Fine-Grained Privacy Leakage Detection in OpenHarmony Apps

AOHAN MEI*, Fudan University, China
GUANGLIANG YANG*, Fudan University, China
XINMING GUO, Fudan University, China
YI WANG, Fudan University, China
FUAN GUI, Fudan University, China
MIN YANG, Fudan University and Shanghai Pudong Research Institute of Cryptology, China

In recent years, the distributed operating system OpenHarmony has gained significant popularity. As the number of OpenHarmony apps grows rapidly, privacy abuse and data leakage have emerged as critical concerns. However, the untyped and highly flexible nature of Ark bytecode poses substantial challenges. We propose HScope, a novel fine-grained program analysis framework designed to directly analyze OpenHarmony app bytecode and identify privacy risks. HScope employs abstract interpretation to model the dynamic behaviors of OpenHarmony apps, enabling precise resolution of indirect calls and the complex inter-component communication mechanisms. We evaluate HScope on a dataset of 300 real-world OpenHarmony apps. The results demonstrate that HScope is both effective and comprehensive, successfully uncovering 39 previously unknown privacy issues (corresponding to 27 apps). These findings highlight HScope's potential as a practical and scalable solution for securing the evolving OpenHarmony ecosystem.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; • **Security and privacy** → **Privacy protections**; *Software security engineering*.

Additional Key Words and Phrases: OpenHarmony Security, Ark-bytecode Analysis, Privacy Leakage Detection, Taint Analysis, Inter-Component Communication Analysis

ACM Reference Format:
Aohan Mei, Guangliang Yang, Xinming Guo, Yi Wang, Fuan Gui, and Min Yang. 2026. Fine-Grained Privacy Leakage Detection in OpenHarmony Apps. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA205 (October 2026), 22 pages. https://doi.org/10.1145/3832296

1 Introduction

In recent years, the distributed operating system OpenHarmony has become increasingly popular, and has gained widespread adoption across critical domains, including personal computers, Internet of Things (IoT), wearables, smart homes, mobile devices, and automotive systems. As of the second quarter of 2025, OpenHarmony holds over 4% global smartphone OS market share and has become the third largest smartphone OS worldwide [23]. Parallel to this, the OpenHarmony app ecosystem is expanding rapidly. By mid-2025, the number of native OpenHarmony apps had surged to over 30,000

*These authors contributed equally to this work.

Authors' Contact Information: Aohan Mei, Fudan University, Shanghai, China, 24212010025@m.fudan.edu.cn; Guangliang Yang, Fudan University, Shanghai, China, yanggl@fudan.edu.cn; Xinming Guo, Fudan University, Shanghai, China, 25213050169@m.fudan.edu.cn; Yi Wang, Fudan University, Shanghai, China, 25213050388@m.fudan.edu.cn; Fuan Gui, Fudan University, Shanghai, China, 24300240005@m.fudan.edu.cn; Min Yang, Fudan University and Shanghai Pudong Research Institute of Cryptology, Shanghai, China, m_yang@fudan.edu.cn.

ACM 2994-970X/2026/10-ARTISSTA205
https://doi.org/10.1145/3832296

[25]. This growth highlights OpenHarmony's pivotal role in cross-device computing, presenting both new opportunities and significant responsibilities.

The proliferation of apps on this new frontier introduces significant privacy and security challenges. Past studies [9][26][8] show that mobile apps routinely access sensitive sources such as location, contacts, sensors, cameras, and credentials. As millions of users adopt OpenHarmony devices and entrust their personal data to apps, the potential for sensitive information misuse escalates dramatically. The sheer volume and rapid growth of OpenHarmony apps make manual security audits infeasible, as such processes are often time-consuming, error-prone, and inherently unscalable.

Automated analysis techniques are therefore essential for identifying privacy risks at scale. For traditional platforms like Android, static taint analysis tools such as FlowDroid [1] have proven effective by tracking sensitive data flows without app source code (i.e. at the bytecode level). However, analyzing OpenHarmony apps presents unique and formidable technical challenges that render existing static analysis techniques largely ineffective. OpenHarmony apps are predominantly written in ArkTS (a TypeScript-based language) and compiled into a proprietary Ark bytecode. This bytecode is designed as an untyped format that embodies the dynamic nature of JavaScript, supporting flexible runtime behaviors and declarative UI development akin to web UI [19, 30]. While this design enhances developer productivity and user experience, it fundamentally complicates systematic privacy risk detection. We distill these obstacles into three major challenges.

- *Challenge#1: Untyped and Highly Dynamic Bytecode.* Traditional static taint analyses depend critically on type information (including function signatures) to construct precise Interprocedural Control Flow Graphs (ICFGs) and to perform data flow reasoning. In Ark bytecode, type information is largely absent and resolved at runtime, while object shapes can be modified dynamically.
  In addition, bytecode operations are quite different from source-level instructions. For example, all variables are anonymized. They cannot be directly accessed using naming like source code. Instead, they can only be accessed through an indirect addressing model [29]. Such an addressing model is implemented in lexical environments. These introduce difficulties in identifying and tracking variables.
  This semantic gap can result in substantial false negatives (where unauthorized data exfiltration goes undetected) or crippling false positives (flooding analysts with irrelevant results).
- *Challenge#2: Indirect and Dynamic Invocation Patterns.* OpenHarmony apps heavily rely on event-driven programming, where callbacks (e.g., UI event handlers and async/await continuations) are registered and invoked indirectly. Unlike Java's statically resolvable interfaces, these invocations often lack compile-time signatures. Simply modeling them as 'unknown targets' inevitably compromises precision.
- *Challenge#3: Inter-Component Data and Control Flow via Shared Object References.* While regular parameter passing is supported in OpenHarmony for inter-component communication (ICC), OpenHarmony permits components to exchange *direct references* to live objects and functions. As a result, a receiver cannot only read but also mutate the sender's objects or invoke callback functions that execute within the sender's context. This breaks two conventional assumptions in static taint analysis: (1) that data flow and control flow can be modeled independently, and (2) that component boundaries enforce isolation of variables and execution contexts. These tightly coupled data-control channels across components pose significant challenges.

In this paper, we present HScope, a fine-grained static analysis framework for detecting potential privacy leaks in OpenHarmony apps. HScope operates directly on Ark bytecode and does not require type annotations. HScope performs abstract interpretation to construct semantic representation of the target program, identifying program identifiers (e.g., variables, classes, and methods) and computing their associated abstract memory objects. Instead of relying on pre-built ICFG, which is difficult to construct under OpenHarmony's dynamic typing and runtime dispatch mechanisms, HScope resolves indirect calls using abstract memory object shapes and builds call graph on the fly. On top of this semantic foundation, HScope incorporates framework-aware modeling (including ICC) to capture implicit control flows and inter-component data dependencies. Finally, HScope conducts fine-grained taint propagation based on the derived semantic information, enabling accurate detection of privacy leakage risks.

To demonstrate the effectiveness and scalability of HScope, we evaluate it on a comprehensive dataset of 300 real-world OpenHarmony apps. Our experiments show HScope successfully identified 48 potential privacy issues, 39 of which were manually confirmed as previously unknown leaks across 27 distinct applications. We have manually confirmed and reported them to the developers. These leaks include the unauthorized transmission of sensitive data such as precise location, device identifiers, phone numbers, and contact list information. Furthermore, when compared to the source-code level analysis tool ArkAnalyzer [5], HScope enables analysis of apps without access to source code. HScope detected 12 additional leaks that ArkAnalyzer failed to report, highlighting its broader coverage and enhanced capability in identifying real-world privacy risks. These results underscore HScope's effectiveness, practicality, and comprehensiveness in analyzing OpenHarmony apps.

To sum up, we make the following contributions:

- We propose a novel fine-grained program analysis methodology for Ark bytecode without relying on type annotations.
- We design and implement HScope that embodies our methodology to detect privacy leaks in real-world OpenHarmony binaries.
- Our evaluation demonstrates HScope's effectiveness, precision, and scalability in discovering previously unknown privacy risks.

2 Background

In this section, we first introduce necessary background information about the OpenHarmony app model. Then, we present a real-world example to illustrate the analysis challenges.

