GlassWing: A Tailored Static Analysis Approach for Flutter Android Apps

Xiangyu Zhang*†, Yucheng Su‡, Lingling Fan*†§, Miaoying Cai*, Sen Chen*,
*DISSec, NDST, College of Cryptology and Cyber Science, Nankai University, China
†Zhongguancun Academy, China
‡Intelligence and Offensive Defense Lab, Xiaohongshu Inc., China

***Abstract*—The variety of mobile operating systems available in the market has led to the emergence of cross-platform frameworks, which simplify the development and deployment of mobile applications across multiple platforms simultaneously. Among these, the Flutter framework promoted by Google has become a widely used cross-platform development framework. To date, no work has provided support for the static analysis of Flutter apps on the Android platform. State-of-the-art static analyzers fail to "see" the implicit invocation between the Dart language used by the Flutter framework and the Dalvik bytecode (DEX) used by the native Android platform, posing a significant threat to the completeness of the mobile software analysis.**

In this paper, we present GlassWing, the first tailored approach to static analysis for Flutter Android apps. GlassWing leverages a data-flow-oriented approach to conduct key program semantic extraction of Flutter apps and discloses the implicit Dart-DEX invocation relations, thereby making cross-language invocation visible. Extensive evaluation on 1,023 popular real-world Flutter apps indicates that GlassWing enhances static analysis of Flutter apps integrated with Soot by parsing 141% more Jimple code lines, extending the call graph with more edges and nodes, and revealing almost 3X potential sensitive data leaks that were previously undetected with FlowDroid. GlassWing sheds light on downstream research fields for Flutter apps (e.g., program graph analysis, taint analysis, and malicious software analysis). Many current and future Android analysis initiatives can be enhanced by seamlessly incorporating GlassWing's insights.

***Index Terms*—Static Analysis, Flutter Android Apps, Cross-platform Framework**

I. Introduction

After years of development, competition in the mobile application market has reached an intense stage, with developers striving to deliver their applications to a wider audience [1]. Currently, popular mobile operating systems (OSes) such as Android [2], iOS [3], and HarmonyOS [4] exist in the market, requiring developers to build applications for each OS to remain competitive [5]. However, developing and maintaining applications across different OSes proves to be a non-trivial task [6]. This process inadvertently slows down the delivery pace of internet products that aim to respond quickly to market changes. In response to this situation, cross-platform development frameworks (e.g., Flutter [7], React Native [8], Weex [9]), introduced a paradigm shift in deploying mobile applications (named as cross-platform mobile apps) across multiple mobile operating systems (named as platforms) [10].

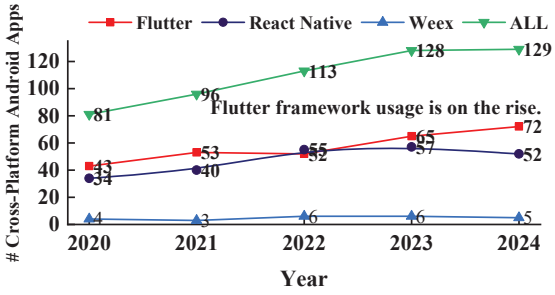


Fig. 1. Usage of identified cross-platform frameworks among the top 1,000 Android apps (ranked by downloads) on Google Play.

The core idea of development by cross-platform frameworks is to develop each app once (in Dart [7], JavaScript [11], etc.) and then compile it for multiple platforms, thereby enhancing development efficiency and consistency [12].

To gain insights into the current Android app market landscape, we conducted a preliminary study of the annual top 1,000 Android apps on Google Play [13], ranked by downloads from Sensor Tower [14] over the past five years. As shown in Fig. 1, the adoption of the Flutter framework has increased annually. We also conducted an empirical study on 11,537 applications from Google Play: 1,023 are Flutter apps (8.9%), 853 are React Native apps (7.4%), and 53 are Weex apps (0.5%). Both studies suggest that Flutter is one of the leading cross-platform development frameworks for modern Android app development. Li et al. [1] also find this trend is even more pronounced in the Tencent App Store,[1] with nearly 50% Flutter app adoption. Impacted and backed by Google, the Flutter framework utilizes Google's custom-developed Dart language and Skia rendering engine [7]. The Flutter app[2] (in Dart) has stood out by achieving code execution efficiency comparable to native Android apps (in Java/Kotlin), creating a performance gap with other cross-platform apps (in JavaScript) using frameworks like React Native and Weex [15]. The endorsement from renowned companies such as Google, eBay, Alibaba, Tencent, and TikTok has further fueled Flutter's rising popularity [16]. However, academic exploration in corresponding static analysis remains limited compared to the

§ Lingling Fan is the corresponding author (linglingfan@nankai.edu.cn).

[1] https://appstore.tencent.com/

[2] Android apps developed using the Flutter framework

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00050

widespread practical use of the Flutter framework.

Samhi et al. [17] have highlighted that the most advanced Android static analysis struggles to fully and automatically handle the implicit invocation of methods by cross-platform development frameworks. It is primarily because most static analyzers are designed for natively developed Android apps. For instance, analyzers such as FlowDroid [18], IccTA [19], Raicc [20], and DroidSafe [21] are unable to analyze cross-platform Android apps, as they leverage Java/Kotlin static analysis techniques to handle `Dex` bytecode [22] in Android apps [1], typically using the Jimple code as their intermediate representation (IR) [23] of the powerful Soot framework [24]. However, they do not support the languages used by frameworks (Dart/JavaScript) and therefore cannot parse the corresponding implicit invocations. To date, only a few efforts [25]–[27] have provided static analysis support for specific cross-platform development frameworks (i.e., React Native) on the Android platform. However, no existing work supports static analysis for the Flutter Android apps, which is ahead-of-time (AOT) [7] compiled into native ARM code [28], employing platform channel mechanisms [29] to facilitate interaction between Dart code and Java/Kotlin code. React Native apps fundamentally differ in programming languages (JavaScript), compilation processes (Interpreted [30]), and framework architectures (bridge [31] or JSI [32]) from the Flutter app (Dart, AOT, channel). These differences imply that existing analyzers for React Native apps cannot be directly applied to the Flutter apps without addressing these unique challenges. Missing the implicit invocations between the Android platform and the Flutter framework leads to an incomplete analysis of the Flutter app behavior (e.g., missing nodes and edges in the *G[3]) [17]. Consequently, attackers can exploit these gaps to bypass advanced static analysis tools like FlowDroid, concealing malicious code. It highlights that even the most sophisticated static analysis tools are ineffective if they rely on an unsound model of the app [17].

To enable static analysis support for Flutter apps, a promising direction involves extracting interaction behaviors of Dart and Dalvik bytecode (DEX, compiled from Java/Kotlin [33]) and disclosing implicit invocations between them. By doing so, we can establish bi-directional communication and support downstream applications (e.g., *G analysis, taint analysis, and malware analysis) to better accommodate Flutter apps. However, due to the intricate nature of cross-language interactions inherent in cross-platform apps and language differences between Dart and Java/Kotlin, we face the following challenges:
C1: Dart Semantics Extraction of Flutter Apps: For Flutter app analysis, basic program semantics from both the Dart and DEX sides are crucial. However, mainstream decompilers [34]–[36] fail to analyze the compiled artifacts of Dart due to its unique low-level structure (which differs from C/C++) [37], making it challenging to comprehend the basic code semantics essential for static analysis of Flutter apps.
C2: Cross-Language Implicit Invocation Identification: Dart-DEX method calls are handled implicitly, without direct code references, making it challenging to directly extract the cross-language invocation. Consequently, the call edges that connect the call relations between both sides are missing, preventing the whole-app analysis.

To this end, to fill the gap of static analysis of Flutter Android apps, we propose GlassWing. Inspired by the Glasswing Butterfly [38] (fluttering wings echo Flutter's logo [7]), our approach aims to achieve a thorough analysis of Flutter apps, as clear as glass. To address **C1**, GlassWing presents a data-flow-oriented method to conduct Dart semantic extraction. GlassWing leverages a dedicated Dart machine code analyzer to extract key program semantics related to Dart-DEX interaction, tackling the intricate nature of Dart to pave the way for cross-language analysis. To tackle **C2**, GlassWing thoroughly analyzes the Dart-DEX call sites to discover the mappings for implicit cross-language calls. Furthermore, it reconstructs the explicit invocation relations between Dart code and Dalvik bytecode. In this way, GlassWing establishes a unified Jimple representation that restores cross-language invocation edges, connecting Dart-DEX invocation relations, thereby making cross-language invocations visible and enhancing static analysis of the Flutter app.

To demonstrate the effectiveness of GlassWing, we conduct experiments on our ground-truth benchmark and 1,023 real-world Flutter apps, evaluating its ability to accurately extract implicit invocations (RQ1), improve static analysis (RQ2), and detect potential sensitive data leaks (RQ3). GlassWing can accurately extracts implicit invocations and sensitive data leaks in our benchmark. On real-world apps, it enhances static analyzers by increasing the number of analyzed methods (by 36.7%), and extracting more call graph nodes (by 28.2%) and edges (by 28.1%), and uncovering previously undiscovered potential sensitive data leaks (nearly 3X), with an acceptable performance overhead (RQ4). Our approach bridges the gap in static analysis of Flutter apps and adapts to the evolving mobile app landscape. The main contributions of our work are summarized as follows:

- We propose GlassWing, the first research work toward conducting the static analysis of Flutter Android apps, which supports cross-language communication between the native Android platform and the Flutter framework.
- We present a data-flow-oriented approach to construct the summarization of Dart-DEX interaction and disclose the cross-language implicit invocation relations to enable static analysis of Flutter Android apps.
- We have demonstrated the effectiveness of GlassWing in identifying implicit invocations, enhancing existing static analyzers, and revealing potential sensitive data leaks that were previously undetectable in Flutter apps.
- GlassWing is fundamental work that bridges the gap in the static analysis of Flutter Android apps, facilitating future advancements. We have released our artifact and results on GitHub website [39].

