnte einige Datenflüsse geben, die während der Laufzeit nicht möglich sind, obwohl sie von der Analyse gemeldet werden. Diese wurden durch eine manuelle Überprüfung entschärft. Diese Arbeit zeigt, dass die Android-Web-Hybridisierung zu (potenziellen) Schwachstellen führen kann, die sich auf die Vertraulichkeits- und Integritätseigenschaften dieser Anwendungen auswirken können.

Anhand der Kommunikationsmuster in Android WebView haben wir festgestellt, dass es für einen Angreifer möglich ist, die Integrität von Apps zu gefährden, indem er einen Wert, z.B. eine Eingabe im Web, über Bridge-Schnittstellen verfälscht. Aus diesem Grund definieren wir eine Informationsflussanalyse der Bridge-Schnittstellen und der damit verbundenen Datenflüsse in hybriden Anwendungen. In einem ersten Schritt schlagen wir ein neuartiges Bedrohungsmodell vor, bei dem wir den Angreifer als jemanden modellieren, der das Verhalten der Android-App als Integritätsverletzung beeinflussen will. Auf der Grundlage dieses Bedrohungsmodells schlagen wir dann eine

bedarfsgesteuerte Analysetechnik vor, um Vertraulichkeits- und Integritätsverletzungen zu erkennen. Unsere Analyse nutzt eine bedarfsgesteuerte Technik, bei der nur der relevante Teil der App für die Analyse des Informationsflusses mit Hilfe von Funktionszusammenfassungen analysiert wird — so dass keine Analyse des gesamten Programms erforderlich ist.

Im zweiten Teil der Arbeit verallgemeinern wir den Ansatz auf die statische Analyse von mehrsprachigen Anwendungen. Zu diesem Zweck untersuchen wir die Frage, ob bestehende einsprachige Analysen kombiniert werden können, um mehrsprachige Programme zu analysieren. Um diese Frage zu bejahen, definieren wir eine Analyse, die einsprachige Analysen für die Call-Graph- und Pointer-Analyse von mehrsprachigen Programmen nutzt. Unsere Analyse nimmt zwei bestehende einsprachige Analysen und analysiert das komplette mehrsprachige Programm. Sie verwendet eine neuartige zusammenfassende Spezialisierungstechnik, die die Informationsflüsse an den Brückenschnittstellen auflöst, indem sie unabhängige Voranalysen (modulo fremde Funktionsschnittstellen) jeder Sprachkomponente verwendet. Wir wenden diese Technik an, um mehrsprachige Android-NDK- und GraalVM-Java-Python-Anwendungen zur Erstellung von Aufrufgraphen zu analysieren.

Zusammenfassend lässt sich sagen, dass wir neuartige Techniken für den Informationsfluss und die Analyse von Aufrufgraphen für mehrsprachige Programme entwickelt haben. Damit begründen wir den Bedarf an statischen Analysen für mehrsprachige Anwendungen und die Anwendungen dieser Analysen für Aufgaben wie die Erkennung von Schwachstellen, das Verstehen von Programmen und vieles mehr.

Contents

Part I

Introduction and Background

Chapter 1

Introduction

1.1 Motivation

Multilingual programming is a programming paradigm where programs are written in multiple languages utilizing the ease of development of functionalities in each language. Recently, there has been a surge in developers adopting this paradigm [43]. It enhances the development speed, as they can reuse the components written in one language and provide an interface to use in another language with minimal effort. For example, they can write performance-critical components in systems programming languages like C++ or Rust and combine them with Java for Android-related tasks. In the last couple of years, multiple environments have increasingly supported this paradigm, examples of which include Android WebView [36, 39], Graal multilingual API [86, 87], and Java JNI [88], among others.

Two of the popular multilingual environments used by mobile developers, among others, are Android NDK [47] and Android WebView [36]. Android NDK (and Java Native Interface) allow writing a part of Java class in C/C++, potentially for performance reasons. Android WebView facilitates writing mobile apps using JavaScript. An upcoming platform is Graal Polyglot, which allows interspersing multiple language combinations such as Java and Python, Java and JavaScript, and many others. In this thesis, design program analyses for such environments.

Multilingual software systems make developers' life easier but comes with a cost. Increasing the number of programming languages and runtime environments means expanding the attack surface area and the possibility of causing bugs at the interface of two languages. Past studies have demonstrated that these attacks are possible [82, 63, 60, 75] and in principle, these systems are at least as vulnerable as their single language components. Here, developers could leverage static analyses to discover potential vulnerabilities. However, static analyses are limited by the target language [13, 108, 70, 57] — which means that these run on a single language — which is a hindrance in the adoption of static analysis in multilingual applications.

1.2 Problem Statement

Central to static analyses is the problem of computing call graphs, pointer analysis, and information flow analysis. Intuitively, call graphs are directed graphs showing the

relations between the calling and called functions. Pointer analysis statically determines the set of objects each variable refers to in the program. Information flow analysis traces the leak of sensitive data through public sinks. This thesis proposes novel frameworks for call-graph, pointer analysis, and information flow analysis of multilingual programs. Our goals in the thesis are:

- **Goal 1.** How vulnerable is the communication between Android and JavaScript running on WebView?
- **Goal 2.** Is it possible to combine various monolingual program analyses available for the languages for analyzing multilingual programs?

Goal 1 aims to understand the limitations of static analysis in analyzing real-world apps and discover the popular programming patterns used in apps using Android WebView through a large-scale study of programming patterns in Android WebView. Based on the study, we also define a novel threat model and a corresponding demand-driven information flow analysis thereof to detect violations under the threat model. In **Goal 2**, we leverage the strength of existing monolingual analysis and devise a novel function summary-based algorithm to combine the analyses for pointer analysis and call graph analysis of multilingual programs. We apply this analysis to Android NDK and GraalVM multilingual programs and demonstrate its efficacy by evaluating through a research prototype on both benchmark and real-world applications.

1.3 Contributions

1.3.1 Empirical Analysis of Android WebView

In the first step, we studied the usage of WebView communication in real-world apps. For the purpose of the study **Goal 1**, we developed a tool named *LuDroid*, based on static data-dependence analysis of WebView-related objects to determine the necessary variables for WebView-related operations. We consider a dataset of 7,500 apps from the Google Play Store (with 196 most popular apps) and 1,000 malware samples for the empirical evaluation. *LuDroid* overcomes some of the challenges to make the empirical study feasible. First, URLs and JavaScript code are passed as strings to WebView interfaces, which is difficult to determine with static analysis. To solve this, we implemented a string analysis that resolves the strings passed to the WebView APIs by resolving the value passed to `StringBuilder` and similar methods. Since the string resolution is based on a static resolution, we get some spurious strings, which we later filter through both manual and semi-automatic methods. Finally, we obtain the code strings passed to the WebView API methods and start with classifying the obtained code string.

We then classify the strings obtained from the previous step. We bucket these strings into URLs and JavaScript code. Out of these, URLs that start with a `file:///` protocol were found in around 41% of these apps. These URLs denote that HTML sources are available in the app itself. We consider the available HTML and JavaScript files in these apps for analysis. In the other case, JavaScript code is being passed through these APIs. These JavaScript code snippets are executed by the WebView browser. We obtain these JavaScript code snippets found in these apps for further analysis.

In this study, we categorized the dataset into benign, frequently used, and malware. In the study, we found that information flow where the underlying Java object from

Android is modified in the JavaScript part of the code via setter methods, which is an example of dataflow from JavaScript to the Android part. We name this data flow as non-trivial information flow, which was found in benign and malware datasets. In another case, we found third-party remote script injection via insecure protocols in both benign apps. Apart from this, we found the use of Android WebView to obfuscate the app to hide its origins. Based on these findings, we developed a demand-driven IFC analysis for hybrid apps, which we explain in the next section.

As a side-effect,*LuDroid* was able to identify the URLs, apart from those with `file://` protocol, which are used to display existing web applications within WebView. To our surprise, twelve percent of these studied apps load the WebPages via the HTTP protocol, which makes it suspectable to phishing attacks. Only a mere six percent of these URLs use the HTTPS protocol, while the remaining 41% of the apps refresh the WebView via `about:blank` URL.

1.3.2 Demand Driven Information Flow Analysis of Android

Based on our observations obtained in the previous part, we extend **Goal 1** and develop a demand-driven analysis strategy that is based on a novel security model. Contrary to the popular notion of source-to-sink flows, we observed a type of flow that modifies the Android objects from the JavaScript side in the previous study. We formulated this problem as a combined IFC analysis on a defined thread model to prevent confidentiality and integrity leaks. Our threat model prioritizes the Android component of the Hybrid app. Permissions guard Android's components while JavaScript and HTML are exposed to the Web. In our threat model, we consider the Android part as the trusted component while the JavaScript component is the untrusted component. A confidentiality violation occurs when the dataflow happens from the Android side and leaks in the JavaScript part of the application. An integrity violation occurs when the bridge methods accept parameters and these methods are invoked within the JavaScript code.

We implemented the analysis in a tool called *IWandDroid*. *IWanDroid* is built on top of the *LuDroid* and uses it to obtain the pre-processed information. The first step is pre-processing, which collects information about the bridge methods and stores it in a database. At this step, the analysis has captured the bridge method information and constructed the JavaScript code passed to the APIs. For this, we modified *LuDroid* with its shortcomings, such as handling interprocedural cases of data dependence.

The IFC analysis starts after it has captured the preprocessed information. The analysis proceeds in three phases, where each phase is based on the IFDS and IDE frameworks. In the first phase, it computes the function summaries of the bridge methods using a reachability analysis to discover the variables that are reachable from the parameters in the bridge summaries. Once we have these summaries, it computes the summaries of the methods that invoke the WebView APIs and shares the bridge object in the second phase. The analysis in the second phase is based on IDE where it computes the security labels of each bridge function-related path. The third phase uses the summaries obtained from the previous phases and computes the IFC labels for variables in the shared JavaScript code.

We evaluated the analysis on a set of microbenchmarks and real-world apps. Due to the unavailability of appropriate microbenchmarks for the Hybrid apps, we developed a

set of microbenchmark suite with application that incorporates the peculiar semantics of the Android WebView. Our results achieve 100% precision in the case of microbenchmarks and identified 17 confidentiality and 15 integrity violations in real-world apps.

1.3.3 Unifying unilingual static analyses for multilingual programs

To tackle **Goal 2**, we leverage existing monolingual analysis, such as analysis for C, C++, Java, and many others, and use those in a multilingual program. Existing analyses for a particular analysis are at a matured state and have incorporated a large semantics of the particular language. Choosing not to use these analyses will require us to write the analysis of multilingual programs from scratch, which is a difficult and tedious task.

To answer this, we propose *Conflux*[1], a modular framework to integrate monolingual static analysis for multilingual applications. Our analysis is based on computing the program summaries at the bridge interfaces and the methods participating in bridge communication using the analysis taken from running the monolingual analysis on these programs. We denote these analyses as pre-analyses. These pre-analyses results are used to compute the function summaries across the bridge functions. Once it has computed the bridge summaries, it unifies these summaries based on our novel summary unification. The unified summary coalesces the summary of all functions participating in the bridge. These unified summaries are then separated into each function with our novel summary specialization technique which is then fed to the pre-analysis for recomputing the analysis. With conflux, we were able to develop two instantiations of the analysis to analyze Android NDK and GraalVM multilingual programs. To the best of our knowledge, we were the first to analyze GraalVM multilingual API programs. To analyze Android NDK, we leveraged WALA [21] analysis for Android and SVF [112] analysis for C++ libraries. For analyzing GraalVM multilingual programs, we use the WALA library analysis for Java and Python programs.

Engineering the framework as a tool requires a huge deal of engineering effort. It requires defining the interfaces to integrate the underlying tool. Summary unification and specialization need the relevant methods to have a common IR representation. In this, we choose the static single assignment (SSA) representation, which is ubiquitous in various program analyses. At other places, it is required to do a localized intra-procedural dataflow analysis to track the APIs from multilingual programs. Nonetheless, with these engineering efforts, one can design and combine existing monolingual program analyses and lift those to analyze multilingual programs.

We evaluated two instantiations of the analysis on a set of microbenchmarks and real-world apps from the Google PlayStore. For GraalVM Polyglot API, we gathered three microbenchmarks from Github. Since GraalVM Polyglot is a relatively [87] new technology, we were unable to get any real-world applications written in that framework. In the case of microbenchmarks, Conflux was able to resolve the call-graph edges in the interlanguage components and identify the correct call-graph in both languages, thanks to the pre-analysis, which surpasses existing tools for analyzing Android NDK.

[1] conflux means the act of blending two components

1.4 Thesis Outline

The rest of the thesis is organized as follows:

Chapter 2 contains the necessary background about static analysis to understand the thesis. Readers familiar with it may choose to skip the chapter.

Chapter 3 presents an empirical study on Android WebView conducted with our tool *LuDroid*. It discusses the methodology with its limitations and presents the results obtained from the study.

Chapter 4 presents a demand-driven information flow analysis for Android *IWanDroid* . Here, it presents the three phases of the analysis and concludes the chapter with the results from an empirical evaluation done on microbenchmarks and real-world apps.

Chapter 5 presents the framework *Conflux* to integrate single language analysis. We discuss the methodology and present the engineering efforts required to integrate the analyses. Finally, the chapter concludes with an empirical evaluation of microbenchmarks and real-world apps.

Chapter 6 presents the related work from the current literature used in preparing the thesis.

Chapter 7 presents the discussions about the limitations of the thesis and concludes with future work.

Chapter 2

Background

In this section, we provide the necessary background to understand the thesis. It introduces the multilingual programming paradigm and dataflow analysis techniques used in the thesis.

2.1 Multilingual Programming Environments

2.1.1 Host and Guest Languages

A set of two languages, where one language is called the host language while the other is called the guest language, forms the pillars of multilingual programming. Host language starts the program execution, such as a `main` method or methods that denote the entry points in an event-driven programming environment. The host language triggers the execution of the guest language code. This trigger may be through an event or direct invocation of the function call. Once the host language has invoked a method in the guest language, subsequent invocations could be from guest to host and *vice-versa*.

A set of program variables and methods defined in the host and guest languages are shared between both languages. In this thesis, we refer to these variables as *bridge variables* and the methods as *bridge methods*. Bridge variables are accessible from the host and guest languages. Similarly, bridge methods are invocable from host and guest languages like a regular method in the program. These bridge variables and methods facilitate the communication between the host language and the guest language. This enables interoperability among the host language and guest languages that form the *central dogma* of multilingual programming (shown in Fig. 2.1).

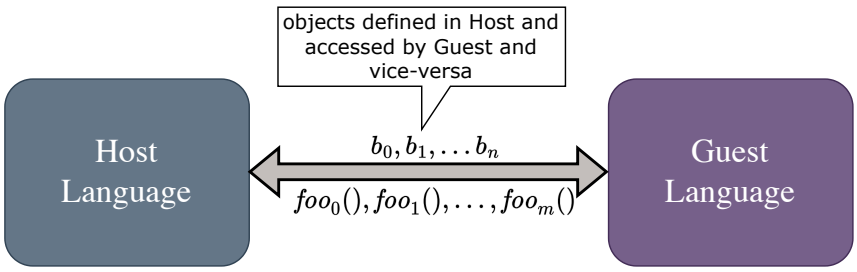


Figure 2.1: Model of Multilingual programming

2.1.2 Runtime and Compiler Support

Host and guest languages are compiled and executed in multiple strategies. To the best of my knowledge, the following two approaches have been used to provide compiler and runtime support:

- Compile the host and guest languages to a common runtime platform or native platform. Since the languages have been compiled to a common runtime target environment, the runtime does not distinguish between host and guest languages but rather sees the bytecode or the native code.
- Dedicated runtime for the guest language embedded in the runtime for the host language. The guest language and host language runtime may communicate through shared memory.

2.2 Examples of the environments studied in this thesis

2.2.1 Android WebView

Android WebView [36] presents an embedded browser that runs within the sandbox of the Android native app. WebView enables the app to run JavaScript code within the runtime environment provided by the embedded browser. At the same time, it can also access Native Android functionalities through bridge objects shared from the Android native side. Android WebView provides interfaces to load URLs and execute JavaScript within the browser runtime. Android WebView runs on the UI thread, with the scope of this thread bound to the UI where it is declared.

Android only allows sharing Java methods through the bridge objects, not the class's fields. The methods are shared as a property of the bridge object, but the behavior varies a little from the regular JavaScript properties. For example, it is not possible to explicitly delete these objects. However, it is not allowed to override the object property [119].

2.2.2 Android NDK

Android NDK [47] is similar to the Java Native Interface and allows Android apps to use the underlying machine's native libraries written in C and C++. The C/C++ part of the app runs in the sandbox of the app itself. It is primarily used for applications that require closer access to memory access. Developers declare the methods that they wish to implement in C++ during the development. Similar to Android WebView, it also allows sharing the Android objects and methods through clearly marked interfaces, which allow communication between the Android objects and the system libraries. The Java objects, and the execution environment are referred through pointers to data structures created for this purpose. Developers can read and write to the Java class fields from C++ like JNI. Garbage collection of the dynamically created objects on the C++ side is not guaranteed. The validity of field read/write is also not guaranteed.

2.2.3 GraalVM Polyglot

GraalVM polyglot is a virtual machine, similar to Java Virtual Machine, which supports running programs in multiple languages. The underlying environment uses Java bytecode. Java and similar languages are compiled to bytecode. Other interpreted languages

are first interpreted via the truffle AST interpreter [87] and optimized before compiling to Java bytecode. It also has a polyglot API, which allows interoperability between the languages supported by the GraalVM. The GraalVM is supported by *Truffle* [87], an AST interpreter which transforms the AST by augmenting it with runtime information, which is subsequently transformed to optimized bytecode by partial evaluation [127]. It also provides the protocols for defining interoperability between data types on the host and guest languages [44]. Like any other managed runtime, GraalVM also supports garbage collection, which frees any unreachable objects.

2.3 Dataflow Analysis

Dataflow analysis computes the information held at different locations of the programs. It can answer questions like whether a variable can hold *null* value at some program location or whether a variable holds a sensitive value. It owes its origins to compiler optimizations. Nowadays, it is being increasingly used for program correctness and software security. Precisely computing a dataflow analysis is impossible due to the undecidability of the halting problem. Therefore, we need approximations to keep it decidable. A monotonic dataflow analysis can approximate this dataflow analysis information and guarantee termination at the same time.

The key ingredients of the dataflow analysis are: (1) a control flow graph of the program to represent the possible control flows in the program, (2) a finite abstract domain $\mathbb{D}$ to represent the interested dataflow information, and (3) a set of functions mapping $\mathbb{D} \mapsto \mathbb{D}$ for each statement s_i in the program to represent the execution semantics of the statement s_i in terms of the abstract domain $\mathbb{D}$.

Control Flow Graph A program is represented as a directed graph called a, *Control Flow Graph* (CFG), to denote the sequence of execution of the statements (or program locations) in a program. A $CFG = (loc, ENTRY, EXIT, E)$ where $E \subseteq loc \cup \{ENTRY, EXIT\} \times loc \cup \{ENTRY, EXIT\}$. loc is the set of program locations, $ENTRY$, and $EXIT$ are special nodes that denote the entry and exit of a function.

An edge $s_i \rightarrow s_j \in E$ means that s_j comes right after s_i in a possible execution of the program, and s_j is called the successor of l_i. A path in a CFG, $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n$, is a sequence of program locations such that s_{i+1} is the successor of s_i.

2.3.1 Monotone Dataflow Analysis

The general framework for dataflow analysis — Montone dataflow analysis — was pioneered in the seventies [31]. A monotonic dataflow analysis requires a set of monotonic transfer functions [31], simply called transfer functions. These specify the changes introduced to the dataflow information.

Definition 1 (Monotonic Transfer Function)**.** *A monotonic transfer function is a function $f_s : \mathbb{D} \rightarrow \mathbb{D}$ defined for a program location s. The function f_s is monotonic, i.e., $\forall a, b \in \mathbb{D}, a \leq b \implies f_s(a) \leq f_s(b)$.*

Definition 2 (Monotonic Transfer Function Space)**.** *A space of monotonic transfer functions $\mathcal{F}$ is a set of monotonic transfer functions f_s if:*

1. *$\mathcal{F}$ contains all functions f_s*
2. *There exists an identity function id, such that $id(s) = s$*
3. *A set of functions $f_s^i \in \mathcal{F},\ i \in [n]$ is closed under composition, i.e., $f_s^n \circ f_s^{n-1} \ldots \circ f_s^1 \in \mathcal{F}$*
4. *Pointwise composition holds for any functions f_s and f'_s, i.e., $f_s \sqcup f'_s \in \mathcal{F}$.*

Definition 3 (Montonic Dataflow Analysis)**.** *A monotonic dataflow analysis is a tuple $(CFG, \mathcal{S}, \mathcal{F})$ defined over a control flow graph CFG, monotonic function space $\mathcal{F}$ and a complete lattice $\mathcal{S} = (\mathbb{D}, \leq, \sqcup, \sqcap)$. Each node n of the CFG is mapped to the transfer function $f_n \in \mathcal{F}$.*

Meet Over Path solution A goal of a dataflow analysis is to compute the *meet-over-path (MOP)* solution. Given a path of program locations, $p : l_0 \to l_1 \to l_2 \to \ldots \to l_i$ and the corresponding transfer functions, $f_0, f_1, \ldots, f_i$, the composite transfer functions for p with some initial state $x_0 \in \mathbb{D}$ is:

$$s_{mop} = f_p(x_0) = f_i \circ f_{i-1} \circ \ldots \circ f_0(x_0)$$

A *MOP* solution for a set of CFG paths is $MOP = \bigsqcup_{p \in paths} f_p(s_p)$, i.e., it composes the transfer functions along each path in the CFG and combines them at the control-flow joins. A *MOP* solution is the most precise solution, albeit undecidable. A minimal fixed-point solution overcomes this drawback by keeping it decidable at the cost of precision.

Fixed Point Solution A fixed point for a function f_s is a point $x \in \mathbb{D}$ such that $x = f_s(x)$. A minimal fixed point $x \in \mathbb{D}$ is such that $\forall x',\ x' = f_s(x'), x' \not\leq x$. The characterization of fixed-point central to static analysis is given by Knaster-Tarski fixed point theorem shown in Theorem 1. It says that the least (or greatest) fixed points exist for a finite lattice and is unique. In this thesis, we have presented the theorem without proof; interested readers may look at the Tarski *et al.* [115] for the proof of this statement.

Theorem 1. *[Kanster-Tarksi Fixed Point Theorem] Let a function $f : \mathbb{D} \to \mathbb{D}$ be defined on $\mathbb{D}$ and $(\mathbb{D}, \leq)$ is a complete lattice then a unique greatest fixed point, $gfp = \bigsqcup\{x \in \mathbb{D} \mid d \leq f(x)\}$, exists. Conversely, a unique least fixed point, $lfp = \bigsqcap\{x \in \mathbb{D} \mid f(x) \leq x\}$, also exists.*

A monotonic dataflow analysis problem $(CFG, \mathcal{S}, \mathcal{F})$ assigns each node of the n of CFG to a dataflow transfer function f. Following Theorem 1, we can infer that a unique fixed-point solution exists for $\mathcal{F}$. This solution is called the maximum fixed-point solution and is obtained by repeated application of the set of transfer functions in $\mathcal{F}$ until a fixed point is reached.

Definition 4 (Maximal Fixedpoint Solution)**.** *A maximal fixed-point solution is the fixed-point solution is $s_{mfp} = \mathcal{F}(s_{mfp})$.*

From Theorem 1, we know that a unique s_{mfp} exists. Recall the discussion in that a monotonic dataflow analysis computes the meet-over-path solution. The maximal

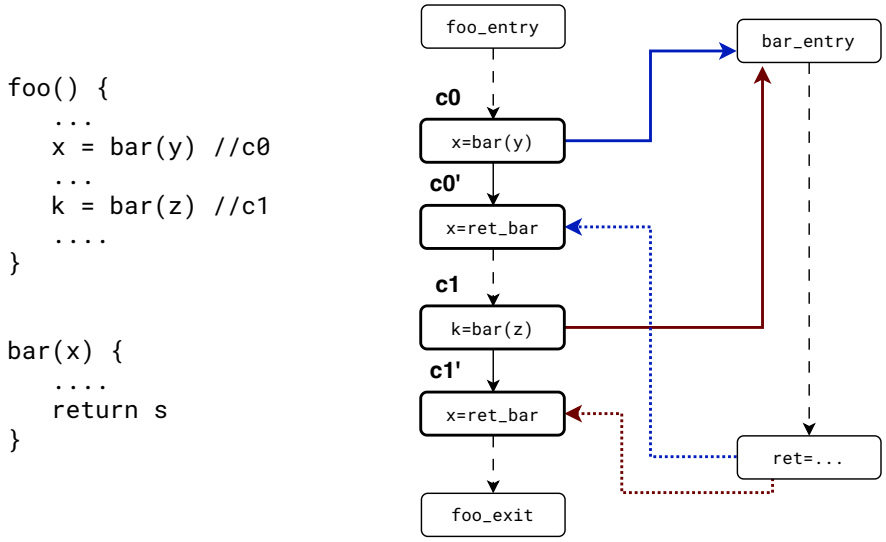


Figure 2.2: Interprocedural Control Flow Graph

fixed-point solution is closer to the meet-over-path solution and over-approximates the meet-over-path solution, i.e., $s_{mfp} \geq s_{mop}$. In case the lattice is distributive, the two solution coincides, i.e., $s_{mfp} = s_{mop}$. Interested readers may refer to [31] for the proof of this statement.

2.4 Interprocedural Dataflow Analysis

In the previous section, we presented a general framework for a monotone dataflow analysis over a CFG for a single function. Here, it is extended to inter-procedural analysis. As the first step, the CFG is extended to include interprocedural edges, which gives the Interprocedural CFG (ICFG). In an ICFG, the node containing the function call s_c is divided into two nodes s_c and s'_c, where s_c is called as the *call-node* and s'_c is a *return-from-call-node*. A *call-edge* $s_c \rightarrow f_{ENTRY}$ is from the call-node s_c to the entry node f_{ENTRY} of the called function and a *return edge* is from $f_{EXIT} \rightarrow s_{c'}$. In Fig. 2.2, the statements $c0$ and $c1$ are represented by the nodes $c0, c0'$ and $c1, c1'$ respectively and the thick edges denote the call (plain) and return edges (dotted). A special edge, which denotes the control flow between the *call-node* and the *return-from-call-node* is called *call-to-return* edge.

In this case, one can define the transfer functions that propagate the value from a call node to the entry node of the called function, which is called as *context-insensitive analysis*. A context-insensitive analysis will coalesce the value of the parameters from the two call nodes at the entry node of the called function. This makes the solution imprecise as it cannot disambiguate the values coming from the call nodes ($c0$ and $c1$) at the entry node of the called function f_{ENTRY}.

The other approach is *context-sensitive*, where it takes the calling context of the parameters into account. In this case, the analysis will propagate the value of the parameter at the calling context. However, statically determining the runtime value of the calling contexts is not feasible (e.g., recursive functions). Again, one has to restore to approximate the calling context as it is difficult to statically determine the exact set of calling contexts. A popular approach, what we call, *call-strings* where a calling

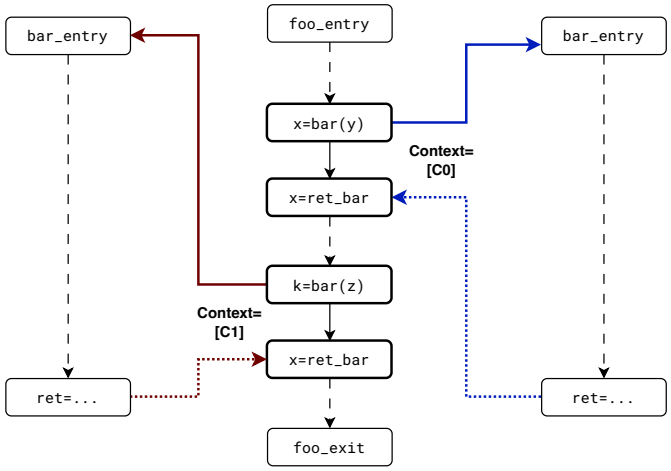


Figure 2.3: Evaluation of CFG on the basis of calling contexts

context is denoted as the state of the runtime call stack. Another approach popular in the case of object-oriented languages is to take the invoking objects as calling contexts, such as `o` in the case of `o.foo()`.

Call String Approach Let $C = \{c_0, c_1, \ldots, c_n\}$ is the set of available calling-contexts in a program. A call-string $cs_k = c_1 c_2 \ldots c_k$ where $c_i \in C$ is a word of length k defined from the alphabets C. Let CS_i is set of all call-strings of length i. A set of call-strings $CS = \{CS_i \mid 0 \leq i \leq k\}$ is the set of call-strings of length i where i is bounded by k. The call-string of the length 0 denotes the entry-point of the analysis, usually the `main` function. In this case, if the $\mathbb{D}$ is the dataflow domain, then the lattice is represented by $(CS, \mathbb{D})$ for the analysis. Consequently, the transfer functions are adapted to the match on the calling context while propagating the value from the calling function to the called function via call-edges and *vice-versa* via the return edges. It facilitates evaluation of the transfer functions based on the calling context as shown in Fig. 2.3.

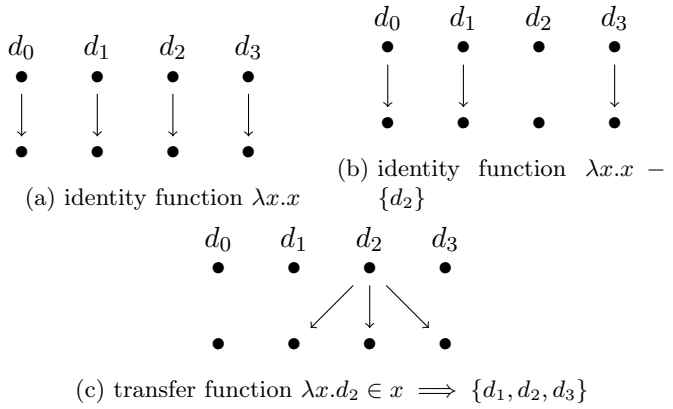
Object String Approach Let $V = \{v_0, v_1, \ldots, v_n\}$ is the set of the invoking objects. On the same lines of call-string, an object-string $os_k = o_1 o_2 \ldots o_k$ is the set of nested invoking objects. A set of object-strings OS_i is the set of object-strings of length i, and $OS = \cup_{0 \leq i \leq k} OS_i$ is the set of object-strings. The rest of the analysis is the same as the call string approach, in this case, the set of call-strings is replaced by the set of object-strings.

Note: Until now, it has been difficult to compare the precision of call-strings over object-strings except a few specialized cases [54] where call-strings have been shown superior to the object-strings approach.

2.5 Interprocedural Finite Distributive Subset

Interprocedural Finite Distributive Subset (IFDS) for an interprocedural control flow graph CFG, domain $\mathbb{D}$ is a class of dataflow analysis which satisfies the following criteria:

- The lattice $\mathcal{S}$ is finite where $\mathcal{S} = 2^{\mathbb{D}}$ for a finite domain $\mathbb{D}$
- The meet operator is either union ($\cup$) or intersection ($\cap$)



(a) identity function $\lambda x.x$

(b) identity function $\lambda x.x - \{d_2\}$

(c) transfer function $\lambda x.d_2 \in x \implies \{d_1, d_2, d_3\}$

Figure 2.4: Representation Relations

- The transfer functions $f : \mathcal{S} \mapsto \mathcal{S} \in \mathcal{F}$ are distributive, i.e., for every $s_0, s_1 \in \mathcal{S}$, $f(s_0 \cup s_1) = f(s_0) \cup f(s_1)$

Definition 5. *IFDS problem is a quintuple $(CFG, \mathbb{D}, M, \mathcal{F})$ where the CFG is an interprocedural control flow graph, $\mathbb{D}$ is the domain of dataflow facts, and $\mathcal{F}$ is a set of functions. A map $M : CFG_{edges} \mapsto \mathcal{F}$ maps the edges in the CFG to the transfer functions.*

The above characterization of dataflow analysis is a special case of the general fixed-point dataflow analysis. However, the novelty of IDFS lies in the fact that it presents an algorithm to compute the minimal fixed point solution through graph reachability. It is achieved by representing the transfer functions as a bipartite graph, the so-called representation relations. Further, a directed graph, *exploded supergraph*, which is computed by the transitive reachability of representation relations, keeps track of the dataflow facts valid at each program location. On reaching a fixed point, an exploded supergraph essentially computes the meet-over-path solution. Owing to the distributive nature of transfer functions, it coincides with a maximal fixed point solution.

A *representation relation* denotes a transfer function as a bi-partite graph defined on $\mathbb{D} \times \mathbb{D}$. The first part denotes incoming data flow facts, while the second denotes outgoing data flow facts. A representation relation specifies the transformations applied by corresponding statements to the set of dataflow facts valid before the statement. Fig. 2.4 shows three examples of these representation relations. In the case of Fig. 2.4a, it denotes the identity function. Each data flow fact is mapped to itself and simply propagates the incoming data flow facts. In the case Fig. 2.4b, each dataflow fact is mapped to itself except d_2. In other words, it removes the d_2 in the resultant if d_2 is reachable, in other cases, it simply propagates the incoming facts. Similary, Fig. 2.4c shows a transfer function that adds $\{d_1, d_2, d_3\}$ if d_2 is reachable.