2.1 OpenHarmony App Programming Model

OpenHarmony is an emerging mobile operating system with a rapidly growing application ecosystem. Unlike Android and iOS, OpenHarmony adopts a new programming language and runtime stack, introducing distinct design choices in language semantics, UI frameworks, and bytecode execution. These differences fundamentally change the structure and behavior of applications, raising new challenges for program analysis and security inspection.

App UI. OpenHarmony apps are primarily developed using ArkTS, a TypeScript-derived language. UI development is facilitated by ArkUI, a declarative UI framework tightly integrated with ArkTS, enabling developers to define UI structure and behavior in a unified and reactive programming model.

The architecture of OpenHarmony apps is modular and component-based. Unlike the Activity-Fragment model in Android, OpenHarmony introduces Ability as the fundamental execution and UI abstraction, decoupling lifecycle management from traditional component hierarchies.

The fundamental unit of UI functionality is Ability. An Ability typically consists of multiple pages/screens, while a page can contain nested UI components (e.g., `Text`, `Image`, and `Button`). These components expose configurable attributes (e.g., `width` and `height`) and event handlers (e.g., `onClick`) to customize appearance and interactivity. The following example illustrates a simple page with a `Text` component whose content updates upon a button click:

```
@Entry @Component
struct MainPage {
    @State message: string = 'Hello'
    build() {
        Column() {
            Text(this.message).fontSize(50)
            Button('Click Me').onClick(() => {
                this.message = 'Clicked!'
            })
        }}}
```

OpenHarmony's declarative model integrates UI structure and behavior within the same code context, resembling modern reactive frameworks but being deeply embedded into the language and runtime. As shown above, event handlers (e.g., `onClick`) are defined inline within the UI declaration, enabling dynamic binding and reactivity. While this integration enhances developer productivity and enables fine-grained control over UI updates, it also complicates static analysis. Data flows and control dependencies are embedded within UI expressions, lacking explicit method boundaries or call sites that traditional analyzers rely on.

Ark Bytecode. In OpenHarmony, ArkTS source code is compiled by the ArkCompiler to Ark bytecode, which is executed by the Ark runtime. Unlike JVM bytecode or Android bytecode, Ark bytecode is intentionally designed to be largely untyped. High-level type annotations and syntactic abstractions from ArkTS are mostly erased during compilation, leaving only low-level instructions and dynamically resolved object structures. Crucially, Ark bytecode is *untyped*. Type annotations and high-level syntactic structures from ArkTS are largely erased during compilation. Object shapes, property accesses, and method dispatches are resolved dynamically at runtime.

Event-based Interactions. OpenHarmony apps are fundamentally event-driven, with control flow orchestrated by the framework runtime rather than explicit program structure. User interactions (e.g., taps and gestures), component lifecycle events (e.g., `onAppear()` and `onPageShow()`), and system callbacks (e.g., sensor updates and network responses) are all delivered through registered handlers, typically using `on*`-prefixed methods or lambda expressions. For example, '`Button().onClick(() => {...})`' registers a closure that executes upon user interaction. These patterns result in pervasive use of higher-order functions and indirect calls. Since event handlers are often defined as anonymous functions or bound methods, their targets cannot be resolved statically without modeling closure capture, 'this' binding, and dynamic dispatch.

Inter-Component Communication. OpenHarmony provides multiple mechanisms for data exchange and navigation between components.

- *Ability-Based Navigation:* The `startAbility()` API allows one Ability to launch another, passing data via a structure analogous to Android's Intent.
- *Navigation Routers:* At the UI framework level, the `router.pushUrl()` API enables page transitions with explicit parameters. The receiving page accesses parameters via `router.getParams()`, whose structure is dynamically typed.
- *UI State Variable Sharing:* OpenHarmony supports fine-grained UI state synchronization between components using decorators like `@Prop`, `@Link`, `@Provide`, `@Consume`, `@Param`, and `@Event`. Specifically, `@Prop` enables unidirectional data synchronization from parent to child

components; `@Link` creates a bidirectional binding between parent and child components; `@Provide` and `@Consume` enable hierarchical state propagation. `@Param` denotes state passed into a component from its parent, supporting unidirectional data synchronization. `@Event` decorates a callback method that a child component uses to output events, primarily working with `@Param` to request the parent component to update the data source, thereby achieving a bidirectional synchronization effect. Mutations to shared UI state variables trigger automatic UI updates, creating implicit data flows that bypass explicit method calls.

The first two channels (i.e., ability-based navigation and navigation routers) perform similarly to Android Intent. Parameters are limited to primitives and serialized objects, ensuring no live references or direct executable callbacks cross Ability boundaries. In other words, only serialized copies of data objects are transmitted. Thus, the boundary between different components is respected. However, the last UI state variable sharing introduces uniqueness. Unlike traditional mobile platforms like Android, this state variable sharing mechanism allows components to exchange not only structured data, but also *function references* (e.g., callbacks) and *live object instances*. This means that (1) when the data shared in inter-component communication is manipulated by the sender or the receiver component, the other component is notified of the change in real time; and (2) when the shared function is executed, the execution context is the sender's. Consequently, these enable implicit data flow and the delegation of control flow across component boundaries.

Together, these language, runtime, and framework characteristics fundamentally differentiate OpenHarmony apps from traditional mobile apps and render existing static analysis techniques inadequate without substantial adaptation.

2.2 Real-World Example

To demonstrate the programming model of OpenHarmony apps, we work through a real-world example. This app (anonymized package name: 'com.conrad.tixxx') contains a previously unknown privacy issue found by HScope. It is an open-source app from GitHub. Figure 1 provides the simplified code snippet compromising user privacy, i.e., OAID (Open Anonymous Device Identifier). OAID is an industry-standard and non-permanent device identity to replace privacy-sensitive persistent hardware identifiers like IMEI or serial number. It can enable compliant advertising attribution and conversion analysis, while preserving user privacy. This privacy information access should be well protected and require users' consent, but this app transmits it silently in the background.

When the app starts, the component `C1` is initialized and opened. During the dynamic rendering of `C1`, a nested component `C2` is created at Line 10. The internal variable `callback` of `C2` is initialized and directly points to the anonymous function specified by `C1` at Line 10 (i.e., `(msg)=>{}`), while the state variable `message` of `C2` is linked to the variable in `C1` with the same name. When `message` is changed in `C1` or `C2`, the other component obtains `message`'s newest value in real time. In `C1`, the `message` is changed at Line 14 and stores the sensitive information OAID. Then, once a button is clicked in `C2` at Line 24, the function pointed to by `callback` is accessed and called at Line 26. Its parameter just holds the value of `message`, i.e., OAID. The anonymous function defined at Line 10 is called and then the function `request()` at Line 4 is further executed. OAID is passed to this essential function `http.post()`, leading to the leakage of OAID.

From the example, we can learn that it is critical to handle the indirect calls (i.e., '`this.callback(this.message)`') passed in inter-component communication. Such a call obscures the data sink, making it hard for static analyzers to detect that the OAID reaches an external transmission point (i.e., `http.post`). Furthermore, another key point is that data and callbacks are not confined to a single component. The critical UI state variable, i.e., `message` is linked and bound by the two

```
1  @Entry @Component
2  struct C1{
3     @State message:string = 'no taint'
4     request(message){
5        http.post(message)  //Sink API
6     }
7     build() {
8     RelativeContainer() {
9        //Initialize component2
10       C2({callback:(msg)=>{this.request(msg)},
11            message:this.message})
12       Text(this.message).onClick(() => {
13          //Read source data
14          this.message = identifier.getOAID()
15       })
16 }}}
```

```
17 @Component
18 struct  C2{
19    callback = (): void => { }
20    @Link message:string
21
22    build() {
23    RelativeContainer() {
24       Button().onClick(()=>{
25          //Call sink API
26          this.callback(this.message)})
27          ……
28 }}}
```

Fig. 1. Simplified Real World Privacy Issue (Case#1)

components C1 and C2. A naive analysis would fail to connect the source to the sink, as the data propagates through component state and the sink is invoked via an indirect callback shared between components. This example underscores a fundamental limitation of traditional static analyzers when applied to modern component-based architectures. Their weakness lies in tracking taint through shared object references and callback-mediated control flows. It motivates the need for a static analysis that precisely resolves inter-component aliasing and indirect function calls.