[3]We use *G to present the existing program analysis graphs, such as Call Graph (CG), Control Flow Graph (CFG), Interprocedural Control Flow Graph (ICFG), and Code Property Graph (CPG), etc.

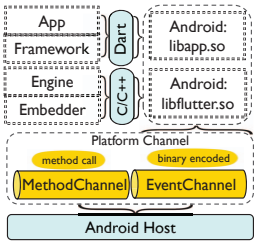


Fig. 2. Platform channel in Flutter.

Fig. 3. A motivating example of an implicit call to the `Leak` method.

II. PRELIMINARIES

A. The Flutter Framework

Flutter. Flutter is Google's primary cross-platform development framework and serves as the official SDK for Google's next-generation mobile operating system, Fuchsia [40]. The core idea of Flutter is to develop each app once in Dart [41] and then compile it for multiple platforms (e.g., Android [2], iOS [3], HarmonyOS [4], etc.). In the Flutter compilation pipeline, source code is initially parsed into an Abstract Syntax Tree (AST), and subsequently transformed into the Dart intermediate language (IL). This Dart IL is a lower-level representation than the AST while remaining independent of specific hardware platforms. Dart IL forms the basis for subsequent compilation and optimization. Thereafter, Dart's Ahead-of-Time (AOT) compiler compiles the IL into native machine code for the target platform, and packages it into a snapshot [42]. As shown in Fig. 2, a compiled Flutter app consists of two dynamic libraries: one is `libapp.so`, which contains the main application logic written in Dart, and the other is `libflutter.so`, an engine responsible for both runtime and rendering for the Flutter framework written in C++ [7]. In the static analysis of Flutter applications, it is necessary to separately extract the Flutter Dart code (i.e., `libapp.so`) and the Android Dalvik bytecode (i.e., `.dex` files) from the APK file.

Platform Channel. Platform channel serves as a communication bridge between Flutter framework and the Android platform, enabling Flutter apps to call native Java/Kotlin APIs from Dart and, in return, receive the data or results from that native Java/Kotlin code [43]. Since Flutter only manages the application rendering layer, system APIs cannot be supported within the Flutter framework [44]. On the other hand, Flutter is still a relatively young framework for Android, so some mature Java/Kotlin libraries only used in the development of the native Android platform (e.g., those for image processing and audio/video codecs) have not yet been implemented in the Flutter framework [45]. Consequently, the Flutter framework provides a lightweight solution for developers through the platform channel mechanism. The platform channel exposes the features of the native Android platform to Dart (Fig. 2), enabling Dart-DEX interaction as if a regular Dart API was being called. Specifically, after a Flutter app declares the platform channel (e.g., Fig. 3, $expchannel$ in Line $D1$), both the Flutter framework (e.g., Line $D2$) and native Android (e.g., Line $J5$) can reuse it. They can achieve cross-platform calls on this channel by registering methods with specific names (i.e., $getBatteryLevel$), allowing the other side to invoke these methods using the registered names. Thus, platform channels essentially provide the mechanism of implicit invocations for Flutter apps, without direct code references.

B. Motivating Example

Current Java-centric static analyzers [1] for Android struggle to adequately analyze cross-platform features within Flutter apps. Fig. 3 illustrates an implicit call scenario. The Dart code invokes the `Leak` method (i.e., $getBatteryLevel$ in Line $J7$) via the method channel (i.e., $expchannel$). This invocation is not resolved within the Dart code; instead, it is implicitly triggered by the Flutter framework to the corresponding method in the Java code [17]. As a result, no explicit call site for this method exists within the Java codebase. Static analyzers [18]–[20], [46]–[58] that rely on explicit call relations to construct a call graph are therefore unable to establish a call edge to the Java method in which the leak occurs. Consequently, the method is erroneously identified as unreachable, leading to an incomplete analysis result.

EBay Motors [59] is an exemplary application using the Flutter framework officially endorsed by Google and has amassed millions of installs across multiple platforms [16]. This app (v1.20.0) stores the Flutter Dart code in the `libapp.so` file. The disassembly of this shared library file generated a 417.7 MB text file with 12,895,862 lines of assembly. However, existing research cannot analyze the Dart code due to the inability of cross-platform analysis, leaving the 12,617 methods in this file unanalyzed.

By constructing a call graph (CG) for Android Flutter applications, we explore the deficiencies in existing research [18]–[20] of disclosing the hidden logic on the native Android Java side. Initially, we used FlowDroid [18], Iccta [19], and Raicc [20] to generate a CG and perform taint analysis for Motors 1.20.0. The best resulting CG contained 3,408 nodes and 9,758 edges, with FlowDroid detecting no sensitive data leaks. Flutter provides platform channels for Dart to access the native Android platform, and further manual inspection of this Flutter app revealed that 13 cross-platform methods exposed on the Java side via platform channels were not included in the CG. By manually submitting these Java channel methods as entry points to FlowDroid, the CG expanded to include 6,931 nodes and 24,901 edges, identifying 12 instances of sensitive data leaks.

Our motivation cases suggest that the relevance of established static analyzers for Android apps to apps using the Flutter framework is dubious [1], [17]. Therefore, our development community urgently needs specialized static analyzers for applications built with the Flutter framework.

III. APPROACH

GlassWing enables the static analysis of Flutter Android apps, uncovering the hidden code logic of Dart-DEX interaction. Specifically, as illustrated in Fig. 4, GlassWing takes APK

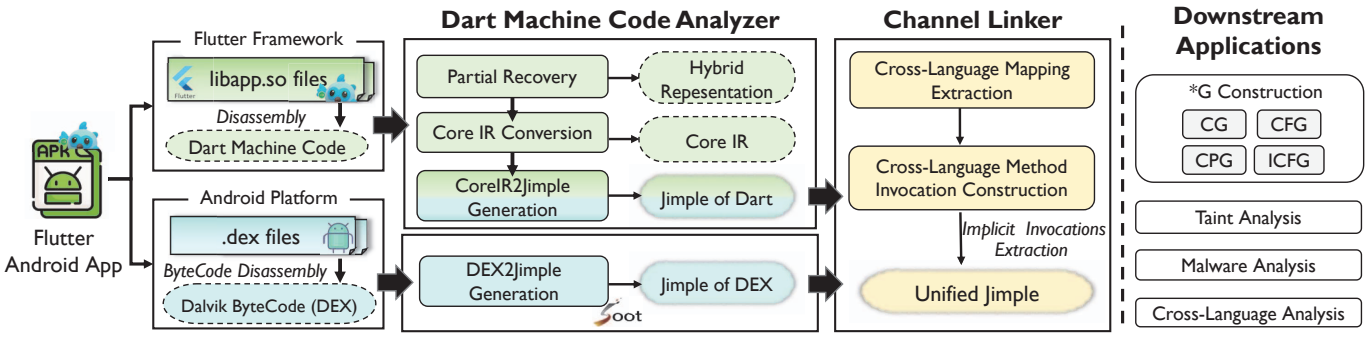


Fig. 4. Overview of GlassWing.

files as input and employs a divide-and-conquer strategy. For Android side, GlassWing utilizes Soot to convert Dalvik bytecode into Jimple IR; whereas for Flutter Dart code, it employs two innovative modules: (1) `Dart Machine Code Analyzer`. This module progressively converts compiled Dart code into Jimple IR, leveraging a data-flow-oriented extraction of Dart semantics, specific to Dart-DEX interaction; and (2) `Channel Linker`, which extracts cross-language implicit invocations and establishes a unified Jimple IR that contains cross-language invocation relations, connecting Dart and Java/Kotlin invocation relations and facilitating the Dart-DEX interaction. As the foundation of cross-language analysis of Android apps, GlassWing supports `Downstream Applications` such as *G analysis (Section IV-B), taint analysis (Section IV-D), and malware analysis (Section V-A).

GlassWing pursues implementing static analysis in Flutter apps, with a particular focus on Dart-DEX call relations and data-flow relations. As effective data-flow analysis relies on a well-designed IR, specific instruction types (i.e., $Assign$, $Load$, $Store$, $Call$, and Ret) have been proven effective for analyzing data flow [24], [60], [61]. Therefore, following the principles of existing IR designs, we define **Core IR** as a data-flow-oriented extraction of Dart semantics in our context, built upon these five key instructions to capture Dart-DEX interactions. GlassWing also considers the general Dart code logic that affects data before the interaction, which is important for the analysis of data-flow relations. Notably, GlassWing Core IR refrains from recovering all Dart source code details, avoiding full decompilation, because undertaking full decompilation would introduce redundant information irrelevant to our core task, consequently increasing analysis complexity and potentially yielding unreliable results.

A. Dart Machine Code Analyzer

The Dart machine code analyzer of GlassWing comprises three steps: (1) Partial Recovery; (2) Core IR Conversion; and (3) CoreIR2Jimple Generation, following a progressive process to advance the conversion to Dart-Jimple to facilitate a tailored static analysis for Flutter apps.