Definition 6. *An exploded supergraph for a CFG=(loc, E) and a domain of dataflow facts $\mathbb{D}$ is a directed graph (N, E) where $N \subseteq (loc \times D)$ and $E \subseteq N \times N$. A node (l, d), where $l \in loc$ and $d \in D$ denotes that the dataflow fact d is valid at the location l.*

IFDS algorithm is a worklist-based algorithm and only computes the relevant part of the exploded supergraph. The IFDS algorithm initializes it with the entry point of

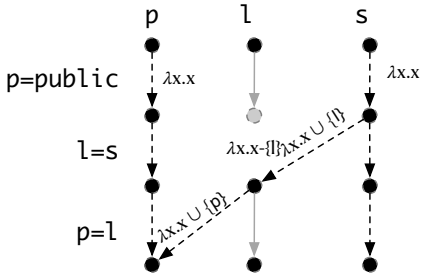


Figure 2.5: IFDS Analysis Example

the CFG and the dataflow fact valid at the initial program location and adds it as a node to the worklist. It then picks up a node from the worklist, and applies the transfer functions to compute the set of reachable nodes. These explored nodes are then added to the exploded supergraph and the worklist. This process continues until it reaches a fixed point, i.e., it is not possible to add any new nodes to the exploded supergraph. Note that, at each step, the representation relation is applied to the set of the valid data flow facts valid at the preceding location. Therefore, a successive application of representation relations computes the composition of representation relations. For brevity, the algorithm is not mentioned here. Interested readers can refer to Reps *et al.* [95] for the algorithm.

IFDS computes function summaries which, are reused later in the analysis to prevent recomputation of the same function. Precisely, a summary is a reachable path from the incoming parameter to the return value. If any such path exists, then it is installed as summary edges at the *call-to-return* edge of the caller function. Later iterations can reuse this summary instead of recomputing the analysis for the called function.

As previously mentioned, IFDS analysis reduces the dataflow analysis problem to a graph-reachability over the exploded supergraph. As the exploded supergraph is explored for each statement in the ICFG, it computes the flow-sensitive dataflow analysis for a class of dataflow analysis problems. Moreover, any dataflow analysis problem, which is expressible in the Kill-Gen framework [95, 31], is also expressible in IFDS. However, many dataflow analyses, such as copy constant propagation or information flow analysis [119], require a more sophisticated lattice that is not a powerset lattice — for example, constant propagation [99] or information flow analysis [119]. This shortcoming of the IFDS is overcome by the Interprocedural Distributive Environment.

Example 1. *Figure 2.5 shows the analysis for the data dependence analysis. The dotted edges and gray edges are the representation relations. The edges are annotated with dataflow functions. Unannotated edges represent the identity function $\lambda x.x$. The dotted edges are part of the exploded supergraph. IFDS analysis starts by initializing the nodes p, l, and s, in the top part of the graph. The bold nodes at each level of the exploded supergraph represent that the variable is valid at that location. Observe that each path in the exploded supergraph denotes a data dependence relation.*

Valid Paths A valid path in an interprocedural CFG is a path in the CFG that respects the function call and returns. An ICFG has *call* and *return* edges to represent a function calls and returns. In figure 2.2, a spurious path in the CFG is the call that follows the call edge of context C_0 and the return edge of context C_1. These paths, if considered will affect the precision of the analysis. IFDS removes these paths from

the analysis by matching the function calls and returns for a precise analysis. A well-known formalization of valid paths is the CFL reachability, where these valid paths are represented in a matching parenthesis, notably a context-free grammar. If each call edge at program location i is labeled with $(_i$ and each return is labeled with $)_i$, then the sequence of such labels over valid call and return edges will be a matched parenthesis. We call these as *same-level valid paths*, as they denote a function call and return pair within the same function. This notion can be extended to the set of valid paths between two program locations s_i and s_j, where s_i and s_j are not in the same function. In this case, it includes the opening parenthesis, $(_i$ but does not include the corresponding closing parenthesis, $)_i$. The set of same-level valid paths, S and the set of valid paths at different levels, V is represented by the context-free grammar given as:

$$S \rightarrow (_i\ S\ )_i\ S \mid \epsilon$$
$$V \rightarrow V\ (_i\ S \mid S$$

Similarly, the *meet over path* solution is extended to the meet-over-valid path solution. Given a set of valid paths in a CFG, a meet over valid paths solution is $\bigsqcup_{p \in ValidPaths} f_p(s_p)$, which the meet over paths solution where the paths are restricted to valid paths. This is interesting to us as IFDS computes the meet-over-valid path solution [95].

2.6 Interprocedural Distributive Environment

Interprocedural Distributive Environment (IDE) analysis extends the capability of IFDS [99] to any dataflow analysis by augmenting the CFG edges with environment transformers, $env(\mathbb{D}, \mathbb{L})$ which are functions from the domain $\mathbb{D}$ to a finite semi-lattice $\mathbb{L}$. The environment transformer are also distributive, i.e., $e_1 \sqcup e_2 = \lambda d \in \mathbb{D}, e_1(d) \sqcup e_2(d)$ for two environment transformers e_1 and e_2.

IDE analysis runs in two phases. In the first phase, it runs IFDS to discover the reachable paths. In the second phase, the analysis accumulates the environment transformers for each path and produces the so-called *Jump Functions*, which are essentially a composition of environment transformers for a particular path. At the program location where two paths meet, the resultant jump function is the join of the corresponding jump functions. In the second phase, it computes the value of the accumulated jump functions.

Example 2. *Information Flow Analysis. An information flow analysis of the information flow labels $\mathbb{L} = L \sqsubseteq H$ for a simple while-language program. The domain $\mathbb{D}$ will be a set of variables. The environment transformers are the functions that map the variable with the corresponding security lattice.*

For example, in Fig. 2.6, assuming a while language with variables and the information flow labels $L, H \in L$. The environment transformers (mentioned along the edges) labels with the corresponding IFC label. The environment transformers are simple identity functions over the labels. id denotes the identity function $\lambda x.x$. In other words, it propagates the incoming value of the computed IFC label. H sets the incoming value

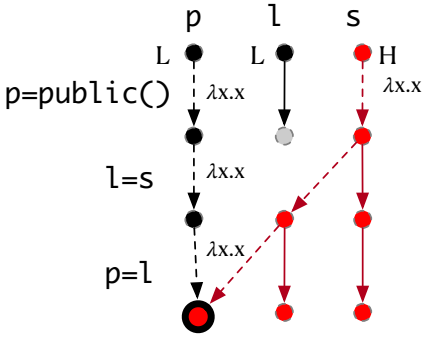


Figure 2.6: IDE Analysis for Information Flow Analysis

to label H. At the last node along the variable p, it determines the IFC function by accumulating and composing the environment transformers along the reachable edges by computing the join of the incoming jump functions. For the straight line path P, it joins the function produced by composing the environment transformers along the paths p ⇢ p and s ⇢ p, which in this case computes H.

IDE also stores the summary edges and corresponding Jump Functions to prevent re-computing the jump function for the same function. In comparison to IFDS, IDE analysis is more powerful as it not only computes reachability but can also be used to compute various other analyses such as type-state [106] or information flow [119]. It also computes the meet-over-path solution [99] like IFDS. In principle, IDFS is a specialization of IDE, where the environment transformers are identity functions.

2.7 Pointer Analysis

Pointer analysis (a.k.a. Points-to analysis) statically computes the set of memory locations or objects referred through a variable. Pointer analysis is a foundational static analysis technique and forms the basis of many other static analyses, such as typestate analysis, dynamic dispatch resolution, call-graph construction, and many others. If a language only contains variables and assignments, then computing a pointer analysis is a simple transitive closure over assignments. However, any practical programming language is far from this ideal. Earlier research has shown that, unsurprisingly, pointer analysis is undecidable [62, 92, 94] and therefore, we need approximations to keep it decidable. A pointer analysis computes a mapping from variables to a set of heap locations called a *points-to set*.

Definition 7. *Points-to Set. Let V be a set of variables defined in a program and H is the set of statically computed heap allocations. Then, a points-to set, $PointsTo : V \mapsto 2^H$ maps V to H.*

2.7.1 Computing Pointer Analysis

Analysis Techniques There are two variants of pointer analysis; one is defined on inclusion-based constraints by Andersen [9], while the other is defined on equality-based constraints by Steengsgard [111]. On an intuitive level, the difference between these two analyses is how they handle operations, such as `x=y`. Steensgard's analysis will combine the corresponding points-to sets into an equivalence class, i.e., $pt(x) \equiv pt(y)$). On the

Operation	Constraint
`x=y`	$pt(y) \subseteq pt(x)$
`x= &y`	$y \in pt(x)$
`x=*y`	$\forall \alpha \in pt(y) \implies pt(\alpha) \subseteq pt(x)$
`*x=y`	$\forall \alpha \in pt(x) \implies pt(y) \subseteq pt(\alpha)$

Table 2.1: Constraints for Andersen's Analysis

other hand, Andersen's analysis will include the points-to set of y, $mathitpt(y)$ to the points-to set of x, i.e., $pt(y) \subseteq pt(x)$. These differences result in the tradeoff between scalability and precision. Andersen's analysis is more precise compared to Steensgard's. Steengsgard's analysis takes linear time in terms of the number of variables, while Andersen's analysis takes cubic time. Naturally, Steensgard's analysis fares better than Andersen's in terms of scalability.

In this thesis, we have used Andersen's analysis. The core of Andersen's analysis is the so-called *cubic algorithm* [78], which resolves the constraints collected from the program statements given in Table 2.1. The first constraint propagates the points-to set of y to x. The second constraint includes the variable y to the points-to set of x. The third constraint transitively propagates the points-to set of the variable α in y to x. The fourth constraint is dual of the third constraint, where it propagates the points-to set of y to the points-to set of the variables α referred by x. As the name suggests, the cubic algorithm takes cubic time in terms of the number of variables to resolve these constraints [78]. Interested readers can refer to the *Static Program Analysis* from Møller and Schwartzbach [78] for the description of the algorithm.

Apart from these techniques, interprocedural pointer analysis requires precise and decidable modeling of other features of programming languages, such as dynamic heap allocation, calling context, and fields of class or structs.

Modelling Heap Allocation As mentioned before, pointer analysis requires approximations to keep it decidable. The first step towards this is statically modeling the heap allocation. The set of heap allocations are statically computed, i.e., these do not represent the exact set of heap allocation in a program, but rather, an abstraction over the heap allocations [91, 104]. A new heap memory can be allocated multiple times within a program as mentioned in 2.1. A heap abstraction technique, *allocation site abstraction*, uses the corresponding program location of the heap allocation to denote the object. For example, in Listing 2.1, it assigns it as `alloc-site-1` and `alloc-site-2` to denote the heap allocations at the site. Another technique is to denote the heap allocation by the corresponding variable, popularly known as *field-based heap abstraction* [105, 58] in the research literature. In field-based heap abstraction, `alloc-site-1` and `alloc-site-2`, the heap allocations are denoted by the variables `x` and `x.next`.

Listing 2.1: Creating a linkedlist

```
1 var x = new ListNode(); // alloc-site-1
2 while (condition) {
3     var y = new ListNode(); // alloc-site-2
4     x.next = y;
5     x = y;
6 }
```

Modelling Context Sensitivity Modelling context-sensitive in the pointer analysis follows the same techniques as for an interprocedural analysis. One approach is to disambiguate function calls based on the call-strings. Assuming that C represents the set of possible call strings, a points-to map in the case of context-sensitive analysis maps the pair of variables and function call contexts (V, C) to the set of heap allocations 2^H. One could also augment the allocation site with the calling context to disambiguate heap allocations occurring in function calls (e.g., factory methods). In this case, the points-to map from the pair of variables in a calling context and heap-allocations in a calling context (V, C) to $(2^H, C)$.

Modelling Field Accesses A pointer analysis can choose to include field variables to disambiguate variables with the field variables. A field-sensitive analysis disambiguates the variable and its fields. Field access `x.f` and `x.g` will be treated as separate variables in case of a field-sensitive analysis. On the other hand, a field-insensitive analysis will only consider the base variable `x` [10] for such variables, at the cost of precision. Another abstraction, which is popular in languages such as JavaScript [30], is a *field-based abstraction*. It only considers the field name but coalesces the variables, thereby distinguishing based on the field name and ignoring the variables.

Part II

Analysis of Android WebView

Chapter 3

Empirical Analysis of Hybrid Apps

3.1 Overview

Hybrid communication leverages APIs like the *loadUrl* or *evaluteJavascript* methods, which, from inside an Android application, can either load a webpage into the WebView or execute Javascript code directly. To improve comprehension of hybrid communication we propose LUDroid, a framework supporting our semi-automatic analysis of hybrid communication on Android. We extract the information flows from Android to hybridization APIs and thus to the JavaScript engine to be executed in the displayed web page (if any), and categorize these flows into benign and transmitting sensitive data. We discover 6,375 sensitive flows from Android to Javascript.

Due to the fact that hybrid apps combine native and web technologies in a single app, the attack surface for malicious activities increases significantly, as potentially untrusted code loaded at runtime, can interfere with the trusted Android environment. In our study with applications from the Google Play Store, we find that 68% of these apps use at least one instance of WebView and 87.9% of these install an active communication channel between Android and JavaScript. This includes exchange of various pieces of sensitive information, such as the user's location. To assess the impact on user privacy a standalone analysis of the Android or Javascript side is thus clearly insufficient. However, very limited work towards linking the assessment of both worlds can be found in the literature. Initial approaches either focus on type errors during hybrid communication [63] and/or only consider a very specific vulnerability arising of hybridization [55, 102, 97, 51, 75, 33, 128].

3.1.1 WebView Interfaces

The WebView classes provides foreign function interfaces to interact with the Web components. Briefly, these APIs come in two variations:

- `loadUrl(url)` and `loadUrl(url, httpHeaders)`. These FFI calls load a remote or local webpage. The local webpage must reside within the file structure of the app. One can also specify *JavaScript* code in the `url` string as `javascript:<source code>`. This lets the WebView know that the URL is a JavaScript code snippets and it runs the specified code. However, in this case, it is not possible to the capture the return value from the JavaScript code, which is supported by `evaluateJavaScript`.

Listing 3.1: An example of WebView

```
1 class MainActivity {
2     protected void onCreate(Bundle savesInstanceState) {
3         WebView wv = (WebView) findViewById(R.id.webview);
4         WebSettings webSettings = wv.getSettings();
5         webSettings.setJavaScriptEnabled(true);
6         wv.addJavascriptInterface(new DeviceProp(), "Android");
7     //case 1: invoke Javascript from Android
8         wv.evaluateJavascript(
9         "const xhr = new XMLHttpRequest();
10        xhr.open('POST', 'http://www.leaker.com');
11        xhr.send('id=Android.getDeviceId()');");
12      /* case 2: load a webpage (potentially executing JavaScript), the
          object "Android" persists as property of the DOM's global object */
13        wv.loadUrl( "http://www.somepage.com");
14    }
15 }
16 class DeviceProp {
17    // expose this function to JavaScript
18    @JavascriptInterface
19    public String getDeviceId() {
20        var tManager = (TelephoneManager)
      ctxt.getSystemService(Context.TELEPHONY_SERVICE);
21        var uid = tManager.getDeviceId(); // get the device ID
22        return uid;
23    }
24 }
```

- `evaluateJavascript(source, returnValue)`. This FFI call makes it convenient to run the specify the JavaScript code to run within the WebView. The first argument source specifies the JavaScript source code and returnValue specifies the callback method to capture and process the return value from JavaScript. If returnValue is a null value then it ignores the return value from JavaScript.

3.1.1.1 Example – WebView

Listing 3.1 depicts a simple hybrid Android app with the communication between the Android and WebView component. Line 3 retrieves a *WebView* object *wv* is retrieved from the Activity's UI via its identifier (line 3). By default execution of JavaScript in a *WebView* object is disabled but can be enabled by overriding the WebView's default settings (line 5). *WebView* provides the means to create a Java interface object that is shared with the WebView and can be accessed via JavaScript. Thus, the Android app's capabilities can be bridged to the Web component via *bridge communication* to, e.g., provide access to various sensors' data. In our example, an object of the class *DeviceProp* (see Listing 3.1) is shared (Listing 3.1, line 6) with the *WebView* object *wv* such that every webpage loaded into *wv* can use this object via the global variable *"Android"* (i.e. "Android" becomes a persistent property of the DOM's global object). Finally, the method *loadUrl* can be used in two ways: (1) to invoke JavaScript code directly (prepending a *javascript:* tag to the passed code) from Android, and (2) to load a custom URL (line 12) (which again may execute Javascript code specified in the web page).

Consider line 8 in Listing 3.1, which reveals that the *loadUrl* method is invoking the

getDeviceId method defined in the *Leaker* class, line 19). This Java method retrieves the Android device's unique ID and returns it to the JavaScript code. Similar Javascript code could also be invoked in the loaded webpage (Listing 3.1, line 13) where it might be leaked to some untrusted web server together with more information the user enters into the web page. Note that state-of-the-art information flow analyses for Android cannot report these flows as they have no information whether the WebView's code actually leaks the shared data (even worse, many do not even consider *loadURL* a sensitive information sink [93]). To further investigate this scenario, access to this webpage is required. Static analysis of these scenarios is non-trivial as any static analysis of JavaScript code is challenging due to its highly dynamic nature [114] and as it additionally requires a careful inspection of the various aspects of the *WebView* class and its bridge mechanism. Therefore, it becomes critical for program analyses to fully understand the behavior of *loadUrl* and *addJavascriptInterface*. The aim of our work is to provide this information by performing a large scale study of real-world apps.

3.2 Contributions

Previous works [63] take a first step into analyzing the data flows from Android to JavaScript but it is far from sound and mostly concentrates on potential type errors when passing data between the two worlds. These analyses such as, BabelView [97], BiFocals [22], and HybridDroid [63], among others [51, 102, 65, 48], focussed only on a specific vulnerability. *BiFocals* (Chin and Wagner) focussed on excess authorization and file-based cross-zone scripting. These attacks expose device file systems to the attacker. *BabelView* (Rizzo et al.) studies the impact of code-injection attacks on Hybrid apps. *HybridDroid* (Lee et al.) focusses on detecting type mismatch bugs in the Android hybrid apps. Similarly, Hassanshahi et al. studied the possibility of web-to-app injection attacks. To better understand the picture of interaction between Android and JavaScript code, a thorough understanding of the methods *addJavascriptInterface*, *loadUrl*, and *evaluateJavascript* is required. We are interested particularly in the uses and potential abuses of this interface in the wild and their implications on the design of a program analysis for hybrid apps.

To start with the study, we developed a tool called *LuDroid* which statically analyzes these apps for the presence of the bridged methods in these apps and extracts the URL and JavaScript code passed through *loadUrl* and *evaluateJavascript* calls. Then, we select a set of benign apps from Google play store and a set of malware apps (from AMD Dataset) for the study to build a corpus of the information extracted by LuDroid, which was subsequently used for a manual analysis. We manually analyzed the extracted URLs and studied their characteristics. Again, from the extracted JavaScript, we aimed to find a common patterns that existed in these apps.

The salient observations obtained from our study are:

1. We observed that both benign apps and malware apps leverage WebView communication. Further, WebViews are frequently used in libraries in both, benign and malware apps.

2. We observed that many apps include third-party libraries through unsecured protocols. This makes it possible to launch simple man-in-middle attacks.

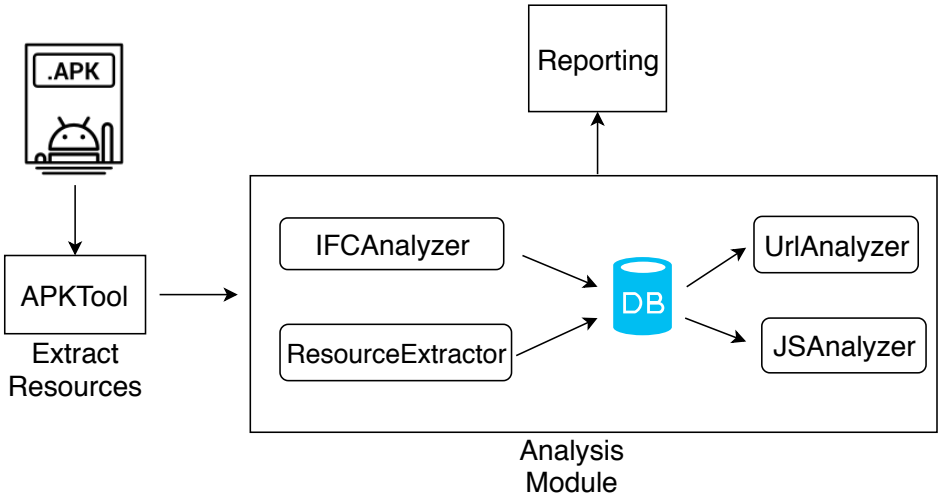


Figure 3.1: The workflow of LUDroid

3. The presence of bridge methods that acts like *setter* methods used to assign values to field in a class is common. Unfortunately, static analysis cannot handle these methods.

4. We identify bridge methods *potential* user-sensitive information flow originating from these methods. These methods could be invoked by the JavaScript code. In this work, we assume that these methods are likely candidates of the data leak.

Remaining of the chapter describes the *LuDroid* tool build for the purpose of the study, the dataset chosen for the study, and presents the results obtained form *LuDroid* on the chosen dataset.

3.3 Methodology

To discover the usage of *loadUrl* and *evaluateJavascript*, we developed the tool chain *LuDroid*. Figure 3.1 presents the workflow of LUDroid's analysis framework which consists of four modules: (1) ResourceExtractor, (2) IFCAnalyzer, (3) JSAnalyzer, and (4) URLAnalyzer. LuDroid decompiles the android app using APKtool [98] to extract the source code of the app in Smali IR [46]. IFCAnalyzer and ResourceExtractor works on the smali module to extract the Bridge interfaces and corresponding security violations, while the IFCAnalyzer and URLAnalyzer works on the extracted information from IFCAnalyzer and ResourceExtractor. In the sequel, we describe each of these modules.

3.3.1 IFCAnalyzer

This module checks for the presence of data originating from sensitive sources in the bridge interfaces. It leverages the sensitive sources defined by [93] to identify the sensitive sources in Android.

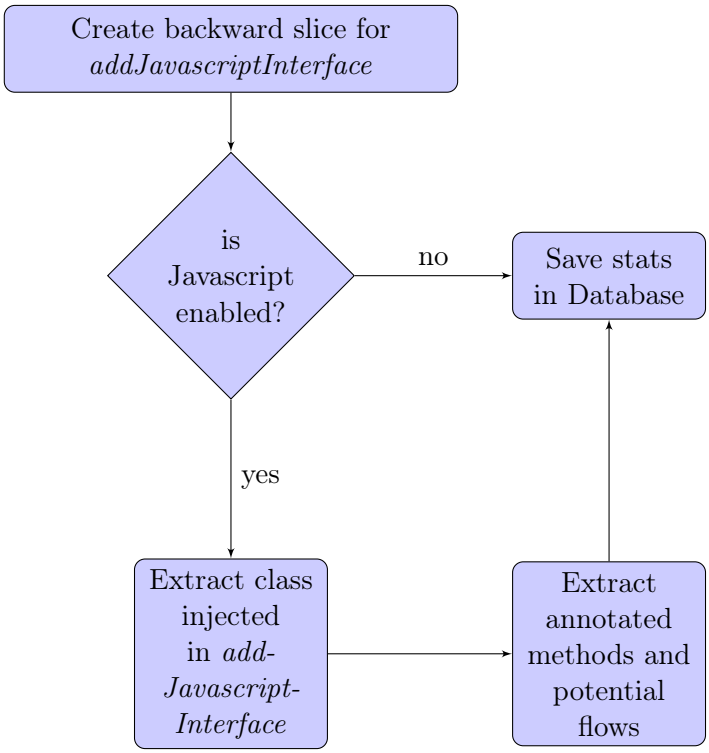


Figure 3.2: The workflow of IFCAnalyzer

The high level workflow of the *IFCAnalyzer* module is illustrated in Figure 3.2. To locate the *WebView* initialization corresponding to each occurrence of the *addJavascript-Interface* method, we compute its backward slice. By using the *addJavascriptInterface* method, the Android object is injected into the relevant *WebView*, with the Java object and the associated object name as its two parameters. When *Javascript* is enabled on the *WebView*, web pages can access and execute methods exposed by the shared Java object (see Listing 3.1). Our focus is on examining the functionality made available by this Java object, and to achieve this, we extract the corresponding class and its supported methods. It is essential to note that not all methods of the Java object are bridged; only those annotated with *@JavascriptInterface* are accessible to Javascript. Our next step involves identifying potential information leaks that may originate from these methods, as Javascript could invoke them and inadvertently expose sensitive information. At the last, we store the analysis results in a database.

3.3.2 ResourceExtractor

The *loadUrl* method loads a specified URL given as string parameter (cf. Listing 3.1, line 13). If the input string starts with *"javascript:"*, the string is executed as JavaScript code (cf. Listing 3.1, line 8). Similarly, *evaluateJavascript* executes the specified JavaScript code. The aim of this module is to extract the URLs and Javascript code passed to these FFI methods. To this end, we create an intra-procedural backward slice on *loadUrl*, extract the input strings, and store them along with their originating class' name. As input strings are often constructed via various String operations, (e.g., using *StringBuilder* to concatenate strings), we extend LUDroid with domain knowledge on the semantics of the Java String (and StringBuilder) class. LUDroid understands the Smali signature of String methods and applies partial evaluation to infer strings created

by various String manipulation methods. However, at the moment we do not support complex string operations such as array manipulation. It is also possible to extend the support for such operations and simple obfuscations. However, in this work, we concentrate on the features of the parameters that can be extracted with reasonable effort. The output of this module is fed to the *UrlAnalyzer* and *JSAnalyzer* modules to interpret and categorize the URLs and Javascript.

3.3.3 JSAnalyzer

JSAnalyzer uses the strings constructed by *ResourceExtractor* and builds a database summarizing the patterns used in raw javascript passed to *loadUrl*. The results in the database are further manually analyzed for the features mentioned in the subsequent paragraphs. The components of JSAnalyzer primarily consist of scripts for automation and reporting. We look for the following properties:

- **Information Flow from Javascript to Android.** The Android SDK allows app developers to annotate *setter* methods with *JavaScriptInterface*. This supports the reuse of existing web-based functionality in Android by transmitting the results from a web-based/JavaScript method to the Android object. It creates an information flow from the external web application to the Android app. In this paper, we identify use-cases of this behavior.
- **Obfuscated and unsecured Code.** Many third-party library developers use code obfuscation to protect their intellectual property. It is also possible to inject remote third-party libraries in JavaScript using unsecured protocols such as HTTP. In this work we identify the patterns in which these libraries are obfuscated or used insecurely.
- **Pass Native Information to Third Parties.** Many apps pass device specific native information to third-party libraries. This user-sensitive information is leveraged by third-party libraries to enhance their services. However, it can be detrimental to the privacy of the user. In this work, we identify cases of passing sensitive information to third-parties.

3.3.4 URLAnalyzer

URLAnalyzer has two functions: (1) it checks the validity of a URL, and (2) extracts its essential features. *URLAnalyzer* reads the passed URL inputs from *ResourceExtractor* and parses them following the RFC 3986 [17] standard. In case the URL is malformed, i.e., it does not adhere to RfC 3986 standard, it reports an error. We obtain the URLs information such as protocol, host, port, search, fragment. Table 3.1 mentions the fields of the extracted data. In addition, it also categorizes the URLs used within the SDKs. These libraries use *loadUrl* to load their custom URLs and provide the intended functionality to other app developers, e.g., Facebook SDK for Android provides Facebook authentication service to other apps.

In this thesis, the discussion is limited only to those URL with `file://` protocols where HTML and JavaScript code bundled in the apps. Other aspects, such as SDKs leveraging FFI calls, unencrypted communication in apps is mentioned in the Appendix A for our readers.

Field	Description
Protocol	The application layer protocol used within the URL.
Host	This can either be a fully qualified domain or an IP address of the corresponding host.
Port	The port (if specified) of the host to which the request is sent.
Path	The path (if specified) on the host the request is sent to. Such a path can for example be specified for HTTP or FTP URLs, but also for local file URLs.
Search	The search part (if specified) of HTTP URLs. This is the remainder of a HTTP URL after the path, e.g., "?x=5&y=9".
Fragment	The fragment is an optional part of the URL that is placed at the end of the URL and separated by a #.

Table 3.1: Data extracted from URLs by URLAnalyzer

Category	#Apps Studied	Category	#Apps Studied
Banking	6	Music	13
Business	10	News	19
Education	8	Shopping	17
Entertainment	16	Social	11
Health	10	Travel	9
Online Payments	25		
Top Grossing	30	Top Apps	22
	Total	196	

Table 3.2: Top apps by categories

3.4 Dataset

We curate four different datasets containing both benign and malware apps. Our rationale to study both is the following: (1) the analysis of benign apps conveys information on how developers use Android-Web hybridization in practice, and (2) analysis on malware provides relevant insights on the use of the feature for malicious purposes. The former exhibits patterns which can be dangerous and potentially exploited. The latter demonstrates that these exploits have been exploited in practice.

We select the benign apps from the Google Play Store and malware from the Argus AMD dataset (Wei et al.). To obtain the benign apps, we crawled the Google Play Store based on the criteria (A) and (B) below. The Argus AMD dataset [122] is the state of the art malware dataset available to the best of our knowledge. In what follows, we describe the datasets chosen for our study.

(A) **Benign apps.** In this dataset we randomly chose 7,500 apps published on the Google Play Store between 2015-2019.

(B) **Frequently Used Apps.** Here, we selected a total of 144 of the top downloaded apps from 11 app categories on the Google Play Store from the following categories: Banking, Business, Education, Entertainment, Health, Music, News, Online payments, Shopping, Social, and Travel. These categories of apps that were among the highest downloaded on Play Store. In addition, we also selected the top downloaded apps (referred as top apps) across all categories and apps that yield the most revenue (referred as top grossing), consisting of 52 apps. In total, we curate 13 categories of apps. Table 3.2 lists these categories together with the apps in each category as of Dec 2019, when they were downloaded.

(C) **Malware:** The Argus AMD dataset contains 24,553 samples from 71 different families of malware. The number of malware in each family range from 4 to 7843. To have a representative from each malware family, we chose at least one from each family, but choosing up to approx. 20% of the malwares from each family. In total, we choose 1000 malware apps for the study.

We present the results of our study in the sequel. All experiments were performed on a MacBook Pro with a 2,9 GHz Intel Core i7 processor, 16 GB DDR3 RAM, and MacOS Mojave 10.14.1 installed. We used a JVM version 1.8 with 4 GB maximum heap size.

3.5 Java-JavaScript communication in Benign Apps

Table 3.5 illustrates the distribution of apps based on different characteristics of hybrid communication. Out of 7,500 apps analyzed, a substantial 68% of them are classified as hybrid apps as they utilize WebView. Of these, 87.9% enable JavaScript (JavaScript), while the remaining 12.1% solely use WebView for static webpages, where JavaScript is not enabled. Among the hybrid apps that enable JavaScript, half of them employ the *addJavascriptInterface* method to create an interface between Android and JavaScript, allowing the transfer of information from Android to JavaScript. A total of 30% of the entire dataset and 43% of the hybrid apps involve the transmission of information from the Android platform to JavaScript.

Table 3.6 displays the top ten app categories along with the types of information they share with JavaScript. However, it is important to note that this research does not delve into what happens to this data on the JavaScript side, specifically whether it might inadvertently leak to untrusted entities. The primary objective is to identify real-world scenarios that necessitate consideration when devising an analysis for hybrid apps.

We observed 6375 sensitive information flows from Android to JavaScript in 18% percent of the total apps in our dataset. The JavaScript code in an app could arise from third party. Out of these, 18% of these flow to URLs located inside the app, i.e., using the *file* protocol. We observed that 82% (or more) of the sensitive information flows could leak to potentially untrusted code. A list of 10 randomly selected apps along with their corresponding components and shared sensitive information from each category (mentioned in Table 3.6) has been shown in the Table 3.4. The majority of these flows include location information, network information and file system access.

Table 3.3: App components with the shared sensitive information (from Android to JavaScript)

App Name	**Category**	**Component Name**	**Information shared**
Instagram	Social	BrowserLiteFragment	Cookies
TASKA AR MARYAM	Entertainment	Map26330	Location (GPS)
Classical Radio	Musik & Audio	MraidView	External storage file system access, Network Information
BLive	Lifestyle	LegalTermsNewFragment	Location
Cat Dog Toe	Board Games	appbrain.a.be	Location, Network Information
N.s.t. A-Tech	Communication	ax	Location, Network Information
Pirate ship GO Keyboard	Personalization	BannerAd	Device ID, Device's Account information, Locale
IQRA QURAN	Books & Reference	Map26330	Location
Logic Traces	Puzzle	SupersonicWebView	Location
FLIR Tools Mobile	Productivity	LoginWebActivity	Network Information

Table 3.4: List of 10 randomly selected apps along with their corresponding components and shared sensitive information

Property	Number of benign apps (total=7500)	Number of Malware Apps (total=1000)
Uses WebView	5,083 (≈ 68%)	572 (≈ 57%)
Enables JavaScript	4,469 (≈ 60%)	572 (≈ 57%)
Injects bridge interfaces	2,256 (≈ 30%)	451 (≈ 45%)

Table 3.5: Hybrid API usage over 7500 benign apps and 1000 malware apps)

Table 3.6: Top ten app categories with type of information shared from Android to JavaScript

App category	Type of information
Social	Cookies, File system
Entertainment	Account Information, File system, Network Information, Location
Music & Audio	Account Information, File system, Network Information
LifeStyle	Activity Information, Application level navigation affordances, Locale
Board Games	Date and Time, Location, Network Information
Communication	Activity Information, File system, Location, Network Information
Personalization	Activity Information, Account Information, File system, Location
Books & Reference	File System, Location, Network Information
Puzzle	File System, Internal Memory Information, Location
Productivity	File System, Internal Memory Information, Network Information

3.6 Java-JavaScript communication in malware apps

Table 3.5 (column 2) depicts the usage of loadUrl features in malwares. Out of the 1000 samples, we observed 57% of these apps use WebView. All of the apps that uses WebView enable JavaScript by default, and 45% of apps injects bridged interfaces defined in Android in WebView.