3 Methodology

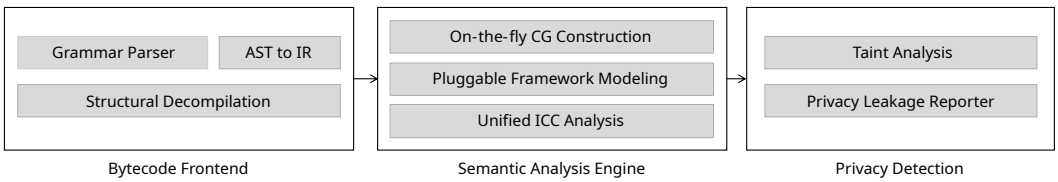

Fig. 2. HScope Design and Workflow

We introduce HScope, a static taint analysis framework for detecting privacy leaks in OpenHarmony applications. In contrast to source-level approaches (e.g., ArkAnalyzer), HScope operates directly on compiled Ark bytecode and does not depend on type annotations. As shown in Figure 2, HScope requires only the application package and a configurable set of taint sources and sinks as input. The analysis begins with the Ark bytecode frontend, which parses the application package, extracts Ark bytecode, and constructs a lightweight intermediate representation (IR) that preserves the original bytecode semantics while making instructions amenable to static analysis. At its core, HScope employs abstract interpretation to model program semantics and compute abstract memory objects associated with program symbols, including variables, classes, methods, objects, and closures. This abstraction provides a conservative approximation of program behavior in the absence of static type information. Instead of relying on a pre-defined ICFG, HScope constructs the call graph incrementally during analysis. Based on this semantic model, HScope performs context-sensitive (1-callsite) and flow-sensitive data-flow analysis, propagating abstract objects through incrementally constructed call graphs and intra-procedural control flows. Indirect control

Program	::=	$\{Function^* \mid Class^* \mid ImportStmt\}$	
Function	::=	$Name(p_1, \ldots, p_n)\{Inst^*\}$	
Class	::=	$Name\{Property^*, Function^*\}$	
ImportStmt	::=	**import** *ID*	
Inst	::=	*Assignment* | *PropRead* | *PropWrite* | *Call*	
	|	*NewObj* | *IfStmt* | *GotoStmt* | *ReturnStmt*	
Assignment	::=	$v_{\text{dst}} = v_{\text{src1}}\ [op\ v_{\text{src2}}]$	; e.g., v1 = v2 or v1 = v2 + v3
PropRead	::=	$v_{\text{dst}} = v_{\text{receiver}}.m_{prop}$	; e.g., v1 = v_obj.propName
PropWrite	::=	$v_{\text{receiver}}.m_{\text{prop}} = v_{\text{src}}$	; e.g., v_obj.propName = v2
Call	::=	$[v_{\text{dst}} =]\ v_{\text{func}}(v^*_{\text{arg}})$	; e.g., v_ret = foo(v1, v2)
NewObj	::=	$v_{\text{dst}} =$ **new** $c_{\text{id}}(v^*_{\text{arg}})$	; e.g., v1 = new MyClass(v_arg)
IfStmt	::=	**if** $(v_{\text{cond}})\ \{Inst^*\}\ [$**else** $\{Inst^*\}]$	
GotoStmt	::=	**goto** i	
ReturnStmt	::=	**return** $[v_{\text{ret}}]$	
v	$\in$	**VReg**	; Virtual registers / variables
ID, m, c, i	$\in$	**MetadataID**	; IDs for strings, classes, instructions, etc.
op	$\in$	$\{+, -, *, /, \ldots\}$	; Operators

Fig. 3. Main Syntax of Structural IR

flows, including event handlers and framework callbacks, are resolved by evaluating the abstract memory objects referenced by dispatch variables at call sites. To support cross-component analysis, HScope further models inter-ability communication, router-based navigation, and shared UI state variables, enabling taint propagation across component boundaries and recovering implicit control and data dependencies. Finally, on top of the recovered semantic information, HScope conducts taint analysis to check whether sensitive data introduced by source APIs can reach security-sensitive sink APIs, and reports potential privacy leakage paths.

In the following subsections, we first present the bytecode frontend and IR construction process (Section 3.1). We then define the abstract domain used for abstract interpretation (Section 3.2) and formalize the abstract semantics for context-sensitive semantic analysis (Section 3.3). Finally, we describe the taint analysis algorithm and privacy leakage detection procedure (Section 3.5 and Section 3.6), followed by implementation details (Section 3.7).

3.1 Bytecode Frontend and IR Construction

The input for our analysis is the OpenHarmony app package, a standard ZIP archive containing all application assets. In this phase, HScope extracts and prepares the necessary artifacts for analysis. We leverage the official 'ark_disasm' tool provided by the ArkCompiler toolchain to decode the Ark bytecode files. From this low-level bytecode, HScope performs a structured decompilation step to construct a more abstract IR. While retaining the register-based and three-address-code style, we explicitly reconstruct critical structures such as conditionals and lexical environments. The main syntax of our IR is defined in Figure 3.

After converting low-level Ark bytecode to our straightforward IR, we further make the following enhancements.

- *Variable Name Recovery:* In Ark bytecode, variables are accessed through an indirect addressing model based on lexical environments rather than explicit source-level names. We recover variable identities by reconstructing these lexical environments, which are typically initialized at the beginning of each function. This allows us to statically map environment slots to variable names for ordinary functions. Function closures are commonly used in callbacks, introducing additional complexity. A closure retains a reference to the lexical

environment of its parent function, identified by the parent's definition location or ID, and accesses variables via relative lexical levels and slot indices. To recover variable names in closures, we first identify the corresponding parent function. This is achieved by leveraging scope-related hints encoded in the closure's function name and by detecting the bytecode instruction in the parent function that initializes the child's lexical environment. With the parent function resolved, we can reconstruct the appropriate lexical environment and recover variable identities.
- *Function Call Name Recovery:* Function names in Ark bytecode are not stored as direct source-level identifiers, but are encoded across multiple structured components (e.g., module, class, and compiler-generated suffixes). We reconstruct human-readable function names by decoupling these components and reassembling them according to Ark's naming conventions.
- *Structured Control Flow:* By reconstructing 'IfStmt' from the bytecode's jump patterns, we create IR statements that are immediately amenable to standard data-flow analysis algorithms. This simplifies CFG and makes control dependencies explicit.

3.2 Abstract Domain

The foundation of our analysis is an abstract domain designed to capture the dynamic and flexible nature of Ark bytecode. We use allocation-site abstraction, where each abstract entity corresponds to a specific point in the code where it was created. Let $\mathcal{L}$ be a finite set of abstract locations, where each $\ell \in \mathcal{L}$ uniquely corresponds to a static allocation site (e.g., object instantiation).

The domain of abstract memory objects, $\hat{A}$, is defined as the union of abstract primitives, objects, and closures, mirroring the core object types:

$$\hat{A} = \hat{P} \cup \hat{O} \cup \hat{C}$$

- *Abstract Primitives ($\hat{P}$)*: A primitive object is a pair $(\hat{p}, \tau)$. Here, $\hat{p}$ represents the object's type (e.g., $\widehat{\mathsf{string}}$, $\widehat{\mathsf{number}}$), and $\tau \subseteq \mathcal{T}$ is a set of taint tags from a finite set of sources $\mathcal{T}$. Attaching taint at this fundamental level allows us to track sensitive information from its origin. A non-empty τ signifies the object is tainted.
- *Abstract Compound Objects ($\hat{O}$)*: To address untyped objects, an abstract object $\hat{o}$ is modeled as a partial map $\hat{o} : \mathsf{Prop} \rightharpoonup \mathcal{P}(\mathcal{L})$. It associates property names with sets of abstract locations, effectively modeling the object's structure and data without relying on static type declarations.
- *Abstract Closures ($\hat{C}$)*: To handle higher-order functions and callbacks, an abstract closure $\hat{c}$ is a pair (f, Γ). Here, f is a function identifier and $\Gamma : \mathsf{Var} \rightharpoonup \mathcal{P}(\mathcal{L})$ is its abstract environment. Capturing this environment is crucial as it preserves the context in which the function was defined, enabling correct analysis when it is invoked later, potentially in a different component.

The entire program state is captured in an abstract state $\sigma \in \Sigma$, which is a tuple $(\mu, \mathcal{S}, \mathcal{R})$. This serves as a global snapshot of the program's memory objects and framework-level communication channels.