*1) **Partial Recovery**:* GlassWing performs preliminary processing on Dart AOT artifacts (in the form of Dart machine code), converting them into a more analyzable Hybrid Representation (HR) through the following three critical partial recovery operations.

- **Register Normalization:** GlassWing normalizes register variants of different bit widths (e.g., 64-bit `X0` and 32-bit `W0`) into a canonical representation using logical registers (e.g., `R0`). This normalization allows the analysis to focus on the register as a value container, irrespective of the access bit width used in specific instructions.
- **Object Pool Resolution:** Dart utilizes an object pool for object management [42]. Its assembly code typically accesses objects via indices (i.e., indirect references) rather than direct addresses, precluding reliance on direct pointers or symbolic references to identify target objects. So GlassWing parses the Dart snapshot, dumping the object pool structure and object information. Subsequently, it identifies indirect references through the object pool base address register (`X27`), establishing mappings between code instructions and the corresponding actual Dart objects within the pool (e.g., Lines A6 to B6 of Fig. 5).
- **Dart IL Recovery:** Analyzing Dart machine code is non-trivial due to its low level of abstraction. GlassWing analyzes the compilation process of the Dart SDK to investigate how Dart IL converts into particular Dart machine code instruction sequences. Based on it, GlassWing then builds a mapping by cataloging IL instructions alongside their corresponding ARM instruction sequences. Consequently, GlassWing recognizes these sequences within the Dart machine code and replaces them with the equivalent high-level IL instructions (e.g., Line A1 to B1, Line A3 to B3 of Fig. 5).

After partial recovery of the Dart AOT artifacts, the code is represented as a Hybrid Representation (HR) (e.g., Block B of Fig. 5).

*2) **Core IR Conversion**:* HR intertwines high-level IL instructions (e.g., `LoadField`, `StoreField`) with remaining low-level ARM instructions (e.g., `mov`, `ldur`), which is impractical to directly perform data-flow analysis of Dart code. This complexity introduces **challenges** for our data-flow-oriented method. (1) The partially recovered HR contains many types of instructions, making it challenging to directly capture data-flow relations between HR instructions. (2) within HR, Dart continues to utilize a hardware location (HL, e.g., registers

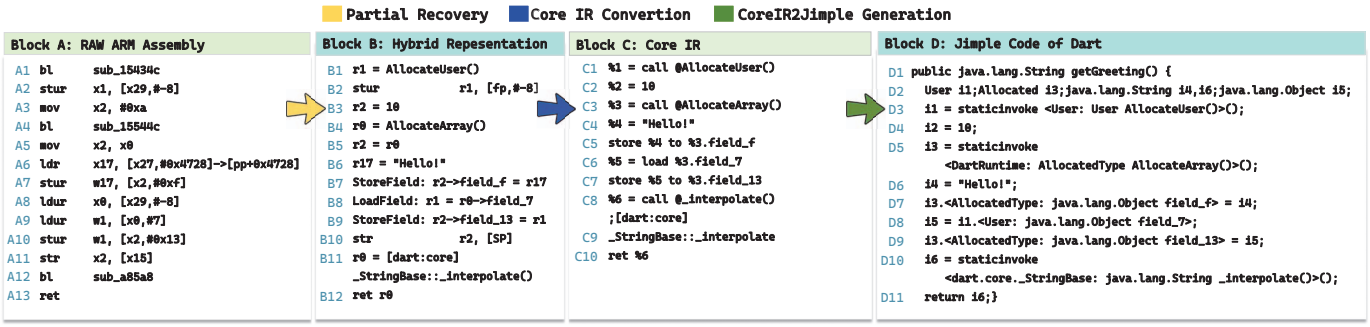


Fig. 5. Processing flow of the Dart machine code analyzer.

or stack) to store and transfer values. HL is subject to reuse, and values transfer among HL. Directly performing data-flow analysis on HL renders the analysis complex and error-prone.

To overcome the analysis obstacles posed by the numerous instruction types in HR, GlassWing converts these instructions into Core IR. GlassWing leverages Algorithm 1 to depict the procedure for converting HR to Core IR. The input is the sequence of HR instructions (I_{HR}), and the output is the list of Core IR instructions (IR_{core}). Specifically, IR_{core} is first initialized as empty and gradually augmented during the conversion. Virtual Register (VR) is used to hold values within the Core IR, in the format `%X`, with a counter cnt to generate unique identifiers (e.g., `%1`, `%2`). Along with IR_{core}, a state map M is employed to maintain a mapping from VR to the corresponding HL. By decoupling value from HL, GlassWing overcomes the difficulties of performing data-flow analysis directly on HL. For each instruction ($instr$) in I_{HR}, GlassWing determines its type using semantic information, and then executes different processing logic based on the type (Lines 2-3). As VR is the container for the data flow, the processing for each instruction type essentially involves how GlassWing operates on VR. We summarize the operations on VR as allocation, substitution, and transfer, serving to capture the data-flow relations of Dart.

• **Virtual Register Allocation**: If $instr$ defines a new value (e.g., $Load$, $Assign$, $Call$), GlassWing checks $instr$ to identify the pre-existing values serving as source operands (OP_{src}) (Line 5). GlassWing summarizes OP_{src} as 3 types: constants, values stored in HL, or return values of functions. If a OP_{src} is stored in a HL, GlassWing queries the state M to retrieve its corresponding current VR for substitution. Then, a new VR (VR_n) is allocated and named based on the current value of the cnt (Line 6). GlassWing generates the corresponding Core IR instruction that assigns the value derived from OP_{src} to VR_n (Line 7). Finally, M is updated by mapping the destination HL (i.e., HL_{dest}) of $instr$ to the VR_n (Lines 8-9). For example, in Fig. 5, consider the instruction at Line B3, where the $OP_{src} = 10$ and the HL_{dest} is `R2`. A new VR, `%2`, is generated, and the Core IR instruction `%2` $= 10$ is created (Line C2). M adds the mapping: `R2` $\rightarrow$ `%2`.

• **Virtual Register Substitution**: If $instr$ consumes a pre-existing value and does not define a new value (e.g., $Store$, Ret), GlassWing identifies the HL involved in the current $instr$ (Line 11), and retrieves the corresponding VR stored for these HL from the state M (Line 12). Finally, it replaces the HL in $instr$ with the retrieved VR (Line 13). For example, in Fig. 5, the instruction (Line B7) involves HL: `R17` and `R2`, and the VR (Line C5) currently mapped to them are `%4` and `%3` respectively. Then, the Core IR instruction `store %4 to %3.field_f` is constructed.

• **Virtual Register Transfer**: If $instr$ performs the value transfer operation in HL, it will cause an update to state M (e.g., register-to-register operation (`MOV`), and stack operations (`ldur` and `str`)). To record the update of M and track the data flow from different HL, GlassWing parses source and destination HL (i.e., HL_{src}, HL_{dest}) of these instructions (Line 15). It then finds the VR_{src} associated with HL_{src} in M (Line 16). Finally, to reflect this transfer of VR, the

Algorithm 1: Core IR Generation

Input: I_{HR}: Sequence of HR instructions
Output: IR_{core}: Sequence of Core IR instructions

1 $cnt \leftarrow 1$, $M \leftarrow \emptyset$, $IR_{core} \leftarrow \emptyset$
2 **foreach** $instr \in I_{HR}$ **do**
3 **switch** *SemanticType(instr)* **do**
4 **case** *ValueDefinition* **do**
 // Virtual Register Allocation
5 $OP_{src} \leftarrow$ FindSource$(M, instr)$
6 $VR_n \leftarrow$ "%cnt"; $cnt \leftarrow cnt + 1$
7 $IR_{core}.add($BuildIR$(instr, VR_n, OP_{src}))$
8 $HL_{dest} \leftarrow$ FindHL$(M, instr)$
9 $M \leftarrow$ UpdateMap(HL_{dest}, VR_n)
10 **case** *ValueUse* **do**
 // Virtual Register Substitution
11 $HL \leftarrow$ FindHL$(M, instr)$
12 $VR \leftarrow$ FindVR(M, HL)
13 $IR_{core}.add($BuildIR$(instr, VR, HL))$
14 **case** *ValueTransfer* **do**
 // Virtual Register Transfer
15 $HL_{src}, HL_{dest} \leftarrow$ FindHL$(M, instr)$
16 $VR_{src} \leftarrow$ FindVR(M, HL_{src})
17 $M \leftarrow$ UpdateMap(HL_{dest}, VR_{src})
18 **case** *Exclusions* **do**
19 continue
20 **return** IR_{core}

mapping in M for HL_{dest} is updated to $HL_{dest} \rightarrow VR_{src}$. (Line 17). For example, (Line B5 of Fig. 5), if `R0` (HL_{src}) currently maps to `%3` (VR_{src}) (i.e., `R0` $\rightarrow$ `%3`), after `R2 = R0`, GlassWing updates M of `R2` (HL_{dest}) to `R2` $\rightarrow$ `%3`.