3.6.1 Patterns of JavaScript usage in Benign Apps

Here, we present the insights on JavaScript code passed to FFIs, i.e., *loadUrl* or *evaluateJavaScript*, identified by *LuDroid*. In our chosen dataset, the JavaScript code passed to *loadUrl* subsumes the JavaScript code passed to *evaluateJavaScript*.

3.6.2 Frequent JavaScript Code in Benign Apps

From the obtained JavaScript code in our dataset, we observe the following: (1) 64% of the the code contain event-driven functionality with the use of JavaScript Event [1] API,

(2) 31% of the code pertains to modification of DOM models, and (3) the remainder 5% construct JavaScript code through inter-procedural function calls. LuDroid cannot construct the Interprocedural strings which makes it unsound. We then further extract a total of 73 distinct type-2 clones, syntactically indentical copy where identifiers have been modified. Given, such as low number of the distinct type-2 clones, it is not surprising to say that most of these originate from third-party libraries.

Program Fragment	**%age of Apps**	**Comments**	**List of selected apps**
`function(){ var event = document.createEvent('Event'); event.initEvent( 'fbPlatformDialogMustClose' ,true,true); document.dispatchEvent(event); }`	32.25	Code from Facebook SDK	com.tivion.usa com.com014.lotto com.t1gamer.lmcs com.ape.games.nlu
`window.HTMLOUT.processHTML( document.getElementById('SearchResults') .innerHTML );`	16.65	Injecting HTML in bridge object *HTMLOUT*	com.www.Fishingmac com.tc.veda com.alx.roseslwp com.mediaapp.mex com.calcrate
`function actionClicked(m, p) { var q = prompt('vungle:' + JSON.stringify({ method: m, cparams: (p ? p : null) })); if (q && typeof (q) === 'string') { return JSON.parse(q).result; } }; function noTapHighlight() { var l = document.getElementsByTagName('*'); for (var i = 0; i < l.length; i++) { l[i].style.webkitTapHighlightColor = 'rgba(0,0,0,0)'; } }; noTapHighlight(); if (typeof (vungleInit) == 'function') { vungleInit($webviewConfig$); }`	4.33	Code from Vungle advertising library	Lucky-Wheel Booboo Guess-The-Flag com.ignm.gesu
`javascript:window.SynchJS.setValue ((function() {try{return JSON.parse(Sponsorpay.MBE. SDKInterface.do_getOffer()).uses_tpn;} catch(js_eval_err){return false;}})());`	1.9	Bridged Communication with Sponsorpay SDK	com.wTieugiano SocialNetwork-Circus BBCNews-Pashto WoRSA

Table 3.7: Frequent resolved javascript code

Table 3.7 lists the four most frequently used JavaScript code fragments and their usage. The most commonly used JavaScript code snippet in our dataset originates from the Facebook SDK. More than 32% of apps, in our dataset, have used it at least once. This code snippet provides the functionality to authenticate with Facebook on the web in case the Facebook mobile app is not present in the user's device. The second most

Listing 3.2: Dynamic Script Loading in loadUrl

```
1 javascript: (function() { var script=document.createElement('script');
2 script.type='text/JavaScript';
3 script.src='http://admarvel.s3.amazonaws.com/js/admarvel_mraid_v2_
      complete.js';
4 document.getElementsByTagName('head').item(0).appendChild(script);})()
```

Listing 3.3: Modifying the bridged Android object named SynchJS

```
1 javascript:window.SynchJS.setValue((function(){
2     try{
3         return JSON.parse(Sponsorpay.MBE
      .SDKInterface.do_getOffer()).uses_tpn;
4     }catch(js_eval_err){
5         return false;
6 }})());
```

common JavaScript code snippet exhibits a peculiar case. In this snippet, a DOM element modifies an Android object. We describe further details in Case Study 3.6.2.2. Similar functionality is manifested by the fourth JavaScript code snippet. In this case, a JavaScript code snippet, originated from an external SDK, modifies an Android object by invoking the setter method. In particular, both cases manifest an interesting scenario where JavaScript modifies Android objects. Finally, the program in the third row originates from the advertisement library Vungle. Details regarding this case are described in Case Study 3.6.2.3.

We describe the four interesting cases relevant to understand developers' intentions.

3.6.2.1 Case Study: Third-party script injection in loadUrl

We identify cases of the third-party script injection in 3 of the 73 code clones. Listing 3.2 illustrates one such script. Instances similar to this script are present in 8.33% of the analyzed apps. The script loads third-party JavaScript code by injecting a script tag into the header of the displayed webpage, resulting in a modification of the global state of the page. In this example, the developers used an unsecured protocol (*HTTP*, cf. Line 3, Listing 3.2). This scenario makes the webpage and thus the whole Android app susceptible to a man-in-the-middle (MITM) attack, where an attacker can intercept the connection and replace the script loaded from `script.src` with malicious JavaScript. Unfortunately, in the case of hybrid apps, this script runs in the trusted sandbox of the app and it is completely oblivious to the user that the script can be replaced by the attacker.

Developers load remote scripts using unsecured protocols which might violate the integrity of the app.

3.6.2.2 Case Study: Information flow from JavaScript to Android

We found cases of dataflow from JavaScript to Android in 8.7% of the investigated apps which is unsurprisingly as "setter" methods can be interfaced via the bridge object to JavaScript. Listings 3.3 and 3.5 display examples of this behavior.

Listing 3.4: Excerpt of source code for SynchJS object in Listing 3.3. Source: Github [34]

```
1 public class SynchronousJavascriptInterface {
2     // JavaScript interface name for adding to web view
3     private final String interfaceName = "SynchJS";
4     private CountDownLatch latch; // Countdown latch to wait for result
5     private String returnValue; // Return value to wait for
6     public String getJSValue(WebView webView, String expression) {
7       latch = new CountDownLatch(1);
8       String code = "javascript:window." + interfaceName +
            ".setValue((function(){try{return " + expression +
            "+\"\";}catch(js_eval_err){return '';}})());";
9       webView.loadUrl(code);
10      try { // Set a 1 second timeout in case there's an error
11        latch.await(1, TimeUnit.SECONDS);
12        return returnValue;
13      } catch [...] return null; }
14    // Receives the value from the JavaScript.
15    public void setValue(String value) {
16      returnValue = value;
17      try { latch.countDown(); } catch (Exception e) {}
18    }
19 }
```

Listing 3.5: Information Flow from JavaScript to Android

```
1 (function() {
2   var metaTags=document.getElementsByTagName('meta');
3   var results = [];
4   for (var i = 0; i < metaTags.length; i++) {
5     var property = metaTags[i].getAttribute('property');
6     if (property && property.substring(0, 'al:'.length) === 'al:') {
7         var tag = { "property": metaTags[i].getAttribute('property') };
8         if (metaTags[i].  hasAttribute('content') ) {
9             tag['content'] = metaTags[i].getAttribute('content');
10    }
11    results.push(tag);
12  } //if end
13 } //for end
14 window.HTMLOUT.processJSON(JSON.stringify(results));
15 })()
```

Listing 3.3 leverages a synchronous communication channel from Android to JavaScript and back. The method *setValue()* which is a setter method defined in the class *SynchronousJavascriptInterface*, excerpted in Listing 3.4, is invoked through a bridged object *SynchJS* . Note that the code of Listing 3.3 is generated in the method *getJSValue* (line 6), where Android executes the parameter expression in the context of the WebView and waits (line 11) for the thread evaluating the JavaScript code to invoke the bridged *setValue* method. Line 3 in Listing 3.3 reads the field *uses_tpn* of an object deserialized from a third-party library method `Sponsorpay.MBE.SDKInterface.do_getOffer` and passes that value to the setter method in *SynchJS*. When this method is invoked inside the WebView's thread, the field *returnValue* is changed at line 16 of Listing 3.4. The implementation then notifies Android's UI thread via a call to *latch.countDown()*, which basically implements a simple semaphore such that the waiting Android thread can continue its execution and return the value retrieved from the WebView (line 12).

Listing 3.5 writes meta-tag information to an Android object. It first constructs the array with properties and content (Line 5-Line 11). This array is then serialized to a

Listing 3.6: Representative Java listing called from *onPageFinished()* in *Liberty Education App*

```
1 public void process() {
2     ....
3     WebView v0 = p0.viewFinished;
4     ...
5     String v1 = "javascript:..."; //String from Listing 3.5
6     v0.loadUrl(v1)
7     ...
8 }
```

string in JSON notation (Line 14) and subsequently passed through Android through the bridge object *HTMLOUT*. Note that *processJSON* method runs in a different thread than the regular Android Code [35], it requires synchronization with the WebView thread.

Android WebViews feature an event system that reacts to many different events in the WebView. The Android SDK allows developers to override the default *WebView chromeless browser* window and specify their own policies and window behavior through extending a Java interface called *WebViewClient*. Interestingly, we also identified many similar code fragments during handling of *WebViewClient* events. As an example, developers can modify the behavior when e.g. the WebView client is closed. Our study found that many developers transfer results from JavaScript to Android after the WebView client terminates by modifying the *onPageFinished()* method in the *WebViewClient* interface to invoke JavaScript. Listing 3.6 shows an example taken from the app *Liberty Education* where the method *process()* is called from method *onPageFinished()* by a series of method calls.

Our study shows intricate cases of using setter methods to permit non-trivial dataflow from JavaScript to Android, in some cases even using (required) synchronization.

Restricting the bidirectional communication impacts the flexibility provided by WebView to developers. Instead static analysis techniques could be leveraged to detect and report similar insecure data flows. However, a simple context-insensitive static analysis on Listing 3.3 using approaches such as HybridDroid [63], or Bae et. al. [15] will be unsound. The unsoundness stems from the analyses' limitation to analyze the described callback communication methods, thus only supporting one-way communication from Android to JavaScript. A precise and sound static analysis would need to consider these non-trivial methods of callback communication that establish a two-way communication channel.

3.6.2.3 Case Study: Device Information to Third-party

This case study illustrates leakage of device information to third-party libraries. Listing 3.7 is taken from the advertising library *Vungle*. Line 10 removes the highlight color from each element. Therefore, the function *noTapHighlight()* makes the app susceptible to a confused-deputy attack such as clickjacking. It obscures user clicks, making users click on advertisements without their knowledge. Additionally, Line 14 can potentially leak Vungle's *WebView* configuration object, which contains identifiable information of a device, to some web server, in this example through the function *vungleInit()*. *Web-*

Listing 3.7: Leaking Sensitive Information

```
function actionClicked(m,p) {
    var q = prompt('vungle:' + JSON.stringify({method:m,
        params:(p?p:null)}));
    if(q && typeof(q) === 'string') {
        return JSON.parse(q).result;
    }
};
function noTapHighlight(){
    var l=document.getElementsByTagName('*');
    for(var i=0; i<l.length; i++){
      l[i].stylewebkitTapHighlightColor=  'rgba(0,0,0,0)';
    }
};
noTapHighlight();
if (typeof vungleInit == 'function') {
    vungleInit($webviewConfig$);
}
```

Listing 3.8: Complex control flow via JavaScript

```
javascript:(function() { Appnext.Layout.destroy('internal'); })()
```

View settings contain sensitive information about the host device that is also used by *WebViewClient*.

3.6.2.4 Case Study: Code Obfuscation in Third-Party Libraries

This case study shows an interesting obfuscation pattern using *loadUrl* to deliberately prevent program analyzers from inferring the intended functionality. Need for obfuscation arises from concerns about safeguarding the intellectual property, or from trying to hide debatable or, worse, malicious behavior.

Appnext is an ad-library which is widely used for app monetization. Listing 3.8 shows a code snippet found in its library code. In this code a Java object *Appnext* is being bridged and used in JavaScript invoked from Android. This functionality could have been directly implemented in Android/Java itself. It is unclear why the programmers chose to implement it by crossing a language-bridge from Android to JavaScript and back, which is even more expensive as an *eval* in JavaScript, instead of the direct invocation. We have discovered this pattern in 25% of the apps, which makes this potential obfuscation pattern prevalent among Android apps. In this case, the anonymous function can be directly replaced by the function call.

In our study, we found an instance of obfuscation where the *WebView* class itself is obfuscated. Various libraries inherit from the *WebView* class and define their own subclass to access *WebView*s functionality. These subclasses are then obfuscated using a code-obfuscator tool. Listing 3.9 shows an instance from the library code *Ad-Locus*. Listing 3.9 defines a class `AdLocusAdapter` which contains the obfuscated method names such as `a(),b()` and `c()`. To improve precision, static analysis needs to consider these libraries and especially the obfuscation patterns present in libraries.

By adding another layer of complexity to inter-language analysis, obfuscation increases the difficulty for program analysis tools to infer the actual functionality. A precise and sound analysis of these patterns is required for useful analysis results. Ob-

Listing 3.9: Representative Java listing of obfuscation in Library Code-AdLocus

```
1 package com.adlocus.adapters;
2 public class AdLocusAdapter {
3     ...
4     protected final WeakReference a;
5     ...
6     private WebView b;
7     ...
8     //direct methods
9     ...
10    public AdLocusAdapter(AdLocusLayout v) { ... }
11    private static WebView a(AdLocusAdapter v) { ... }
12    private void a(AdLocusLayout v) { ... }
13    ...
14    //virtual methods
15    public void a() { ... }
16    public void b() { ... }
17    public void c() { ... }
18    ...
19 }
```

Program Fragment	**%age of Apps**
`document.loginForm.j_password.value = "...";`	9
`document.getElementById('access-pin').value = "...";`	9
`document.getElementById('access-code').value = "...";`	9
`document.logon1.PASSWORD1.value = "...";`	9
`document.form1.selBankID.value = "...";`	5
`document.form1.consumerEmail.value = "...";`	2

Table 3.8: JavaScript code passed to evaluateJavascript . Sensitive values are masked by . . . to protect the confidentiality of the information in these apps.

fuscation patterns in Android apps are discussed in detailed in a recent large scale study [125].

We also analyzed the most frequently used apps from the Google PlayStore to study the JavaScript code passed to the method evaluateJavascript . In particular, we found differences between the usage of evaluateJavascript and *loadUrl* in these apps. In Section 3.6.4, we identified some usage patterns of *loadUrl*, many of which similar to the benign apps in Section 3.6.2. However, in the top 196 apps, our study found a significant difference between the usage of evaluateJavascript and *loadUrl*. We found that 26 of these 196 apps use at least one instance of evaluateJavascript . These usages contain 158 instances of JavaScript code fragments, forming 16 type-2 code clones. In what follows, we elaborate two cases of frequent JavaScript code in these apps. We also observe an unusual case which – to our surprise – violates basic security practices.

In this section we present the insights of our study of *loadUrl* and *evaluateJavascript* on frequently used apps. We collected the most frequently used 196 apps from the Play Store and categorized it into 13 categories. These categories include the most downloaded apps (referred to as Top Apps) and apps that yield high revenues (referred

Category	#Apps Studied
Banking	6
Business	10
Education	8
Entertainment	16
Health	10
Online Payments	25
Music	13
News	19
Shopping	17
Social	11
Top Grossing	30
Top Apps	22
Travel	9
Total	196

Table 3.9: Top apps by categories

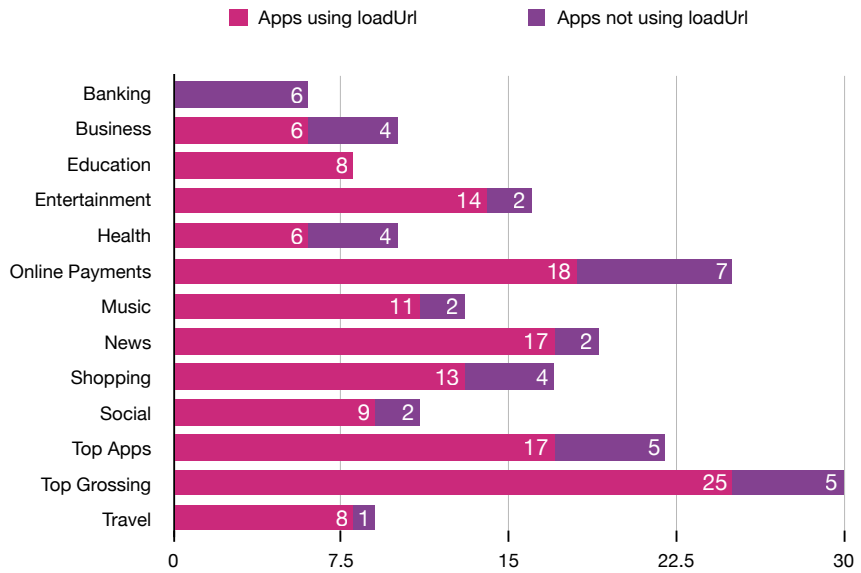


Figure 3.3: Apps using *loadUrl* in each category

to as Top Gross). Table 3.9 lists our categories alongside the number of apps studied in each category.

3.6.3 JavaScript from *loadURL* in Top 196 Apps

Using *LUDroid*, we prepared a corpus of JavaScript string and objects' fields (containing a JavaScript string, potentially a constant) passed to *loadUrl*. We performed a manual analysis of the corpus. We observed that none of the apps in this category injects third-party scripts remotely over an unsecured protocol.

Listing 3.10: Code from Facebook SDK found in all of the top apps category

```
(function() { var event = document.createEvent('Event');
    event.initEvent('fbPlatformDialogMustClose',  true,true);
    document.dispatchEvent(event);
})();
```

In addition, we also provide insights on the *usage* of *loadUrl* in these apps. Figure 3.3 lists the percentage of apps using *loadUrl*. We observed that none of the banking apps we studied use *loadUrl*. In this app corpus we founds 50% usage of *loadUrl* in categories other than Banking. These results suggest that a majority of the popular apps, except those related to security-critical banking tasks, leverage *loadUrl*. In what follows, we describe our insights on the usage of *loadUrl* in each of the categories.

In top-grossing and top apps categories, more than 70% of apps use *loadUrl*. On manual inspection, we observed that the use of *loadUrl* in these apps originates from SDKs. Similar to the benign apps in the last section, either the SDKs pass raw JavaScript or refer to a field variable within the SDK. Approximately 76% of these are mainly related to mobile payments or social networks. Social networking SDKs account for one-third of the usage among these apps. Listing 3.10 lists the most frequent code we observed in this category of apps. Other categories of SDKs prevalent are advertisement (approx. 10%), e-commerce (approx. 5.77%), and also file sharing SDKs such as *DropBox* (approx. 2%). The file-sharing SDK used internally within the *DropBox* app and is used for authentication. These cases highlight the ubiquity of *loadUrl* use among the top downloaded apps and strengthen our results from the benign apps (cf. Section 3.6.2).

In the education apps, such as *EdX* and *Coursera*, we observed that all apps leverage at least one instance of *loadUrl*. We observed a total of 21 instances of its use in 8 apps. Approximately two-third of the apps leverage *loadUrl* provided by the Facebook SDK by passing raw JavaScript or accessing the URL through a field defined in the SDK. Fortunately, none of the SDKs were untrusted. This pattern is similar to those observed in benign apps studied in Section 3.6.2. We also observed another a similar pattern of resetting the webpage using `about:blank` in one app. A minority (approx. 10%) of these apps pass the parameter to *loadUrl* through inter-procedural function calls. The usage of *loadUrl* in the entertainment category is similar to those in education. Here, also two-third of the apps leverage *loadUrl*. Additionally we observed the usage of the *vungle* ad library, which was also present in the benign apps.

Next, we consider another category: online payments. We observed that 76% of the apps pass object fields to *loadUrl*, while raw JavaScript strings account for the remaining 24%. All of these object fields are from their respective apps. The raw JavaScript is a simple facebook login fallback code which is used in case the Facebook app is not installed on the user's device. This is in contrast with our previous observation that object fields passed to *loadUrl* originate from SDKs.

In social apps, we perceive the use of features that modify an app's look-and-feel through Web APIs. Consider the program fragment in Listing 3.11, where the app uses Web APIs to modify the header section of the app, which is implemented in WebView. Line 2 selects the element named *MTopBlueBarHeader*, and line 3 removes all styling. In this category we found two unique type-2 source code clones, and no instances of

Listing 3.11: Modifying an app's look-and-feel

```
1 (function() {
2   var topbar = document. querySelector('#header[data-sigil =
      "MTopBlueBarHeader"]');
3   topbar.setAttribute('style', 'display:none');
4 })()
```

passing objects' fields to *loadUrl*. In other categories, such as shopping and travel, we observed a single type-2 clone of raw JavaScript shown in Listing 3.10. The same code snippet is most frequent among benign apps as well (Section 3.6.2). Line 2 initializes a DOM event handler that listens to the event fbPlatformDialogMustClose, and line 3 dispatches the corresponding event.

3.6.4 Use of evaluateJavascript in frequently used apps

3.6.4.1 Case Study — Controlling Page Navigation

In our study, we found apps using JavaScript to control page navigation on Android, which is similar to using a regular web page. Consider the code snippet from the payment apps *JioMoney* and *LiquidPay*, where the developers use handlers for keyboard events in JavaScript : `handleBackKey()`, `LoginSubmitClick()`, and `document.getElementById('loginIdxSubmitBtn').cli`
The first one registers a custom handler for handling the back key on Android devices, while the second one is for logging in via a submit button, and the third one automatically initiates a click event. This case study highlights sophisticated use of evaluateJavascript corresponding to the sophisticated use in *loadUrl* (cf. Section 3.6.2 and Section 3.6.4).

3.6.4.2 Case Study — LiquidPay

Table 3.8 displays six code snippets we identified using *LuDroid*. The payment app *LiquidPay* uses sensitive information in plain-text in five of its Activities. All of these five Activities use sensitive data such as a password or Bank-ID or perform a sensitive operation such as User Login. The developers include sensitive values in plain-text such as `document.getElementById('access-code').value = "\dots"` or `document.form1.selBankID.value = "\dots"` in *WebView* (cf. Table 3.8). We have masked the sensitive information by . . . to protect the confidentiality of the information of this app. This case study exemplifies a misuse of *WebView*.

To the best of our knowledge, we did not find any relevant work on static analysis of evaluateJavascript . Our study on the most frequently used apps dataset reveals that developers have found sophisticated use cases for evaluateJavascript . In principle evaluateJavascript provides an alternative to *loadUrl*. Although evaluateJavascript is meant to only evaluate the passed JavaScript expression, it is also possible to mimic the functionality of *loadUrl* by manipulating DOM properties. Again, this makes the existing analyses [63], [15] unsound. A more precise and sound analysis has to also consider the data-flow between Android and evaluateJavascript .

Listing 3.12: Malware injecting external results in Android object

```
1 window.HTMLOUT.processHTML(  document.getElementById('SearchResults').
    innerHTML);
```

3.6.5 JavaScript in Malwares

We manually analyzed 1000 malware samples and studied the JavaScript passed to *loadUrl* and evaluateJavascript . To our surprise, we did not find any instances of evaluateJavascript use in malware. Of all malware samples we studied, 121 samples pass *JavaScript* code to *loadUrl*, while 451 samples use *loadUrl* with an object's field variable. The 121 instances of raw JavaScript constitute eight distinct code clones (type 2) . We further performed a manual analysis of these eight unique JavaScript codes and found two of these JavaScript codes to be similar (type 2 clone) to those in the benign apps. A manual analysis of the field variables again revealed that these originate from SDKs, similar to those we found in benign apps (cf. Table A.1).

3.6.6 Case Study — Injecting external objects

We found a usage pattern in 88 of the 121 distinct malware codes (approx 72.73%), where the malware injects results from the WebView into a shared Android object. Listing 3.12 shows one such instance where the DOM element named *SearchResults*, is passed to the method *processHTML*, and the resulting HTML is accessed through the bridge object *HTMLOUT*. This pattern is similar to injecting external results in Android in Section 3.6.2. However, in this case, it is more dangerous. By manually searching for the package names, we find approximately 63% of the malware are from repackaged apps. It points out that these apps either have a malicious activity or *WebView*. Having a malicious *WebView* is more dangerous as it potentially exposes it to the Web.

Further, the activities (or WebView) in these apps run with the same privileges as a regular app. For the other 37% apps, we could not find any information on the Google Play Store. Therefore, these apps are taken down by the user, or otherwise, these apps performed some suspicious operation. However, with the current data that we have, it is not possible to give further comments.

3.6.7 Case Study — Clickjacking

As another case study we identified is a clickjacking attack. Listing 3.13 shows a code snippet found in five out of 121 malware (approx. 4.13%). Similar to the case study in Section 3.6.2, this code snippet is also susceptible to a clickjacking attack. The function *noTapHighlight* in Listing 3.13 masks user-clicks by removing the feedback given to the users (by removing the highlighted color), thus, making them unaware of the accidental user-interactions with the device. Line 2 of the function *actionClicked* serializes the arguments passed to the method. Line 3 returns the value of the field *result* after a deserialization of the previous statement. We discover the source code in malware, and therefore, the intent of the clickjacking attack could potentially be malicious. However, owing to the limitations of *LuDroid*, we could not ascertain the exact sequence of user-interactions that invokes these functions.

The source code mentioned here is a type-2 clone of the source code in Listing 3.7,

Listing 3.13: Malware susceptible to clickjacking

```
function actionClicked(t, u) {
    var r = prompt('showToast' + JSON.stringify({ method: 'showToast',
    params: (u ? [t, u] : [t]) }));
    if (r && typeof r === 'string') { return JSON.parse(r).result; }
};
function noTapHighlight() {
    var l = document.getElementsByTagName('*');
    for (var i = 0; i < l.length; i++) {
        l[i].style.webkitTapHighlightColor = 'rgba(0,0,0,0)';
}};
noTapHighlight();
```

Listing 3.14: Malware using SDK

```
(function() {
    var flurryadapter = window.flurryadapter = {};
    flurryadapter.flurryCallQueue = [ ];
    flurryadapter.flurryCallInProgress = false;
    flurryadapter.callComplete = function(cmd) {
        if ( this.flurryCallQueue.length == 0 ) {
            this.flurryCallInProgress = false;
            return;
        }
        var adapterCall = this.flurryCallQueue.pop();
        this.executeNativeCall(adapterCall);
        return "OK";
    };
    //Remaining code from Flurry SDK
})();
```

which was also observed in the benign apps in Section 3.6.2.3. However, it makes the scenario more dangerous than those in benign apps as the same technique is applied to apps with malicious intent. This case study reveals the severity of vulnerable patterns (Section 3.6.2) if exploited by malicious apps. We attempted to map the respective source codes between benign apps and malware. Unfortunately, a similarity could not be established owing to the limitations of decompilation. Decompilation mangled the variables and class names in many cases, so building a one-to-one mapping between malware and benign apps is not feasible.

3.6.8 Case Study — Analytics SDK in malware

We identify a case study where, to our surprise, we found malware using the analytics SDK *Flurry*. Consider Listing 3.14 where the function defines an adapter to listen to the Flurry analytics services. The body of the anonymous function implements the functionality possibly required for handling of call requests stored in the call queue. We found this pattern in 8 of 121 apps (approx. 6.61%) of apps.

3.6.9 Case Study — SDK usage in malware

We also identified a case where malware uses SDKs, especially those SDKs concerned with advertising. The SDK usage pattern is prevalent in 48.1% of the studied malware samples. Out of these, we found 78.2% (approx. 37% of all malware) originating from

Category	Benign	Frequently Used	Malware
Third-party script injection	✓	✗	✗
Non-Trivial Information Flow	✓	✗	✓
Obfuscation by Third-Party	✓	✗	✗
Privacy leak	✓	✓	✓

Table 3.10: Summary of the observed JavaScript behavior in studied apps

SDKs meant for advertising. For example, consider the use of the advertising library *TapJoy* in malware `com.nuttyapps.sally.makeup.salon` and `skloo.mobile.pud`. Note that the former APK is probably derived from a game called "Sally's Makeup Salon", the latter an "Ultimate Drinking" app. Malware is often piggybacked to popular benign software and distributed via non-regulated app stores as a trojan horse in order to be installed voluntarily by naive users. In related work, Lee and Ryu have shown that the TapJoy library performs various sensitive operations. These include launching new activities, retrieving sensor information, and fetching location information and app information [65]. The last three operations potentially harm user privacy. However, this case is more dangerous because of the malicious intent of the built-in malware. Launching new activities through JavaScript leads to a *App to Web Injection attack* [49], [65] where the app (in this case, malware) initiates new malicious activities through injected JavaScript.

A closer investigation of which functionality stems from the regular application and in which form the attached malware is beyond the scope of this paper. However, we can see that malware leverages hybrid app activities in order to satisfy its malicious intents and that analyses that aim at analyzing the hybrid communication are bound to provide a precise model of the communication patterns detected in this study. Therefore our work may also serve as a "testbed" for such analyses.

3.6.10 Comparison of JavaScript Usage Across Datasets

In this chapter, we saw the various uses of JavaScript embedded in Android apps. Besides, we also discovered the vulnerable use case of leaking private information through clickjacking. Here, we summarize our observations on the JavaScript strings passed to *loadUrl* for the three categories of apps: benign apps, frequently used apps, and malware apps.

Table 3.10 presents the behavior observed across the datasets of apps. We observed that (potential) privacy leak is a prevalent behavior of the JavaScript code passed to *loadUrl* in all of the app categories. Non-trivial information flows from JavaScript to Android is the next typical behavior, observed only in the benign apps and malware dataset. Interestingly, the frequently used apps do not show this behavior as they, primarily, use it through field objects (mostly constant strings) of the SDKs. Obfuscated third-parties were observed only in benign apps. Furthermore, frequently used apps and malware were free from third-party script injection attacks. It shows that at least the

frequently used apps adhere to basic security practices.

3.7 Discussion

Using LUDroid we were able to derive a plenitude of novel statistics covering information flow information, URL statistics and statistics on JavaScript code. Additionally, we detected URL based vulnerabilities. At this point LUDroid is not a stand-alone analysis tool but merely supports manual inspection and calculation of statistical data. The goal of this work is to present interesting insights on how the bridge between Android and JavaScript is used in the wild in order to facilitate the design of automatic program analysis that take both sides of the hybrid app into account.

Other common limitations of static analysis are native code, reflection, dynamic control flow and obfuscation and the fact that strings like the URLs passed to *loadURL* may be constructed at runtime. All of these obstacles have been investigated in separate lines of research [45, 74, 117, 56, 69, 41] so we consider them orthogonal to the insights we are aiming at in this study. Note however, that *loadURL* is also a dynamic language feature that may execute code that is constructed at runtime based on a string parameter. Insights gained in studies that target dynamic code execution (e.g. for JavaScript [96]) are also relevant to understand the semantics of *loadURL*.

Native code In Android applications it is possible to include native code (e.g. C/C++ code) via the Java Native Interface (JNI). LUDroid does not support the analysis of native code at the current point of time. The support of native code implies high additional complexity, as assembler semantics needs to be included into the analysis. Additionally, we have observed native code rarely during our experiments.

Reflection, dynamic control flow and obfuscation In Java reflection is a method for accessing code features during runtime. With reflection it is possible to make call and field access decisions at runtime dependent on strings or other values that can even be fetched during runtime. Theoretically, it is therefore possible to make the runtime control flow and data flow completely dependent on these value, preventing every static analysis. At the current point of time LUDroid does not support analysis for reflection. Given this limitation, it becomes clear that LUDroid can not analyze an app that intentionally uses reflection to prevent analysis (obfuscation). While some support for reflection may be implemented in the future, it is in general not possible to resolve reflection with dynamic values with any static analysis approach. However, we would like to stress that we have not encountered the described massive usage of reflection in any of the inspected benign apps.

Limited String analysis LUDroid requires String analysis to resolve the arguments passed to the *loadURL* method. For the cases in that the string argument is being manipulated before being passed we implemented domain knowledge of the Java *String* class as well support for the *StringBuilder* class. However, LUDroid does not support advanced string manipulations such as array based string manipulations at the current point of time.

3.8 Remarks

In this work, we present a large-scale analysis of *loadURL* usages in real world applications. The statistical results include numerous features, such as information flow data, URL statistics and javascript code features on a set of 7,500 randomly selected applications from the Google Playstore. We implemented our semi-automatic analysis approach in a tool called LUDroid that computes the data by using slicing techniques. One of the major insights obtained from this study was the bi-directional communication between Android Java and JavaScript with bridge interfaces, which paved the way for an information flow analysis of bridge methods presented in the next chapter.

Chapter 4

Information Flow Analysis of Hybrid Apps

4.1 Introduction

In this chapter, we introduce a information flow analysis for hybrid apps. Android apps store a plentitude of private data safeguarded by the android permission model. However, it is uncertain how to regulate the flow of the guarded data as soon as the permission model. It is more prominent in the case of hybrid apps, where the data might flow from one language to another. In the last chapter, we saw a large-scale analysis of hybrid apps and conducts an information flow analysis on WebViews. Unfortunately, the approach is imprecise as it does not track flows inside JavaScript, marking a data-flow *leak* if it flows from Java to JavaScript.