- $\mu : \mathcal{L} \to \mathcal{P}(\hat{A})$ is the abstract store, mapping every location to the set of abstract objects it may hold.
- $\mathcal{S} : \mathsf{Comp} \times \mathsf{StateKey} \rightharpoonup \mathcal{P}(\mathcal{L})$ models inter-component state bindings (e.g., via '@Prop' or '@Link').
- $\mathcal{R} : \mathsf{Route} \rightharpoonup \mathcal{P}(\mathcal{L})$ models route-parameter bindings in routers.

3.3 Abstract Semantics

We define the program's behavior through a set of transfer rules that specify how each instruction transforms the abstract state σ. These rules form the core of our abstract interpreter. These rules are context-aware. For brevity, we assume a contextualized environment $\rho_{ctx} : \text{Var} \rightharpoonup \mathcal{P}(\mathcal{L})$ that resolves local variables to their abstract locations.

Base Operations. The following rules model the fundamental operations on abstract memory objects.

Object Allocation: $\ell \leftarrow \texttt{new } C()$

$$\frac{\hat{o}_{\text{init}} = \lambda p.\, \emptyset \quad \text{(empty shape)}}{\mu' = \mu[\ell \mapsto \mu(\ell) \cup \{\hat{o}_{\text{init}}\}]}$$

Assignment: $x \leftarrow y$

$$\frac{\rho_{ctx}(x) = L_x \quad \rho_{ctx}(y) = L_y}{\mu' = \mu[\ell_x \mapsto \{\mu(\ell_x) \cup \mu(\ell_y)\}]}$$

Property Write: $x.p \leftarrow y$

$$\frac{\rho_{ctx}(x) = L_x \quad \rho_{ctx}(y) = L_y \quad \forall \ell_x \in L_x, \forall \hat{o} \in \mu(\ell_x) \cap \hat{O}}{\mu' = \mu[\ell_x \mapsto (\mu(\ell_x) \setminus \{\hat{o}\}) \cup \{\hat{o}[p \mapsto \hat{o}(p) \cup L_y]\}]}$$

Property Read: $x \leftarrow y.p$

$$\frac{\rho_{ctx}(y) = L_y \quad L_x = \bigcup_{\ell_y \in L_y, \hat{o} \in \mu(\ell_y) \cap \hat{O}} \hat{o}(p)}{\rho'_{ctx} = \rho_{ctx}[x \mapsto L_x]}$$

Call Operations. Handling functions, especially high-order functions and indirect calls, is critical for resolving control flow. Our rules model this by tracking abstract closures and their captured environments.

Function Definition: $f = \lambda(\vec{p}).body$, with free vars $\vec{v}$

$$\frac{\Gamma = \{v \mapsto \rho_{ctx}(v) \mid v \in \vec{v}\} \quad \hat{c} = (f, \Gamma) \quad \ell_f \text{ is allocation site for } f}{\mu' = \mu[\ell_f \mapsto \mu(\ell_f) \cup \{\hat{c}\}] \quad \rho'_{ctx} = \rho_{ctx}[f \mapsto \{\ell_f\}]}$$

Function Call: $x \leftarrow y.f(\vec{a})$

$$\frac{\rho_{ctx}(y) = L_y \quad \forall \ell_y \in L_y \quad \text{(each } \ell_y \text{ becomes a new context } ctx')}{\text{For each } ctx', \text{ yielding return locations } L_{\text{ret},ctx'} \text{ where } \rho'_{ctx} = \rho_{ctx}[x \mapsto \rho_{ctx}(x) \cup \bigcup_{ctx'} L_{\text{ret},ctx'}]}$$

Event Handlers. The rules for implicit control flow are analogous, resolving target functions from the abstract store and invoking them within the appropriate new contexts. An event registration call `c.on('event',h)`, where `event` represents the event name and h is the event handler, is canonicalized into a property write, e.g., $c.\texttt{on\_event} \leftarrow h$. When the framework dispatches 'event' on 'c', our analysis models it as an indirect call to the closures stored in the '`on_event`' property of the abstract object for 'c'.

Inter-Component Communication. Implicit flows across components are modeled via location summaries passed through $\mathcal{S}$ and $\mathcal{R}$. These summaries are sets of abstract locations. Crucially, OpenHarmony introduces a fundamental distinction from traditional platforms (e.g., Android):

- *Ability-based Navigation and Router & State Channels:* Parameters are restricted to primitives and serialized objects. No live references or executable callbacks may cross boundaries. This is modeled by object capture. Only abstract objects $\hat{A} \in \hat{P} \cup \hat{O}$ are copied, and new abstract locations ℓ_{copy} are allocated in the target component. Original locations are not propagated.

Below, we detail the location summary propagation principle. Let $\mathcal{M} : \text{Path} \rightharpoonup \mathcal{P}(\mathcal{L})$ be a global summary map, where Path is a transition identifier (e.g., route string and state key pair).

- *Sender (Ability Navigation/Router):* When an object v is passed, for each $\ell \in \rho(v)$ and each $\hat{A} \in \mu(\ell)$, allocate a *fresh* location ℓ_{copy} in the target context, initialize $\mu_{\text{target}}(\ell_{\text{copy}}) = \{\hat{A}\}$, and propagate ℓ_{copy} instead:

$$\mathcal{M}[p] = \mathcal{M}[p] \cup \ell_{\text{copy}}$$

- *Sender (State):* When an object v is passed, its location ℓ is bound to a path p:

$$\mathcal{M}[p] = \mathcal{M}[p] \cup \ell$$

- *Receiver:* When the corresponding retrieval API is called, locations are restored from $\mathcal{M}[p]$:

$$\rho'[x] = \mathcal{M}[p]$$

This mechanism uniformly handles:

Implicit Data Flow: When L_v points to $\hat{O}$, and the UI state channel permits location sharing, then:

$$\forall \ell \in L_v,\ \forall \hat{o} \in \mu(\ell) \cap \hat{O}$$

Any update to $\hat{o}$ in one component is visible to all components sharing ℓ. Thus, the mutation made by sender or receiver is observable to the other. The abstract store μ is global to the analysis, so all references to ℓ see the same evolving $\mu(\ell)$.

Control Flow Delegation: When L_v points to $\hat{C}$, and the channel permits location sharing, then:

$$\forall \ell \in L_v,\ \forall \hat{c} = (f, \Gamma) \in \mu(\ell) \cap \hat{C}$$

Invocation from any component executes f under environment Γ. This models the receiver triggering logic defined and scoped in the sender.

It is worth noting that for the routers, the target component is first determined. Specifically, when `pushUrl()` is called, we check the `url` specified by `pushUrl()` to map routes to component identifiers. Hence, the analysis adds control-flow edge to the entry point of the component associated with 'U'.

$$\frac{\texttt{(Router Push)} \quad \textit{router.pushUrl(\{url:U, params:P\})} \quad \rho_{ctx}(P) = L_P}{\mathcal{R}' = \mathcal{R}[U \mapsto \mathcal{R}(U) \cup L_P]}$$

Our current implementation provides robust models for OpenHarmony's core communication patterns: UI event handling, router-based navigation, and declarative state management (e.g., `@Prop` and `@Link`). However, we acknowledge that OpenHarmony is a complex and evolving ecosystem. Certain advanced inter-component mechanisms, such as data exchange via shared file persistence, are currently out of scope and represent areas for future work. Our analysis is therefore sound with respect to the features we model.

3.4 Context-Sensitive Semantic Analysis Engine

HScope's analysis engine implements the abstract semantic analysis defined above. To handle dynamic calls and framework APIs, it operates in three phases: (1) building call graph on-the-fly, (2) modeling framework-specific semantics, and (3) unifying UI data access. The analysis starts from entry points (e.g., `onWindowStageCreate`) and iterates via a worklist until convergence.