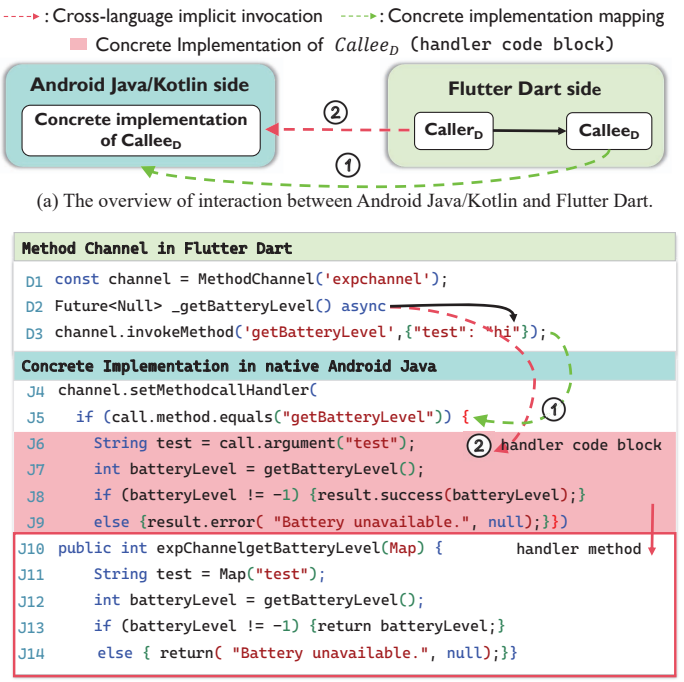
• **Exclusions**: GlassWing skips annotations about stack frame management (e.g., `EnterFrame`, `LeaveFrame`) or runtime checks (e.g., `CheckStackOverflow`), as well as other instructions not requiring representation at the Core IR level (via the `continue` operation), since they do not directly contribute to the data flow representation within the Core IR.

*3) **CoreIR2Jimple Generation**:* Leveraging the converted Core IR as input, this step paves the way for GlassWing to be supported by the standard Android analysis toolchain (e.g., Soot, FlowDroid). First, GlassWing performs the **Java type inference** of the VR based on the type of the value assigned at its initial definition (e.g., `%2` can be inferred as type `Int`). GlassWing determines the function's return type based on the type of the VR used in the return instruction (e.g., the return type can be inferred as `String` from `%6`). If inference is not possible, the default type `java.lang.Object` is utilized. Next, GlassWing performs the **ID resolution**. It ascertains the function's name using symbol table information and determines the Jimple method's ID based on the previously inferred return type. Subsequently, GlassWing executes the **method body generation**. For each VR, it generates a local variable within the method body. Then, based on predefined conversion rules, GlassWing converts each Core IR instruction ($Assign$, $Store$, $Load$, $Call$, Ret) into its corresponding Jimple statement ($assignment$, $field\ write$, $field\ read$, $invocation$, $return$), respectively (e.g., Block D of Fig. 5).

B. Channel Linker

So far, GlassWing converts native Android Java/Kotlin (AJ) and Flutter Dart (FD) into Jimple code separately. However, taking cross-language call relations into account, linking AJ and FD is a key **challenge**. Dart-to-DEX calls in Flutter apps are inherently implicit, without direct code references, hindering the establishment of cross-language calls. To address the challenge, GlassWing employs the channel linker (via channel resolution as shown in Fig. 6) to disclose implicit Dart-to-DEX calls, as platform channels expose features of Dart, facilitating cross-language interaction. GlassWing establishes the actual Dart-DEX call edges by refactoring channel methods into directly invokable Jimple methods, ultimately linking the channel.

*1) **Cross-Language Mapping Extraction**:* Detecting cross-language interactions is a prerequisite for conducting cross-language analyses [62]. In Flutter apps, platform channels provide the message-passing mechanism for cross-language interaction. Crucially, this mechanism relies on implicit invocation (as noted in Section II-A), leading to no explicit cross-language interaction (e.g., method calls), thus posing a challenge to fully comprehend the interaction between AJ and FD. Therefore, we must identify the implicit invocation relation between the two sides to enable their interaction. Flutter apps have their explicit call relations (i.e., $caller_D \rightarrow callee_D$) on separate sides (i.e., FD side), as shown in Fig. 6. However,



(a) The overview of interaction between Android Java/Kotlin and Flutter Dart.

(b) Implementation of Method Channel between native Android Java and Flutter Dart

Fig. 6. The interaction between Android platform and Flutter framework (Channel Resolution).

due to development features of these apps, the explicit callee methods (i.e., $callee_D$) on the FD side have no concrete implementation (called "virtual method" in this paper), leaving the concrete implementation in the AJ side. In this step, we aim to extract the call sites of each side and map the virtual methods on one side to their concrete implementation on the other side (i.e., ① in Fig. 6), serving as a foundation for subsequent method invocation relation construction in Section III-B2.

Specifically, we analyze the code obtained from the previous step to identify call sites and explicit method calls independently on both sides. We illustrate the process in Fig. 6. To elucidate the explicit call relations on separate sides (i.e., callers and virtual callees), we first locate instances of the channel class, which are used to identify the call sides of cross-language interaction methods in the code. We record these instances along with the channel names used in cross-language interactions ($MethodChannel$ in Line $D1$). Next, we track the control flow until we reach the call site of a channel instance, flagged by the $invokeMethod$. At this point, we have identified the virtual methods, i.e., $callee$ ($getBatteryLevel$ in Line $D3$). We then backtrack to the parent methods that call these callees and note the $caller$ on each side (Line $D2$). By analyzing the virtual callee methods, we extract the label (e.g., "getBatteryLevel") and its parameters, which are typically in the form of a parameters map (e.g., {"test" : "hi"} in Line $D3$). Then, GlassWing intends to identify the handler code block of the invocation (i.e., concrete implementation) on the opposite side, allowing to map the virtual methods and their concrete implementations on the other side.

Subsequently, we trace the cross-references until we reach the call site of a channel instance, flagged by the $setMethodCallHandler$ (Line $J4$), which is a callback function used to handle method calls. We then analyze the callback functions to extract logic that checks specific labels (e.g., Line $J5$), and match them with the recorded virtual methods to locate the corresponding code blocks that represent the concrete implementations of the virtual methods from the other side (e.g., Lines $J6 \sim J9$). Finally, we extract call sites on each side and map the virtual methods to their concrete implementations on the opposite side.

*2) **Cross-Language Method Invocation Construction**:* At this point, GlassWing has established the mapping $caller \rightarrow callee$ and $callee \rightarrow concret\ implementation\ of\ callee$ (i.e., handler code block). It seems that we can establish a mapping from $caller \rightarrow concrete\ implementation\ of\ callee$ using the two previous mappings; however, it only represents a logical abstraction and cannot be directly used to construct an invocation edge for static analysis. Without the concrete invocation edge, static analyzers still cannot detect or trace any Dart-DEX implicit invocations. Therefore, we need to build the concrete invocation edge between $caller$ and the concrete implementation of $callee$ (i.e., ② in Fig. 6), making cross-language invocation visible. Nevertheless, it is challenging to construct the concrete invocation edge because the $concrete\ implementation\ of\ callee$ exists as a handler code block rather than an actual method due to the implicit invocation feature of the channel. As a result, the $caller$ cannot directly reference the handler code block, hindering the construction of the concrete invocation edge. To overcome it, we refactor the handler code block into a Jimple method (i.e., handler method) to enable innovation.

Firstly, for the handler code block, we create a new empty Jimple method (i.e., handler method) within the class of the concrete implementation of $callee$ (e.g., Lines $J10 \sim J14$). This handler method is designed to accept a parameter (of map type), which encapsulates all the call arguments from the Dart call site (Line $J10$). The statements in the original handler code block that read the call arguments are rewritten to read from this map (Line $J11$). However, in Flutter apps, methods across different channels may use the same label, thus $caller$ may invoke these methods with the same name, leading to naming conflicts during invocation. To resolve it, GlassWing combines the channel name and the extracted label as the name of the handler method (Line $J10$). After that, we must also address how the handler returns data back to the *caller* through the Flutter channel's result API. Furthermore, the Flutter channel returns data to the *caller* through the result API (Lines $J8 \sim J9$) [63]. However, static analyzers struggle to accurately model the data flow of the result API due to its involvement with asynchronous callbacks [17]. To address this challenge, we convert the result API into a return statement (Lines $J13 \sim J14$), which directly returns data in the current execution stream. Finally, GlassWing sequentially inserts the remaining statements from the handler code block into the handler method. After refactoring the handler code block, we establish a direct invocation edge between AJ and FD using the handler method. Finally, GlassWing establishes the explicit cross-language invocation and a unified Jimple presentation to support subsequent analysis of the Flutter Android apps.

IV. Effectiveness Evaluation

To evaluate the effectiveness of GlassWing, we aim to conduct the experiments by answering the following research questions (RQs).

- **RQ1**: How well can GlassWing identify platform channels?
- **RQ2**: How well can GlassWing enhance Soot-based static analysis for Flutter Android applications?
- **RQ3**: How effective is GlassWing in detecting previously inaccessible potential sensitive data leakage issues in Flutter Android applications?
- **RQ4**: How efficient is GlassWing on real-world apps?

We ran all our experiments on an Ubuntu computer with two Intel Xeon Platinum 8378A CPUs and 1024GB RAM. For implementation, GlassWing uses APKtool [64] to decompile `.dex` file into Dalvik bytecode, and employs a modified Blutter [65] as part of the Dart machine code analyzer. In the pre-processing stage for FlowDroid, GlassWing outputs Dart Jimple methods, integrating them into the Soot scene [24]. Subsequently, we register these methods as additional entry points for FlowDroid, thereby extending its analysis scope.