Numerous methods [65, 63, 55, 102, 97, 51, 75, 33, 128] have been proposed to detect security and privacy issues in Android hybrid apps. However, most of them only explore web vulnerabilities' impact on hybrid apps. Lee et al. [63] introduced the first framework to uncover *type bugs* resulting from the misuse of hybrid APIs and to analyze the flow of information from the Android native side to JavaScript. However, their analysis mainly focuses on type bug analysis and provides only a basic taint analysis to track information flows. An improvement over this work proposed by Bae *et al.* [15] covers additional peculiar properties of the language while , Their approach mainly concentrates on ad platforms and overlooks many unique aspects of the bridge interface. Moreover, these approaches fails to consider violations of flows from JavaScript to Java.

In this chapter, we propose a demand-driven static analysis-guided framework to discover information flows in Hybrid apps. Our framework comprehensively tracks confidentiality violations, i.e., for (sensitive) data flows from Java to JavaScript and then to a public sink. Besides, our framework can identify integrity violations for *interface object* manipulations from the web.

The semantics of WebView is complex, and prior approaches [63, 15] have only covered parts of it. We further extend these findings to uncover a few peculiar semantics essential to track information flows of WebView concerning using *interface objects*. For instance, in this chapter, we present cases where flow-insensitive updates of *interface objects* are available to the enclosed JavaScript. These cases occur due to the complex execution

```
1 Class BridgeJava {
2 String g;
3 ...//some code
4 @JavascriptInterface
5 public String getValue() {
6   // getter method
7   return this.g;
8 }
9 @Override
10 protected void onCreate(Bundle ...)  {
11   WebView wv = new WebView(this);
12   WebSettings wvsettings = wv.getSettings();
13         // enable JavaScript
14   wvsettings.setJavaScriptEnabled(true);
15         //add interface object
16   BridgeJava ifcObj = new BridgeJava(this);
17   wv.addJavasriptInterface(ifcObj,"ifcObj");
18         //load JavaScript in Web
19   wv.loadUrl("file:///index.html");
20 }
```

(a) Java Side

```
1 function set() {
2   var x = interfaceObj.getValue();
3 }
```

(b) JavaScript code in index.html

Figure 4.1: A Simple Java and JavaScript communication in Android

behavior of WebView and the semantic differences between Java and JavaScript.

The interface object facilitates information flow between Java and JavaScript; consequently, analyzing it and its use in JavaScript. We devise a demand-driven approach where we begin our analysis on the JavaScript side and initiate the analysis of interface objects only if they appear in JavaScript. The key insight is that the information shared with JavaScript is linked to the interface object's lifecycle. Thus, our framework leverage IFDS/IDE framework [95, 99] to create a data-flow summary framework of the interface objects, with security labels attached to their associated properties. Next, our framework begins an independent analysis of JavaScript, allowing for using an interface object in two ways. The first involves invoking a *bridge method* without any parameters, while the second involves invoking a *bridge method* with input parameters. In the first case, we use the precomputed security labels obtained from interface object summaries. While the second case poses a potential threat to integrity as the input parameters, computed in JavaScript, can alter the interface object or native side code.

4.2 Threat Model

To set the stage for our analysis, we detail the threat model concerning the information flow in hybrid apps. Figure 4.3 shows a high-level overview of the threat model, including confidentiality and integrity violations. In our work, we are interested in communication anomalies between Java and JavaScript in a hybrid Android app; other channels of information leakage in a hybrid app are ignored in this work. Our threat model assumes Java as the host and Information channels within Java safe. JavaScript

```
1 Class BridgeJava {
2 // Previous code ommitted
3 @JavascriptInterface
4 public String setValue(String secret) {
5         this.g = secret;
6 }
7
8 @Override
9 protected void onCreate(Bundle ...)  {
10 // Previous code ommitted
11 //boilerplate code to get deviceId
12         setValue(deviceId);
13 // Previous code ommitted
14         wv.loadUrl("file:///index.html");
15 }
```

(a) Extended Java Side

```
1 function set() {
2         var x = interfaceObj.getValue();
3         Sink(x); //New code to Sink secret x
4 }
```

(b) Extended JavaScript Side

Figure 4.2: A Simple Java and JavaScript communication in Android

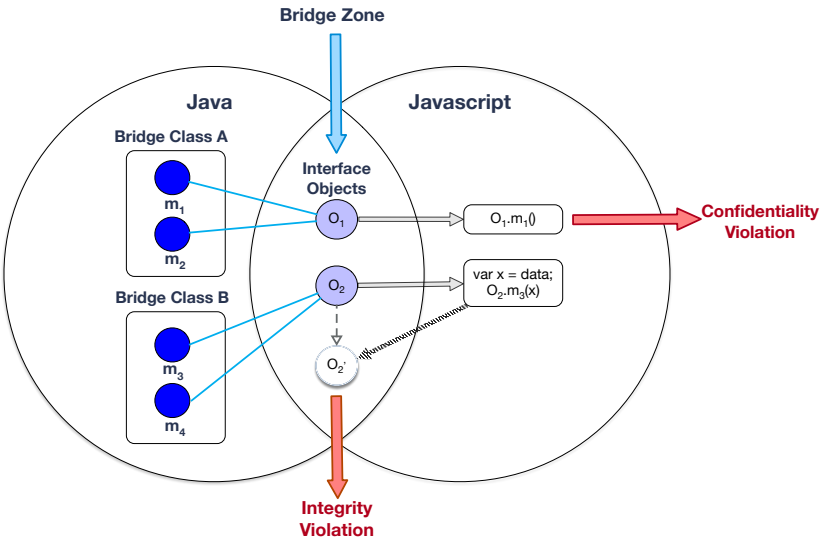


Figure 4.3: Hybrid App Threat Model

works as a guest language and can access selective functionalities shared by the Android side.

A confidentiality violation occurs if sensitive information flows from Android to JavaScript and then flows through a sink in JavaScript. In Figure 2, JavaScript access *BridgeClassA*'s method m_1 with the interface object O_1 and outputs the result via a public sink. If method m_1 releases sensitive information, then this flow is considered a confidentiality violation. To elaborate on this violation, we extend the example in Figure 4.1 to demonstrate a confidentiality violation via Figure 4.2. In particular, the *BridgeClass* is extended to include a setter method *setValue* (Line 4 in Figure 4.2a)

```
1 javascript: window.SynchJS.setValue(function(){
2   try {
3      return JSON.parse(Sponsorpay.MBE.SDKInterface.getOffer()).uses_tpn;
4   } catch(error) {
5      return false;
6   }
7 })() // call setValue
```

Figure 4.4: Javascript updating Interface Object

and *onCreate* method assigns a sensitive information such as device id or location information via *setValue* method (Line 12 in Figure 4.2a). The *getValue* method call in JavaScript returns this sensitive information and leaks via a sink at Line 3 in Figure 4.2b. In our threat model, an integrity violation occurs when a JavaScript code snippet modifies an interface object. Updating an Android object could interfere with the sensitive properties of the (secure) Android system. Consider the example code snippet taken from a large-scale app [83, 118] in Figure 4.4. In this example, a JavaScript code snippet modifies the Android object by invoking a *setter* method of the bridged class. In particular, the *setValue* method is invoked via the interface object *SynchJS*; the interface object is updated based on the return value of an object deserialized from a third-party library.

4.3 Peculiar Semantics of Interface Object

Android Webview establishes an asynchronous communication channel between the Android native side and the Web, where both sides run in different execution environments; this results in many unexpected cases arising due to complex inter-operation semantics. Bae *et al.* [15] provided crucial semantics for some of these operations. Their study explored error-prone runtime behaviors such as resolving overloaded method calls, first-class java methods, multiple bridge objects, and nested bridge interfaces. In this section, we further extend their findings to include pivotal semantics required for a sound and precise information flow analysis of hybrid apps.

Observation 1 (Flow-insensitive update of the Interface Object)**.** *Updates to the interface objects (field-updates)* after *the* loadUrl *method call are also reflected in the corresponding JavaScript code.*

Field updates of a Bridge class via the interface object are asynchronously imposed to JavaScript, i.e., any updates to the interface object after the *loadUrl* or *evaluateJavascript* method calls are applied to the enclosed JavaScript code. This phenomenon occurs due to the complex initialization process of the Chromium thread (part of the WebView browser). As is, the WebView class is a part of Android's View class and runs as a View thread, and as per the Android OS design [5, 37], WebView uses the first view thread as the UI thread. Thus, the Chromium initialization is postponed until the view thread (UI thread) has been established. The example in Figure 4.5 explains this observation. Line 3 assigns field `g` a low value, Line 4 invokes the *loadUrl* method with an input URL with inbuilt JavaScript code, and finally Line 5 updates the field `g` a sensitive value. In normal scenarios, the effect of Line 5 in the Chromium thread should depend on whether the Chromium thread executes before the UI thread; however, as

```
1 //... code from Figure 1 a
2  wv.addJavasriptInterface(ifcObj,"ifcObj");
3 +ifcObj.g = "publicData";
4  wv.loadUrl("file:///index.html");
5 +ifcObj.g = "secret";
```

Figure 4.5: Modifying interface objects on host

```
1 ifcObj =  {
2         getValue: function () {
3                 return "temp_property";
4         }
5 };
6 var x = ifcObj.getValue();
```

Figure 4.6: Interface Object Definition in JavaScript

```
1 x =  {
2   printHello: function () {
3         console.log("Hello");
4   }
5 };
6 delete x.printHello
7 x.printHello() //method not found exception
8 delete ifcObj.getValue
9 ifcObj.getValue() // invokes the getValue method
10 delete ifcObj // deletes the interface object
```

Figure 4.7: Deleting interface Objects

explained, the Chromium thread waits for initialization until the UI thread completes its task, and thus, Line 5 updates are applied to the enclosed JavaScript.

Observation 2 (Interface Object redefinition)**.** *The bridge methods of the interface objects can be redefined in JavaScript. On the JavaScript side, interface objects, like other JavaScript variables, can be modified by updating the definitions of the bridge methods.*

JavaScript is a dynamically typed language, and the semantics are sometimes puzzling. Incidentally, the bridge variable can be reassigned to another variable or simply redefined. Figure 4.6 redefines the interface object with a new *getValue* function, i.e., the Java bridge method *getValue* is *not* available at invocation in Line 6.

Observation 3 (Interface object deletion)**.** *Interface objects can be deleted in JavaScript; however, the associated properties, e.g. methods, cannot be deleted, and JavaScript silently ignores the deletion.*

As is, JavaScript semantics allows the deletion of the object properties. For example, in Figure 4.7 at Line 6, the `printHello` property of the variable `x` is deleted, and thus, it is not accessible in Line 8. However, the same does not hold with interface objects. While you can completely delete an interface object (Line 10), deleting its properties is silently ignored. For example, Line 8 does not yield any results, and Line 9 is successfully executed.

4.4 Modular Inter-language Analysis

4.4.1 Overview of IFDS and IDE

IFDS and IDE analysis has been introduced in this thesis in Section 2.4 and Section 2.6. To summarize, an IFDS problem has three inputs: (1) interprocedural control flow graph $ICFG$, (2) domain of dataflow facts D, and (3) set of flow functions. A flow function $f : S \times d \mapsto d'$ maps each statement S from the input domain d to the output domain d' where $d, d' \subseteq 2^D$. Flow functions in IFDS are of four types:

1. normal flow functions express the dataflow function for a statement s in an intraprocedural CFG for a single function
2. call-to-begin function propagates the dataflow facts from the call-site to the called functions
3. call-to-return propagates the dataflow facts from the call-site to the return site on the side of the caller function
4. return-flow propagates the dataflow facts returned by the called function to the return-site of the caller function

IFDS algorithm greedily explores the set of dataflow facts. At each step, it applies the flow function only on the set of reachable dataflow facts reachable in the previous step. IDE [99] is an extension of IFDS which overcomes the shortcoming of the IFDS by augmenting statements with an environment transformer $t : (2^d \to l) \to (2^d \to l')$ [99]. An environment $env : 2^d \to l$ maps the dataflow facts $d \in D$ to the lattice element l belonging to a meet (or join) semi-lattice L. In other words, environment transformers are similar to transfer functions. The environment transfers make it possible to specify dataflow transfer functions based on the dataflow lattice. In IDE, the environment transfer functions are composed and propagated along the reachable paths explored in IFDS.

4.4.2 Preprocess–Bridge Object Dynamics on Java Side

The preprocessing step captures the bird's eye view of Java-JavaScript communication in an Android app. Given an apk file, this step aims to compute the essential information, e.g., bridge classes, concerning Java-JavaScript communication using the *WebView* APIs. Algorithm 1 details the basic steps involved in the preprocessing. The first step is to look for the key WebView APIs that enable Java-Javascript communication, i.e., *setJavaScriptEnabled(true)*and *addJavaScriptInterface()*. In Line 6, the *locateAddJSIFaceAPI* method call returns the line number (if any) of the *addJavaScriptInterface* method call. In the next step, the settings of the associated WebView object are extracted. If JavaScript is enabled on this WebView object, the core part of the information collection begins.

There are four central pieces of information a JavaScript-enabled *WebView* contains. First, the Java class that is bridged to JavaScript. Second, the interface object of this bridged class that can be accessed from within the JavaScript. Third, the bridged class's methods annotated with *@JavascriptInterface*, i.e., these methods can be accessed from the JavaScript code, and finally, the JavaScript code snippet that can leverage this bridged class. Extracting the bridge class and interface object poses some challenges to

Algorithm 1 ExtractBridge$_{Java}$

Require: an APK file $\mathcal{A}$
Ensure: a set of Java-JavaScript communication relations $\mathcal{T}$

```
1: IClasses ← ∅
2: T ← {(∅ | ∅ | ∅ | ∅)}
3: M ← ∅
4: IClasses ← Decompile(A)
5: for class ∈ IClasses do
6:     line ← locateAddJSIFaceAPI(class)
7:     webView = extractWebViewSettings(line)
8:     if JavaScript is enabled on webView then
9:         iFaceObj ← extractInterfaceObj(line)
10:        iFaceObjAliases ← computeMayAlias(iFaceObj)
11:        for mayaliasObj ∈ iFaceObjAliases do
12:            bridgedClasses ← CHA(mayaliasObj)
13:            for bridgedClass in bridgedClasses do
14:                M ← collectAnnotatedMethods(bridgedClass)
15:            end for
16:        end for
17:        lineJS ← forwardTrace(line, ("loadUrl"||"evaluateJS"))
18:        if lineJS located then
19:            js ← extractJavaScript(lineJS)
20:            T ← T ∪ (bridgedClass | iFaceObj | M | js)
21:        end if
22:    end if
23: end for
```

Table 4.1: Preprocessed Java-JavaScript Communication data

Bridge Class	Interface Object	Bridged Methods	JavaScript	Web Page
BridgeJava	ifcObj	getValue(), setValue(String)		file:///index.html

overcome. First, the aliases of the interface variable may be unresolved. Considering the privacy-sensitive and conservative nature of our framework, we compute the may aliases[110] of the interface object (Line 10). Next, for each alias, a class hierarchy scan [24, 78] is performed to obtain all methods that can be bridged via this interface object, i.e., all (annotated) methods of the bridge class and its parent class (Line 11-13). Finally, a forward trace starting from the *addJavaScriptInterface(..)* until methods *loadUrl(...)* or *evaluateJavascript(...)* is performed. From these methods, the input webpage or the JavaScript code snippet is extracted (Line 17 and 19). This step is performed for all *WebView* objects in the app, and the corresponding entries are stored in a relational table (Line 20). Notably, the *loadUrl(...)* method can contain an input webpage that contains the JavaScript code snippet. Besides, it can also include the JavaScript code snippet as the input string (appended with "JavaScript:" String). In the former case, we store the path of the input webpage to download it later for extracting the embedded JavaScript. Table 4.1 shows the sample output of the preprocessing step for the code snippet in Fig. 4.1 and Fig. 4.2.

4.4.3 Analysis of JavaScript and Bridge Interface

The exchange of information between Java and JavaScript is facilitated by the interface object; thus, analyzing this interface object and how it is used in JavaScript would

yield the desired outcome. Consequently, we devise a demand-driven approach where we begin our analysis on the JavaScript side and initiate the analysis of interface objects only if they appear in the JavaScript code. The JavaScript code is extracted for each entry in the Java-JavaScript communication relation table (e.g., Table 4.1). In our approach, if multiple JavaScript code snippets are present in a web page, we aggregate them in the order they appear on the web page. Next, we perform a shallow check to see the occurrences of the corresponding interface object in the JavaScript code. If no occurrences are found, then the analysis concludes for this entry due to the lack of code sharing between Java-JavaScript for this entry.

In our work, we embed our analysis in the IDE framework [95, 99] for a demand-driven IFC analysis. IDE is an extension of IFDS — IFDS only computes reachability. IDE framework enables us to track properties, for example, tracking if the variables has closed the file handler object after opening it. In the IDE framework, edges are annotated with dataflow functions and composed along the reachable paths to obtain the dataflow solutions. We leverage this property to define our IFC analysis.

Our analysis begins with the analysis of the interface object. The key insight in bridge communication is that the information that can be shared with JavaScript is bound to the lifecycle of the corresponding interface object, i.e., having a flow-insensitive, context-, and field-sensitive sound analysis of the shared interface object would yield the necessary information for our analysis without the need of analyzing complete (and potentially huge) Java code base. Besides, the analysis needs to take care of our first observation (strong update of the interface object).

With the summary of the interface object, our analysis starts IFC analysis of standalone JavaScript code. In particular, we use the IFC labels obtained from the summaries of interface object and influenced paths to escape the re-analysis of the interface objects. Besides, our analysis needs to keep track of special properties of interface objects, i.e., observations 2 and 3. Finally, to analyze the integrity violation, our analysis computes the access paths that influence the parameters of the bridge methods. The parameters of the bridged methods are exposed to the JavaScript side and are potential candidates for integrity violations. Therefore, our analysis performs a dataflow analysis to analyze which access paths are reachable from the parameters embedded in the IFDS framework. We store these summarized access paths for each bridged method. Next, we explain each of these stages in detail.

4.4.3.1 IFC Analysis for interface objects

Figure 4.8 provides a basic introduction to our two-phase approach for the IFC analysis of interface objects. The inputs to the analysis include a set of bridge methods and Interface objects (obtained from the preprocessing step), the method invoking interface objects (referred to as *entrypoint methods*), and an interprocedural control flow graph (CFG). The IFC analysis is embedded in the IDE framework and computes the security labels of the access-graphs. IDE framework needs a set of environment transformers, an interprocedural CFG, and a domain for the analysis. The analysis proceeds in two phases: (Phase 1) computes the summary information for bridged methods, and (Phase 2) computes the IFC analysis using the summaries at each stage. Phase 1 runs on bridge methods and computes the reaching definition summaries via a forward analysis along the edges of the CFG. Phase 2 starts from the *entrypoint method* and computes the IFC

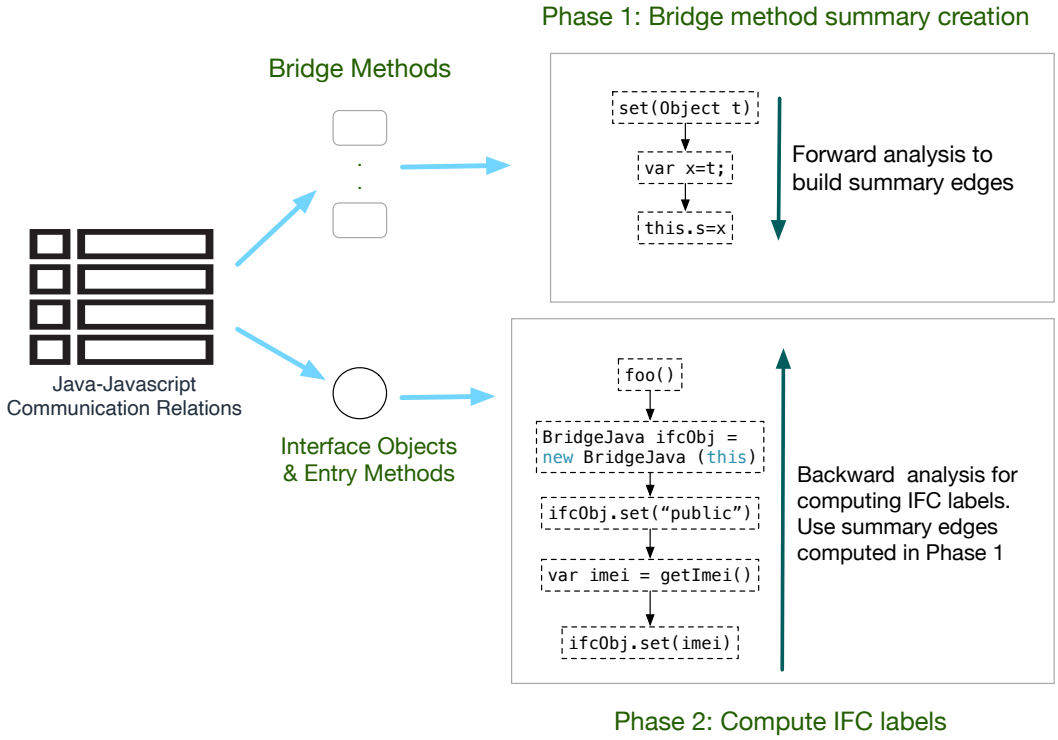


Figure 4.8: Overview of IFC analysis of bridged objects

labels in the reverse direction of the control flow. In the case of call-to-bridge interfaces in Phase 2, the analysis invokes the summaries of Phase 1 without re-analyzing the bridge methods. Next, we describe our analysis in detail.

Definition 8. *Domain of the analysis (or dataflow facts) $\mathcal{D}$ is a set of access-graphs, where an access-graph [58, 67] is a pair $(v, \mathbf{F})$ of a local variable v and a field graph $\mathbf{F}$. The Field graph $\mathbf{F}$ is a directed graph where nodes are labelled with field names and edges represents as field-accesses.*

A local variable v will be represented as $(v, \emptyset)$ where $\emptyset$ denotes no field graph. Similarly, a field access such as $v.f$ will be represented as (v, F) where f is a node in the the field-graph F. At each program point s_i, the domain t of access graphs provides a set of available variables and access paths. For a given security lattice $\mathcal{L}$, each access graph is assigned to a security label $\ell \in \mathcal{L}$. Given a security lattice $\mathcal{L}$, the environment transformer $env : (A, \mathcal{L}) \rightarrow (A, \mathcal{L})$, encodes the information flow functions which updates the security labels of the access-paths.

Phase 1 computes the dataflow summaries for bridge functions. For each bridge method m, our analysis computes the reaching definition of the variables defined in the scope of m and fields of the bridge class. The corresponding dataflow functions are shown in Table 4.2. Let t be the set of access graphs valid before the statement. **New flow** creates a new access graph $\langle x, \oslash \rangle$ to t. **Field Store** adds a field f to the field graph of source y and adds $\langle x, F \rangle$ to t. **Field Load** matches the head of the field graph of the source y and copies the rest to the destination variable x. **Call-to-start** function propagates the related access graphs from the caller function to the called

New flow: `x = new T()` $f_s(t) = t \cup (x, \varnothing)$
Normal Flow s : `x = y` $f_s(t) = \underline{\textbf{if}}\ y.\mathbf{F} \in t\ \underline{\textbf{then}}\ t \cup (x, \mathbf{F})\ \underline{\textbf{else}}\ t$
Field Store: `x.f = y` $f_s(t) = \underline{\textbf{if}}\ y.\mathbf{F} \in t\ \underline{\textbf{then}}\ t \cup (x, cons(f, \mathbf{F}))\ \underline{\textbf{else}}\ t$
Field Load: `x = y.f` $f_s(t) = \underline{\textbf{if}}\ y.\mathbf{F} \in t \wedge head(\mathbf{F}) = f\ \underline{\textbf{then}}\ t \cup \{(x, \mathbf{F}_{i+1}) \mid \forall\ \mathbf{F}_{i+1} \in tail(\mathbf{F})\}\ \underline{\textbf{else}}\ t$
Call-to-start: `x = f(m)` for a function declaration `f(a)` $f_s = \underline{\textbf{if}}\ (m, \mathbf{F}) \in t\ \underline{\textbf{then}}\ (a, \mathbf{F})\ \underline{\textbf{else}}\ \emptyset$
Exit-to-return: `x = f(m)` $f_s = \underline{\textbf{if}}\ (return_f, \mathbf{F}) \in t\ \underline{\textbf{then}}\ (x, \mathbf{F})\ \underline{\textbf{else}}\ \emptyset$

Table 4.2: Flow Functions for Forward Analysis in Phase 1

function. **Exit-to-return** function propagates the return value from the called to the caller function.

Phase 2 begins with the *entrypoint method* of the interface object. The analysis begins with a backward analysis from the last *definition* of the interface object in *entrypoint method*. Next, the dataflow facts are propagated in the reverse direction of control-flow edges in the interprocedural CFG. The reason for choosing a backward dataflow analysis is two-fold. First, it gives a single entrypoint for IFDS, i.e., the last definition of the interface object. Second, the analysis is effectively confined to influenced paths in the *entrypoint method*.

The dataflow functions are defined in Table 4.3. Note that the flow functions are analogous to those mentioned in Table 4.2; however, their direction has been reversed except the **New flow** function. **Normal flow** propagates the dataflow facts from the destination to the source. **Field Load** and **Field Store** propagate the fields from the destination to the source base variable. **Call-to-bridge** reverses the edges in the summary information of the bridge method computed from the previous step. For each edge (s, s'), an edge label *edgeLabel* is also propagated to compute the security labels of the access-graphs.

Algorithm 2 presents the pseudocode for our analysis of the interface objects. The analysis begins with the summary creation (via forward analysis) for each bridge method b (Line 1). Line 2 initializes all variables with self-edges and a security label. Next, the flow functions (Table 4.2) are invoked for each transition of the statements with the bridge method b (Line 3). Once the analysis computes the summary of each bridge method, Phase 2 begins. Phase 2 begins with the entrypoint method, i.e., the method with WebView initialization. After the initialization process (Line 6), Line 5 starts the backward analysis starting at the last update of the interface object. This time flow functions (Table 4.3) propagate in the reverse direction of the CFG. The summary information computed in Phase 1 is leveraged whenever a bridge method is invoked in

New flow: `x = new T()` $f_s(t) = t \cup (x, \oslash)$
Normal Flow: `x = y` $f_s(t) = \underline{\textbf{if}}\ y.\mathbf{F} \in t\ \underline{\textbf{then}}\ t \cup (x, \mathbf{F})\ \underline{\textbf{else}}\ t$
Field Load: `x = y.f` $f_s(t) = \underline{\textbf{if}}\ y.\mathbf{F} \in t\ \underline{\textbf{then}}\ t \cup (x, \mathbf{F}')\ \underline{\textbf{else}}\ t\ \underline{\textbf{where}}\ \mathbf{F} = cons(f, \mathbf{F}')$
Field Store: `x.f = y` $f_s(t) = \underline{\textbf{if}}\ y.\mathbf{F}' \in t\ \underline{\textbf{then}}\ t \cup (x, \mathbf{F})\ \underline{\textbf{else}}\ t\ \underline{\textbf{where}}\ \mathbf{F} = cons(f, \mathbf{F}')$
Call-to-start: `x = func(m)` for a function declaration `f(a)` $f_s = \underline{\textbf{if}}\ (a, \mathbf{F}) \in t\ \underline{\textbf{then}}\ (m, \mathbf{F})\ \underline{\textbf{else}}\ \emptyset$
Exit-to-return: `x = f(m)` for a function declaration `f(a)` $f_s = \underline{\textbf{if}}\ (x, \mathbf{F}) \in t\ \underline{\textbf{then}}\ (return_f, \mathbf{F})\ \underline{\textbf{else}}\ \emptyset$
Call-to-Bridge: `x = b.func(p)` $f_s = g^{-1}(t)$ where g^{-1} is the set of access-graphs reachable in the reverse direction of the summary of bridge function from *exit* to *entry*

Table 4.3: Flow Functions for Backward Analysis in Phase 2

Algorithm 2 IDEPhases

Require: Bridge Method b
Require: WebView entrypoint method m

1: **function** PropagatePhase1
2: Initialize($\langle b, v\rangle \rightarrow \langle b, v\rangle, id$) for every variable v in b
3: Propagate($flowFunction_1$) ▷ forward analysis
4: **end function**

5: **function** PropagatePhase2
6: Initialize($\langle m, v\rangle \rightarrow \langle m, v\rangle, id$) for every variable v in m
7: Propagate($flowFunction_2$) ▷ backward analysis with summaries
8: ComputeInformationFlowLabels()
9: **end function**

Phase 2. In this case, for every summary edge $\langle s_i, t_i\rangle \xrightarrow{edgeLabel} \langle s_j, t_j\rangle$ obtained from Phase 1, the analysis propagates the edge $\langle s_j, t_j\rangle \xrightarrow{edgeLabel} \langle s_i, t_i\rangle$ in Phase 2 (via **Call-to-bridge** rule in Table 4.3). Once the propagation stage has been completed, i.e., it has saturated, the analysis computes the IFC labels of the access-graphs by resolving the IFC functions.

Algorithm 3 shows the standard algorithm for IDE analysis with aliasing information. Lines 4-8 initialize the path edges to the worklist and update the worklist by populating the path edges and IFC functions for discovered edges. Line 16 computes the transitive closure of the reachable edge with the current edge (n_j, n_{j+1}) and populates the worklist.

Algorithm 3 IDEBridge

Require: Set of labeled access paths

1: $PathWorkList \leftarrow \emptyset$
2: $IFCFunction[\langle \mathcal{S}, l\rangle \rightarrow \langle \mathcal{S}, l\rangle] \leftarrow \emptyset$
3: $IFCLabel[node] = \emptyset$ ▷ IFCLabel : $Nodes \mapsto IfcLattice$

4: **procedure** Initialize($edge$, $edgeLabel$)
5: $PathWorkList.add(edge)$
6: $IFCFunction[edge] = edgeLabel$
7: **end procedure**

8: **procedure** Update($path, pathlabel$)
9: $PathWorkList.add(path)$
10: $IFCFunction[path] = pathlabel$
11: **end procedure**

12: **procedure** Propagate($flowFunction$)
13: **while** $PathWorkList$ is not empty **do**
14: $edge = PathWorkList.pop()$
15: **let** $edge = \langle n_i, l_i\rangle \rightarrow \langle n_j, l_j\rangle$
16: **for** $flowFunction(\langle n_j, l_j\rangle \xrightarrow{edgeLabel} \langle n_{j+1}, l_{j+1}\rangle)$ **do**
17: $pathLabel \leftarrow IFCFunction(edge) \sqcup edgeLabel$
18: Update($\langle n_i, l_i\rangle \rightarrow \langle n_{j+1}, l_{j+1}\rangle$, $pathLabel$)
19: **if** $n_k \in aliases(n_{j+1})$ **then**
20: Update($\langle (n_i, l_i)\rangle \rightarrow \langle n_k, l_k\rangle, pathLabel$)
21: **end if**
22: **end for**
23: **end whilereturn** $PathWorkList$
24: **end procedure**

25: **procedure** ComputeInformationFlowLabels
26: **for** node $n = \langle n_g, l_g\rangle$ in esg **do**
27: $IFCLabel(n_g) \leftarrow \bigsqcup_e IFCFunction(e)$ ($\forall$ incoming paths e at n_g)
28: **end forreturn** $IFCLabel$
29: **end procedure**

Line 19 handles propagating the aliases for the discovered access-paths.