On-the-Fly CG Construction. Pre-defining a complete call graph for ArkTS is infeasible due to untyped bytecode. Our engine constructs call graph incrementally using a worklist of $\langle pc, ctx \rangle$ units, where ctx can be the abstract location ℓ of the receiver object or the program point of the

Algorithm 1 Context-Sensitive Taint Analysis Algorithm

Initialize worklist W with entry point $\langle pc_{\text{entry}}, ctx_{\text{global}} \rangle$
Initialize analysis map $M[pc][ctx] \leftarrow \bot$ (for all pc, ctx)
Initialize $M[pc_{\text{entry}}][ctx_{\text{global}}] \leftarrow \sigma_{\text{init}}$
while $W \neq \emptyset$ **do**
    Pop $\langle pc, ctx \rangle$ from W
    $\sigma_{\text{in}} \leftarrow M[pc][ctx]$
    Let i be the instruction at pc
    $\sigma_{\text{out}} \leftarrow [\![i]\!]_{ctx}(\sigma_{\text{in}})$ ▷ Apply transfer function
    Check for leaks at pc
    for all successor $\langle pc', ctx' \rangle$ of $\langle pc, ctx \rangle$ **do**
        $\sigma_{\text{prev}} \leftarrow M[pc'][ctx']$
        $\sigma_{\text{new}} \leftarrow \sigma_{\text{prev}} \sqcup \sigma_{\text{out}}$ ▷ Join states
        if $\sigma_{\text{new}} \neq \sigma_{\text{prev}}$ **then**
            $M[pc'][ctx'] \leftarrow \sigma_{\text{new}}$
            Add $\langle pc', ctx' \rangle$ to W
        end if
    end for
end while
return All reported leaks

call statement. When encountering a `Call`, it resolves targets from the callee's abstract object. For each abstract closure $\langle f, \Gamma \rangle$, it generates ctx', enqueues $\langle f_{\text{entry}}, ctx' \rangle$, and records return-flow links.

Pluggable Framework Modeling. Framework semantics (e.g., event registration) cannot be captured by generic heap operations. We use a transfer function dispatcher to process instructions: standard operations (e.g., `NewObject`) use default handlers on μ, while framework calls delegate to specialized semantic handlers. For instance, the router handler for `router.pushUrl` updates the global route map R, while the event handler for `emitter.on("event",h)` registers the callback under a literal event name in μ. For dynamically computed names, we conservatively aggregate to a single dynamic-event key, a soundness-preserving trade off for scalability. This design allows easy extension to new APIs.

Unified ICC Analysis. App data can flow through ICC. Our UI manager maps all to the abstract state $\sigma = (\mu, S, R)$. All data reading and writing operations route through this layer. In OpenHarmony, UI components communicate by sharing state variables that reference live objects. HScope models these shared states as abstract heap objects, and introduces UI Manager as a unified abstraction to resolve aliasing across components. Specifically, when a shared state variable is updated in one component, the corresponding abstract object is updated in the global abstract store, making the change visible to all components that reference it. This enables HScope to capture both data flow and implicit control flow induced by ICC. These cross-component interactions are also incorporated into the call graph and taint propagation, allowing HScope to track privacy flows across component boundaries.

3.5 Taint Analysis

Based on the program semantics generated by the above context-sensitive abstract memory object analysis, the taint analysis is performed to detect flows from sources to sinks. The taint tag propagation is conducted in two levels: the symbol level and the abstract object level. In particular, a taint environment M is applied to record all tainted symbols and objects. For tag injection, when a symbol is tainted, all its pointed-to objects are also tainted. For tag propagation (e.g., assignment $x = y$), taint tags flow at the symbol level $x \leftarrow y$, and also at the object level. The latter does not

require extra processing, as y's abstract objects have been included in x's. Thus, the taint tag is propagated automatically at the object level. A leak is reported if an object with a non-empty taint set reaches a sink.

The analysis is implemented as a worklist-based algorithm, as shown in Algorithm 1. The algorithm systematically explores the program's contextualized semantic space until a fixpoint is reached. The join operator ($\sqcup$) in Line 12 merges abstract states by performing a component-wise union of their maps and taint sets. To guarantee termination for programs with loops and recursion, standard widening techniques are applied.

3.6 Privacy Leakage Detection

In this module, HScope conducts taint analysis based on the results of our semantic analysis engine and checks privacy leakage risks.

Taint Semantics and Policy. HScope follows our taint analysis algorithm (Section 3.5) to implement taint tracking. Taint is introduced at source APIs or event callback parameters. Taint flows through assignments, property accesses, and combinations. A leak is flagged when a tainted variable reaches a sink parameter. Source and sink definitions, along with taint tags, are externalized in a JSON policy for easy updates. We defined a comprehensive taint policy covering various privacy categories. Table 1 enumerates the representative APIs used as sources and sinks in our experiments. To explicitly summarize the taint propagation semantics, Table 2 lists the representative rules used by HScope.

Table 1. Taxonomy of Privacy Sources and Sinks in HScope Policy

Category	Type	Representative APIs	Sensitive Data Description
Device Identity	Source	`deviceInfo, identifier`	Persistent or non-permanent identifiers (e.g., OAID)
Location	Source	`geoLocation`	Precise GPS coordinates and cell tower information
Telephony	Source	`telephony.sms, contact`	Phone numbers and contact address book entries
Personal Info	Source	`calendar, account`	User schedules and account-related metadata
Network	Source	`wifiManager, connection`	IP/MAC addresses and network topology information
Network	Sink	`http.post, webSocket`	Unauthorized data exfiltration to remote servers
Diagnostic	Sink	`HiLog, console.log`	Publicly readable diagnostic side-channels
Messaging	Sink	`sms.sendMessage`	Automated data leakage via text messaging

Table 2. Representative taint propagation rules in HScope

IR Instruction Category	Taint Propagation Direction
Assignment (e.g., $x = y + z$)	From z and y to x
Property Read (e.g., $x = y.p$, $x = y[p]$)	From $y.p$ to x
Property Write (e.g., $y.p = x$, $y[p] = x$)	From x to $y.p$ and y, which may cause over-tainting
Function Return	From return value to caller
Function Call (e.g., $z = x(y)$)	From y to callee parameter

Actionable Leak Reporting via Taint Provenance Tracking. Identifying a source-sink pair is insufficient for checking. HScope records taint provenance by augmenting the abstract state with a taint-flow graph. When a transfer function computes $\tau_{new} = \tau_{src1} \vee \tau_{src2}$, it stores back-pointers to τ_{src1}, τ_{src2}, and the responsible instruction (pc). Upon detecting a leak, HScope performs a backward traversal from the sink, reconstructing a complete, context-sensitive data-flow path (a sequence of

$\langle pc, ctx \rangle$ pairs) that traces how sensitive data flowed from source to sink, providing a precise flow trace.

3.7 Implementation

We implemented the prototype of HScope with 33,757 lines of Python code. The choice of Python prioritizes rapid prototyping and seamless integration with underlying tools.

To process ArkTS bytecode, we use the official `ark_disasm` tool for disassembly. For parsing ArkTS bytecode syntax, we leverage the general-purpose parser generator `tree-sitter` [24]. 1,252 lines of JavaScript code are provided to tree-sitter as grammar rules.

Critically, all subsequent components (including AST-to-IR conversion, context-sensitive abstract interpretation, taint propagation, and leakage detection) are implemented entirely from scratch. This is necessary because, to our knowledge, no existing static analysis tools support fine-grained and context-sensitive analysis of ArkTS bytecode or its OpenHarmony framework semantics.

4 Experiments

In this section, we describe our evaluation process. Generally, we evaluate HScope by answering the following research questions:

- *RQ1: How accurate is HScope in detecting privacy leaks?*
- *RQ2: Can HScope detect real privacy leaks in real-world wild OpenHarmony apps?*
- *RQ3: Does the semantic analysis engine actually work as expected?*
- *RQ4: How does HScope compare to state-of-the-art static techniques in detecting OpenHarmony privacy leaks?*

4.1 Dataset Preparation

To evaluate the effectiveness of HScope, we prepared a multi-faceted evaluation environment consisting of a controlled micro-benchmark and an extensive collection of real-world applications.

Micro-benchmark: OH-Bench. Due to the absence of a standardized privacy leak test suite for OpenHarmony, we developed OH-Bench, a benchmark consisting of 50 test cases across 11 categories (e.g., aliasing and ICC). OH-Bench follows the design principles of the Android benchmark 'DroidBench' [21]. Additionally, it specifically targets the unique semantics of ArkTS and the OpenHarmony framework, such as reactive state sharing and router-based navigation.