A. RQ1: Platform Channel Identification

*1) **Setup**:* To evaluate the effectiveness of GlassWing in extracting cross-language invocation relations between Dart and Java/Kotlin, we utilize platform channels as the evaluation metric for RQ1, as platform channels provide the mechanism for implementing cross-language implicit invocations. To this end, we select apps from the officially recommended open-source Flutter examples [16] to construct a ground truth dataset. We evaluate the accuracy of GlassWing in identifying these platform channels on our benchmark dataset. Leveraging the proven capability, we then apply GlassWing to a real-world dataset to observe the number of platform channels identified, aiming to demonstrate GlassWing's performance in identifying cross-platform interactions in complex Flutter apps.

- ***Ground-Truth Benchmark:*** We select all 30 available open-source example apps, recommended by Flutter officials [16], to constitute our ground-truth benchmark. Given these apps' open-source nature, we conducted thorough manual inspections to precisely identify and enumerate each platform channel therein, and the result is cross-validated by three authors. These manually validated platform channels constitute the ground truth for evaluating the identification capability of GlassWing. Concurrently, to facilitate the analysis of sensitive data leaks for RQ3, we inject specific leaks into the identified platform channel within the benchmark dataset.
- ***Real-World Dataset:*** To evaluate the effectiveness of GlassWing in identifying platform channels in real-world apps, we collected all APKs from the AndroZoo repository [66] in 2024 and retained 11,537 apps after deduplicating. Given that these apps may include apps not developed by the

Table I: Ground-Truth Benchmark

ID	Name	#Ch	Ours	ID	Name	#Ch	Ours	ID	Name	#Ch	Ours
1	Telsavideo	1	1	11	Weight Tracker	1	1	21	Squawker	1	1
2	Amiibo	2	2	12	Light Wallet	1	1	22	Airdash	2	2
3	Authpass	1	1	13	Jidoujishou	10	10	23	Trireme	1	1
4	Immich	4	4	14	Meditation	1	1	24	Timy	5	5
5	Bluebubbles	1	1	15	Yubico	17	17	25	ShockAlarm	2	2
6	LibreTrack	2	2	16	ServerBox	2	1	26	Hacki	2	2
7	Osram	1	1	17	ESSE	1	1	27	Thingsboard	1	1
8	Kalium	1	1	18	Vidar	2	2	28	Lighthouse	3	3
9	Natrium	1	1	19	Voiceliner	2	1	29	Group-track	1	1
10	Blink Comp	2	2	20	Cake Wallet	5	4	30	Piggyvault	1	1
Total: 77				**TP:** 74				**FP:** 0		**FN:** 3	
Precision: 100%								**Recall:** 96.1%			

TP = True Positive, FP = False Positive, FN = False Negative

Flutter framework, affecting the analysis result, we further selected the Flutter apps by identifying the presence of the `libapp.so` file containing Dart code as the criterion. Finally, we obtained a dataset of 1,023 real-world Flutter apps for our analysis, including 72 popular Flutter apps found among the top 1,000 apps on Google Play.

*2) **Result**:* As shown in Table I, across our ground-truth benchmark apps, GlassWing achieved a recall of 96.1% (74/77) for platform channel identification. Upon manual investigation, all 74 channels reported by GlassWing were confirmed to be true positives, achieving a precision of 100%. This result demonstrates that GlassWing can accurately identify analyzable platform channels, which in turn indicates its capability to accurately extract implicit call relations in Flutter apps.

We further analyzed the platform channels (3) that GlassWing failed to identify (i.e., false negatives), revealing two reasons for these misses: (*i*) Unimplemented Handlers (2): The channels are declared in the Dart code, but their corresponding handler methods are not implemented on the DEX side. Since these channels represent ineffective execution paths, their exclusion does not influence GlassWing's analysis. (*ii*) Unused Channels (1): These channels are implemented on the DEX side but are never invoked from Dart code. Such channels constitute dead code from a runtime perspective, and thus, GlassWing excludes them from the set of analyzable channels.

On the real-world dataset, GlassWing identified 11,253 platform channels. To examine precision, we manually reviewed 100 platform channels (sampled to achieve a 95% confidence level with a margin of error of $\pm 10\%$, i.e., $\alpha = 0.05$, $E = 0.10$), balancing practical effort [67]. The result confirmed all reviewed platform channels were true positives, demonstrating GlassWing's capability to extract the implicit invocation relations in complex apps. The true positives results are attributed to GlassWing's design—specifically, its two-sided validation strategy, which confirms a platform channel by matching an invocation in Dart with its corresponding handler method in Java/Kotlin.

To further provide insights into GlassWing's performance in identifying cross-platform interaction mechanisms in complex Flutter applications, we analyzed different types of identified channels. According to our analysis, GlassWing identifies that each Flutter app uses an average of 8.4 *MethodChannels*, 0.3 *EventChannels*, and 2.3 *BasicChannels*. The *MethodChannel* is the most commonly used channel (accounting for 76.4%), serving as the primary interaction mechanism. It allows the Flutter framework to perform method calls and data exchanges with the native Android platform, which aligns with the development needs of Flutter apps that require frequent method calls [43]. Furthermore, we observe that the vast majority (65.7%) of platform channels in Flutter apps originate from third-party plugins. Developers rely on these plugins, which provide native features such as camera access and location services, to simplify development and save time. However, it highlights the need to consider the quality of third-party plugins used for platform channels, which may introduce bugs or security issues, posing a more severe impact than those caused by the developer's own code.

Answer to RQ1: On the ground-truth benchmark apps, GlassWing accurately identified all analyzable platform channels (i.e., implicit invocation relations). In complex real-world scenarios, GlassWing also demonstrated its effectiveness in identifying platform channels.

B. RQ2: Static Analysis Enhancement

*1) **Setup**:* To evaluate the effectiveness of GlassWing in enhancing static analysis for Flutter applications, we integrated GlassWing into the Soot framework and extended FlowDroid to analyze the real-world dataset from RQ1 (i.e., 1,023 Flutter apps, including the most popular 72 Flutter apps from the top 1,000 of Google Play). Since the static result produced by the Soot framework heavily relies on the availability of the Jimple code and the size of the call graph, followed by existing related work [25], [33], [68], we evaluate GlassWing from the two key aspects: **(*i*)** the usability of the Jimple code, measured by the number of *SootMethods* extracted and the lines of Jimple code (LOCs), which is evaluated by comparing Soot with/without GlassWing; **(*ii*)** the size of the call graph, measured by the number of nodes and edges, which is evaluated by comparing Flowdroid with/without GlassWing. Note that, due to the time and memory-intensive nature of FlowDroid analysis, following existing work [25], [33], [68], we set a timeout of 30 minutes per app for our experiments. The integration procedure is mentioned in our implementation of evaluation (Section IV).

Note that in this RQ, we do not set a ground-truth benchmark for evaluation due to the vast number of *SootMethods*, lines of Jimple code, and the nodes and edges in the call graph, making it extremely difficult to collect data manually, thus making it nearly impossible to design a reliable ground-truth benchmark. Additionally, existing related work [25], [33], [68] also did not provide a ground-truth benchmark for validation in this context, thus our setup aligns with standard practices.

*2) **Result**:* **Volume of Code and Methods:** As shown in Table II, incorporating GlassWing results in an increase in the number of *SootMethod* along with lines of Jimple code (LOCs) extracted during static analysis. Specifically, the number of methods extracted increased by 36.7%, from 37,715 to 51,556, and the LOCs increased by 141%, from 624,990 to 1,506,226.

Table II: Average #Methods and #LOCs with/-out GlassWing

Soot		Soot+GlassWing		Improvement	
#Methods	#LOCs	#Methods	#LOCs	#Added Methods	#Added LOCs
37,715	624,990	51,556	1,506,226	13,841 (**+36.7%**)	881,236 (**+141%**)

Table III: Average #Nodes and #Edges with/-out GlassWing

FlowDroid		FlowDroid+GlassWing		Improvement	
#Nodes	#Edges	#Nodes	#Edges	#Added Nodes	#Added Edges
5,327	21,757	6,830	27,887	1,503 (**+28.2%**)	6,130 (**+28.1%**)

This growth demonstrates that GlassWing effectively adds executable code that traditional static analyzers do not consider, enhancing the comprehensiveness of the analysis.

Size of the Call Graphs: Table III shows that FlowDroid enhanced with GlassWing increases the size of the graph, i.e., the number of nodes (i.e., methods) and edges (i.e., cross-language implicit invocation relations). According to Table III, the average number of nodes increases by 28.2%, from 5,327 to 6,830, and the number of edges increases by 28.1%, from 21,757 to 27,887. The increase indicates that GlassWing successfully identifies a large number of previously unreachable nodes and edges, enhancing code coverage. Note that the channel represents the invocation relations between the Flutter framework and the Android platform. However, due to the reuse of channels, the call relations discussed in RQ1 constitute only a part of the edges counted in RQ2. To validate the new graph additions, we manually inspected a random sample of 96 new nodes and corresponding edges ($\alpha = 0.05$, $E = 0.10$ [67]) via reverse engineering, confirming all of them to be true positives. Furthermore, we computed graph expansion metrics, which revealed that each newly introduced node is associated with 4.1 new edges on average. This result indicates that GlassWing adds meaningful and relevant call relations rather than isolated noise nodes. GlassWing effectively analyzes the executable code that traditional static analyzers overlook, avoiding treating these parts of the code as dead code [17], thereby enhancing the analysis. The additional code contributes to thoroughly identifying potential security vulnerabilities and overcoming performance bottlenecks.

Answer to RQ2: GlassWing enhances the static analysis of Flutter apps by discovering previously overlooked executable code. GlassWing increases the number of parsed methods by 36.7%, lines of Jimple code by 141%, call graph nodes by 28.2%, and edges by 28.1% compared to Soot and FlowDroid.