Figure 4.9 demonstrates two phases via a simple example. Phase 1 discovers the reachable access-paths for the bridge method *set* and *get*. This step yields path edges such as, $\langle set_0, t\rangle \dashrightarrow \langle set_3, this.s\rangle$ and $\langle get_0, this.s\rangle \dashrightarrow \langle get_2, this.s\rangle$ among others. Phase 2 uses these summaries for the analysis, e.g., the edge $\langle foo_5, ifcObj.secret\rangle \dashrightarrow \langle foo_4, imei\rangle$ is created by using the summary of *set* method and matching the input parameters of the calling and called function. As is, Phase 2 employs a backward analysis; therefore, on an intuitive level, it tracks *which* variables are influenced by *what* other variables. Consequently, the edge is directed from *ifcObj.secret* to *imei*. When the reachability of $\langle foo_4, imei\rangle$ is determined, the analysis determines if the information is from a sensitive source and updates the IFC function accordingly. Similarly,

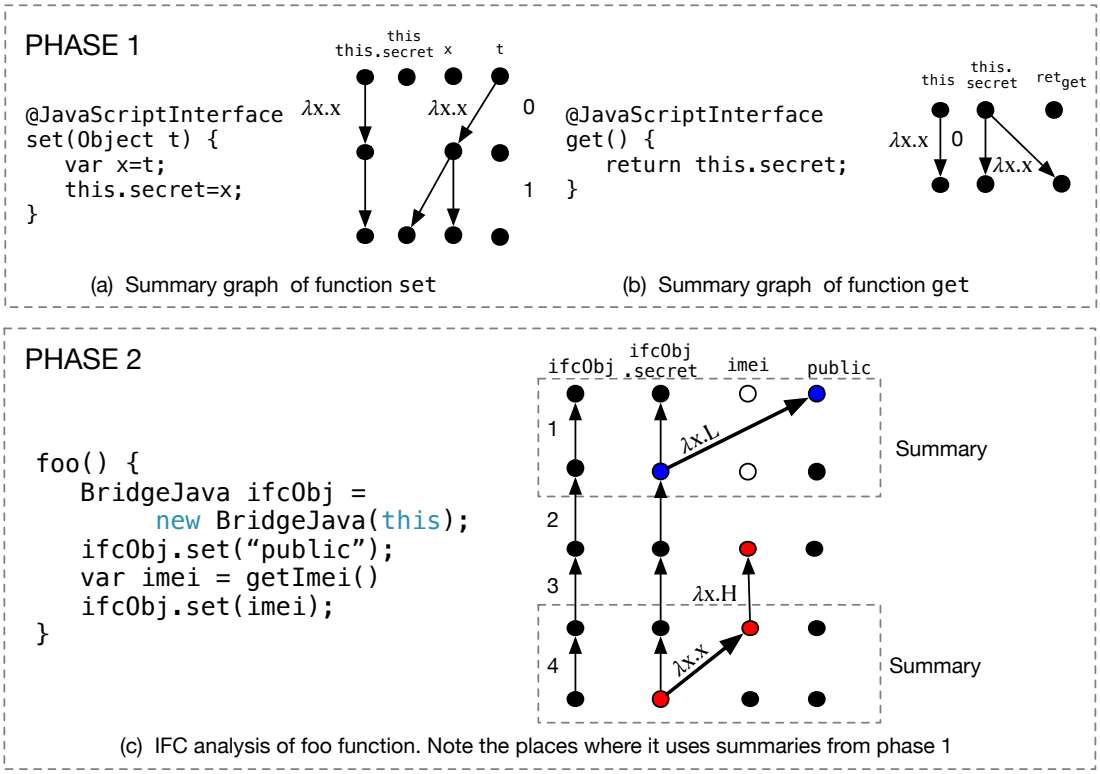


Figure 4.9: IFC with Bridge Objects

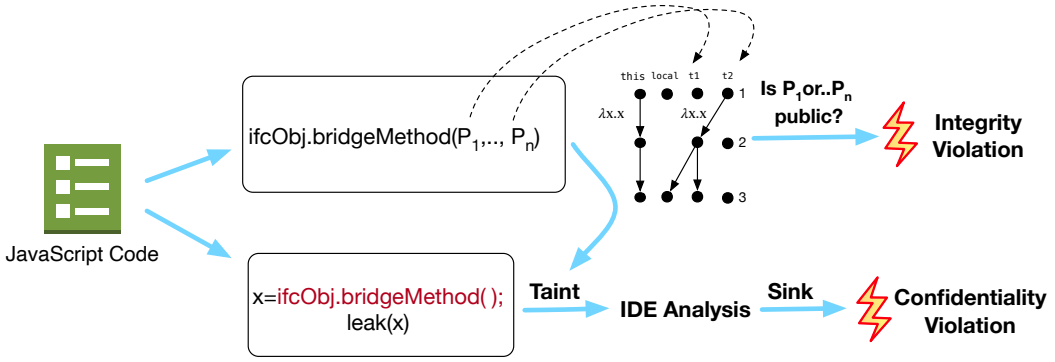


Figure 4.10: IFC analysis in JavaScript

$\langle foo_2, ifcObj.secret\rangle \dashrightarrow \langle foo_1, public\rangle$ is updated via the summary edge obtained from Phase 1. Once the analysis has discovered all of the reachable variables and accumulated the IFC functions along the paths, the function ComputeInformationFlowLabels is called in Phase 2 to determine the flow-sensitive IFC labels for *ifc.secret* as *high* at foo_5 and *low* at foo_2.

4.4.3.2 Javascript Analysis

After obtaining the interface objects' summaries with IFC labels, the stage is set to perform a standalone confidentiality and integrity analysis of Javascript code. In the presence of bridge function calls (via the interface object), there could be two potential scenarios (as shown in Figure 4.10): (1) a bridge function with some input parameters is invoked from JavaScript, and (2) a bridge function without any parameters is called from JavaScript. In the first case, the interface object may get updated due to the

influence of input parameters from JavaScript. In this case, the analysis injects the values of input parameters into the summary computed in Phase 1 and updates the interface object accordingly. Besides, if the input parameters contain a public channel, i.e., the values can be influenced by an adversary, our analysis considers this an integrity violation. In the second case, the pre-computed summaries with IFC labels are replaced, and the corresponding return value (if any) is tainted. If the results (if any) flow to a public sink in both cases, we mark it as a confidentiality violation.

Algorithm 4 IDE_{js}

Require: m_{js}: Bridge Javascript method
Require: *IfcValue*: IfcValue for bridged interfaces $PathWorkList \leftarrow \emptyset$
1: **procedure** AnalyzeJavascript
2: **for** every v in Variables(m_{js}) **do**
3: Initialize($\langle m_{js}, v\rangle \rightarrow \langle m_{js}, v\rangle, id$)
4: **end for**Propagate($flowfunctions_{js}$)
5: **end procedure**

6: **procedure** $resolveTarget(\langle s, f\rangle, PathWorkList)$ $targetfunctions \leftarrow \emptyset$
7: **for** path $\langle s_0, f_0\rangle \rightarrow \langle s, f\rangle \in PathWorkList$ **do**
8: **if** $type(f_0)$ is a function **then**
9: $targetFunctions \leftarrow f_0$
10: **end if**
11: **end for**return targetfunctions
12: **end procedure**

Algorithm 4 presents the pseudocode of the JavaScript analysis. Similar to the analysis of interface objects, the JavaScript analysis is also embedded in the IFDS/IDE framework. On a high level, the approach is quite similar to the standard IFDS framework, with the exception of handling bridge interfaces via the interface object. In summary, the flow functions are extended to include summary information from the interface object. Besides, these functions provided handling for peculiar cases (Section 4.3) with respect to the use of interface objects in JavaScript. Next, we explain the flow functions in detail.

Table 4.4 contains the flow functions for the Javascript analysis. The execution of these flow functions is similar to those mentioned in Table 4.3. In particular, **New**, **Normal**, **Field Store**, and **FieldLoad** operations are same as described in Table 4.2. We further extend the flow functions to analyze language properties unique to JavaScript and the use of interface objects. One of them is the ability to delete variables and fields, which needs to be modeled in flow functions. Besides, Observation 3 needs to be considered. **Delete variable** and **Delete field** model these behaviors. In the case of **Delete variable**, it removes the variable x from the set of access-graphs. **Delete field** matches the corresponding field in the head of the access graphs and deletes it with the exception of interface objects. Consequently, these access graphs are not propagated further in the process. **Call-to-start** resolves the target function with a backward dataflow analysis elaborated in the next paragraph. In case a bridge function is called, **Call-to-bridge** inserts the function summary edges from Phase 1.

Javascript is a dynamic language, which poses challenges to IFDS/IDE analysis — the object properties may refer to an object or a function; worse, Javascript also allows

Table 4.4: Extended Flow Functions for JavaScript Analysis

New flow: `x = new T()` $f_s(t) = t \cup (x, \oslash)$
Delete Variable $f_s(t) = \underline{\textbf{if}}\ x.\mathbf{F} \in t\ \underline{\textbf{then}}\ t - (x, \mathbf{F})\ \underline{\textbf{else}}\ t$
Delete field $f_s(t) = \underline{\textbf{if}}\ x.\mathbf{F} \in t \wedge head(\mathbf{F}) = f \wedge !x.aliasof(ifcObj)\ \underline{\textbf{then}}\ t - (x, \mathbf{F})\ \underline{\textbf{else}}\ t$
Call-to-start `x=f'(m)` for a function declaration `f'(a)` $f_s = \underline{\textbf{if}}\ (a, \mathbf{F}) \in t\ \underline{\textbf{then}}\ (m_r, \mathbf{F})\ \underline{\textbf{else}}\ \emptyset$ where $m_r = resolveTarget(\langle a, f\rangle, PathWorkList)$
Call-to-Bridge: `x = b.func(p)` $f_s = g(t)$ where g is the set of access-graphs reachable in the *forward* direction of the summary of bridge function from *entry* to *exit*

adding or removing properties as functions. Therefore, it makes it difficult to determine the calling target based on class hierarchies or rapid types. One option is to analyze the properties and compute the set of possible calling targets for that property (CFA analyses). However, this approach is imprecise and will result in many spurious calling targets as such analyses are flow-insensitive. Our analysis performs a *on-the-fly closure analysis* to resolve the function targets.

On-the-fly closure analysis Procedure *ResolveTarget* in Algorithm 4 at Line 6 presents the pseudocode to resolve target functions in **Call-to-start**. Intuitively, the analysis performs a backward dataflow analysis to discover the reachable access-paths and checks if the corresponding access-path is of function type. To our advantage, IFDS stores the reachable-path edges, and therefore for a reachable variable v at a location f_l, there exists a path edge $\langle f_0, v_0\rangle \dashrightarrow \langle f_l, v\rangle$ from the variable v_0 defined in location f_0. At this stage, if the type of the variable v_0 is a function, it returns the function as a calling target. Note that this analysis is field- and flow-sensitive as compared to existing field-based approaches [30]. Since the flow functions strongly update the access-paths at assignments, the analysis is also flow-sensitive and resolves the targets without explicitly doing a closure analysis. However, this approach is not immune to control-flow joins, such as the same paths defined in the if and else part of the block, which is an acceptable imprecision of the analysis. In the spectrum of the analysis, this can be a may-call analysis.

4.5 Implementation

We implemented *IwanDroid* by extending *LuDroid* to extract the bridge details. In particular, we enhanced the string processing of LuDroid by providing the knowledge of string library. We also introduced class hierarchy analysis in LuDroid to resolve dynamic dispatch calls, which was an inherent limitation of LuDroid. For persistence of bridge information, the data was stored in a database which was used by the next phase of the analysis.

In the second phase of the analysis, we use the bridge function analysis and JavaScript

analysis is implemented on the top of IFDS solver from the Wala [121] library. The IFDS analysis in Wala requires the following steps: (1) the set of entry points, i.e., the method from where the analysis will start, (2) an Interprocedural CFG (ICFG) of the whole program, (3) set of transfer functions. The entry points are determined by mapping the bridge method summary from preprocessing step to those in Wala library. For the ICFG, we use a precomputed ICFG with one-level of calling context-sensitivity. Access-paths are represented in a traditional bitset, where each bit represents an access-path. A set bit denotes that the access-path is valid at that program locations. Transfer functions are represented as the transformations over these bitsets. Further, to track the source of sensitive data, it uses the sources and sinks list from FlowDroid [106].

4.6 Evaluation

We empirically evaluate effectiveness of *IwanDroid* in identifying information flows (via WebView) in Android hybrid apps. This study answers the following research questions:

RQ1: How does *IwanDroid* compare to state-of-the-art tools in benchmarks?

RQ2: Does *IwanDroid* scale to large-scale apps?

Multiple approaches [63, 118, 65, 19, 97, 51, 22] have been proposed for the security analysis of WebView in Android hybrid apps; however, most of these studies focus on a specific vulnerability type or a set of vulnerabilities. To the best of our knowledge, there are two static analysis guided tools, called HybriDroid [63] and LUDroid [118], designed specifically for the information flow analysis of WebView. Therefore in this study, we focus on comparing *IwanDroid* with HybriDroid and LUDroid.

Dataset Selection To effectively evaluate RQ1, our choice of apps is motivated by two goals: (1) to have a fair comparison with HybridRoid and LUDroid, (2) to include various cases concerning information flows in WebView. Concerning these goals, we select all apps from the benchmarks these tools provide. Unfortunately, these benchmarks do not contain many cases (including peculiar cases discussed) essential for analyzing information flows via interface objects. To remedy this, we extend these benchmarks to include all missing potential cases (to the best of our knowledge) essential for analyzing information flows via interface objects; finally, we end up with 19 apps.

To effectively evaluate RQ2, i.e., to compare the scalability, we downloaded apps from the popular open-source app hosting site FDroid [73]. Our choice of large-scale apps is motivated by two goals: (1) exclude toy-apps, i.e., the app size should be at least 2 MB, (2) the app should contain at least one instance of *addJavascriptInterface* with JavaScript enabled. After applying these criteria, we ended up with 687 apps. All experiments were conducted on a personal MacBook Pro with 64 GB RAM and an Apple M1 Max processor.

4.6.1 RQ1: Comparison on Benchmarks

Table 4.5 presents the evaluation results of HybriDroid, LUDroid, and *IwanDroid* on benchmarks. The first and second columns show the app identifiers. The third column shows if the app has at least one component where *addJavascriptInterface* is invoked

Table 4.5: Results on micro-bench by HybriDroid, LUDroid, and *IwanDroid*

App ID	App Name	JS Enabled	HybriDroid		LUDroid	*IwanDroid*
			version 2016	current version		
1	HelloCordova	no	NA	NA	NA	NA
2	HelloHybrid	yes	success	failed	success	success
3	HelloScript	yes	success	failed	failed	success
4	HelloScript_simple	yes	success	failed	success	success
5	HelloScript_test	yes	success	failed	success	success
6	HybridAPIArgNum	yes	success	failed	success	success
7	NormalAliasFlowTest	no	NA	NA	NA	NA
8	NormalAliasFlowTest_objfield_false	no	NA	NA	NA	NA
9	NormalAliasFlowTest_objfield1	no	NA	NA	NA	NA
10	StrongUpdate	yes	failed	failed	success	success
11	StrongUpdateCaseA	yes	failed	failed	success	success
12	StrongUpdateCaseB	yes	failed	failed	failed	success
13	StrongUpdateCaseC	yes	failed	failed	failed	success
14	JSUpdateCaseD	yes	failed	failed	failed	success
15	JSUpdateCaseE	yes	failed	failed	failed	success
16	JSUpdateCaseF	yes	failed	failed	failed	success
17	JSUpdateCaseG	yes	failed	failed	failed	success
18	DynamicJSCaseH	yes	failed	failed	failed	failed
19	DynamicAliasCaseI	yes	failed	failed	failed	failed

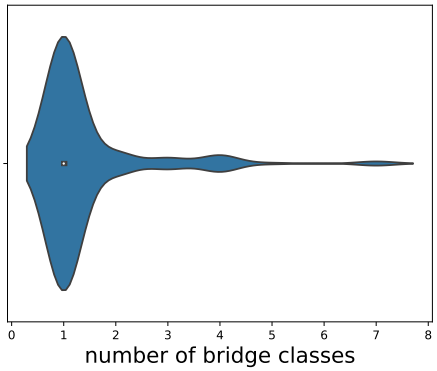


Figure 4.11: Bridge Class distribution

with JavaScript enabled. A *success* label indicates that all the information flows via WebView (if any) are correctly reported. A *failed* lable suggests that the tool either missed an information flow (false negative) or incorrectly reports it (false positive). We obtained HybriDroid from its publicly available Github repository [8], while we obtained LUDroid directly from its authors.

The latest commit (#a5bc657) of HybriDroid was submitted on July 2019. Thus, we configured it to evaluate this latest version. However, HybriDroid at this version could not run on any benchmarks and crashed with an error message pointing to problems in JavaScript analysis. To remedy this, we obtain an older version (#db54abe) nearing the time of its paper acceptance. With this version, we were able to run it on some benchmarks; however, it failed to provide any results in the majority of the cases, i.e., it successfully reported only five of 15 information flows. We observed that while HybriDroid handles simple flows well, e.g., for app id 3 and 4, it fails to tackle any complex flow, e.g., app id 11-13. Besides, it struggles with JavaScript analysis and fails to analyze integrity violations, i.e., flows from JavaScript to Java. Similarly, in our experience LUDroid did not perform automatic analysis of JavaScript code. It reported information flows as a leak if sensitive data is shared from the native side to JavaScript, which results in many false positives. Besides, it does not understand any peculiar flows and ignores them completely. In the end, we obtained six correctly reported flows.

Finally, *IwanDroid* successfully detected 13 information flows; four apps do not enable JavaScript communication and hence do not require analysis by *IwanDroid* . In the remaining two cases, it failed to report a leak correctly. In particular, *IwanDroid* handles simple String operations but struggled for cases where the input JavaScript was dynamically created. Similarly, it reported a false positive when the interface variable was dynamically created under two aliases; however, this is a general limitation of static analysis, and *IwanDroid* conservatively reports such flows.

IwanDroid significantly outperforms the state-of-the-art tools and correctly reports 13 of 15 information flows in benchmarks. It identifies confidentiality violations, i.e., flows from Android to JavaScript and finally to a public sink, and integrity violations, i.e., Javascript invading Android sensitive data.

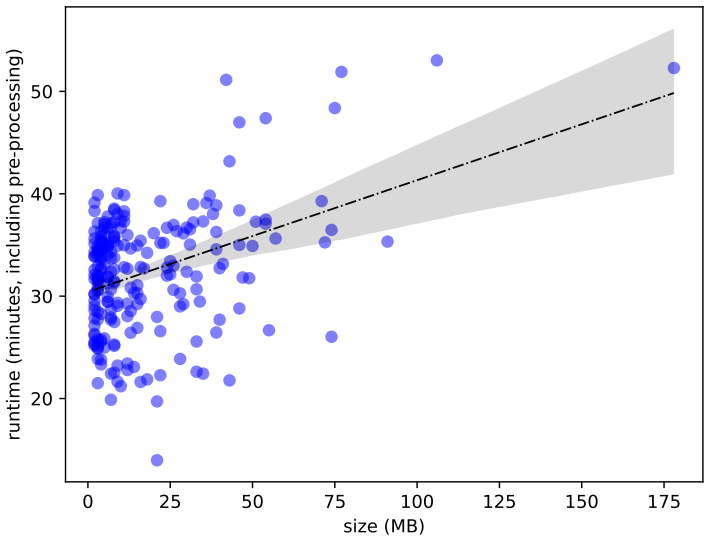


Figure 4.12: Relation between runtime and size of the apk file

4.6.2 RQ2: Scalability of *IwanDroid*

Before delving into the specifics of RQ2, we present a few noteworthy observations made during the evaluation. The distribution of bridge classes shared by each app in our dataset is depicted in Figure 4.11 . On average, each application shares two bridge classes and five bridge methods (with JavascriptInterface annotation).

HybridDroid has encountered crashes on several apps, while in others, it failed to report any IFC violations. The reason behind this could be attributed to the lack of maintenance. On the other hand, LUDroid does not perform Javascript analysis and produces no information leak output. Thus, for large-scale apps, we did not manually analyze its output (a set of tables). *IwanDroid* identified 136 confidentiality and 142 integrity violations. Two authors manually validated 20 of each violation, and *IwanDroid* accurately identified 17 confidentiality violations and 15 integrity violations. In some cases, *IwanDroid* reported false positives, which were due to its conservative design. For instance, it reported three integration violations when Javascript modified the ad's banner size. Figure 4.12 shows the runtime distribution of *IwanDroid* on large scale apps; the runtime increases with the size of app. The runtime also includes the time to decompile the app. Next, we present a case study containing a confidentiality and integrity violation.

4.6.3 Case Study — Bookreader App [28]

We present a case study of the integrity and confidentiality violation reported by *IwanDroid* . Listing 4.1 shows a data leak via the *customAjax* method and Listing 4.2 shows an integrity violation via the *customSubmit* method. In this case study, two bridge methods (native side methods), namely *customAjax* and *customSubmit*, are accessed on the JavaScript side via the interface object *interception*. Line 4 of the JavaScript code snippet (Listing 4.1) invokes *customAjax* method via the interface object *interception*. The return value of *customAjax* encapsulates the user-sensitive information in

plain text on the Java side, which is sent via an XMLHttp request on the JavaScript side. Line 3 of the JavaScript code snippet (Listing 4.2) injects data taken from the form attributes collected on click of submit button (injects the parameters via *customSubmit* method).

Listing 4.1: Confidentiality violation detected by *IwanDroid*

```
1 XMLHttpRequest.prototype.send =
2 function(form) {
3         var params = this.params;
4         var response = interception.customAjax(
5                 params.method, params.url, params.user, params.password,
                      JSON.stringify(this.header), form);
6         ... // rest of code
7 }
```

Listing 4.2: Integrity violation detected by *IwanDroid*

```
1 function interceptor(e) {
2 ... // populate the variable "aa"
3 interception.customSubmit(
4 form.attributes['method'] === undefined ? null :
      form.attributes['method'].nodeValue,
5 form.attributes['action'] === undefined ? null :
      form.attributes['action'].nodeValue,
6 form.attributes['enctype'] === undefined ? null :
      form.attributes['enctype'].nodeValue,
7 JSON.stringify({"form" : aa}));
8 }
```

4.7 Remarks

IwanDroid is susceptible to a few limitations. First, *IwanDroid* does not handle dynamically constructed strings in *eval* in Javascript. In particular, we do not handle code generated at runtime through *eval*. *eval* makes static analysis difficult by introducing source code generated through string manipulations. In principle, it is possible to augment techniques from resolution of *eval* operations with our analysis. In the case of javascript code (in *loadUrl* method) manipulated dynamically, e.g., via string operations, we implement a very basic String resolver to handle these cases via tracking calls to String operation APIs. A more advance dynamic analysis could supplement our analysis to resolve such dynamically constructed Javascript.

This chapter proposed a framework for identifying confidentiality and integrity violations in hybrid apps. Our analysis devises a demand-driven approach and leverages IFDS/IDE frameworks to compute *related* functions' summaries to reuses these in discovering the inter-language integrity and confidentiality violations. Our prototype *IwanDroid* successfully detected such violations in benchmark and large-scale apps. In the next part of the thesis, we generalize the idea of using function summaries across other multilingual frameworks for fundamental static analysis technique, namely, pointer and call-graph analysis.

Part III

Unifying Analyses

Chapter 5

Unifying Unilingual Analyses for Multilingual Programs

5.1 Introduction

In the last chapter, we saw a demand-driven analysis for for analyzing information flows in Android Hybrid apps. In this chapter, we shift our focus from analysis of specific language combinations to proposing a general framework for integrating single language analyses to multilingual programs.

Of late, many multilingual programming environments have emerged, such as GraalVM and Android NDK, while many other interpreted languages are using FFIs for reusability. Example of these include the FFI communication in R language with C [52], Python with C [79], among many others. Developers often have incentives in using these techniques such as to implement their performance critical components in languages such as C and export it to other languages. This has accelerated the adoption of polyglot programming in software development [60, 43, 27, 72, 6, 7, 65, 63, 116, 100, 15]. Again, a limited understanding of these interoperations, could include errors in the code, ranging from type-bugs to security bugs [60, 43, 72].

In this aspect, we investigate into integrating existing foundational analyses — pointer analyses and call-graph analyses in this chapter. Last couple of years of research have resulted in sophisticated static analysis for single language programs [84, 101, 117]. Here, we propose a technique to combine those analyses for multilingual programs. Such an approach to inter-language analysis can benefit from utilizing the language specific heuristics and optimizations [66]. To this end, we present a modular technique for call-graph and pointer analysis that combines different monolingual analysis. A modular technique, in which each language module is studied independently and the findings combined for a whole-program analysis, appears to be promising for scalability and precision [23].

Integrating analysis results of several language models in a modular style is one of the main obstacles. Existing approaches, such as Lee *et al.* [66] translate data-flow summaries of the guest language's operations to the host language. After injecting these summaries into the host program, this program is analyzed. However, there are two primary disadvantages to this approach. First, due to the complexity of modern

languages' syntax and semantics, translating all conceivable consequences of the guest language to the host language is a complex and potentially error-prone undertaking. It entails modeling every language aspect of the guest language and transforming them into the host language's semantics. Lee *et al.* [66] consequently only convert a *subset* of the guest language's semantics to the host. Second, the translated program is analyzed using the model of the host language and thus cannot take advantage of the guest language's unique characteristics, preventing language-specific optimizations for a most appropriate result of the guest module.

Another drawback of present techniques is that they rely on explicit host-guest language boundaries [79]. These boundaries are identified via explicit syntactic markers, which are typically predefined function calls, signatures, or data structures involved in interlanguage communication. Static analyses can detect these predefined syntactic structures and define models that replicate their behavior in the host language. However, current multilingual programming environments such as *GraalVM* or Android's *WebView* allow code from both languages to be mixed, blurring these explicit boundaries. For instance, WebViews allows messages from the guest to the host as well, which are not linked to explicit syntactic markers but communicate via a shared object. Prior research demonstrated that these functionalities are widespread in contemporary applications [118]. Furthermore, it is unavoidable that future multilingual programming environments will offer similar patterns, making it impractical to rely entirely on static language boundaries.

5.1.1 Approach in a nutshell

We propose a novel strategy to address these challenges. Our modular technique is oblivious of the particular language models: It can instantiate various combinations of host and guest analyses to take advantage of the language-specific models, thereby inheriting the precision, scalability, and soundness improvements of the underlying analyses. Notably, our technique extends existing host and guest language analyses to include essential information concerning call and data flows to their counterparts. Given a multilingual model, our analysis finds the communication endpoints and features (shared classes/functions) of a host that a guest can use. With this information, an *interlanguage call graph* is created by joining the missing call flow edges in the host's and guest's individual call graphs. To resolve the data flows, our technique first builds an intra-procedural function summary of all the interoperating functions discovered in the interlanguage call graph. The function summaries are determined using intra-procedural points-to-analysis according to an *Andersen style* [109] constraint scheme. Next, these summaries are resolved such that the points-to sets of variables shared across languages contain the correct and complete references. To this end we propose innovative *summary specialization* and *unification* techniques. Unification is an alias-aware technique that unifies the variables' must-aliases across host and guest. The resolved summaries are then translated to be consistent with the host and guest's analysis models. Once the call and data flows are resolved, the analysis is performed as a whole by merging the host and guest results independently.

We implemented our approach into a tool called Conflux. To evaluate the efficacy and generalizability of our approach, we applied Conflux to two multilingual language models; 1) Java-C communication via the Android NDK [47] and 2) Java-Python communication in GraalVM [87]. Conflux outperforms the previous state-of-the-art and

$$
\begin{array}{lcl}
m \in \textit{host methods} & := & m_1 \mid m_2 \mid m_3 \mid \cdots \mid m_n \\
i \in \textit{InterfaceVars} & := & i_1 \mid \cdots \mid i_n \\
v \in \textit{variables} & := & v_1 \mid v_2 \mid v_3 \mid \cdots \mid v_n \mid \textit{InterfaceVars} \\
g \in \textit{fields} & := & g_1 \mid g_2 \mid g_3 \mid \cdots \mid g_n \\
s \in \textit{statements} & := & v = \textbf{new}\ T() \mid i = \textbf{new}\ T() \mid \textbf{return}\ v \\
 & & v' = v.m() \mid v' = v.g \mid v'.g = v \mid \textbf{if}\ (e)\ \{s\}[\textbf{else}\ \{s\}] \\
 & & v = \textbf{eval}(\textit{program}) \mid \textbf{op}\ s_1\ s_2 \mid s_1; s_2 \mid \textbf{while}\ (e)\ \{s\}
\end{array}
$$

(a) **Host Language**

$$
\begin{array}{lcl}
f \in \textit{guest methods} & := & f_1 \mid f_2 \mid f_3 \mid \cdots \mid f_n \\
w \in \textit{variables} & := & v_1 \mid v_2 \mid v_3 \mid \cdots \mid v_n \mid \textit{InterfaceVars} \\
h \in \textit{fields} & := & h_1 \mid h_2 \mid h_3 \mid \cdots \mid h_n \\
t \in \textit{statements} & ::= & v = \textbf{new}\ T() \mid w' = w.f() \mid \textbf{return}\ v \\
 & & w' = i.m() \mid w' = w.h \mid \textbf{if}\ (e)\ \{t\}[\textbf{else}\ \{t\}] \\
 & & w'.h = w \mid \textbf{op}\ t_1\ t_2 \mid t_1; t_2 \mid \textbf{while}\ (e)\ \{t\}
\end{array}
$$

(b) **Guest Language**

Figure 5.1: Host and guest language DSL (PolyDSL)

successfully analyzed 31 out of 39 apps using Java-C communication in Android NDK. Besides, Conflux successfully analyzed two of three Java-Python apps using GraalVM APIs. To the best of our knowledge, Conflux is the first multilingual analysis framework that can be applied to multiple multilingual language models.

5.2 Modular Inter-language Analysis

There are various multilingual language models (Java-C, Java-Javascript, Java-python, Javascript-wasm, Java-Javascript-wasm, etc.) and communication settings: Some host and guest language models operate on a heterogenous execution environment, e.g., Java-Javascript. At the same time, some host and guest language models operate in a homogeneous environment, e.g., GraalVM. Similarly, in some models, even the mutation/replication of the Interface object is permitted (Java-Javascript), while in others, it is not feasible/possible (e.g., Java-C).

In this work, we propose a modular analysis framework for handling multilingual language models based on *imperative/OO paradigms*. To this end, we define a DSL, called PolyDSL, sufficiently abstracted from the general multilingual model's semantics to ease the presentation and explanation of our approach.

A core multilingual model PolyDSL, depicted in Fig. 5.1, presents the syntax required for host-to-guest and guest-to-host communication in a multilingual model following the foreign function interface (FFI) model[1]. This language is based on an in-depth examination of existing multilingual programming environments, specifically GraalVM and Android hybrid apps. The host and guest languages have been defined as standard imperative/OO programs with the addition of inter-language communication

[1]Other potential communication facilities, such as middleware-based remote method invocation, have been covered by other research (e.g. [117, 13, 68]) and are thus out of scope of this approach.

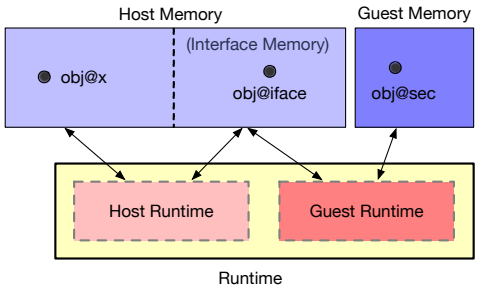


Figure 5.2: Memory Model

constructs. Traditional details such as class and field syntax are omitted to concentrate on the analysis. The functions, variables, field variables, and statements in both the host and guest languages are *equivalent*.

Interface variables (*InterfaceVars*) and *eval* are the two key features that make it possible for the host and guest to communicate with each other. The host language construct *eval* executes a guest language program or expression and returns the corresponding result to the host. The host language defines *InterfaceVars* for the guest to access host functionalities. InterfaceVars are host objects that the guest can use to invoke host methods. Technically, they are ordinary objects that serve as a bridge between a host's class and the guest. In PolyDSL, one host method can create at most one InterfaceVars object, and the guest can access all methods of the corresponding InterfaceVars object. Multilingual environments that allow a subset of the host language features to be accessed by the guest are a specific instance of this premise.

5.2.1 Operational Semantics

Memory Model. Figure 5.2 shows the memory model of the host and guest execution in PolyDSL. It has two separate heap memories: one for the host and the other for the guest. *Interface* memory is the part of the host's heap memory accessible to the guest. Each heap memory is a map from access paths to heap objects.

Semantics. The state of the program is a quintuple, $(\mathcal{H}_H, \mathcal{H}_G, \mathcal{H}_S, \mathcal{E})$. $\mathcal{H}_H$ and $\mathcal{H}_G$ refer to the heap memories accessible by the host and guest language, respectively. $\mathcal{H}_S$ refers to the heap memory accessible by both the host and guest language. $\mathcal{E}$ refers to the evaluation context, and it can be either $\mathcal{E}_H[m]$ or $\mathcal{E}_G[f]$, which refers to the evaluation of method m in the host language and function f in the guest language, respectively. Our model assumes that the interoperable types are valid, i.e., if the guest function contains a **return** statement that returns the value to the host method, then it is type-compatible with the host language. Type transformation is represented via the function $\mathit{transformType}(o_i, T_G, T_H)$, which type-casts the object o_i from guest type T_G to the host type T_H.

$$\mathit{transformType}(o_i, T_G, T_H) = \begin{cases} \hat{o}_i & \text{transformation from } T_G \text{ to } T_H \text{ is defined} \\ \textbf{error} & \text{otherwise} \end{cases}$$

Figure 5.3 shows the rules for the operational semantics. Many of the trivial rules have been omitted to keep the focus on the main topic. HeapAllocHost updates the

HeapAllocHost
$$(\mathcal{H}_H, \mathcal{H}_G, \mathcal{H}_S, \mathcal{E}_H[m])\ \{v = \text{new T}()\}\ (\mathcal{H}_H \texttt{ ++ } [v \mapsto obj_t], \mathcal{H}_G, \mathcal{H}_S, \mathcal{E}_H[m])$$

HeapAllocGuest
$$(\mathcal{H}_H, \mathcal{H}_G, \mathcal{H}_S, \mathcal{E}_G[f])\ \{v = \text{new T}()\}\ (\mathcal{H}_H, \mathcal{H}_G \texttt{ ++ } [v \mapsto obj_t], \mathcal{H}_S, \mathcal{E}_G[f])$$

HeapAllocInterface
$$(\mathcal{H}_H, \mathcal{H}_G, \mathcal{H}_S, \mathcal{E}_H[m])\ \{i = \text{new T}()\}\ (\mathcal{H}_H, \mathcal{H}_G, \mathcal{H}_S \texttt{ ++ } [i \mapsto obj_t], \mathcal{E}_H[m])$$

Eval
$$(\mathcal{H}_H, \mathcal{H}_G, \mathcal{H}_S, \mathcal{E}_H[m])\ \{\mathbf{eval(m_g)}\}\ (\mathcal{H}_H, \mathcal{H}_G, \mathcal{H}_S, \mathcal{E}_G[m_g])$$

BridgeFunctionCall
$$\frac{i \in keys(\mathcal{H}_S) \qquad \texttt{i.h() in function f}}{(\mathcal{H}_H, \mathcal{H}_G, \mathcal{H}_S, \mathcal{E}_G[f])\ \{i.h()\}\ (\mathcal{H}_H, \mathcal{H}_G, \mathcal{H}_S, \mathcal{E}_G[h])}$$

GuestToHostReturnSuccess
$$\frac{o_i = \mathcal{H}_G[r] \qquad v \text{ is variable at return site} \qquad transformType(o_i) = \hat{o}_i}{(\mathcal{H}_H, \mathcal{H}_G, \mathcal{H}_S, \mathcal{E}_G[f])\ \{\mathbf{return}\ r\}\ (\mathcal{H}_H \texttt{ ++ } [v \mapsto \hat{o}_i], \mathcal{H}_G, \mathcal{H}_S, \mathcal{E}_H[m])}$$

Figure 5.3: Interlanguage Operation Semantics for the PolyDSL

Listing 5.1: Host language code

```
1  foo() {
2    i = new BridgeClass(); // obj@iface
3    eval("set()"); // invoke the guest code
4    // do some computation
5    x = eval("getAndLeak()");
6    return x;
7  }
8  BridgeClass.setSecret(secret) {
9     this.secret = secret;
10 }
11 BridgeClass.getSecret() {
12    return this.secret;
13 }
```

host memory with the new object obj_i. It only performs this operation in the evaluation context of the host language. HostAllocGuest is analogous to host, albeit with the guest language's heap memory and evaluation context. HeapAllocInterface updates the shared memory with the corresponding memory of the objects (modified by either the host or the guest program). Eval evaluates the guest language program by switching the evaluation context from the host to the guest. GuestToHostReturnSuccess returns the transformed value of the object in the guest language to the host.