Real-world App Dataset. Our evaluation utilizes 300 OpenHarmony apps collected from diverse sources, including the official developer portal [14] as well as code-hosting platforms such as GitHub, Gitee, and GitLab. We selected projects compatible with recent OpenHarmony versions and maintained within the past six months to ensure ecological representativeness. Please note that directly crawling apps from the official market is currently *infeasible*. This is because OpenHarmony enforces robust anti-sniffing measures (e.g., package encryption) and restrictive app package protection. We contacted the OpenHarmony OS development team, and they have confirmed this status. Consequently, our reliance on open-source apps ensures the feasibility, transparency, and verifiability of our evaluation.

4.2 Performance Overhead

To evaluate the practicality and scalability of HScope, we further measured its analysis overhead on the 300 real-world OpenHarmony applications used in our evaluation. All experiments were conducted on a machine equipped with an Intel Core i7-14700 CPU, 32 GB of RAM, and Ubuntu 24.04 LTS. Across 300 applications, the average analysis time was 206 seconds, and the average memory consumption was 1,523 MB per application.

Instead of performing a whole-program analysis, HScope constructs the call graph on the fly and adopts a highly targeted, entry-point-driven strategy. By strictly initiating analysis from framework-defined entry points and UI rendering functions, we ensure only reachable, execution-relevant code is processed.

4.3 Effectiveness on Micro-benchmark

To evaluate the precision and recall of HScope, we conducted experiments on OH-Bench, our curated micro-benchmark consisting of 50 test cases. As summarized in Table 3, HScope achieves an overall Precision of 87.5%, a Recall of 87.5%, and an F1-score of 0.88.

HScope demonstrates high effectiveness in resolving OpenHarmony-specific patterns. Specifically, it achieves a 100% success rate in categories such as *Indirect Calls*, *UI-State Sharing*, and *ICC/Router*. This validates that our abstract interpretation-based modeling of framework semantics is technically sound. In contrast, the baseline tool ArkAnalyzer yields a Precision of 81.8% and a Recall of 22.5% (F1-score: 0.35). The significant gap in recall highlights ArkAnalyzer's limitations in handling the dynamic and indirect control flows pervasive in the OpenHarmony ecosystem.

False Positives and Negatives. To better understand the tool's limitations, we manually investigated the 5 false positives (FP) and 5 false negatives (FN). The primary factor for FPs is the over-approximation of composite data structures. For instance, HScope currently treats several structures like `HashMap` as single abstract entities. Consequently, if one element is tainted, any property access to that object is considered tainted, leading to sibling-property over-tainting. FNs are mainly attributed to complex dynamic behaviors such as string-based dynamic imports and reflection.

Table 3. Detection Results on Micro-benchmark (OH-Bench)

Category (No. of Cases)	GroundTruth	TP	FP	FN
Aliasing & Arrays (6)	6	6	0	0
Lifecycle & Tasks (6)	6	6	0	0
Indirect Calls (5)	5	5	0	0
UI-State Sharing (4)	4	4	0	0
ICC / Router (3)	3	3	0	0
Language (Dynamic) (11)	10	7	0	3
No Leak (Controlled) (8)	0	0	5	0
Others (OS, Import, etc.) (7)	6	4	0	2
Total (50)	**40**	**35**	**5**	**5**

4.4 Real-world Privacy Detection

We applied HScope to our real-world app dataset consisting of 300 OpenHarmony apps. Across the 300 applications, the average size of source code is 21,548 lines (ranging from 356 to 73,585 lines), while the average size of bytecode is 110,426 instructions (ranging from 2,106 to 443,528 instructions). The average number of UI pages and components is 23 and 28, respectively. HScope successfully finds 72 sources accessed by the dataset apps, while 48 of them are potentially leaked. We manually verified these potentially risky flows from sources to sink APIs, and confirmed that 39 of these sensitive data flows are feasible. These 39 verified sensitive flows involve 27 OpenHarmony apps. Table 4 presents more result details. Below we first analyze the cause of the imprecision and then discuss several crucial findings.

Table 4. Analysis Results of 27 Confirmed Privacy Leaks in OpenHarmony Apps

Repo	App Name (anonymized)	# Sources	# Leaks	Source Types	Analysis Difficulties
Official Repo	com.huawei.boxxx	1	1	Personal Info	indirectcall
	com.huawei.btxxx	1	1	Personal Info	indirectcall
	com.huawei.caxxx	1	1	Device Identity	indirectcall
	com.huawei.coxxx	1	1	Personal Info	indirectcall
	com.huawei.lixxx	1	1	Personal Info	inter-component
	com.huawei.exxxx	1	1	Personal Info	inter-component
	com.huawei.goxxx	1	1	Device Identity	indirectcall
	com.huawei.ofxxx	1	1	Personal Info	indirectcall
	com.huawei.caxxx	1	2	Device Identity	indirectcall
	com.huawei.coxxx	1	6	Network	indirectcall
	com.huawei.adxxx	1	1	Device Identity	inter-component
	com.huawei.arxxx	1	1	Device Identity	inter-component
Public Repo	com.example.onxxx	1	1	Location	indirectcall
	com.teach.ibxxx	1	1	Location	indirectcall
	io.calsul.mexxx	1	1	Personal Info	inter-component
	com.conrad.tixxx	1	1	Device Identity	inter-component
	com.example.wexxx	1	1	Location	inter-component
	com.swift.maxxx	1	1	Location	inter-component
	com.recruitment.texxx	2	2	Personal Info, Location	indirectcall
	com.atomicservice.caxxx	1	1	Telephony	indirectcall
	com.itheima.faxxx	1	1	Location	inter-component
	io.smdsbz.moxxx	1	2	Device Identity	inter-component
	com.yang.hmxxx	1	1	Device Identity	indirectcall
	com.example.haxxx	1	1	Device Identity	indirectcall
	io.github.wlxxx	1	1	Device Identity	indirectcall
	com.example.noxxx	1	1	Device Identity	indirectcall
	ohos.samples.obxxx	1	5	Telephony	indirectcall

False Positives. Among the 48 potential leaks, 9 cases were identified as false positives, resulting in a precision of 81.3%. Our manual audit revealed that these FPs primarily stem from two sources of imprecision: (1) Field-insensitivity in taint analysis, where HScope conservatively treated an entire object as tainted due to a single sensitive property, leading to sibling-property over-tainting. (2) Path-insensitivity, involving execution paths that are logically unreachable due to complex conditional branches that our static engine approximated. Despite these conservative estimates, HScope effectively prunes the search space for manual auditors.

Finding#1: Mobile apps frequently access privacy data. Among the 300 analyzed apps, sensitive data access remains pervasive, indicating widespread use of such data in practical development. From the manually validated true positives, HScope identifies 39 distinct taint flows across 27 apps, spanning five source categories: Personal Info, Device Identity, Location, Telephony, and Network. Notably, Device Identity (e.g., OAID) and Personal Info are the most frequently exfiltrated risk vectors.

Finding#2: Logging APIs dominate data exfiltration, posing systemic risks. Of the 39 confirmed taint flows, 32 (82.1%) exfiltrate sensitive data via logging APIs, while only 7 (18.0%) use network APIs. Logs in OpenHarmony are often publicly readable, enabling any co-resident app to harvest leaked data and turning logging into a de facto side channel. At the app level, 21 apps leak via logging, compared to 6 via network. This reinforces that developers underestimate the security implications of diagnostic output.

Finding#3: Official template apps propagate privacy risks at scale. We discover that 12 official OpenHarmony template apps, intended as starter kits for new developers, contain verifiable privacy leaks. Since developers commonly clone and modify these templates, privacy risks are inadvertently

inherited and amplified across the ecosystem. This copy-paste propagation turns pedagogical artifacts into systemic security liabilities, posing a critical concern for platform maintainers.

Finding#4: Inter-component communication and indirect calls impede precise taint tracking. As shown in Table 4, all found risky apps involve indirect calls (via function pointers and dynamic dispatch) during taint propagation, further challenging static resolution. Additionally, 10 of the 27 confirmed leaky apps rely on inter-component communication (via routers and state variables) to transfer sensitive data and delegate control. These mechanisms break conventional inter-procedural analysis assumptions. These patterns validate our motivation for designing HScope with cross-component and context-sensitive modeling.