C. RQ3: Sensitive Data Leak Identification

*1) **Setup**:* To evaluate whether GlassWing can effectively identify potential sensitive data leaks in Flutter apps, we used the real-world datasets from RQ1. We also designed a systematic injection scheme for our ground-truth benchmark from RQ1, tailored to the unique features of Flutter apps. Specifically, we adopted the community-recognized SuSi [69] to categorize FlowDroid's default sources and sinks into seven source categories and three sink categories [67]. Subsequently, we injected data leaks into the corresponding DEX-side handler methods for each known platform channel (one per channel, 74 in total), thus constructing our ground-truth benchmark for RQ3. We verified that the injected leakage paths covered all of the aforementioned source/sink categories, which underpins the representativeness and systematicity. A complete list of all injected source-sink pairs is detailed on our website [39]. Based on the two datasets, we compared the number of potential leaks detected by FlowDroid with/without GlassWing and the timeout of FlowDroid is set to 30 minutes per app, as in RQ2.

*2) **Result**:* GlassWing with FlowDroid extracts all types of sensitive data leaks in the ground-truth benchmark apps, with no false positives and no false negatives. However, FlowDroid alone fails to detect leaks on our benchmark, as it is not designed to handle the cross-platform feature of Flutter apps.

On the real-world dataset, running FlowDroid alone detected only 3.1 potential sensitive data leaks on average, while the application of GlassWing resulted in an average detection of 8.4 potential sensitive data leaks, shown in Fig. 7. Our tool enhances the detection capabilities for potential sensitive data leaks in Flutter applications by bridging Dart-DEX interaction.

We manually verified the precision of the identified potential sensitive data leaks on a statistically significant random sample ($\alpha = 0.05$, $E = 0.10$ [67]) via reverse engineering. Our analysis revealed that 94.7% (90/95) of the data flows were true positives. Regarding the five false positives: (*i*) three are attributable to the inherent limitations of the underlying taint analysis capability of FlowDroid [18]. (*ii*) The remaining two false positives stem from the limitations in GlassWing's current handling of asynchronous operations. Specifically, our tool currently adopts a conservative assumption for asynchronous callbacks, which presumes that these callbacks are executed immediately. In scenarios involving complex asynchronous operations, this assumption can lead to infeasible data flows, a well-known challenge in static analysis [52].

GlassWing analyzes the data flow of Flutter apps to identify potential sensitive data leaks, further utilizing the SuSi [69] classification to understand the nature of these leaks. SuSi's classification helps us systematically categorize the sources and

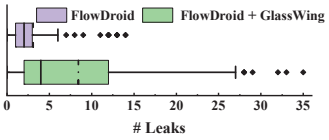


Fig. 7. Distribution of detected leaks

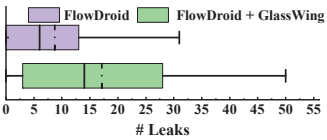


Fig. 8. Distribution of detected leaks in malware, see Section V-A

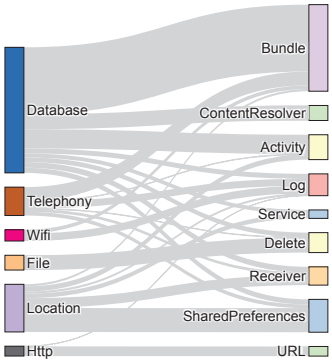


Fig. 9. Sankey diagram of data leaks

sinks of leaks by feature, ensuring more precise localization and data flow analysis. The most common source method was *<android.database.Cursor: java.lang.String getString(int)>*, occurring 838 times, suggesting that reading string data from databases is the most frequent data source in Flutter apps. Such prevalence is mainly because databases are central components for storing and retrieving data in modern application development. Flutter apps rely on dynamic data, being frequently updated from databases, as seen in apps like social media, shopping, and news reading. In a multi-device usage environment, Flutter apps might need to synchronize data states across different devices. Reading string data from databases is an important step in syncing and updating information [70]. The most common sink method was *<android.os.Bundle: void putString(java.lang.String,java.lang.String>*, occurring 285 times, typically used for passing string data between different parts or components of an application. Such high frequency is mainly because Flutter applications use Android's API to pass information through Intents carrying Bundles for functionality implementation [71].

To better understand and visualize how the potential sensitive data leaks discovered by GlassWing, we illustrated it using a Sankey diagram (shown in Fig. 9). The main sources of sensitive data leaks are Database, Location, and Telephony. Sensitive information primarily leaks into Bundle, SharedPreferences, and Activity. Indeed, our analysis shows the effectiveness of GlassWing in identifying the potential sensitive data leaks of complex Flutter apps.

Answer to RQ3: GlassWing can effectively help identify potential sensitive data leaks in Flutter apps. It detects all injected leaks in our ground-truth benchmarks and, on average, detects nearly 3X more potential leaks per app in the real-world dataset, revealing potential sensitive data leaks that traditional analyzers overlook.

D. RQ4: Efficiency

*1) **Setup**:* In the complex real-world scenario (of the 1,023 apps), we evaluated the efficiency (in terms of memory and time) of integrating GlassWing into FlowDroid, comparing the results with the baseline of running FlowDroid standalone.

*2) **Result**:* **Memory Overhead:** As illustrated in Fig. 10, GlassWing integrated with FlowDroid consumed an average of 2.1 GB of memory (median: 1.7 GB), versus 1.9 GB (median: 1.4 GB) for the FlowDroid baseline, representing a 17.4% memory overhead. **Time Overhead:** As shown in Fig. 11, GlassWing with FlowDroid took an average of 389.5s (median: 284.9s), compared to 204.3s (median: 139.9s) for the baseline.

We consider this overhead a reasonable and worthwhile trade-off. In return, GlassWing enhances the analysis scope for Flutter apps. Our dataset includes large-scale applications (e.g., Google Classroom, eBay Motors) from major companies (e.g., Google, eBay), and GlassWing proved capable of analyzing them. The median overheads exhibit a moderate increase (21.4% for memory, 103.6% for time), indicating the overhead of GlassWing is manageable for the majority of apps. The higher average values, driven by a few large-scale applications, highlight the scalability of GlassWing in its ability to process these complex targets.

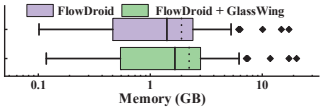


Fig. 10. Memory consumption per app (in GB).

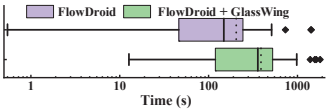


Fig. 11. Analysis time per app (in seconds).

Answer to RQ4: GlassWing extends the scope of static analysis for Flutter Android apps while introducing an acceptable performance overhead, demonstrating its reasonable scalability.

V. Discussion

In this section, we offer an enriched perspective on GlassWing by discussing its potential applications in Flutter malware analysis, limitations and threats to validity.

A. Potential Application

Table IV: Average #Nodes and #Edges with/-out GlassWing on 355 malwares

FlowDroid		FlowDroid + GlassWing		Improvement	
#Nodes	#Edges	#Nodes	#Edges	#Added Nodes	#Added Edges
7,317	37,738	7,733	39,577	416 (+5.7%)	1839 (+4.9%)

The immaturity of analysis techniques for the young Flutter framework presents an exploitable avenue for malware developers. By mid-2023, the Android Fluhorse [72] family began directly implementing the malicious payload within the Flutter codebase (called Flutter malware). It highlights a trend where malware leverages framework features to evade detection. Consequently, the in-depth analysis of Flutter malware emerges as a potential application.

We attempt to use GlassWing for the analysis of Flutter malware, laying the foundation for future automated identification of Flutter malware. Specifically, we downloaded all the APK malware samples from the VirusShare [73] database from 2020 to 2022 and identified 355 samples developed using the Flutter framework. We analyzed these samples using GlassWing, which showed an increase in the average number of nodes (+5.7%) and edges (+4.9%) per sample after enhancement through GlassWing (Table IV). As shown in Fig. 8, we also employed the GlassWing-enhanced FlowDroid to analyze the samples and found that, on average, it revealed an additional 9.6 potential leakage issues per sample. Additionally, we find that compared to popular benign apps in Fig. 7, a higher number of potential leaks are detected in malware, reflecting more potential issues within the malware, which aligns with its tendency to engage more extensively in the theft of sensitive user information. Through the analysis provided by GlassWing, we can gain a deeper understanding of the structure and behaviors of Flutter

malware in the future, thereby supporting the design of more effective defense strategies.

B. Limitations and Threats to Validity

1) ***Limitations****:* Limitations of GlassWing stem from the following aspects, which also highlight potential avenues for future research.

• Advanced Dart Features. GlassWing's design prioritizes addressing the foundational challenge of analyzing cross-language calls in Flutter apps. Consequently, support for certain advanced Dart language features remains limited. Specifically, GlassWing simplifies Dart's asynchronous streams to synchronous ones. The modeling of custom widget state from internal mechanisms of the UI framework and exception handling is currently limited. To accommodate Dart's dynamic nature, GlassWing employs type inference to mitigate its impact. In its current implementation, 51.3% of Dart types are resolved to `java.lang.Object`.

• Lifecycle Modeling. GlassWing's analysis focuses on methods pertinent to platform channel interactions and does not currently model the component lifecycle. Consistent with prior work [17], [25], [33], we assume that platform channels are reachable throughout a component's lifetime, which may cause the static analyzer to miss lifecycle-dependent entry points.