Listing 5.1 and Listing 5.2 employ our language constructs to exemplify a simplified multilingual model. Line 2 in Listing 5.1 declares an interface variable i of the host class *BridgeClass*, granting the guest language access to all methods of *BridgeClass* via i. In Lines 3 and 5, the host uses the *eval* method to invoke guest language code. In Listing 5.2, guest accesses the *BridgeClass*'s *setSecret* and *getSecret* methods at Line 17, 20.

Listing 5.2: Guest language example

```
15 set() {
16   v = new Secret(); //obj@sec
17   i.setSecret(v);
18 }
19 getAndLeak() {
20   g = i.getSecret();
21   leak(g);
22   return g;
23 }
```

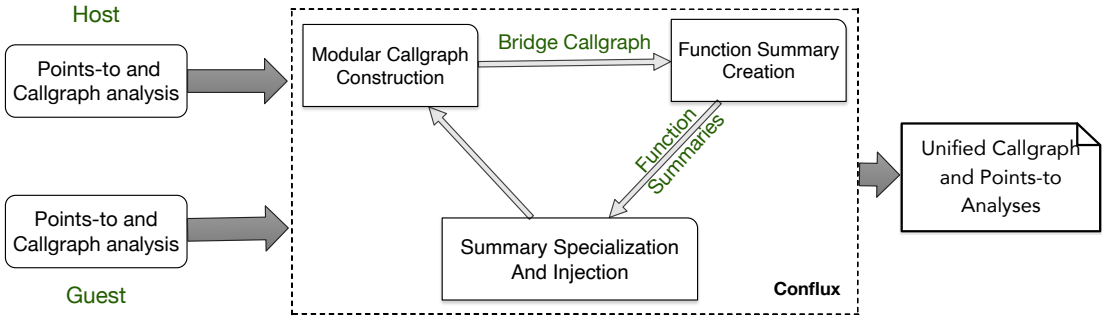


Figure 5.4: Analysis Overview

Example 3. *We illustrate some essential operations in Listing 5.1 and Listing 5.2 via the operational semantics. The creation of the interface object at Line 2 proceeds as* $([],[],[],\mathcal{E}_H[m])$`i = new BridgeClass()`$([],[],[i \mapsto obj@iface],\mathcal{E}_H[m])$*, i.e., it adds the interface object obj@iface to the shared memory following the rule* HeapAllocInterface*. The following statement is an* eval *operation, so the new state becomes* $([],[],[i \mapsto obj@iface],\mathcal{E}_G[f])$*, which switches the execution to the guest language following the rule* Eval*. In Line 16 (Listing 5.2), the new state becomes* $([],[v \mapsto obj@sec],[i \mapsto obj@iface],\mathcal{E}_G[f])$*, following the rule* HeapAllocGuest*. The following statement accesses the bridge method and the state transforms to* $(\mathcal{H}_H,\mathcal{H}_G,\mathcal{H}_S,\mathcal{E}_G[setsecret])$*, following the rule* BridgeFunctionCall*, which is then evaluated with the standard rules.*

Analysis Overview Fig. 5.4 depicts the components of our analysis. The framework builds upon a scalable context-sensitive pre-analysis for the host and guest languages. It then computes other building blocks of the analysis, a call graph (constructed with a modular analysis, Section 5.2.3) and intra-procedural function summaries (Section 5.2.4). Next, we describe all components of our framework in detail.

5.2.2 Preliminaries

Our approach leverages the points-to and callgraph analyses computed independently for the host and guest languages (i.e., modulo the foreign function interfaces). The model of the points-to analysis is depicted in Fig. 5.5a. It includes a set of variables V, a set of heap objects H, and a collection of contexts C. C can be modeled in various notions; by using objects as contexts (object sensitive), by invoking function call sites (call-site sensitive), and many others. Context set C improves the analysis by removing unrealizable paths (such as paths from unmatched function calls). A context *ctxt* is a string constructed using elements from C. For the sake of presentation, we choose to omit contexts in the sequel.

V : Set of variables	C : Set of contexts
H : Set of heap objects	$Ctxt^j \subseteq c_1^j \times c_2^j \times \ldots \times c_n^j,\ c_i^j \in C$

(a) Model of Pointer Analysis

$\langle i, ctx\rangle \mapsto \{\texttt{obj@iface}\}$
$\langle x, ctx\rangle \mapsto \emptyset$
$\langle g, ctx'\rangle \mapsto \emptyset$
$\langle v, ctx'\rangle \mapsto \{\texttt{obj@sec}\}$

(b) Points-to analysis

$\langle foo, ctx\rangle \rightarrow \langle bar, ctx\rangle$
$\langle set, ctx'\rangle \rightarrow \langle setSecret, ctx'\rangle$
$\langle getAndLeak, ctx'\rangle \rightarrow \langle getSecret, ctx'\rangle$
$\langle getAndLeak, ctx'\rangle \rightarrow \langle leak, ctx'\rangle$

(c) Callgraph analysis

Figure 5.5: Representation of Call-graphs and points-to analysis taken as input

A points-to analysis generates a mapping from variables to a set of heap objects, referred here as *variable points-to set* (under a context *ctxt*). A variable points-to mapping, $VarPointsTo : V \mapsto \mathcal{P}(H)$, is a mapping from the set of variables to heap objects. A points-to set for a variable $v \in V$ is a set of heap objects $h \subseteq H$. In the presence of fields in a programming language, one adds one more resolution by mapping object and field pairs to sets of heap objects, referred to as *field points-to set*. If f is a field for a heap object o_i, its points-to set is the set of heap objects, O_{if}, referred through the field f of the heap object o_i. A field points-to mapping, $FldPointsTo : H \times Field \mapsto \mathcal{P}(H)$ maps each pair of a heap object and a field to a set of heap objects.

In the example in Listing 5.1 and 5.2, a pre-analysis of the host language infers that i and v under some contexts refer to `obj@iface` and `obj@sec`, respectively (Fig. 5.5b). Since x captures an FFI call's return value, the analysis ignores this operation. Similarly, in the guest language, g captures the value of an FFI call, which is also ignored. As expected, the pre-analysis result is incomplete and misses crucial data flows. Listing 5.1 and 5.2 show that the methods `set` and `getAndLeak` are reachable from `foo`. Section 5.2.3 extends the pre-analysis to resolve these edges by constructing an inter-language callgraph. Further, the guest object `obj@sec` defined at Line 16 is written to the BridgeClass's field `secret` through the bridge invocation at Line 17. At Line 20, it reads from this field into g and at the end returns it to the variable x in `foo`. Therefore, g and x should refer to `obj@sec`. We resolve these dataflows with our modular callgraph construction and summary specialization approach (Section 5.2.4.3).

5.2.3 Call-Graph Construction

The modular call-graph construction determines the edges for inter-language function calls happening through *eval* and interface variables. It builds upon the incomplete call-graphs obtained from the pre-analysis phase. The process begins with an incomplete host call graph and expands it to include all bridge class methods; our analysis assumes that all bridge class methods can be invoked from the guest. Initially, the extended host call-graph (G_{host}) includes potentially disconnected nodes (we include methods of the bridge class even if the host code does not invoke them). The next objective is to complete the missing edges in the host and guest call graphs (G_{guest}), i.e., to add the corresponding edges from host to guest (via *eval*) and vice versa (via *interface variables*).

Algorithm 5 describes the modular call graph generation. At first we merge the

Algorithm 5 ModularCallgraphAnalysis

Require: Host and guest call graph G_{host}, G_{guest}
Ensure: Interlanguage callgraph G
Ensure: Bridge callgraphs $B_{m_h}, m_h \in G_{host}$
1: $G \leftarrow G_{host} \cup G_{guest}$
2: $E_{bc} \leftarrow \emptyset$
3: **for eval**$(m_g) \in m_h$ **do**
4: **if** $m_g \in G_{guest}$ **and** $m_h \in G_{host}$ **then**
5: ▷ Extend G to include an edge (m_h, m_g)
6: **if** $i \in InterfaceVar(m_h)$ **and** $i \in use(m_g)$ **then**
7: $E_{bc} \leftarrow E_{bc} \cup DiscoverBridgeEdges(i, m_h, m_g)$
8: **end if**
9: **end if**
10: **end for**$G \leftarrow G \cup E_{bc}$
11: $B_{m_h} = makeGraph(E_{bc})$

Algorithm 6 DiscoverBridgeEdges

Require: Interface variable i, host language method m_h, guest language method m_g
Ensure: Bridge call edges E_{bc}
1: **for** invocation $x.m()$ in m_g where $x \in MayAlias(i)$ **do**
2: add *bridge call edge* (m_h, m) to E_{bc}
3: **end for**
4: ▷ *alias* as a parameter to another guest function
5: **for** invocation $m'_g(*, x, *)$ in m_g where $x \in MayAlias(i)$ **do**
6: $DiscoverBridgeCalls(x, m_h, m'_g)$
7: **end for**

independent host and guest call graphs. The next step identifies guest methods invoked via *eval* in a host method (Line 3), for which an edge is added from the host to the guest method (Line 5). The guest language can also access the host's methods via interface objects. Such flows are added at Line 6 (explained in Algorithm 6). The key idea is to search the guest methods for invocations of bridged host methods via interface variables. If such a call exists, edges between the corresponding guest and host methods are added together with edges from the original host method to the bridged host methods. The latter edges are referred to as *bridge call edges*. They denote the indirect call relationship to host language methods through bridge objects in the guest language.

Definition 9 (Bridge call edge.)**.** *Let i be an interface variable shared from a host method $m_h \in G_{host}$ and $m_g \in G_{guest}$ such that m_g invokes a bridged method $m_b \in G_{host}$ via the interface variable i (or its aliases). Then, a bridge call edge, denoted as E_{bc}, is a transitive call edge from m_h to m_b in the interlanguage call graph G.*

Figure 5.6 shows the transitions involved in creating an interlanguage call graph for the example in Listing 5.1. Step 0 depicts the call graph obtained during the pre-analysis phase. Initially, G_{host} has no edges originating from *foo* as the pre-analyses is oblivious to edges from foreign function calls. The host call graph is extended to include the bridge methods (step 1). The call graph is expanded further to include edges that perform indirect invocation of guest methods through eval (step 2). In the last step, the call graph is extended to include the bridge call edges (step 3); *setSecret* and *getSecret* are called via the interface variable *i*. Therefore, bridge call edges (*foo*,

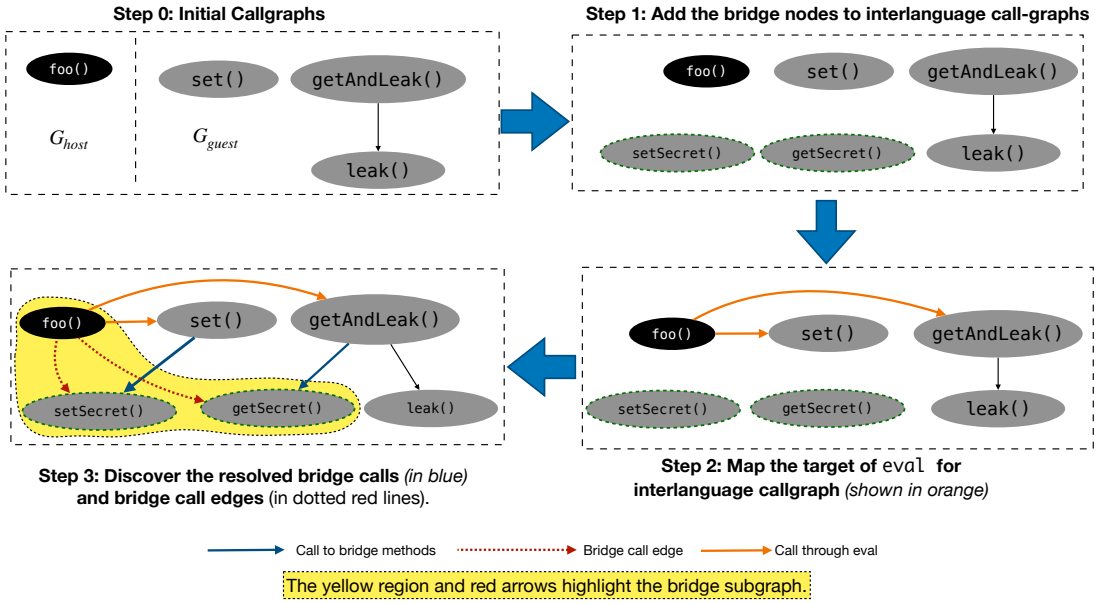


Figure 5.6: Call-graph for program in Listing 5.1 and Listing 5.2

Algorithm 7 ANALYZE

Require: $\mathcal{P}_h$ Host language pre-analysis, $\mathcal{P}_g$ Guest language pre-analysis
Require: G_{host}, G_{guest} host and guest language callgraphs
Ensure: Analysis $\mathcal{P}_{modular}$

```
repeat
    G, B_{m_h} ← ModularCallgraphAnalysis(G_host, G_guest)
    F ← SpecializeSummary(B_{m_h}, P_g)
    for f ∈ functions(B) do
        R^f ← Projection(f)
        P_modular ← InjectSummary(R^f, P_h, P_g, F^f)
    end for
until P_modular is unchanged (i.e., reaches a fixed point)
```

setSecret) and (*foo, getSecret*) are added to G. Bridge call edges are used to determine the methods required for summary-specialization and unification in our analysis. We define the notion of bridge callgraph as an directed graph formed by bridge edges and nodes in Definition 10.

Definition 10 (Bridge callgraph)**.** *A bridge callgraph* $B_{m_h}(G_{host}, E_{bc})$ *is a connected intra-language graph such that* E_{bc} *comprises all bridge call edges originating in* $m_h \in G_{host}$*.*

The shaded part of Fig. 5.6 (step 4) shows the bridge callgraph originating in *foo()* for the example in Listing 5.1 and Listing 5.2. An indirect call from *foo* to *setSecret* and *getSecret* happens through the bridge object i. The bridge callgraphs are the core data structures used in summary specialization, unification, and injection.

5.2.4 Pointer Analysis for Interlanguage Operations

We now present the analysis's core pillar – *summary specialization*. Summary specialization resolves the dataflows occurring in the bridge callgraph from the guest to

$$\textsc{Assign}\ \frac{x = y}{[y] \subseteq [x]} \qquad \textsc{Load}\ \frac{v = x.f}{[x.f] \subseteq [v]} \qquad \textsc{Store}\ \frac{\texttt{x.f = v}}{[v] \subseteq [x.f]}$$

$$\textsc{New}\ \frac{\texttt{v = new Obj()} \qquad pts(v) = \mathcal{P}(v, ctxt) \qquad ctxt = CallContext(v)}{\forall o_i \in pts(v) : \{o_i\} \subseteq [v]}$$

$$\textsc{Func}\ \frac{\texttt{x = func(args)} \qquad pts(x) = \mathcal{P}(x, ctxt) \qquad ctxt = CallContext(x)}{\forall o_i \in pts(x) : \{o_i\} \subseteq [x]}$$

Figure 5.7: Constraint system ($\mathcal{C}_s$) for intra-procedural function summaries

the host and vice versa. The procedure is divided into three steps. Initially, it computes function summaries leveraging the information gained during the pre-analysis. The function summaries are determined using an intra-procedural points-to-analysis. The second step involves merging and propagating these intra-procedural summaries to the functions engaged in the bridge callgraph to resolve the dataflows occurring via the bridge. At this stage, the analysis has identified the dataflows from guest to host through interface variables. These resolved summaries are injected into the corresponding guest methods in the third step. Once we have inserted summaries into the bridge methods, the analysis naturally shifts to the whole program analysis. Algorithm 7 shows the overall workflow of our approach. It runs by computing the call graphs and unifying the pre-analyses via summary specialization until it no longer changes, i.e., achieves a fixed point. Next, we explain each technique, i.e., SpecializeSummary, Unification, and InjectSummary in detail.

5.2.4.1 Function Summary

Function summaries in our analysis are obtained through a light-weight intra-procedural pointer analysis. Our function summaries have the form $\mathcal{AP} \mapsto H \cup \emptyset$, where H symbolizes the collection of heap objects, $\mathcal{AP}$ signifies the set of access-paths, and $\emptyset$ indicates that the associated summary has not been resolved. An access path consists of the base variable (quintessentially a local variable) followed by a sequence of field accesses (denoted as `x.f.g.h`). We choose the access path representation because it inherently models field-sensitivity [67] and is bounded, making it suited for data-flow analysis [58].

The analysis is based on a straightforward constraint-based approach in which the constraints are reinforced with resolved values from the pre-analysis. We create the function summaries using constraints of an Andersen's style analysis (i.e., inclusion-based) [109]. These constraints have been shown to be precise, and they have become the de-facto representation for pointer analysis in programming languages that support heap manipulation [110]. In general, inclusion-based constraints face scalability issues due to the cubic time and quadratic space complexity on the number of variables; yet, the function summaries in this phase are calculated intra-procedurally and hence confined to a small number of variables.

Fig. 5.7 shows the constraint system to generate a function summary. We use the symbol $[var]$ to denote the constraint variable for var, $\mathcal{P}$ denotes the pre-analysis, and $pts(var)$ denotes the points-to set for variable var. Assign, Store, and Load are standard Andersen style constraints. The rule New queries the points-to set of the variable from the pre-analysis and constructs a constraint for each heap object. In the case of function calls Func directly obtains the points-to set of the result-

Algorithm 8 SPECIALIZESUMMARY

Require: Bridge callgraph $B_{m_h}(G_{host}, E_{bc})$
Require: Guest pre-analysis $\mathcal{P}_g$
Ensure: Function summary constraints $\mathcal{F}$
1: **for** function node $N_i \in G_{host}$ **do**
2: ▷ Generate constraints from constraint system in Fig. 5.7
3: $\mathcal{C}_i^N \leftarrow MakeConstraints(N_i, \mathcal{C}_s)$
4: **for** guest function node N_g invoking $N_i(\bar{q})$ (calling bridge method) **do**
5: ▷ $\bar{q}$ is list of formal parameters
6: let $N_i(\bar{t})$ be the function invocation in N_g
7: ▷ $\bar{t}$ is the list of actual parameters
8: $\mathcal{P}_g(t_j) \subseteq [q_j]$ to $\mathcal{C}_i^N$ where $t_j \in \bar{t}, q_j \in \bar{q}$ ▷ add points-to set of t_j from $\mathcal{P}_g(t_j)$
9: **end for**
10: **for** $c_i \in \mathcal{C}_i^N$ **do**
11: **if** c_i contains `this.fld` **then**
12: replace *this* in c_i by the interface variable b
13: **end if**
14: **end for**
15: $\mathcal{F}_i^N = \text{CUBICSOLVER}(\mathcal{C}_i^N)$
16: $\mathcal{F} = \mathcal{F} \cup \mathcal{F}_i^N$
17: **end for**
18: UNIFYSUMMARY$(\mathcal{F}, B_{m_h})$

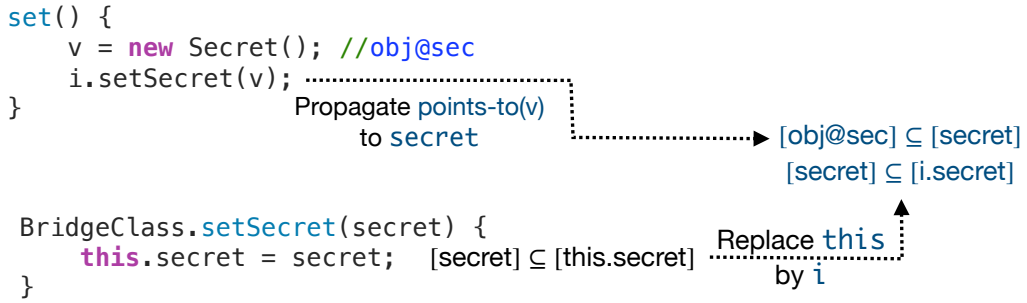


Figure 5.8: Function summary constraints for `setSecret`

variable (capturing the return value) from the pre-analysis, which already contains inter-procedural analysis. The collected constraints are then resolved to function summaries. Algorithm 8 illustrates the procedure to acquire function summary constraints. Given a bridge callgraph and the pre-analysis, it first generates the function summary constraints for the bridge callgraph's function nodes. Note that (except for the root) the function nodes in a bridge callgraph comprise the (bridged) functions of the host, which provide functionality to the guest. After obtaining the constraints for these functions, the next step is to map their formal parameters to the actual invocations' parameters in the guest. Fig. 5.8 illustrates the procedure from Line 5 to Line 14. These constraints are then sent into a cubic solver, which generates the actual function summaries.

Fig. 5.9 shows the constraints and function summaries generated for our example program. The second column shows the constraints and the third column the resolved summaries after the cubic solver. In the function foo, we have a single constraint

Function	Constraints	Function Summary
foo	$\{obj@iface\} \in [i]$ $\emptyset \subseteq [x]$	$[i] \mapsto \{obj@iface\}$ $[x] \mapsto \emptyset$
setSecret	$[secret] \subseteq [i.secret]$ $\{obj@sec\} \in [secret]$	$[i.secret] \mapsto \{obj@sec\}$
getSecret	$[i.secret] \subseteq [\$ret]$	$[i.secret] \mapsto \emptyset$ $[\$ret] \mapsto \emptyset$

Figure 5.9: Summary constraints and resolved summaries

Algorithm 9 UnifySummary

Require: Function Summaries $\mathcal{F}$
Require: Bridge subgraph B
Ensure: Unified function summaries $\mathcal{F}$
1: **for all** $(x(.e)^* \mapsto \phi_1, y(.e)^* \mapsto \phi_2) \in \mathcal{F} \times \mathcal{F}$ **and** $aliases(x, y)$ **do**
2: $\mathcal{F}[x(.e)^*, y(.e)^* \mapsto \phi_1 \cup \phi_2]$
3: **end for**

$\{obj@iface\} \subseteq [i]$ which is resolved to $[i] \mapsto \{obj@iface\}$. In the function setValue, we have two constraints $\{obj@sec\} \in [secret]$ and $[secret] \subseteq [i.secret]$; they are resolved to $[i.secret] \mapsto \{obj@sec\}$. In the function getValue, we have a single constraint $[i.secret] \subseteq [\$ret]$ resolved to $[\$ret] \mapsto \emptyset$ and $[i.secret] \mapsto \emptyset$.

5.2.4.2 Unification

As evident in Fig. 5.9, in the setSecret function i.secret corresponds to the guest object *obj@sec*; however, in the getSecret function, i.secret remains unknown, despite the fact that the access paths are identical based on the same interface variable i. To address this, we combine these summaries according to the access path.

Algorithm 9 shows the algorithm to unify summaries. Two summaries with the same access path are combined by a union operation, i.e., if $ap \mapsto \phi_1$ and $ap \mapsto \phi_2$, then the unification results in $ap \mapsto \phi_1 \cup \phi_2$. The unification is also alias-aware, i.e., aliases of the base variable in ap are also unified. For simplicity, we assume that the access paths are not nested. This is a fair assumption since most dataflow analyses are performed on an intermediate language, which simplifies the language to such a syntax for analysis. Fig. 5.10 shows the result of unification for the bridge callgraph shown in Fig. 5.6. This step unifies the summaries based on the access paths and aliased access paths. First, the analysis resolves the data flow across bridge functions by comparing the access paths, and then it updates the corresponding points-to sets. Therefore, $i.secret$ points to the guest object *obj@sec* across all functions participating in the bridge callgraph.

5.2.4.3 Summary Injection

After summary unification, the analysis has resolved the data flows across the bridge callgraph. The analysis, in particular, resolved the operations in bridge methods, i.e., host language functions called from the guest interface. Two tasks remain: First, translating the analysis summaries to be compatible with the analysis models adopted by

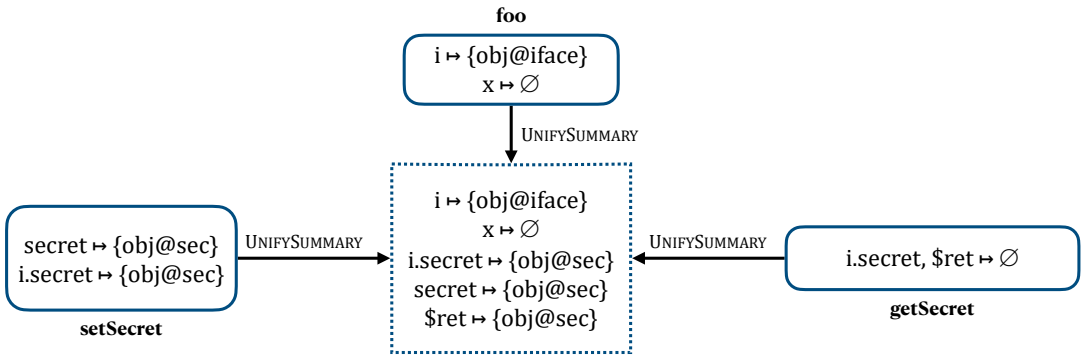


Figure 5.10: Unification for the bridge callgraph in Fig. 5.6

the host and guest languages. Second, propagating the translated summaries to the invoking functions at the guest side.

The summaries obtained via unification are in the access path format; however, we need to translate them to the host and guest languages' analysis models to reuse them in the host and guest. Notably, we translate the access path format to their native format. The two levels of sensitivities involved in the underlying analysis, calling context and field sensitivity, are of particular relevance. Since the bridge functions are invoked through interface variables in the guest applications, the context of the host method is assigned in the bridge callgraph. Following that, we classify field sensitivity according to how the underlying analysis handles field operations; as field-sensitive, field-based, or field-insensitive. The analysis determines the appropriate field sensitivity representation depending on the type of underlying analysis.

Considering the running example, the analysis resolves that the getSecret function returns `obj@sec`. The return value is captured by the variable g in getAndLeak; however, at this stage g has not been resolved and still points to $\emptyset$. Consequently, the analysis misses that g exposes a sensitive value at Line 21. Summary injection performs this step by retrofitting the pre-analysis with the unified summary. First, the unified summary information is translated to the host language's analysis model. The pre-analysis updates cause an incremental update of the variables impacted by the injected dataflows. Next, these updates (translated summaries) are propagated to the affected guest functions.

To show the soundness of SpecializeSummary, we use the notion of safe-approximation of pointer analysis from Andersen's Ph.D. thesis [9] defined in 1. Intuitively, a safe approximation of a pointer analysis is if it computes a set of all objects that a variable may refer to in runtime. Therefore, an analysis is sound if it produces a safe approximation. Further, our analysis assumes that the input analyses are also safe approximations.

Definition 11. *(Safe Approximation). Let C be the set of objects pointed by an access-path $v.f$ in all possible concrete executions of the program. An analysis computes the safe approximation if* $\mathsf{Analysis}(v.f) \supseteq C$.

Lemma 1. *(Safe approximation of* UnifySummary*). Assuming $x(.e)* \mapsto O_1$ and $y(.e)* \mapsto O_2$ are safe approximations, UnifySummary terminates and on termination also computes the safe approximation of $x(.e)*, y(.e)*$.*

Proof. Observe that lines 1 and 3 in Algorithm 9 join the function summaries for each

Algorithm 10 InjectSummary

Require: Projection $\mathcal{R}^f$
Require: pre-analysis host $\mathcal{P}_h$, guest pre-analysis $\mathcal{P}_g$
Require: Function Summary $\mathcal{F}^f$ of a function f
Ensure: updated pre-Analysis $\mathcal{P}_{modular}$

$\mathcal{P} \leftarrow \mathcal{P}_g \cup \mathcal{P}_h$
$\mathcal{R}^f_\Delta \leftarrow \emptyset$ ▷ Updated Projection
for all $(accesspath, heap) \in \mathcal{F}^f$ **do**
  if *accesspath* is a simple variable *var* **then**
    $\mathcal{R}^f_\Delta = \mathcal{R}^f_\Delta \cup \{(var, heap, ctxt)\}$ for any $ctxt \in \mathcal{R}^f$
  end if
  if *accesspath* is a field access x.f **then**
    FieldSensitive analysis: $\mathcal{R}^f_\Delta = \mathcal{R}^f_\Delta \cup \{(o_i, f, heap) \mid \forall o_i \in \mathcal{P}(x)\}$
    FieldInsensitive analysis $\mathcal{R}^f_\Delta = \mathcal{R}^f_\Delta \cup \{(o_i, heap) \mid \forall o_i \in \mathcal{P}(x)\}$
  end if
end for ▷ Update the guest projection to include resolved host projection
for $f' \in G_{guest}$ invoking f, which is a bridge method **do**
  ▷ Propagate host function's (f) return variable to guest (f'), adapting variable name.
  $\mathcal{R}^{f'} = \mathcal{R}^{f'} \cup \mathcal{R}^f_\Delta[\$ret]$

end for
$\mathcal{P}_{modular} = \mathcal{P} \cup \mathcal{R}^{f'}$

pair of aliased variables. Therefore, it produces $x(.e)*, y(.e)* \mapsto O_1 \cup O_2$, which is also a safe approximation. Termination is straightforward as the algorithm runs over a finite pair of aliased variables. □

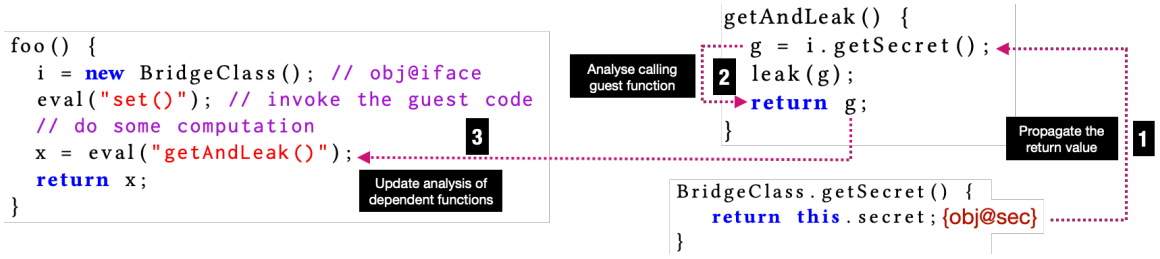
Theorem 2. *(Termination and correctness). The procedure* SpecializeSummary *terminates and computes a safe approximation F_s for a function f.*

Proof. Observe that the constraints compute Andersen's style inclusion-based pointer analysis. Objects are represented by the set of abstract heap objects taken from the input analyses. Rule New and Func include the objects from the input analyses. The remaining constraints are standard Andersen's analysis constraints, which propagate the solutions from the constraint variables on *lhs* to that on the *rhs*. Considering Lemma 1, UnifySummary computes the safe approximation over aliased variables pairs.