4.5 Ablation Study

To quantify the specific contributions of our modeling components for OpenHarmony features, we conducted an ablation study using the real-world dataset of 300 apps. We evaluated three variants of HScope by disabling the following modules: *Closure Handling*, *Router Logic*, and *UI State Sharing*. The results are summarized in Table 5.

Impact of Closure Handling. The results indicate that the closure-handling module is the most critical component of HScope. Disabling this module led to a significant drop in detection capability, with TP decreasing from 39 to 15. This confirms that modern ArkTS applications rely heavily on anonymous functions and closures for event handling and asynchronous tasks; without precise context-sensitive closure modeling, a large portion of the data flow graph remains disconnected.

Impact of Router and State Variable Modeling. Disabling the router-handling module resulted in the loss of 16 confirmed leaks, while disabling state variable modeling led to 14 missed detections. These modules are essential for tracking flows that traverse component boundaries. Disabling state handling, in particular, also suppressed several control-flow detections, as reactive state changes often trigger implicit UI updates that traditional analyzers fail to capture.

The results demonstrate that while a basic taint analysis might be functional, it is insufficient for the complex and dynamic operations frequently used in the OpenHarmony ecosystem. Our specialized modeling for these features is vital for maintaining high detection recall.

Table 5. Ablation Study Results on Real-world Apps

Variant	TP	FP
HScope @W.O. Closure Handling	15	9
HScope @W.O. Router Logic	23	9
HScope @W.O. State Variable Modeling	25	5
HScope (Full Implementation)	**39**	**9**

4.6 Comparison

Preparation. The landscape of static analysis for OpenHarmony is still in its emerging stage. ArkAnalyzer[5] is the open-source source-code level static analysis framework, making it the natural choice as our primary baseline. Although HarmonyFlow[27] also operates in this domain, it is closed-source and only available through a web-based service. We tested the service in August 2025 and January 2026 and found that it consistently failed to produce results for real-world application packages, making it unsuitable for a verifiable comparative study. We contacted the authors of HarmonyFlow but received no response. Consequently, ArkAnalyzer remains the most

```
1  @Entry @Component
2  struct ExceptionReport {
3    onPageShow () {
4      //Receive value from Selection
5      param=router.getParams()
6      console.log(param.address) //Sink API
7    }
8    build() {
9      ……
10     //Go to Selection
11     Button().onClick(
12       router.pushUrl({
13         url: "pages/Selection"})
14     )
15     ……
16   }}
```

```
17 @Entry @Component
18 struct Selection{
19   @State address: string
20   aboutToAppear () {
21     this.address = getLocation() //Source API
22   }
23   build () {
24     Row().onClick(() => {
25     //Go back to ExceptionReport
26     router.back({
27       url: "pages/ExceptionReport",
28       params: {
29         address: this.address
30       })}}
```

Fig. 4. Case#2: Simplified Location Exfiltration

rigorous and transparent baseline for our evaluation. However, ArkAnalyzer operates exclusively on ArkTS source code, while HScope analyzes compiled bytecode. To ensure a fair comparison, we evaluate both tools on the same set of apps: we feed ArkAnalyzer the original ArkTS source, and provide HScope with the corresponding bytecode compiled by ArkCompiler. This setup isolates the impact of analysis design on detection capability.

Since ArkAnalyzer is not designed for taint analysis, we adopt a lenient yet principled methodology to assess its ability. We consider a data flow 'detected' by ArkAnalyzer, if the call graph or pointer assignment graph generated by ArkAnalyzer contains a feasible path connecting the privacy source and sink.

Comparison Results. To establish a baseline for fundamental analysis capabilities, we evaluated both tools on our 50-case micro-benchmark. As summarized in Section 4.3, HScope significantly outperforms ArkAnalyzer, particularly in terms of Recall (87.5% vs. 22.5%) and F1-score (0.88 vs. 0.35). This stark disparity stems from ArkAnalyzer's heavy reliance on source-level patterns, which often fail to capture the complex runtime dispatching and closure-captured objects that our bytecode-level abstract interpreter resolves with higher precision.

In our comprehensive check of the real-world dataset, ArkAnalyzer failed to identify 12 critical data flows across 11 of the 27 confirmed leaky applications. These failures originate from several architectural limitations in ArkAnalyzer, most notably its inability to handle inter-component data flows where a parent component invokes a child via complex delegates. Furthermore, the analysis lost track of data flows at indeterminate conditional branches and lacked specialized modeling for `AppStorage`, preventing it from tracking sensitive data that propagates through this platform-specific global state. Notably, we verified that ArkAnalyzer did not identify any privacy leaks that HScope failed to detect, underscoring the superior coverage and practical utility of our framework.

4.7 Case Study

Case#1: State Variable Sharing. We revisit the open-source app from GitHub introduced in Section 2.2. HScope detects an OAID sharing enabled by a reactive state variable (i.e. `message`). In component `C1`, when component `C2` is initialized at Line 10, an anonymous function is delegated to `C2`, and the state variable `message` is bound. HScope updates the abstract object of `C2`'s `callback` and `message`. For `callback`, its object points to the anonymous function in `C1`. For `message`, its

abstract location is identical to that of `message` in `C1`. HScope models this by aliasing `C2.message` to ℓ_{msg}, the same location as `C1.message`.

At Line 14, `identifier.getOAID()` writes a tainted object tagged τ_{oaid} into ℓ_{msg}. Because both components reference ℓ_{msg}, `C2.message` inherits τ_{oaid} implicitly. In this process, the data flows even when no assignment occurs in `C2`. When a button in `C2` is clicked at Line 24, the function held in the abstract object of `callback` is called. ℓ_{msg}'s corresponding object is transmitted as the parameter. Finally, when the sink API at Line 5 is called, the function reads from its parameter carrying τ_{oaid}. HScope reports the risk. HScope's precision in tracking taint through aliased abstract locations is critical for detecting leaks in reactive UI frameworks.

Case#2: Taint Flow via Router. In app '`com.swift.maxxx`' (anonymized), HScope detects a physical location propagating across components via the OpenHarmony router. First, the page ExceptionReport goes to the page Selection via `router.pushUrl()`. The taint originates at `getLocation()` at Line 21. The return object is stored into the abstract location ℓ_{addr} bound to `this.address` and tagged with τ_{loc}.

During navigation to previous page `ExceptionReport` (Line 26), `router.back()` passes `this.address` as a parameter. HScope models this as a transfer. It resolves the target component symbolically, maps ℓ_{addr} to the receiver's abstract location ℓ_{target}, and propagates τ_{loc}. When the sink API `console.log(...)` (Line 6) is called, the argument resolves to ℓ_{target}. Since it still carries τ_{loc}, a risk is reported. This case validates HScope's ability to track taint across framework-mediated boundaries by preserving and propagating abstract location bindings.

5 Related Work

Mobile Privacy and Security. A vast body of research has investigated the access and exfiltration of sensitive data within mobile ecosystems. For the Android platform, *FlowDroid* pioneered highly precise, lifecycle-aware static taint analysis by employing a context-, flow-, field-, and object-sensitive approach [1]. *Amandroid* introduced a comprehensive inter-component data-flow framework designed to reason about complex component interactions through environment modeling [28], while *IccTA* further refined this by specifically tracking privacy leaks that traverse Android component boundaries via Intent mechanisms [15]. To enhance formal guarantees, *HornDroid* leveraged Horn clauses and SMT solving to create a sound abstraction of Android's operational semantics, providing the first formal proof of soundness for such analysis techniques [3]. Addressing scalability, *R-Droid* utilized static program slicing to prune irrelevant code paths, thereby reducing the complexity of data-flow graphs for large-scale auditing [2]. Beyond internal flows, *ComDroid* exposed vulnerabilities in Android's Intent-passing mechanism, illustrating how misconfigured components lead to unauthorized data interception [7]. While static methods offer broad coverage, dynamic systems like *TaintART* provide real-time monitoring by implementing multi-level information-flow tracking within the Android Runtime (ART) via register-level tagging [22].