• Code Obfuscation. Flutter current default obfuscation does not alter platform channel name strings, thus leaving GlassWing name-based identification method unaffected. However, when facing more advanced obfuscation techniques, GlassWing effectiveness may be impacted. For apps employing obfuscation beyond the default settings, we attempt to use existing deobfuscation techniques [74]–[76] to address them.

• Platform Plugin Support. Currently, GlassWing only supports the analysis of Flutter Android apps, but the methodology is also applicable to other platforms. We plan to extend our support to more platforms in future work.

These limitations exist as fully supporting a new language is a major undertaking, and many are known hard problems in static analysis [77]. We plan to explore and address these challenges, continuously maintaining GlassWing to ensure support for new features and adaptation to the framework evolution.

2) ***Threats to Validity****:* (*i*) Evaluation on Real-World Apps. For closed-source applications, obtaining complete ground truth for binary-disassembly platform channels and data leaks is inherently unattainable [77], which limits our ability to conduct recall analysis. To mitigate this threat, we instead evaluate the recall of GlassWing using official open-source Flutter apps as ground truth. For the precision analysis, we manually validate the platform channels and leaks identified in our real-world dataset. However, given the manual effort constraints, we employ a random sample ($\alpha = 0.05$, $E = 0.10$ [67]) to select a statistically representative subset for this verification. (*ii*) Manually Injected Leaks. The manually injected leaks in our evaluation may pose a threat to the validity of GlassWing's leak-revealing capability. To mitigate this threat, we designed a systematic injection scheme, guided by the community-recognized SuSi [69] categories and designed to cover all leak types defined by FlowDroid's default sources and sinks.

VI. Related Work

A. Cross-Language Analysis of Android Apps

Current cross-language static analysis efforts target specific cross-platform frameworks (i.e., React Native) [25] and specific interfaces (i.e., Java Native Interface (JNI)) [68], [78]–[83] to identify interactions between DEX $\leftrightarrow$ cross-platform framework code (JavaScript) and DEX $\leftrightarrow$ native code (C/C++), converting components written in other languages into Jimple code. Lee et al. [27] proposed the HybriDroid framework to detect errors and information leaks in hybrid applications by analyzing communication between Java and JavaScript. More research has focused on cross-language analysis between Java/Kotlin and native code (C/C++). Various works in Android malware detection [79], dynamic taint analysis [80]–[82], and sensitive data leak detection [68], [78] emphasize Native Development Kit (NDK) analysis. However, they all overlook the prevalence of cross-platform frameworks in current mobile app development. To date, only a few efforts have provided static analysis support for specific cross-platform frameworks (i.e., React Native [25]). Nevertheless, their methodologies are not directly applicable to Flutter apps (Dart, AOT, implicit call). To tackle the technical challenges, we introduce data-flow-oriented summarization to uncover the implicit cross-language calls and link them through our channel linker.

B. Static Analysis of Android Apps

Static analysis is an important means of evaluating Android applications [18]–[20], [84]–[95]. Since the development of Soot [24], it has become one of the most popular frameworks for analyzing Android applications [96], with several tool prototypes relying on Soot to perform their respective static analyses. Many studies focus on detecting sensitive data leaks [18]–[20], [97], identifying malicious behavior [89]–[92], and inter-component communication vulnerabilities [93]–[95], [98]. However, existing research primarily focuses on Android applications based on Java/Kotlin, overlooking other programming languages that are increasingly popular in Android development, such as Dart, used by the Flutter framework. This oversight has led to significant effectiveness challenges when dealing with modern Android applications.

VII. Conclusion

In this paper, we introduce GlassWing, the first approach for conducting a tailored static analysis of Flutter Android apps, which enhances cross-language communication between the native Android platform and the Flutter framework. Our work lays the foundation for future developments to advance the static analysis of Flutter apps.

Acknowledgments

This work was supported by the National Key R&D Program of China (Grant No.2024YFE0203800), the National Natural Science Foundation of China (No. 62472309), and the grant from Zhongguancun Academy (Grant No. 20240302).

REFERENCES

[1] Haonan Chen, Daihang Chen, Yonghui Liu, Xiaoyu Sun, and Li Li. Are your android app analyzers still relevant? In *Proceedings of the IEEE/ACM 11th International Conference on Mobile Software Engineering and Systems*, pages 69–73, 2024.

[2] Android. https://www.android.com/, 2024.

[3] ios 18. https://www.apple.com/ios/ios-18/, 2024.

[4] Huawei Technologies Co., Ltd. Harmonyos. https://www.harmonyos.com/en/, 2024. Accessed: 2024-10-10.

[5] Manuel Palmieri, Inderjeet Singh, and Antonio Cicchetti. Comparison of cross-platform mobile development tools. In *ICIN '12*, pages 179–186. IEEE, 2012.

[6] Matias Martinez and Sylvain Lecomte. Towards the quality improvement of cross-platform mobile applications. In *MOBILESoft '17*, pages 184–188. IEEE, 2017.

[7] Flutter. https://github.com/flutter/flutter, 2024.

[8] React native. https://github.com/facebook/react-native, 2024.

[9] Weex. https://github.com/alibaba/weex, 2024.

[10] Matteo Ciman and Ombretta Gaggi. An empirical analysis of energy consumption of cross-platform frameworks for mobile development. *Pervasive and Mobile Computing*, 39:214–230, 2017.

[11] Douglas Crockford. *JavaScript: The Good Parts: The Good Parts*. " O'Reilly Media, Inc.", 2008.

[12] Andreas Sommer and Stephan Krusche. Evaluation of cross-platform frameworks for mobile applications. 2013.

[13] PLAY Store. Google play store, 2024.

[14] Sensor Tower, Inc. Sensor tower: App store optimization and mobile app market intelligence.

[15] Gustav Tollin and Lidekrans Marcus. React native vs. flutter: A performance comparison between cross-platform mobile application development frameworks, 2023.

[16] Flutter showcase. https://flutter.dev/showcase.

[17] Jordan Samhi, René Just, Tegawendé F Bissyandé, Michael D Ernst, and Jacques Klein. Call graph soundness in android static analysis. In *ISSTA '24*, pages 945–957, 2024.

[18] 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. *ACM sigplan notices*, 49(6):259–269, 2014.

[19] 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. IEEE, 2015.

[20] Jordan Samhi, Alexandre Bartel, Tegawendé F Bissyandé, and Jacques Klein. Raicc: Revealing atypical inter-component communication in android apps. In *ICSE '21*, pages 1398–1409. IEEE, 2021.

[21] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.

[22] Gabor Paller. Understanding the dalvik bytecode with the dedexer tool, 2009.

[23] Michael Otten and Milos G Pacak. Intermediate languages for automatic language processing. *SEN Report Series Software Engineering*, 2:105–118, 1971.

[24] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010.

[25] Yonghui Liu, Xiao Chen, Pei Liu, John Grundy, Chunyang Chen, and Li Li. Reunify: A step towards whole program analysis for react native android apps. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1390–1402. IEEE, 2023.

[26] YONGHUI LIU, XIAO CHEN, PEI LIU, JORDAN SAMHI, JOHN GRUNDY, CHUNYANG CHEN, and LI LI. Demystifying react native android apps for static analysis. *ACM Trans. Softw. Eng. Methodol*, 1(1), 2024.

[27] Sungho Lee, Julian Dolby, and Sukyoung Ryu. Hybridroid: static analysis framework for android hybrid applications. In *ASE '16*, pages 250–261, 2016.

[28] Peter Knaggs and Stephen Welsh. *ARM: Assembly Language Programming*. Bournemouth University, School of Design, Engineering, and Computing, 2004.

[29] Flutter. Platform-specific code — flutter. https://docs.flutter.dev/platform-integration/platform-channels, 2023. Accessed: 2024-10-20.

[30] Eric Masiello and Jacob Friedmann. *Mastering React Native*. Packt Publishing Ltd, 2017.

[31] Akshat Paul, Abhishek Nalwaya, Akshat Paul, and Abhishek Nalwaya. Native bridging in react native. *React Native for Mobile Development: Harness the Power of React Native to Create Stunning iOS and Android Applications*, pages 165–186, 2019.

[32] Tamara Ranisavljević, Darjan Karabašević, Miodrag Brzaković, Gabrijela Popović, and Dragiša Stanujkić. React native: A brief introduction to modern cross-platform mobile application development. *Quaestus*, (21):120–136, 2022.

[33] Jikai Wang and Haoyu Wang. Nativesummary: Summarizing native binary code for inter-language static analysis of android apps. In *ISSTA '24*, pages 971–982, 2024.

[34] Chris Eagle. *The IDA pro book*. no starch press, 2011.

[35] Chris Eagle and Kara Nance. *The Ghidra Book: The Definitive Guide*. no starch press, 2020.

[36] Radare Team. Radare2 github repository, 2017.

[37] Boris Batteux. The current state & future of reverse engineering flutter™ apps. https://www.guardsquare.com/blog/current-state-and-future-of-reversing-flutter-apps, June 2022.

[38] Wikipedia contributors. Greta oto — wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Greta_oto, 2024. Accessed: 2024-10-31.

[39] papersubmit anonymous. Glasswing: A step towards whole program analysis for flutter android apps. https://github.com/glasswing-ase25/GlassWing, 2025.

[40] Ly Hoang. State management analyses of the flutter application. 2019.

[41] UUAS Swathiga, P Vinodhini, and V Sasikala. An interpretation of dart programming language. *DRSR Journal*, 11(3):144–149, 2021.