Regarding termination, the New and Func initialize a finite number of heap objects. The remaining rules propagate the corresponding finite points-to sets until it saturates (CubicSolver). From Lemma 1, UnifySummary runs over the finite set of summaries computes. □

Algorithm 10 shows the steps for summary injection. For a given function f, it takes the analysis snapshot of f as an input. We denote this snapshot as a *projection* of a function. Formally, we define the *projection* as:

Definition 12 (Projection)**.** *A projection $\mathcal{R}^f = \bigcup_{v \in def(f)} \{(v, h, ctxt) \mid h = pointsTo(v, ctxt)\}$ for a function f is a set of points-to sets for the variables defined in the function f under a context ctxt.*



(a) Summary Injection in methods

Method	Before Summary Injection	After Summary Injection
foo	$\langle i, ctx\rangle \mapsto \{\texttt{obj@iface}\}$ $\langle x, ctx\rangle \mapsto \emptyset$	$\langle i, ctx\rangle \mapsto \{\texttt{obj@iface}\}$ $\langle x, ctx\rangle \mapsto \{\texttt{obj@sec}\}$
setSecret	$\langle i.secret, ctx'\rangle \mapsto \{\texttt{obj@sec}\}$	$\langle i, secret, ctx\rangle \mapsto \{\texttt{obj@sec}\}$
getSecret	$\langle i.secret, ctx'\rangle \mapsto \{\texttt{obj@sec}\}$ $\langle ret, ctx'\rangle \mapsto \{\texttt{obj@sec}\}$	$\langle i, secret, ctx\rangle \mapsto \{\texttt{obj@sec}\}$ $\langle ret, ctx\rangle \mapsto \{\texttt{obj@sec}\}$
set	$\langle v, ctx'\rangle \mapsto \{\texttt{obj@sec}\}$	$\langle i.secret, ctx'\rangle \mapsto \{\texttt{obj@sec}\}$
getAndLeak	$\emptyset$	$\langle g, ctx'\rangle \mapsto \{\texttt{obj@iface}\}$

(b) Analysis before and after summary injection

Figure 5.11: Summary Injection for Listing 5.1 and Listing 5.2

For the function f in the bridge graph, our analysis gets the function's context and produces a projection of the analysis. Intuitively, projection is the points-to sets of the variables defined in the function for a context *ctxt*. Algorithm 10 uses the projection of the analysis to inject the dataflow facts obtained from unification. Along with the projection, the algorithm accepts the function summary of a function f in access-path format, and information on how the function handles field operations — either FIELDSENSITIVE or FIELDBASED or FIELDINSENSITIVE. Earlier research [109, 30, 107, 40, 18, 77] shows that these are the primary techniques for handling field operations, and hence, we confine our algorithm to those. Line 3 to 11 translate the obtained summary information in access-path representation to the analysis-supported representation. In Line 5, the translated points-to-set for the simple variables are added to the projection. Line 8 incorporates the translated points-to-set for field-sensitive analysis and Line 9 for the field-insensitive variables. In the following step, the modified host projection (of the bridge method) is propagated to the calling guest method (connected via the bridge method edge in bridge graph). Finally, the pre-analyses are revised and translated projections are included. With these revisions, the analysis has successfully resolved all key operations for (one) bridge callgraph's inter-language operation.

Fig. 5.11 summarizes the output of the summary injection to the pre-analysis. The second and third columns depict a snapshot of the analysis prior to and following the summary injection. The return value of the getSecret method has been updated to {obj@sec} as a result of the summary unification. Since *getAndLeak* invokes the function *getSecret*, the summary injection automatically propagates the updated return value to *getAndLeak*, and the points-to set of g now contains {obj@sec}. The bridge callgraph includes the connection from *getAndLeak* to *foo*, and consequently, the {obj@sec} is propagated to the variable x.

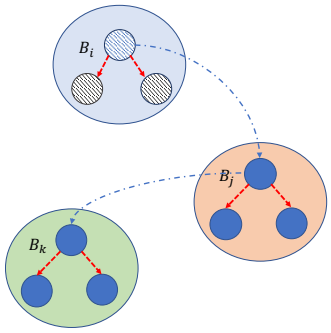


Figure 5.12: Multiple Bridge callgraphs

Lemma 2. *Line 3 to 11 (Algorithm 10) preserves monotonicity. For a function f, let $\mathcal{R}^f$ and $\widehat{\mathcal{R}}^f$ be its projections before and after summary injection. Then, $\mathcal{R}^f \subseteq \widehat{\mathcal{R}}^f$.*

Proof. Given the safe approximation of the function f as F_s. Without loss of generality, let F_s is a set of summary relations, say, $\{(a,h) \mid (a,h) \in F_s\}$. At each step in the loop between Line 3 and Line 11 transforms a summary relation (a_1, h_1) to the corresponding field-sensitive or field-insensitive analysis and adds it to $\mathcal{R}^f$. Therefore, $\widehat{\mathcal{R}}^f = \mathcal{R}^f \cup \bigcup_{i=0}^{n} (a_i, h_i)$ which is monotonic. □

Theorem 3. InjectSummary *preserves monotonicity and terminates.*

Proof. Line 12-15 (Algorithm 10) propagates the return value in the bridge subgraph to the method in the guest callgraph and adds it to $\mathcal{R}^f$. Finally, at line 16, it adds the $\mathcal{R}^f$ to the unified input analyses $\mathcal{P}$ and gives $\mathcal{P}_{modular} = \mathcal{R}^f \cup \mathcal{P}$. Therefore, it preserves monotonicity. □

5.2.5 Multiple Bridge Calls

As is, bridge calls originating from different host functions will produce multiple bridge subgraphs, thus (potentially) generating a list of disconnected bridge subgraphs as shown Fig. 5.12. Assume bridge subgraph B_j comes after B_i, B_k after B_j in the topological ordering of the callgraph G. If the analysis resolves the data flows for the bridge subgraph B_i, then InjectSummary will propagate the changes down the call-graph. In the next iteration of the algorithm, say for B_j, the analysis needs to start with the updated information incorporating the changes introduced by B_i. We use a topological ordering of the bridge subgraphs and a worklist initialized with the topological order to process the list of bridge subgraphs.

5.2.6 Limitations

Figure 5.1 only allows programs as parameters to eval, so no dynamically computed target. In practice this is not a restriction in contemporary programs but could be lifted with additional analysis [42, 11]. Our approach uses may aliases of the interface object on the guest side. Thus, our analysis may yield imprecise results when a bridge method has the same name and arity as the guest method. However, this is a general limitation of static analysis; besides, we have not seen such occurrences in our dataset. The language assumes that *bridge interfaces* are specified and their type is known to the analysis to identify *InterfaceVars*. This is a fair assumption as modern languages

specify the syntax and semantics of these markers, e.g., Android defines WebView APIs for bridge interface. Considering this, our analysis assumes that methods invoked via interface objects are statically resolved, i.e., dynamic binding is not applicable to the bridge class's methods invoked via interface objects; our analysis obtains statically resolved interface objects from the pre-analysis. Further, the language assumes that the objects are bound to the heap, i.e., it does not consider the pointer reference and dereference operations.

5.3 Discussion

Figure 5.1 only allows programs as parameters to eval, so no dynamically computed target. In practice this is not a restriction in contemporary programs but could be lifted with additional analysis [42, 11]. Our approach uses may aliases of the interface object on the guest side. Thus, our analysis may yield imprecise results when a bridge method has the same name and arity as the guest method. However, this is a general limitation of static analysis; besides, we have not seen such occurrences in our dataset. The language assumes that *bridge interfaces* are specified and their type is known to the analysis to identify *InterfaceVars*. This is a fair assumption as modern languages specify the syntax and semantics of these markers, e.g., Android defines WebView APIs for bridge interface. Considering this, our analysis assumes that methods invoked via interface objects are statically resolved, i.e., dynamic binding is not applicable to the bridge class's methods invoked via interface objects; our analysis obtains statically resolved interface objects from the pre-analysis. Further, the language assumes that the objects are bound to the heap, i.e., it does not consider the pointer reference and dereference operations.

5.4 Implementation

We developed a prototype implementation of our approach called Conflux. Conflux currently supports analysis of Java-C programs and Java-Python programs, and with minor modifications, it can generalize to other programming language combinations. Conflux is implemented in Java, consists of about 5k lines of code, and builds on top of the WALA analysis framework [21] for analyzing Java and Python. For analyzing C/C++ programs, we choose SVF [112, 113], a popular tool for C/C++ dataflow analysis. Conflux can be instantiated to run with other analyses with some engineering efforts.

For analyzing Android NDK programs, Conflux takes the android apk file as input. It uses the *retdec* decompiler [14] to lift the NDK libraries bundled in the apk file to LLVM. For analyzing Graal Polyglot APIs, it takes the Java jar files and Python sources as input.

5.5 Evaluation

We empirically evaluate *Conflux*'s efficacy in resolving the inter-language operations in multilingual applications. To demonstrate the generalizability of *Conflux* to other interoperable languages, we evaluate *Conflux* on two multilingual programming models: (1) Java-C communication in Android NDK apps and (2) Java-Python applications using GraalVM. Java-C communication is manifested in Java (or Android) applications that use system-level features for various reasons, such as performance or interfacing with

the hardware. Java-Python apps in our dataset use the GraalVM Polyglot API. Our evaluation is designed to address the following research questions:

1. How does *Conflux* compare to the state-of-the-art multilingual analyses in terms of precision and scalability? (Section 5.5.1)
2. Can *Conflux* generalize to other multilingual programming models? (Section 5.5.2)

All the experiments were performed on a laptop running macOS 13 with a 10-Core M1 Pro processor and 32 GB RAM.

5.5.1 Comparison to the state-of-the-art

Recent years have seen a few research tools [66, 100, 79, 123, 20] to address the analysis of multilingual apps. C-summary [66] and Jucify [100] are two recent multilingual analysis tools, and they have been shown to overcome limitations of previous works. Thus, we compare *Conflux* with C-summary [64] and Jucify [26].

Subject apps. To evaluate the first research question, our criteria for selecting the subject apps is dictated by two aims: (1) include apps from the C-summary and Jucify dataset to avoid selection bias, and (2) evaluate *Conflux* on various Java and C communication patterns. Considering these choices, we compiled 23 Android JNI apps from the widely used NativeFlowBench [12, 123] and 16 popular apps from the Google play store. NativeFlowBench is an Android app benchmark that includes various inter-language communication features. Each app in this benchmark has a ground truth.

Next, we assess all tools on the following parameters: (1) whether it can discover the callgraph edges from the host-to-guest language and vice-versa, and (2) whether it can discover the reachable methods in the guest language to generate a whole-program call-graph. Table 5.1 and 5.2 contains the subject apps and shows the evaluation results of *Conflux* , C-Summary, and Jucify on them. The first column shows the subject apps.

Our Experience with C-summary The evaluation uses a few symbols to illustrate the results. The second column contains the results obtained from C-summary. The symbol □ indicates that the analysis was able to detect the function signature but failed to synthesize the function definition. In our experiments, in 11 out of 19 apps, C-summary failed to synthesize a function definition. This happens when a program has complex pointer operations (such as pointer arithmetics) or method overloading; for example, programs use string library operations such as `strcpy`, which do not have the source code. The symbol ⊤ indicates the tool could identify the function signature but generated an empty program such as `Java.Top()`, which essentially returns the set of all heap allocations (imprecise analysis result). In six out of 19 apps, C-summary generated an empty program. This happens when object field load and store methods are accessed in native code. For example, in the benchmark app *complexdata_ stringop*, C-summary generates an empty program for the Java methods `GetFieldID` and `GetObjectClass`. The symbol ✗ indicates that a tool could neither identify the function signature nor synthesize the program. In six out of 19 apps, C-summary showed this behavior. This happens in two cases: (1) intent operations over native classes and (2) operations not handled by C-summary such as arrays or global variables, which are reported limitations of their framework [66]. Thus, C-summary could not synthesize the required program for

Table 5.1: Analysis of JNI from NativeFlowBench microbenchmark

Benchmark	**C-Summary Generated**	**JuCify**						**Conflux**						
		Analyze	N_{bj}	E_{bj}	N_{aj}	E_{aj}	IL	**Analyze**	N_J	E_J	N_C	E_C	IL	Time (in sec)
NativeFlowBench														
complexdata	☐	↝	26	26	9	7	2	✓	364	940	8	10	2	< 1
complexdata stringop	✗	↝	26	26	9	7	2	✓	5802	13015	9	10	1	< 1
dynamic register multiple	✗	↝	18	18	0	0	2	✓	382	979	9	13	3	< 1
heap modify	☐	↝	25	25	7	7	1	✓	6889	15272	11	12	1	< 1
icc javatonative	✗	↝	17	17	2	2	0	✗	366	895	13	14	0	< 1
icc nativetojava	☐	↝	27	27	5	5	0	✓	7865	14891	8	9	1	< 1
leak	☐	↝	18	18	4	3	1	✓	367	894	3	4	1	< 1
leak array	☐	↝	18	18	5	4	1	✓	383	968	4	5	1	< 1
leak dynamic register	☐	↝	18	18	4	3	1	✓	367	895	9	11	1	< 1
method overloading	☐	↝	18	18	3	2	1	✓	307	751	3	5	2	< 1
multiple interactions	☐	↝	18	18	3	2	1	✓	384	980	11	13	2	< 1
multiple libraries	☐	↝	18	18	5	4	1	✓	382	978	3	5	1	< 1
noleak	☐	↝	18	18	3	2	1	✓	367	894	1	2	1	< 1
noleak array	☐	↝	18	18	5	4	1	✓	5795	13008	4	5	1	< 1
nosource	⊤	↝	13	13	1	0	0	✓	306	746	1	2	1	< 1
pure	✗	✗	No call graph was generated					✗	6581	14686	39	42	0	< 1
pure direct	✗	✗	No call graph was generated					✗	6581	14686	30	48	0	< 1
pure direct customized	✗	✗	No call graph was generated					✗	380	958	30	48	0	< 1
set field from arg	⊤	↝	13	13	12	11	1	✓	373	908	4	5	1	< 1
set field from arg field	⊤	↝	31	31	12	11	1	✓	314	763	4	5	1	< 1
set field from native	⊤	✗	25	25	7	7	0	✓	310	762	14	15	1	< 1
source	⊤	✗	27	27	9	9	0	✓	6892	15271	7	8	1	< 1
source clean	⊤	✗	27	27	9	9	0	✓	385	985	4	5	1	< 1

Table 5.2: Analysis results on large-scale apps.

Benchmark	C-Summary	JuCify	Conflux						
	Analyze	IL Detected	Analyze	N_J	E_J	N_C	E_C	IL	Time (in sec)
Airmessage	-	✗	✓	53987	30886	169	141	4	30
Apple Music	-	✗	✓	51867	28760	8334	3055	8	72
Arte	-	✗	✗	57224	31237	74	31	0	480
ClearScore	-	✗	✗	50834	31063	344	321	0	30
CommonsLab	-	✗	✓	28817	16576	447	348	14	15
FPLayAndroid	⊤	✗	✓	5052	2316	294	238	14	240
FairEMail	⊤	✗	✓	59766	40214	3854	1116	5	85
JioMoney	-	✓	✗	50002	22904	299	232	2	27
NetGuard	-	✓	✗	12159	6448	333	168	2	5
PCAPdroid	⊤	✗	✓	54270	28835	613	303	3	29
Proton VPN	⊤	✗	✓	56217	31253	56217	8	1	54
Spotify	-	✓	✗	59690	33736	439	177	0	600
Spotify Lite	-	✗	✓	51728	31234	843	434	8	390
TermOne	⊤	✗	✗	47352	24365	48	30	0	22
Termux	⊤	✗	✓	15656	9603	77	36	6	6
TimidityAE	⊤	✗	✓	15278	8275	187	127	6	1

further analysis by Android analysis tools, thus hiding the functions reachable through the bridge interfaces. In many cases it could only identify the function definitions used to find reachable native functions from the Java (Android) side.

Our Experience with JuCify The third column contains the results obtained from Jucify. $\precapprox$ denotes that Jucify's call graph underapproximates the actual call graph. N_{bj}, E_{bj} shows the number of nodes and edges in the call graph before JuCify. N_{aj}, and E_{aj} show the change in the number of nodes and edges (except the interlanguage edges) in the call graph after JuCify. In our evaluation, JuCify does not report the inter-language edges properly; thus, we manually went through the call graph logs to identify interlanguage edges reported in IL (for microbenchmark). Due to the sheer volume of the logs and lack of truth value in large-scale apps, we did not manually analyze them. JuCify partially detects the interlanguage edges in 15 out of 23 apps. In three apps, it fails to generate the call graph for the whole program owing to the limitations of the underlying Soot framework. During the evaluation, we observe that Jucify cannot correctly compute the call graph for the native component due to the limitations of symbolic execution of the binary, especially for those operations arising out of the pointers (such as pointers to `JNIEnv`). Unfortunately, these operations are ubiquitous owing to the interfaces of JNI communication, which have been addressed by *Conflux*.

***Conflux*'s Results** *Conflux* uses SVF (for C/C++) and Wala (for Java) as the pre-analyses. We configured SVF to run with the entrypoints defined in the host language analysis. Based on the entrypoints, it triggers SVF to analyze the guest functions in the environment and collects the results. Consequently, SVF identifies the reachable functions in the guest language and *Conflux* combines the analysis result with those obtained from Wala to build an inter-language call graph. The columns N_J and E_J show the number of call graph nodes and edges identified by the respective pre-analyses. The numbers are significantly high considering the size of the benchmarks because Wala includes a number of libraries by default [91], which increases the call graph size. The column IL reports the interlanguage edges in the call graph, i.e., the edges

which go from Java to C and vice-versa. Column E_C shows the edges identified by SVF. *Conflux* identifies the inter-language edges in 19 out of 23 benchmarks, i.e., it identifies the methods which would have been previously reported unreachable with Wala. In the remaining four cases, *Conflux* does not identify the entry point where native code is invoked via intents, which is expected as middleware-based inter-language communication is out of the scope of this study. Intent resolution tools [68, 124, 117] could supplement *Conflux* to resolve these entrypoints.

In 19 out of 23 subjects, *Conflux* builds the complete inter-language call graph. *Conflux* leverages the strength of individual analyses and maps required call edges from host-to-guest and vice-versa.

Scalability To evaluate the efficacy of *Conflux* on real-world apps, we downloaded 50 popular and recent apps from the Google play store. To filter out NDK apps, we vetted their decompiled code for the presence of shared object files, resulting in 16 apps. These apps include popular apps such as Spotify, and Apple Music. We share a few apps in our dataset with those evaluated by C-summary.

The first part of the bottom half of Table 5.1 shows the results obtained on these apps from C-Summary. Since C-summary is designed to run on source code, we ran it on seven apps that are open-source. In all of these apps, C-summary synthesized imprecise programs (statement containing `Java.Top()`).

The second part of Table 5.1 shows the results of our analysis on these apps. Our analysis scales for many popular apps and analyzes relatively large apps such as *Spotify* within 10 minutes (median around 2 minutes). Among these, *Conflux* was able to identify inter-language call graph edges in 12 out of 16 apps. Again, in the other four apps, guest functions were called through intents, which are out of scope. To validate the number of inter-language edges, we scanned the decompiled files for the presence of native markers in the binary. Two authors independently validated these results.

In 12 out of 16 large apps, *Conflux* successfully builds the interlanguage call graph in about 2 minutes on average. It demonstrates that combining the individual analyses via *Conflux* is a promising direction for a scalable dataflow analysis in multilingual applications.

5.5.2 RQ2: Analysis of Java-Python programs with GraalVM

To evaluate the generalizability of our approach, we applied Conflux to Java-Python-based GraalVM programs. GraalVM's Polyglot API is relatively new, and we could not find many projects using Java-Python-based communication on GitHub. To curate the apps, we selected projects with the keyword graalpython and those importing the class (or subclasses) of org.graalvm.polyglot.Context. Out of the seven results we obtained on Github, we searched for the presence of Python code snippets and files. Finally, we confirmed two applications with Java as the host language and Python as the guest (Table 5.3). We also develop an additional microbenchmark for analysis, in total, having three micro-benchmarks for study. In BM-1 [2], Java setter and getter methods are invoked from Python. BM-2 [3] invokes Java library methods from Python. In BM-3 [4], Java methods are called from Python (as host).

Table 5.3: Java-Python apps in GraalVM .

Benchmark	Wala-Java	Wala-Python	Conflux
BM-1	✓	✓	✓
BM-2	✓	✓	✓
BM-3	✗	✓	✗

Conflux leverages the Wala framework as the pre-analysis, configured to one-callsite-sensitive and field-sensitive for both languages, to obtain the analysis of Java and Python programs. Conflux successfully analyzed BM-1 and BM-2, and the inter-language function calls are correctly resolved. BM3 uses Java as the guest language and Python as the host. Therefore, Wala-Java failed to analyze the program as it requires a main method, whereas the guest language is used as a library.

RQ2: Conflux can be configured to multiple polyglot frameworks. To exemplify, Conflux is configured to analyze GraalVM programs, and successfully resolves the bridge interfaces in two of three Java-Python apps.

5.6 Remarks

In this chapter, we leveraged existing language analyses for analyzing multilingual programs. We proposed a series of algorithms for the same and demonstrated our approach's efficacy with a prototype implementation called Conflux. Our framework can be employed on combinations of host and guest analyses to take advantage of the language-specific models. Our analysis neither requires us to remodel the analysis for multilingual applications nor translate the existing semantics of the guest language to the host languages, consequently saving a lot of effort in this process. We evaluated Conflux on multiple apps from two multilingual language models and demonstrate its efficacy and generalizability.

Part IV

Outlook

Chapter 6

Related Work

6.1 Analysis of Hybrid Apps

Numerous analyses have been proposed for the analysis of Android Hybrid Apps in the recent past. The analyses range from detecting web-specific vulnerabilities to the cross-language analysis of the Android WebView. This section walks through all of these works while showing how our empirical study on WebView and information flow analysis on WebView compares against these works.

BabelView by Rizzo et al. [97] models JavaScript as a black box. They leverage static taint analysis to detect unwanted information flows, where the WebView class is instrumented by the *BabelView*, which models the attacker. This instrumented app is then analyzed with a taint analysis, such as FlowDroid [13], to look for possible vulnerabilities. However, they do not analyze the JavaScript code. In comparison, our analysis *IwanDroid* analyzes both Java and JavaScript components, resulting in fine-grained analyses instead of using JavaScript as a black box.

Zhang et al. [130] performed a large-scale study of the WebView APIs to classify them into four categories of web resource manipulation. Mutchler et al. [80] conducted a large-scale study of apps using WebView aiming at the security vulnerabilities present in these apps. This study focuses only on particular types of vulnerabilities, and they did not consider the misuse of JavaScript in `loadURL`. In comparison, our empirical study with *LuDroid* performs a large study of the patterns of use of bridge interfaces and JavaScript patterns by developers.

There is a large body of work on detecting web-specific vulnerabilities. Hidhaya et al. [51] described the "supplementary event-listener injection attack" in Android WebViews. Essentially, such attacks add additional event listeners to the WebView. They further proposed a tool for automated detection of this vulnerability and its mitigation. Li et al. [71] discovered a new type of WebView-based attack that they call Cross-App WebView Infection (XAWI), where malicious content in WebView of one app redirects it to another activity, which subsequently launches a malicious page. Mandal et al. [75] proposed a static analysis tool to detect various vulnerabilities in Android Infotainment applications. To this end, they leverage abstract interpretation to track infotainment apps' specific bugs, such as external files' world-writable permission on Android apps, among others. Like *BabelView*, they view the usage of WebView as malicious, which is

not a precise behavior to track. Fratantonio et al. [33] proposed a static analysis tool to detect malicious application logic in Android apps. Their approach is based on various known static analysis techniques, such as symbolic execution and inter-procedural control-dependency analysis. In comparison to these works, our empirical study on WebView again does a comprehensive study of vulnerabilities in WebView. The information flow analysis focuses on detecting information flow bugs with a novel threat model, which has not been the case in prior works. Brucker and Herzberg [19] proposed a static analysis to detect security vulnerabilities in Cordova (Java-Javascript) based applications. This work is close to *IWandDroid* but only confined to Cordova applications.

Neugschwandtner et al. [82] proposed two attack scenarios based on when the client or server is compromised. This work is orthogonal to compromised attack scenarios. Rather, it is related to providing analysis tools for developers.

Yang et al. [128] examined so-called "Origin Stripping Vulnerabilities" caused by wrongly using the *loadURL* method. Kim et al. [59] leveraged abstract interpretation to design a static analysis that finds privacy leaks in Android applications. Targeting excess authorization and file-based cross-site scripting attacks, Chin et al. [22] proposed Bifocals, a tool to detect these vulnerabilities. However, these analyses focused on one particular part of the problem. Our study aims to analyze all programming patterns that may potentially lead to vulnerabilities.

Generally, the analysis of unencrypted communication in Android apps is a well-explored topic [90, 50, 29]. For example, Pokharel et al. [90] demonstrated eavesdropping attacks on VoIP apps. However, to the best of our knowledge, no previous work has analyzed the security consequences of unencrypted communication caused by *loadURL*, as done in *LuDroid*. In contrast, our large-scale study with *LuDroid* (developed by us) studies the usage of JavaScript in hybrid apps. In this work, we discovered the intricate programming patterns used by developers through a large-scale study. In contrast to all these, *IwanDroid* proposes a confidentiality and integrity violations-based threat model, which is a fine-grained notion of information leaks.

Lee et al. [63] proposed HybriDroid, a static analysis for hybrid apps based on the WALA framework. They discussed the semantics of WebView communication, including type conversion semantics between Java and JavaScript. In addition, HybridDroid performed taint analysis, although it does not consider the peculiar semantics of WebView. In a later work, Bae [15] formalized an incomplete semantics of the android interoperations between Java and JavaScript geared toward detection for `MethodNotFound`, again with limited handling of information flow. In addition, HybriDroid does not provide a comprehensive analysis of hybridization APIs. Compared to this thesis, we present the full picture of non-trivial data and control flows that occur in Android WebHybridization. *IwanDroid* performs an information flow analysis of Hybrid apps based on the additional observations obtained from *LuDroid*.

6.2 Static Analysis of Multilingual Applications

Multilingual analysis. Lee *et al.* [66] proposed a modular semantic summary extraction technique for analyzing Java-C/C++ interoperations. Monët *et al.* [79] used abstract interpretation for analyzing Python and C programs. JN-SAF [123] by Wei *et al.*

used a function summary-based bottom-up dataflow analysis for analyzing Java Native applications. All these works rely on explicit boundaries between the host and guest. Also, these works are tied to particular host and guest languages, while our work is language-agnostic. Buro *et al.* proposed an abstract interpretation-based framework [20] for analyzing multilingual programs by combining analysis of while and expression language. Their work proposes a framework based on abstract interpretation to combine a while and expression language. In comparison, *Conflux* provides a framework for the pointer and call-graph analyses by combining two unilingual analyses for imperative programming language.

Recently, a couple of works have also appeared on combining the analysis. These works are in the spirit of *Conflux*. Youn *et al.* (2023) proposed a declarative multilingual analysis [129] embedded in the CodeQL framework to fuse the data flows at the bridge interfaces. This work is tied to the CodeQL framework while in this thesis, we have proposed a general framework for combining analyses of imperative/OO languages. On another note, Negrini *et al.* (2023) [81] proposed a generic framework, *Lisa*, for multilingual static analysis based on a common intermediate representation. They provide the necessary infrastructure to write program analyses for various languages, where the developers can customize the analysis for each language and combine those for a multilingual analysis. *Lisa* performs a complete dataflow analysis and could benefit from the call graphs generated by Conflux for an interprocedural analysis.

A number of works have been proposed on analyses of specific multilingual platforms. JET [70], Li and Tan [103], Kondoh and Onodera [61] analyze bugs occurring at Java JNI interfaces. Fourtounis *et al.* [32] scanned native binaries to extract the dataflow facts used for the analysis in Java. Compared to them, our approach is generic and not restricted to the JNI calls. One of the closest works to *Conflux* which we have compared in the thesis, is *Jucify* by Samhi et al.. They mapped the native code to Soot IR for a unified analysis, which again does not benefit from the heuristics to analyze each language. Turcotte *et al.* [120] implements a type checker for the correctness of types for Lua-C programs. In comparison, Conflux implements a pointer and call-graph analysis.

Modular Analysis of monolingual applications It is orthogonal to our work, yet these form the backbone of our analysis as input analysis. JAM [84] by Nielsen *et al.* proposed a modular call-graph construction for analysis of *Node.js* applications. Tiwari *et al.* [117] used persistent summaries for resolving inter-component communication in Android. ModAnalyzer by Schubert *et al.* [101] leverages the pattern used by C++ developers to perform a modular analysis of C++ applications and libraries. Dilling *et al.* [25] and Whaley and Rinard [126] proposed compositional pointer analyses for C and Java, respectively. Illous *et al.* [53] proposed a separation logic-based technique for compositional shape analysis.

Modular analysis for monolingual applications also deserves a mention here, as these form the backbone of our work. These works are orthogonal, yet they can form the backbone of our analysis. Dilling *et al.* [25] proposed a compositional pointer analysis over function summaries. Dilling *et al.* [25] proposed a combined escape and points-to analysis for Java, which utilizes a novel program representation – points-to escape graph, which denotes the dataflow of the object allocated in one region to another. All of these analyses are based on a C-like abstract language.

Compared to these works, *Conflux* presents a technique for unifying existing pointer analyses. The approach allows us to unify the data flow across language boundaries by unifying the dataflow summaries at the bridge interfaces. Conflux can take advantage of the monolingual analysis and combine it with multilingual analysis with some engineering effort. Conflux is the first framework, to the best of our knowledge, to combine pointer analyses. This differentiates our approach from the other proposed static analyses of multilingual apps. C-Summary [66] and Jucify [100], which are the closest neighbors to Conflux, map the native C/C++ code to corresponding Soot/Wala IR. In Conflux, we deliberately avoid taking this path. Many static analyses rely on the heuristics particular to a language, and transforming it to a common IR loses these benefits.

Chapter 7

Conclusion

This chapter reflects on the goals introduced in the first chapter and briefly outlines the some of the future directions. The thesis started with the two goals in the 1.2 of Chapter 1. This chapter reflects on those goals (Section 7.1) and presents the future work that could emerge from the thesis (Section 7.2).

7.1 Reflections

Goal 1. How vulnerable is the communication between Android and JavaScript running on WebView?

In this thesis, we developed two tools to study and analyze the problem. The first tool, *LuDroid*, performs a lightweight static analysis to extract the JavaScript and analyze the bridge interfaces. *LuDroid* detected these code snippets, which were analyzed manually. With *LudDroid*, we empirically analyzed the available JavaScript code on benign apps and malware. We discovered the presence of remote libraries over unsecured protocols, cases of non-trivial data and control flows that are realizable, and obfuscation. However, it is unsurprising that these problems exist in multilingual applications, as many studies have reported them with single-language applications. Nonetheless, we can conclude from the study that the situation is similar with hybrid apps.

In the next step, we define an information flow analysis for hybrid apps to detect data leaks at the bridge interfaces. Inspired by the confidentiality-integrity model, we observed the similarities between the confidentiality-integrity model and information flow control in hybrid apps. The tool, *IWanDroid*, performs a demand-driven analysis, which is localized to the detected bridge objects and the corresponding JavaScript code. This analysis escapes the need for an explicit whole program analysis to analyze both Java and JavaScript, which is expensive in terms of time. Our evaluation shows that *IWanDroid* analyzes dataflow anomalies in 13 of 15 benchmark applications. In terms of large apps, it identified 136 integrity and 142 confidentiality violations.

To conclude this question, the interlanguage communication between Android Java and JavaScript is vulnerable. However, this thesis shows we can devise techniques to catch any information flow vulnerabilities.

Goal 2. Is it possible to combine various monolingual program analyses available for the languages for analyzing multilingual programs?

This problem was inspired by the observation that existing monolingual analyses have reached a level of sophistication where they can analyze the quirky semantics of each language. We hypothesized reusing these analyses for a multilingual analysis is a promising step. We devised the technique, *Conflux*, and instantiated it for two implementations, with Android NDK apps (Java and C/C++) and Graal VM polyglot API (Java and Python). Our evaluation demonstrated that combining analyses with *Conflux* framework is possible and practical. Nonetheless, engineering challenges remain while combining analyses, such as writing interfaces for each framework and interfacing with the intermediate representation.

7.2 Future Work

The rise in the multilingual programming paradigm will raise the need for advanced programming tools to aid developers in reliable and secure software development. In this direction, the adoption of static analysis techniques has grown in popularity. However, static analysis of multilingual software is in its infancy. Static analysis of multilingual programs is complex as it requires abstracting the semantics of interaction between the languages.

7.2.1 Static Analysis of Multilingual Applications

This dissertation proposed a framework, *Conflux*, for combining existing static analyses. It will be interesting to apply it to other multilingual systems. An example is WebAssembly binaries, similar to dynamically linked assembly libraries. Programmers can call methods defined in WebAssembly binary from JavaScript; in this case, WebAssembly as a guest language is embedded in JavaScript. Therefore, a combined call-graph analysis of JavaScript and WebAssembly could benefit from a framework like *Conflux*.

In addition, it will also be interesting to study the feasibility of combining different multilingual environments, in particular, *what* analyses we can combine and *how* to overcome the implementation details. Of special interest will be combining Datalog [57] based static analysis, or exploring other program representations, such as value-flow graphs [112], or points-to graph for combining static analyses.

Further, multilingual systems can combine more than two languages. As WebView can execute JavaScript, Android Java can invoke a JavaScript code, which in turn, invokes a WebAssembly library. Unsurprisingly, runtime systems support these operations [38]. Static analysis of such applications will be intriguing as it requires modeling the three languages, and combining function summaries could help in these analyses.

7.2.2 Information Flow Analysis of Multilingual Applications

Information flow analysis is essential in multilingual program analysis to prevent illicit data flow. Central to the ideas presented in this topic is the role of function summaries in unifying analyses and combining various localized analyses. The current state of

the research on static analysis primarily uses these function summaries to analyze a single language. In this thesis, we have explored it to model the interlanguage dataflow in Android WebView. These ideas could be applied to model the semantics of other interlanguage environments. These analyses will have to precisely model the runtime semantics of interlanguage communications that have not been explored yet, such as communication through Android Intents and other message-passing mechanisms.

7.2.3 Interlanguage Analysis of languages with different paradigms

In this thesis, we have analyzed programming languages that follow an imperative programming paradigm. However, foreign function interfaces are also present in functional programming languages, such as Haskell and OCaml. It will be interesting to see a combined analysis of functional and imperative languages. It will be challenging due to the different variants of context-sensitivity used in analysis of these languages [16, 76].