[42] PlugFox. Introduction to dart vm. https://plugfox.dev/introduction-to-dart-vm/, 2023.

[43] Lukas Dagne. Flutter for cross-platform app and sdk development. 2019.

[44] Michael Gonsalves. Evaluating the mobile development frameworks apache cordova and flutter and their impact on the development process and application characteristics. 2019.

[45] YESIM UZUN. Integration of flutter framework in real-life applications: Technical and development practices.

[46] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *ISSTA '16*, pages 318–329, 2016.

[47] Chaoran Li, Xiao Chen, Ruoxi Sun, Jason Xue, Sheng Wen, Muhammad Ejaz Ahmed, Seyit Camtepe, and Yang Xiang. Natidroid: Cross-language android permission specification. *arXiv preprint arXiv:2111.08217*, 2021.

[48] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. *arXiv preprint arXiv:1612.04433*, 2016.

[49] Daoyuan Wu, Debin Gao, Robert H Deng, and Chang Rocky KC. When program analysis meets bytecode search: Targeted and efficient inter-procedural analysis of modern android apps in backdroid. In *DSN '21*, pages 543–554. IEEE, 2021.

[50] Kadiray Karakaya and Eric Bodden. Sootfx: A static code feature extraction tool for java and android. In *SCAM '21*, pages 181–186. IEEE, 2021.

[51] Tarek Mahmud, Meiru Che, and Guowei Yang. Acid: an api compatibility issue detector for android apps. In *ICSE '22*, pages 1–5, 2022.

[52] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *ICSE '15*, volume 1, pages 89–99. IEEE, 2015.

[53] Felix Pauck and Heike Wehrheim. Jicer: Simplifying cooperative android app analysis tasks. In *SCAM '21*, pages 187–197. IEEE, 2021.

[54] Ying Wang, Yibo Wang, Sinan Wang, Yepang Liu, Chang Xu, Shing-Chi Cheung, Hai Yu, and Zhiliang Zhu. Runtime permission issues in android apps: Taxonomy, practices, and ways forward. *TSE*, 49(1):185–210, 2022.

[55] Jordan Samhi, Li Li, Tegawendé F Bissyandé, and Jacques Klein. Difuzer: Uncovering suspicious hidden sensitive operations in android apps. In *ICSE '22*, pages 723–735, 2022.

[56] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, and Geguang Pu. Efficiently manifesting asynchronous programming errors in android apps. In *ASE '18*, pages 486–497, 2018.

[57] Sen Chen, Lingling Fan, Chunyang Chen, and Yang Liu. Automatically distilling storyboard with rich features for android apps. *TSE*, 49(2):667–683, 2022.

[58] Sen Yang, Sen Chen, Lingling Fan, Sihan Xu, Zhanwei Hui, and Song Huang. Compatibility issue detection for android apps based on path-sensitive semantic analysis. In *ICSE '23*, pages 257–269. IEEE, 2023.

[59] Motors for eBay. Motors cars for ebay - android app. https://apkpure.net/motors-cars-for-ebay/com.motorsforebay.carsauctionapp.

[60] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

[61] Bowen Zhang, Wei Chen, Hung-Chun Chiu, and Charles Zhang. Unveiling the power of intermediate representations for static analysis: A survey. *arXiv preprint arXiv:2405.12841*, 2024.

[62] Tobias Roth, Julius Näumann, Dominik Helm, Sven Keidel, and Mira Mezini. Axa: Cross-language analysis through integration of single-language analyses. 2024.

[63] Aakanksha Tashildar, Nisha Shah, Rushabh Gala, Trishul Giri, and Pranali Chavhan. Application development using flutter. *International Research Journal of Modernization in Engineering Technology and Science*, 2(8):1262–1266, 2020.

[64] Apktool. https://apktool.org/, Oct 2024. Accessed: 2024-10-23.

[65] Blutter. https://github.com/worawit/blutter, 2024.

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

[67] Ronán M Conroy et al. The rcsi sample size handbook. *A rough guide*, pages 59–61, 2016.

[68] 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. In *ICSE '22*, pages 1232–1244, 2022.

[69] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Susi: A tool for the fully automated classification and categorization of android sources and sinks. *University of Darmstadt, Tech. Rep. TUDCS-2013-0114*, 2013.

[70] Roee Hay, Omer Tripp, and Marco Pistoia. Dynamic detection of inter-application communication vulnerabilities in android. In *ISSTA '15*, pages 118–128, 2015.

[71] Hao Zhou, Xiapu Luo, Haoyu Wang, and Haipeng Cai. Uncovering intent based leak of sensitive data in android framework. In *CCS '22*, pages 3239–3252, 2022.

[72] Flutter-based android malware. https://filestore.fortinet.com/fortiguard/research/flutter.pdf, 2023.

[73] Virusshare. https://virusshare.com/, 2024.

[74] jadx. https://github.com/skylot/jadx, 2024.

[75] Heejun Jang, Beomjin Jin, Sangwon Hyun, and Hyoungshick Kim. Kerberoid: A practical android app decompilation system with multiple decompilers. In *CCS '19*, pages 2557–2559, 2019.

[76] Geunha You, Gyoosik Kim, Sangchul Han, Minkyu Park, and Seong-Je Cho. Deoptfuscator: Defeating advanced control-flow obfuscation using android runtime (art). *IEEE Access*, 10:61426–61440, 2022.

[77] Chengbin Pang, Tiantai Zhang, Ruotong Yu, Bing Mao, and Jun Xu. Ground truth for binary disassembly is not easy. In *USENIX Security '22*, pages 2479–2495, 2022.

[78] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code. In *CCS '18*, pages 1137–1150, 2018.

[79] Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. Droidnative: Automating and optimizing detection of android native code malware variants. *computers & security*, 65:230–246, 2017.

[80] Lei Xue, Chenxiong Qian, Hao Zhou, Xiapu Luo, Yajin Zhou, Yuru Shao, and Alvin TS Chan. Ndroid: Toward tracking information flows across multiple android contexts. *TIFS*, 14(3):814–828, 2018.

[81] Mingshen Sun, Tao Wei, and John CS Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *CCS '16*, pages 331–342, 2016.

[82] Wen Li, Jiang Ming, Xiapu Luo, and Haipeng Cai. {PolyCruise}: A {Cross-Language} dynamic information flow analysis. In *USENIX Security '22)*, pages 2513–2530, 2022.

[83] Chenxi Zhang, Yufei Liang, Tian Tan, Chang Xu, Shuangxiang Kan, Yulei Sui, and Yue Li. Interactive cross-language pointer analysis for resolving native code in java programs. In *ICSE '25*, pages 612–612. IEEE Computer Society, 2025.

[84] Xiangyu Zhang, Lingling Fan, Sen Chen, Yucheng Su, and Boyuan Li. Scene-driven exploration and gui modeling for android apps. In *ASE '23*, pages 1251–1262. IEEE, 2023.

[85] Xian Zhan, Lingling Fan, Sen Chen, Feng We, Tianming Liu, Xiapu Luo, and Yang Liu. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications. In *ICSE '21*, pages 1695–1707. IEEE, 2021.

[86] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. Storydroid: Automated generation of storyboard for android apps. In *ICSE '19*, pages 596–607. IEEE, 2019.

[87] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. Large-scale analysis of framework-specific exceptions in android apps. In *ICSE '18*, pages 408–419, 2018.

[88] Xian Zhan, Lingling Fan, Tianming Liu, Sen Chen, Li Li, Haoyu Wang, Yifei Xu, Xiapu Luo, and Yang Liu. Automated third-party library detection for android applications: Are we there yet? In *ASE '20*, pages 919–930, 2020.

[89] Bozhi Wu, Sen Chen, Cuiyun Gao, Lingling Fan, Yang Liu, Weiping Wen, and Michael R Lyu. Why an android app is classified as malware: Toward malware classification interpretation. *TOSEM*, 30(2):1–29, 2021.

[90] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *MobiSys '12*, pages 281–294, 2012.

[91] Li Li, Kevin Allix, Daoyuan Li, Alexandre Bartel, Tegawendé F Bissyandé, and Jacques Klein. Potential component leaks in android apps: An investigation into a new feature set for malware detection. In *QRS '15*, pages 195–200. IEEE, 2015.

[92] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *2012 Seventh Asia joint conference on information security*, pages 62–69. IEEE, 2012.

[93] Joshua Garcia, Mahmoud Hammad, Negar Ghorbani, and Sam Malek. Automatic generation of inter-component communication exploits for android applications. In *ESEC/FSE '17*, pages 661–671, 2017.

[94] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective {Inter-Component} communication mapping in android: An essential step towards holistic security analysis. In *USENIX Security '13*, pages 543–558, 2013.

[95] Alireza Sadeghi, Hamid Bagheri, and Sam Malek. Analysis of android inter-app security vulnerabilities using covert. In *ICSE '15*, volume 2, pages 725–728. IEEE, 2015.

[96] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. The soot-based toolchain for analyzing android apps. In *MOBILESoft '17*, pages 13–24. IEEE, 2017.

[97] Sen Chen, Lingling Fan, Guozhu Meng, Ting Su, Minhui Xue, Yinxing Xue, Yang Liu, and Lihua Xu. An empirical assessment of security risks of global android banking apps. In *ICSE '20*, pages 1310–1322, 2020.

[98] Andrea Romdhana, Alessio Merlo, Mariano Ceccato, and Paolo Tonella. Assessing the security of inter-app communications in android through reinforcement learning. *Computers & Security*, 131:103311, 2023.