7.3 Concluding Remarks

This thesis studies the static analyses for multilingual programs where a significant challenge in analyzing multilingual programs is modelling the data flow between the languages. The methods presented in the thesis navigate this challenge in the form of function summaries. The underlying theme is to construct the function summaries of the data flow at the interfaces and propagate the resolved data flows to dependent methods. Towards the end, we give a glimpse of the work that could emerge from the insights obtained in the thesis. We hope that future research on multilingual program analysis will gain those insights.

Appendix A

Empirical Study on URLs in WebViews

LUDroid resolved 3075 distinct URLs. In addition it found 4980 URLs dynamically created using SDKs. Figure A.1 shows the distribution of protocols in the resolved URLs passed to the *loadURL* method. 40.81% of the URLs use the *file* protocol pointing to the device's (trusted) local files, while the remaining point to external (potentially trusted) hosts. Naturally, developers have more control over these offline local files. While this is good for trusted entities, malicious entities could easily launch phishing attacks by designing offline pages that look similar to trusted web pages. Only good user practices can prevent these attacks from happening: Ideally, APKs should not be downloaded from other sources than the official Play Store. Additionally, users should properly verify app metadata and permissions. As local web pages come bundled with the APK files, they can be taken into account during analysis. However, an analysis might need to consider several security aspects such as identifying phishing attacks, discovering privacy leaks, or finding keyloggers.

In addition to local file URLs, we discovered that in 41.24% of the resolved cases the URL argument was *"about:blank"*, which displays an empty page. According to Android's WebView [36] documentation *about:blank* should be used to "reliably reset the view state and release page resources". As an example, we discovered *about:blank* in the WebViewActivity class of the app *com.zipperlockscreenyellow*. In this class the

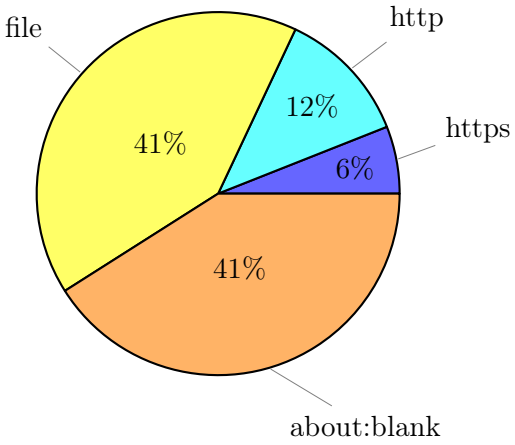


Figure A.1: Distribution of protocols (rounded values for clarity)

Listing A.1: WebViewActivity class in com.zipperlockscreenyellow (manually translated to Java and simplified)

```
1 webView.removeAllViews();
2 webView.clearHistory();
3 webView.clearCache();
4 webView.loadURL("about:blank");
```

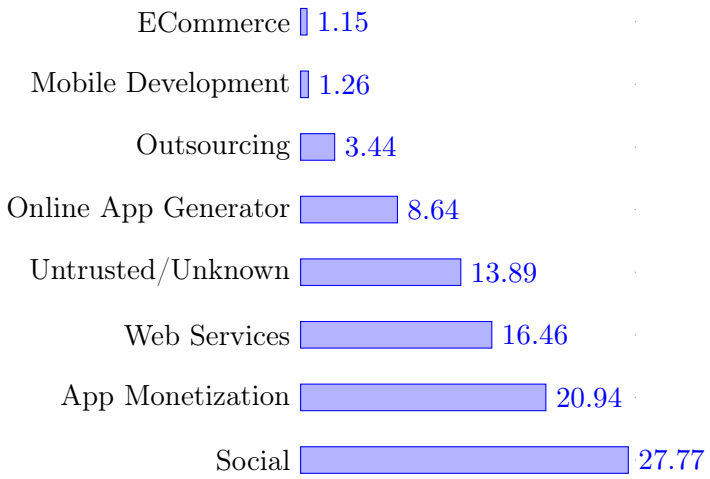


Figure A.2: Distribution of SDK usage in apps by categories (Top eight)

Table A.1: Selected list of Top-8 SDK hosts, its app share, and a common use case found in the top SDK categories

Category	**Host**	**Percent**	**Common Use Case**
Social Networking	Facebook	20.32	Authentication
App monetization	Vungle	1.75	Monetize Apps by targeted advertising
Web Services	Google	10.58	Authentication
Online App Generator	SeattleClouds	5.99	Unknown (obfuscated)
Outsourcing	biznessapps	1.89	Unknown (obfuscated)
Mobile Development	PhoneGap	1.52	Platform-independent development
E-Commerce	Amazon	1.1	Sales
Others	Ons	0.8	Rendering ebooks

method *killWebView* releases the view's resources (see Listing A.1). After clearing the history and the cache this method opens a blank page in the WebView.

Considering network URLs, the most-frequently loaded hosts per category in the analyzed apps are listed in Table A.1. We found that Facebook and Google SDKs are widely used in apps, primarily for authentication purposes. In addition app monetization and customer analytics SDKs are found in 18.04% of the apps (cf. Figure A.2). Figure A.2 displays all host categories sorted by their share. We found that a majority of the analyzed apps use social networking SDKs or app monetization SDKs.

Mobile application development frameworks such as Cordova or PhoneGap allow de-

Table A.2: Five out of 365 loadURL calls using the insecure HTTP protocol

Package Name	**URL**
com.JLWebSale20_11	`http://www.dhcomms.com/applications/dh/cps/google/main_agreepage01.html`
net.pinterac.leapersheep. main	`http://pinterac.net/dev/leapersheep/index.php?viewall=1`
com.quietgrowth.qgdroid	`http://docs.google.com/gview?embedded=true&url=http://www.rblbank.com/pdfs/CreditCard/fun-card-offer-terms.pdf`
net.lokanta.restoran. arsivtrkmutfagi	`http://images.yemeksepetim.com/App_Themes/static-pages/terms-of-use/mastercard/mobile.htm`
com.cosway.taiwan02	`http://ecosway.himobi.tw`

velopers to use HTML/CSS and JavaScript to develop mobile apps. These libraries primarily use bridge communication between native Android and web technologies [89]. In our study we found that 1.26% of the apps use these frameworks for mobile application development (referred in Figure A.2 as Mobile Development).

The 535 URLs that point to a network resource only reference 147 distinct paths. This indicates that in many cases identical host and path combinations were requested by multiple apps. In approximately 1.68% of the external URLs the host's port was specified. Additionally, 20.37% of the URLs (HTTP/HTTPS) specify an argument pattern.

While evaluating URLs we gained several relevant security insights. We found that 11.87% of the calls to *loadUrl* resulted in unencrypted network traffic, making a total number of 365 communications. Table A.2 shows five examples of unencrypted HTTP URLs together with the packages in the corresponding app. The usage of unencrypted protocols with *loadURL* may result in eavesdropping and phishing vulnerabilities. Another security threat is caused by untrusted SDKs using *loadURL*. We found a total of 13.89% apps use untrusted SDKs. However, in this context untrusted may or may not refer to a malicious SDK. It is non-trivial to classify untrusted SDKs as malicious due to absence of common patterns in these SDKs. Therefore, we take a conservative approach where an SDK is untrusted if there is no public information available on the web. Clearly, security testing of untrusted SDKs is imperative to ensure the integrity of one's code. However, many programmers include desired functionalities into their projects without considering the security implications.

Another interesting observation is the usage of online app development platforms. These development platforms allow users to build an application with minimal technical effort and programming background. From the collected data, we found using manual inspection that approximately 8.64% (cf. Figure A.2) of the apps use an online app generation platform. A potential threat to these applications is that the developer/app provider using the online app generators neither has the knowledge about the internal

details of these apps nor do they perform rigorous testing. A recent study on these online app generators (OAG) found serious vulnerabilities for various OAG providers [85]. Again, programmers should not rely blindly on the quality of external tools and perform additional validation of the resulting app's security properties. Unfortunately, OAGs are particularly intriguing to developers with low technical expertise, so the creators of these platforms have a responsibility.

We discovered 18 instances (see e.g. URL for *com.quietgrowth.qgdroid* in Table A.2) where a call to *loadURL* was used to display a PDF via Google Docs, which is considered a misuse of WebView. To deliver content such files to users the WebView documentation recommends to invoke a browser through an *Intent* instead of using a WebView [35]. It appears that developers prefer users to stay inside the app for viewing documentation, and thus rather use *WebView* to accomplish this task.

List of Figures

List of Tables

Bibliography

[1] Event - Web APIs | MDN — developer.mozilla.org. https://developer.mozilla.org/en-US/docs/Web/API/Event. [Accessed 31-07-2023].

[2] 08 2022. URL https://github.com/vpaz11/python-graal-java.

[3] 08 2022. URL https://github.com/shelajev/graalpython-java-example.

[4] 08 2022. URL https://github.com/vspaz/graalvm-java-python-interoperation/tree/main/src/main/java/org/vspaz.

[5] Android webview and the ui thread, Dec 2022. URL https://chromium.googlesource.com/chromium/src/+/HEAD/android_webview/docs/threading.md.

[6] Rcpp for seamless r and c++ integration, 2022. URL http://www.rcpp.org.

[7] Rpy2: Python in r, March 2022. URL https://rpy2.github.io/doc/v2.9.x/html/index.html.

[8] Hybridroid. GitHub Repo, 2023. URL https://github.com/SunghoLee/HybriDroid.

[9] Lars Ole Andersen and Peter Lee. Program analysis and specialization for the c programming language. 2005. URL https://api.semanticscholar.org/CorpusID:20876553.

[10] Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. Porting doop to soufflé;: A tale of inter-engine portability for datalog-based analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2017, pages 25–30, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5072-3. doi: 10.1145/3088515.3088522. URL http://doi.acm.org/10.1145/3088515.3088522.

[11] Vincenzo Arceri and Isabella Mastroeni. Analyzing dynamic code: A sound abstract interpreter for evil eval. *ACM Trans. Priv. Secur.*, 24(2), jan 2021. ISSN 2471-2566. doi: 10.1145/3426470.

[12] ArgubLab. Nativeflowbench. URL https://github.com/arguslab/NativeFlowBench.

[13] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327848. doi: 10.1145/2594291.2594299. URL https://doi.org/10.1145/2594291.2594299.

[14] Avast. Avast/retdec: Retdec is a retargetable machine-code decompiler based on llvm., 2022. URL https://github.com/avast/retdec.

[15] Sora Bae, Sungho Lee, and Sukyoung Ryu. Towards understanding and reasoning about android interoperations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 223–233, 2019. doi: 10.1109/ICSE.2019.00038.

[16] George Balatsouras and Yannis Smaragdakis. Structure-sensitive points-to analysis for C and C++. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, pages 84–104, 2016. doi: 10.1007/978-3-662-53413-7_5. URL https://doi.org/10.1007/978-3-662-53413-7_5.

[17] Tim Berners-Lee, Roy Fielding, and Larry Masinter. Uniform resource identifier (uri): Generic syntax. Technical report, W3C, 2004.

[18] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640108. URL http://doi.acm.org/10.1145/1640089.1640108.

[19] Achim D. Brucker and Michael Herzberg. On the static analysis of hybrid mobile apps. In Juan Caballero, Eric Bodden, and Elias Athanasopoulos, editors, *Engineering Secure Software and Systems*, pages 72–88, Cham, 2016. Springer International Publishing. ISBN 978-3-319-30806-7.

[20] Samuele Buro, Roy L. Crole, and Isabella Mastroeni. On multi-language abstraction. In David Pichardie and Mihaela Sighireanu, editors, *Static Analysis*, pages 310–332, Cham, 2020. Springer International Publishing. ISBN 978-3-030-65474-0.

[21] IBM T.J. Watson Research Center. Wala t.j. watson libraries for analysis. https://github.com/wala/WALA, 2022.

[22] Erika Chin and David Wagner. Bifocals: Analyzing webview vulnerabilities in android applications. In *Revised Selected Papers of the 14th International Workshop on Information Security Applications - Volume 8267*, WISA 2013, pages 138–159, New York, NY, USA, 2014. Springer-Verlag New York, Inc. ISBN 978-3-319-05148-2. doi: 10.1007/978-3-319-05149-9_9. URL http://dx.doi.org/10.1007/978-3-319-05149-9_9.

[23] Patrick Cousot and Radhia Cousot. Modular static program analysis. In R. Nigel Horspool, editor, *Compiler Construction*, pages 159–179, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-45937-8.

[24] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101, Berlin, Heidelberg, 1995. Springer-Verlag. ISBN 3540601600.

[25] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 567–577, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306638. doi: 10.1145/1993498.1993565. URL https://doi.org/10.1145/1993498.1993565.

[26] Jordan Samhi et al. Jucify github tool. Github, May 2023. URL https://github.com/JordanSamhi/JuCify.

[27] Stack Exchange. Why are multiple programming languages used in the development of one product or piece of software?, March 2022. URL https://softwareengineering.stackexchange.com/questions/370135/why-are-multiple-programming-languages-used-in-the-development-of-one-product-or/370146#370146.

[28] F-Droid. Iwanddroid. URL https://f-droid.org/repo/com.github.axet.bookreader_416.apk.

[29] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 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*, pages 50–61. ACM, 2012.

[30] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In *ICSE*, 2013. doi: 10.1109/ICSE.2013.6606621. URL files/ICSE-2013-Approximate.pdf.

[31] Chris Hankin Flemming Nielson, Hanne Riis Nielson. *Principles of Program Analysis*, volume 1. Springer Berlin, Heidelberg, 1999.

[32] George Fourtounis, Leonidas Triantafyllou, and Yannis Smaragdakis. Identifying java calls in native code via binary scanning. New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380089. doi: 10.1145/3395363.3397368. URL https://doi.org/10.1145/3395363.3397368.

[33] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 377–396, May 2016. doi: 10.1109/SP.2016.30.

[34] Gustavo Genovese. Github, aug 2012. URL "https://github.com/gcesarmza/androidSamples/blob/master/webview/src/main/java/com/gustavogenovese/webview/SynchronousJavascriptInterface.java".

[35] Google. Building web apps in webview, dec 2018. URL https://developer.android.com/guide/webapps/webview.

[36] Google. Webview, September 2018. URL https://developer.android.com/reference/android/webkit/WebView.

[37] Google. Build web apps in webview, December 2022. URL https://developer.android.com/develop/ui/views/layout/webapps/webview.

[38] Google. Webassembly v8, Sep 2023. URL https://v8.dev/docs/wasm-compilation-pipeline.

[39] Google. Webview ui thread, June 2023. URL https://chromium.googlesource.com/chromium/src/+/HEAD/android_webview/docs/threading.md.

[40] Neville Grech and Yannis Smaragdakis. P/taint: Unified points-to and taint analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA):102:1–102:28, October 2017. ISSN 2475-1421. doi: 10.1145/3133926. URL http://doi.acm.org/10.1145/3133926.

[41] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Heaps don't lie: Countering unsoundness with heap snapshots. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi: 10.1145/3133892. URL https://doi.org/10.1145/3133892.

[42] Neville Grech, George Kastrinis, and Yannis Smaragdakis. Efficient Reflection String Analysis via Graph Coloring. In Todd Millstein, editor, *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, volume 109 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:25, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-079-8. doi: 10.4230/LIPIcs.ECOOP.2018.26. URL http://drops.dagstuhl.de/opus/volltexte/2018/9231.

[43] Manel Grichi, Ellis E. Eghan, and Bram Adams. On the impact of multi-language development in machine learning frameworks. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 546–556, 2020. doi: 10.1109/ICSME46990.2020.00058.

[44] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. High-performance cross-language interoperability in a multi-language runtime. New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336901. doi: 10.1145/2816707.2816714. URL https://doi.org/10.1145/2816707.2816714.

[45] Sascha Groß, Abhishek Tiwari, and Christian Hammer. Pianalyzer: A precise approach for pendingintent vulnerability analysis. In López et al. [74], pages 41–59. ISBN 978-3-319-98988-4. doi: 10.1007/978-3-319-98989-1_3. URL https://doi.org/10.1007/978-3-319-98989-1_3.

[46] Ben Gruver. Smali/baksmali, 2019. URL https://github.com/JesusFreke/smali.

[47] Ben Gruver. Android ndk, 2023. URL https://developer.android.com/ndk.

[48] Behnaz Hassanshahi, Yaoqi Jia, Roland H. C. Yap, Prateek Saxena, and Zhenkai Liang. Web-to-application injection attacks on android: Characterization and detection. In Günther Pernul, Peter Y A Ryan, and Edgar Weippl, editors, *Computer Security – ESORICS 2015*, pages 577–598, Cham, 2015. Springer International Publishing. ISBN 978-3-319-24177-7.

[49] Behnaz Hassanshahi, Yaoqi Jia, Roland H. C. Yap, Prateek Saxena, and Zhenkai Liang. Web-to-application injection attacks on Android: Characterization and detection. In Günther Pernul, Peter Y A Ryan, and Edgar Weippl, editors, *Computer Security – ESORICS 2015*, pages 577–598, Cham, 2015. Springer International Publishing. ISBN 978-3-319-24177-7.

[50] Dongjing He, Muhammad Naveed, Carl A Gunter, and Klara Nahrstedt. Security concerns in android mhealth apps. In *AMIA Annual Symposium Proceedings*, volume 2014, page 645. American Medical Informatics Association, 2014.

[51] S Fouzul Hidhaya, Angelina Geetha, B Nandha Kumar, Loganathan Venkat Sravanth, and A Habeeb. Supplementary event-listener injection attack in smart phones. *KSII Transactions on Internet and Information Systems (TIIS)*, 9(10): 4191–4203, 2015.

[52] https://www.rdocumentation.org. Foreign: Foreign function interface, Sep 2023. URL `https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/Foreign`.

[53] Hugo Illous, Matthieu Lemerre, and Xavier Rival. Interprocedural shape analysis using separation logic-based transformer summaries. In David Pichardie and Mihaela Sighireanu, editors, *Static Analysis*, pages 248–273, Cham, 2020. Springer International Publishing. ISBN 978-3-030-65474-0.

[54] Minseok Jeon and Hakjoo Oh. Return of cfa: Call-site sensitivity can be superior to object sensitivity even for object-oriented programs. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022. doi: 10.1145/3498720. URL `https://doi.org/10.1145/3498720`.

[55] Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, New York, NY, USA, 2014.

[56] Zeliang Kan, Haoyu Wang, Lei Wu, Yao Guo, and Guoai Xu. Deobfuscating android native binary code. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, ICSE '19, pages 322–323, Piscataway, NJ, USA, 2019. IEEE Press. doi: 10.1109/ICSE-Companion.2019.00135. URL `https://doi.org/10.1109/ICSE-Companion.2019.00135`.

[57] George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 423–434, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320146. doi: 10.1145/2491956.2462191. URL `https://doi.org/10.1145/2491956.2462191`.

[58] Uday P. Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs. *ACM Trans. Program. Lang. Syst.*, 30(1):1–es, nov 2007. ISSN 0164-0925. doi: 10.1145/1290520.1290521. URL https://doi.org/10.1145/1290520.1290521.

[59] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. ScanDal: Static analyzer for detecting privacy leaks in android applications. In Hao Chen, Larry Koved, and Dan S. Wallach, editors, *MoST 2012: Mobile Security Technologies 2012*, Los Alamitos, CA, USA, may 2012. IEEE. URL http://ropas.snu.ac.kr/scandal/.

[60] Pavneet Singh Kochhar, Dinusha Wijedasa, and David Lo. A large scale study of multiple programming languages and code quality. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 563–573, 2016. doi: 10.1109/SANER.2016.112.

[61] Goh Kondoh and Tamiya Onodera. Finding bugs in java native interface programs. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 109–118, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580500. doi: 10.1145/1390630.1390645. URL https://doi.org/10.1145/1390630.1390645.

[62] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, December 1992. ISSN 1057-4514. doi: 10.1145/161494.161501. URL http://doi.acm.org/10.1145/161494.161501.

[63] S. Lee, J. Dolby, and S. Ryu. Hybridroid: Static analysis framework for android hybrid applications. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 250–261, Sep. 2016.

[64] Sungho Lee. C-summary artifacts, Sep 2022. URL https://github.com/SunghoLee/c-summary.

[65] Sungho Lee and Sukyoung Ryu. *Adlib: Analyzer for Mobile Ad Platform Libraries*, pages 262–272. Association for Computing Machinery, New York, NY, USA, 2019. ISBN 9781450362245. URL https://doi.org/10.1145/3293882.3330562.

[66] Sungho Lee, Hyogun Lee, and Sukyoung Ryu. Broadening horizons of multilingual static analysis: Semantic summary extraction from c code for jni program analysis. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, pages 127–137, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367684. doi: 10.1145/3324884.3416558. URL https://doi.org/10.1145/3324884.3416558.

[67] Johannes Lerch, Johannes Späth, Eric Bodden, and Mira Mezini. Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, pages 619–629. IEEE Press, 2015. ISBN 9781509000241. doi: 10.1109/ASE.2015.9. URL https://doi.org/10.1109/ASE.2015.9.

[68] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *2015*

IEEE/ACM 37th IEEE International Conference on Software Engineering, volume 1, pages 280–291, 2015. doi: 10.1109/ICSE.2015.48.

[69] Li Li, Tegawend'e F. Bissyand'e, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 318–329, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4390-9. doi: 10.1145/2931037.2931044. URL http://doi.acm.org/10.1145/2931037.2931044.

[70] Siliang Li and Gang Tan. Jet: Exception checking in the java native interface. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 345–358, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450309400. doi: 10.1145/2048066.2048095. URL https://doi.org/10.1145/2048066.2048095.

[71] Tongxin Li, Xueqiang Wang, Mingming Zha, Kai Chen, XiaoFeng Wang, Luyi Xing, Xiaolong Bai, Nan Zhang, and Xinhui Han. Unleashing the walking dead: Understanding cross-app remote infections on mobile webviews. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 829–844. ACM, 2017.

[72] Zengyang Li, Xiaoxiao Qi, Qinyi Yu, Peng Liang, Ran Mo, and Chen Yang. Multi-programming-language commits in oss: An empirical study on apache projects. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 219–229, 2021. doi: 10.1109/ICPC52881.2021.00029.

[73] F-Droid Limited and Contributors. F-droid, 2023. URL https://f-droid.org.

[74] Javier López, Jianying Zhou, and Miguel Soriano, editors. *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II*, volume 11099 of *Lecture Notes in Computer Science*, 2018. Springer. ISBN 978-3-319-98988-4. doi: 10.1007/978-3-319-98989-1. URL https://doi.org/10.1007/978-3-319-98989-1.

[75] Amit Kr Mandal, Agostino Cortesi, Pietro Ferrara, Federica Panarotto, and Fausto Spoto. Vulnerability analysis of android auto infotainment apps. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pages 183–190. ACM, 2018.

[76] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-cfa paradox: Illuminating functional vs. object-oriented program analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 305–315, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300193. doi: 10.1145/1806596.1806631. URL https://doi.org/10.1145/1806596.1806631.

[77] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14 (1):1–41, January 2005. ISSN 1049-331X. doi: 10.1145/1044834.1044835. URL https://doi.org/10.1145/1044834.1044835.

[78] Anders Møller and Michael I. Schwartzbach. Static program analysis, November 2022. Department of Computer Science, Aarhus University, http://cs.au.dk/~amoeller/spa/.

[79] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. A multilanguage static analysis of python programs with native c extensions. In Cezara Drăgoi, Suvam Mukherjee, and Kedar Namjoshi, editors, *Static Analysis*, pages 323–345, Cham, 2021. Springer International Publishing. ISBN 978-3-030-88806-0.

[80] Patrick Mutchler, John Mitchell, Chris Kruegel, and Giovanni Vigna. A large-scale study of mobile web app security, 2015. URL http://www.ieee-security.org/TC/SPW2015/MoST/papers/s2p3.pdf.

[81] Luca Negrini, Pietro Ferrara, Vincenzo Arceri, and Agostino Cortesi. *LiSA: A Generic Framework for Multilanguage Static Analysis*, pages 19–42. Springer Nature Singapore, Singapore, 2023. ISBN 978-981-19-9601-6. doi: 10.1007/978-981-19-9601-6_2. URL https://doi.org/10.1007/978-981-19-9601-6_2.

[82] Matthias Neugschwandtner, Martina Lindorfer, and Christian Platzer. A view to a kill: Webview exploitation. In *Presented as part of the 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats*, Washington, D.C., 2013. USENIX. URL https://www.usenix.org/conference/leet13/workshop-program/presentation/Neugschwandtner.

[83] BBC news. Bbc news pashto. Google play store, 2020. URL https://play.google.com/store/apps/details?id=com.mobiroller.mobi883825667676&hl=en&gl=US.

[84] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. Modular call graph construction for security scanning of Node.js applications. In *Proc. 30th International Symposium on Software Testing and Analysis (ISSTA)*, July 2021.

[85] M. Oltrogge, E. Derr, C. Stransky, Y. Acar, S. Fahl, C. Rossow, G. Pellegrino, S. Bugiel, and M. Backes. The rise of the citizen developer: Assessing the security impact of online app generators. In *2018 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 102–115, 2018. doi: 10.1109/SP.2018.00005. URL doi.ieeecomputersociety.org/10.1109/SP.2018.00005.

[86] Oracle. oracle/graaljs: A ecmascript 2021 compliant javascript implementation built on graalvm. with polyglot language interoperability support. running node.js applications! URL https://github.com/oracle/graaljs.

[87] Oracle. Graalvm. https://www.graalvm.org, 2023.

[88] Oracle. Java native interface. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/, 2023.

[89] Adobe PhoneGap. Phonegap native bridge, aug 2010. URL "https://phonegap.com/blog/2010/08/13/phonegap-native-bridge/".

[90] Shasi Pokharel, Kim-Kwang Raymond Choo, and Jixue Liu. Can android voip voice conversations be decoded? i can eavesdrop on your android voip communication. *Concurrency and Computation: Practice and Experience*, 29(7):e3845, 2017.

[91] Jyoti Prakash, Abhishek Tiwari, and Christian Hammer. Effects of program representation on pointer analyses - an empirical study. In *Fundamental Approaches to Software Engineering - 24th International Conference, FASE 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, pages 240–261, 2021.

[92] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994. ISSN 0164-0925. doi: 10.1145/186025.186041. URL http://doi.acm.org/10.1145/186025.186041.

[93] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, 2014. ISBN 1-891562-35-5. doi: 10.14722/ndss.2014.23039.

[94] Thomas Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst.*, 22(1):162–186, January 2000. ISSN 0164-0925. doi: 10.1145/345099.345137. URL http://doi.acm.org/10.1145/345099.345137.

[95] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 49–61, 1995. doi: 10.1145/199448.199462. URL https://doi.org/10.1145/199448.199462.

[96] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do. In Mira Mezini, editor, *ECOOP 2011 – Object-Oriented Programming*, pages 52–78, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22655-7.

[97] Claudio Rizzo, Lorenzo Cavallaro, and Johannes Kinder. Babelview: Evaluating the impact of code injection attacks in mobile webviews. *arXiv preprint arXiv:1709.05690*, 2017.

[98] Connor Tumbleson Ryszard Wiśniewski. Apktool. https://ibotpeaches.github.io/Apktool/, 2020.

[99] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1):131–170, 1996. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(96)00072-2. URL https://www.sciencedirect.com/science/article/pii/0304397596000722.

[100] Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F. Bissyandé, and Jacques Klein. Jucify: A step towards android code unification for enhanced static analysis, 2021.

[101] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. Lossless, Persisted Summarization of Static Callgraph, Points-To and Data-Flow Analysis. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:31, Dagstuhl, Germany,

2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-190-0. doi: 10.4230/LIPIcs.ECOOP.2021.2. URL https://drops.dagstuhl.de/opus/volltexte/2021/14045.

[102] Mohamed Shehab and Abeer AlJarrah. Reducing attack surface on cordova-based hybrid mobile apps. In *Proceedings of the 2Nd International Workshop on Mobile Development Lifecycle*, New York, NY, USA, 2014. ACM.

[103] Li Siliang and Tan Gang. Finding bugs in exceptional situations of jni programs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 442–452, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605588940. doi: 10.1145/1653662.1653716. URL https://doi.org/10.1145/1653662.1653716.

[104] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, April 2015. ISSN 2325-1107. doi: 10.1561/2500000014. URL http://dx.doi.org/10.1561/2500000014.

[105] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:26, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-014-9. doi: 10.4230/LIPIcs.ECOOP.2016.22. URL http://drops.dagstuhl.de/opus/volltexte/2016/6116.

[106] Johannes Späth, Karim Ali, and Eric Bodden. Ideal: Efficient and precise alias-aware dataflow analysis. volume 1, New York, NY, USA, oct 2017. Association for Computing Machinery. doi: 10.1145/3133923. URL https://doi.org/10.1145/3133923.

[107] Johannes Späth, Karim Ali, and Eric Bodden. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proc. ACM Program. Lang.*, 3(POPL):48:1–48:29, 2019. doi: 10.1145/3290361. URL https://doi.org/10.1145/3290361.

[108] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 387–400, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: 10.1145/1133981.1134027. URL http://doi.acm.org/10.1145/1133981.1134027.

[109] Manu Sridharan and Stephen J. Fink. The complexity of andersen's analysis in practice. In Jens Palsberg and Zhendong Su, editors, *Static Analysis*, pages 205–221, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-03237-0.

[110] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. *Alias Analysis for Object-Oriented Programs*, pages 196–232. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-36946-9. doi: 10.1007/978-3-642-36946-9_8. URL https://doi.org/10.1007/978-3-642-36946-9_8.

[111] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 32–41, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917693. doi: 10.1145/237721.237727. URL https://doi.org/10.1145/237721.237727.

[112] Yulei Sui and Jingling Xue. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 265–266, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342414. doi: 10.1145/2892208.2892235. URL https://doi.org/10.1145/2892208.2892235.

[113] Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 40 (2):107–122, 2014.

[114] Kwangwon Sun and Sukyoung Ryu. Analysis of javascript programs: Challenges and research trends. *ACM Computing Surveys (CSUR)*, 50(4):59, 2017.

[115] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285 – 309, 1955. doi: pjm/1103044538. URL https://doi.org/.

[116] A. Tiwari, J. Prakash, S. GroB, and C. Hammer. Ludroid: A large scale analysis of android—web hybridization. In *2019 IEEE 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 256–267, Los Alamitos, CA, USA, oct 2019. IEEE Computer Society. doi: 10.1109/SCAM.2019.00036. URL https://doi.ieeecomputersociety.org/10.1109/SCAM.2019.00036.

[117] Abhishek Tiwari, Sascha Groß, and Christian Hammer. IIFA: modular inter-app intent information flow analysis of android applications. *CoRR*, abs/1812.05380, 2018. URL http://arxiv.org/abs/1812.05380.

[118] Abhishek Tiwari, Jyoti Prakash, Sascha Groß, and Christian Hammer. A large scale analysis of android — web hybridization. *Journal of Systems and Software*, 170:110775, 2020. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2020.110775. URL https://www.sciencedirect.com/science/article/pii/S0164121220301898.

[119] Abhishek Tiwari, Jyoti Prakash, and Christian Hammer. Demand-driven information flow analysis of webview in android hybrid apps. volume 1, Florence, Italy, oct 2023. IEEE. doi: 10.1145/3133923.

[120] Alexi Turcotte, Ellen Arteca, and Gregor Richards. Reasoning About Foreign Function Interfaces Without Modelling the Foreign Language. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:32, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-111-5. doi: 10.4230/LIPIcs.ECOOP.2019.16. URL http://drops.dagstuhl.de/opus/volltexte/2019/10808.

[121] WALA. Watson libraries for program analysis, Jan 2023. URL http://wala.sourceforge.net/wiki/index.php/Main_Page.

[122] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 252–276, Cham, 2017. Springer International Publishing. ISBN 978-3-319-60876-1.

[123] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. Jnsaf: 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*, CCS '18, pages 1137–1150, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243835. URL https://doi.org/10.1145/3243734.3243835.

[124] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Trans. Priv. Secur.*, 21(3), April 2018. ISSN 2471-2566. doi: 10.1145/3183575. URL https://doi.org/10.1145/3183575.

[125] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Traynor, and Sascha Fahl. A large scale investigation of obfuscation use in google play. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, pages 222–235, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6569-7. doi: 10.1145/3274694.3274726. URL http://doi.acm.org/10.1145/3274694.3274726.

[126] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 187–206, New York, NY, USA, 1999. Association for Computing Machinery. ISBN 1581132387. doi: 10.1145/320384.320400. URL https://doi.org/10.1145/320384.320400.

[127] Christian Wimmer and Thomas Würthinger. Truffle: A self-optimizing runtime system. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, page 13–14, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450315630. doi: 10.1145/2384716.2384723. URL https://doi.org/10.1145/2384716.2384723.

[128] Guangliang Yang, Jeff Huang, Guofei Gu, and Abner Mendoza. Study and mitigation of origin stripping vulnerabilities in hybrid-postmessage enabled mobile applications. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 742–755. IEEE, 2018.

[129] Dongjun Youn, Sungho Lee, and Sukyoung Ryu. Declarative static analysis for multilingual programs using codeql. *Software: Practice and Experience*, n/a (n/a). doi: https://doi.org/10.1002/spe.3199. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3199.

[130] Xiaohan Zhang, Yuan Zhang, Qianqian Mo, Hao Xia, Zhemin Yang, Min Yang, Xiaofeng Wang, Long Lu, and Haixin Duan. An empirical study of web resource manipulation in real-world mobile applications. In *27th USENIX Security Symposium 18*, pages 1183–1198. USENIX Association, 2018.