On the iOS side, *PiOS* demonstrated the feasibility of statically analyzing Objective-C binaries to detect privacy leaks in third-party libraries, specifically addressing the challenges posed by dynamic message passing [10]. Together, these works define the state-of-the-art for privacy analysis on mature platforms. However, these systems fundamentally rely on the assumption of strictly typed bytecode and well-defined, imperative component invocation mechanisms (e.g., Android Intents). As we demonstrate through our empirical study of 300 apps, OpenHarmony's use of dynamically-typed Ark bytecode and its reactive, event-driven programming model fundamentally invalidate these assumptions, necessitating a new analysis paradigm.

Taint Analysis. Taint analysis, encompassing both static and dynamic approaches, remains a cornerstone technique for detecting unauthorized data exfiltration. Fundamental research has established formalisms for dynamic taint propagation [20] and its practical implementation on mobile platforms [11]. In the Android ecosystem, enhancing precision for data leak detection has been a primary focus. For instance, *DroidSafe* introduced comprehensive API modeling to achieve high-accuracy information flow analysis [13], while *EdgeMiner* addressed the challenge of 'blind spots' in control-flow graphs by automatically identifying implicit callbacks registered in the framework [4]. Furthermore, to mitigate precision loss in complex object-oriented code, researchers have proposed context- and object-sensitive frameworks [18] and reflection-aware engines like *Ripple* to recover hidden call targets [31]. While these techniques effectively resolve imprecision in imperative Java/Android environments, adapting them to OpenHarmony requires fundamentally new abstractions.

OpenHarmony App Security. Academic interest in OpenHarmony security has flourished alongside the ecosystem's rapid growth. One primary research trajectory focuses on application quality through dynamic analysis and testing. For instance, *HapTest* provides an extensible framework for diverse dynamic security assessments [16], while *HmTest* utilizes a hybrid approach, integrating static guidance with dynamic exploration for automated GUI testing [6]. Another line of research addresses ecosystem migration, such as the *UITrans* framework, which employs large language models to facilitate UI code translation from Android to OpenHarmony [12]. While valuable, these dynamic methods are inherently constrained by code coverage limitations and the difficulty of triggering specific privacy-leakage paths.

Regarding static analysis, *HM-SAF* introduced a cross-layer framework using a memorized summary-based algorithm to track data flows specifically between Java and Native layers [32]. However, as the ecosystem transitions toward the 'pure' ArkTS architecture, *HM-SAF*'s reliance on cross-language bridges limits its applicability to modern reactive ArkUI applications. Recently, *ArkAnalyzer* was introduced as a comprehensive framework for source-level program analysis of ArkTS [5]. Despite its fine-grained modeling, *ArkAnalyzer* primarily operates on source code, which often lacks the explicit runtime semantics required to resolve complex closure captures and reactive state synchronizations. Instead, we propose a bytecode-level static analysis framework specifically designed to detect potential privacy leaks in OpenHarmony apps.

6 Limitations

HScope provides powerful semantic analysis, but it is not perfect. HScope is constrained by the scope and precision of its abstract semantics. While the analysis explicitly models key OpenHarmony and ArkTS mechanisms, certain highly dynamic language features (such as reflection, string-based dynamic imports, and runtime-generated property accesses) are conservatively approximated or remain unsupported. To ensure scalability, HScope adopts summary-based abstractions for some composite data structures (e.g., maps and records), which may introduce over-approximation and imprecise taint propagation across sibling fields. Finally, the analysis employs bounded context sensitivity to balance precision and efficiency, which may limit its ability to fully resolve deeply nested or highly polymorphic callback interactions.

7 Conclusion

The rapid emergence of OpenHarmony has outpaced the availability of dedicated security analysis tools, creating a critical gap in automated analysis capabilities. A robust and automated tool for security and privacy vetting is urgently needed. This paper introduces HScope, a context-sensitive static analysis engine designed specifically to address the unique challenges of ArkTS

apps. HScope supports on-the-fly call graph construction, pluggable framework modeling, and unified ICC analysis. This design allows HScope to precisely model complex data flows that traverse asynchronous callbacks, reactive UI state variables, and framework-specific communication channels that would confound traditional static analyzers. Our evaluation shows HScope is effective and scalable for vetting the security and privacy of real-world OpenHarmony apps. We hope the development of HScope opens avenues for future research and fosters further innovation in the OpenHarmony ecosystem.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. This work was supported in part by the National Natural Science Foundation of China (Grant Nos. U25A6023 and 62202106). Guangliang Yang and Min Yang are the corresponding authors. Min Yang is a faculty member of Shanghai Pudong Research Institute of Cryptology, Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

Ethics and Privacy Statement

This work does not have any ethical concerns. HScope is designed for fine-grained privacy leakage detection in real-world OpenHarmony apps. For the privacy leakage concerns, we have sent emails to all developers to report our findings with detailed results. In our paper, we have anonymized all app names. Thus, there is no sensitive information about the developers and apps being exposed.

Data Availability

The implementation of HScope is integrated into our open-source multilingual program analysis framework Lian. The specific version corresponding to this paper is archived on Zenodo [17], while the latest development version is publicly available at https://github.com/yian-lang/lian. HScope can be activated using the "`--lang abc`" or "`-l abc`" command-line option.

References

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/2594291.2594299

[2] Michael Backes, Sven Bugiel, Erik Derr, Sebastian Gerling, and Christian Hammer. 2016. R-Droid: Leveraging Android App Analysis with Static Slice Optimization. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. doi:10.1145/2897845.2897927

[3] Stefano Calzavara, Ilya Grishchenko, and Matteo Maffei. 2016. HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving. In *IEEE European Symposium on Security and Privacy*. doi:10.1109/EuroSP.2016.16

[4] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. 2015. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*. doi:10.14722/ndss.2015.23140

[5] Haonan Chen, Daihang Chen, Yizhuo Yang, Lingyun Xu, Liang Gao, Mingyi Zhou, Chunming Hu, and Li Li. 2025. ArkAnalyzer: The Static Analysis Framework for OpenHarmony. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. doi:10.1109/ICSE-SEIP66354.2025.00018

[6] Yi-Ge Chen, Yu-Jia Fan, Si-Nan Wang, Yi-Da Tao, and Ye-Pang Liu. 2025. HmTest: Automated Testing of HarmonyOS Apps via Model-Driven Navigation and Reinforcement Learning. *Journal of Computer Science and Technology* 40 (2025), 1006–1021. doi:10.1007/s11390-025-5142-4

[7] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*. doi:10.1145/1999995.2000018

[8] Paula Delgado-Santos, Giuseppe Stragapede, Ruben Tolosana, Richard Guest, Farzin Deravi, and Ruben Vera-Rodriguez. 2022. A Survey of Privacy Vulnerabilities of Mobile Device Sensors. *ACM Comput. Surv.* (2022). doi:10.1145/3510579

[9] Fahimeh Ebrahimi, Miroslav Tushev, and Anas Mahmoud. 2021. Mobile app privacy in software engineering research: A systematic mapping study. *Information and Software Technology* (2021). doi:10.1016/j.infsof.2020.106466

[10] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2011. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. https://seclab.nu/static/publications/ndss2011pios.pdf

[11] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*.

[12] Lina Gong, Chen Wang, Yujun Huang, Di Cui, and Mingqiang Wei. 2025. UITrans: Seamless UI Translation from Android to HarmonyOS. In *Proceedings of the 16th International Conference on Internetware*. doi:10.1145/3755881.3755933

[13] Michael I Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. 2015. Information-flow analysis of android applications in droidsafe. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*. doi:10.14722/ndss.2015.23089

[14] HUAWEI Developers. 2026. OpenHarmony Developer Portal. https://developer.huawei.com/consumer/cn/market/. Accessed July 22, 2026.

[15] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick D. McDaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE)*. doi:10.1109/ICSE.2015.48

[16] Farong Liu, Mingyi Zhou, Yakun Zhang, Ting Su, Bo Sun, Jacques Klein, Xiang Gao, and Li Li. 2025. HapTest: The Dynamic Analysis Framework for OpenHarmony. In *FSE 2025 Industry Track (Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering)*. doi:10.1145/3696630.3728565

[17] Mei, Yang, Guo, Wang, Gui, and Yang. 2026. *ISSTA-2026-AEC-camera-ready*. doi:10.5281/zenodo.21507308

[18] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.* (2005). doi:10.1145/1044834.1044835

[19] React Team. 2026. React. https://react.dev/.

[20] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland / SP)*. doi:10.1109/SP.2010.26

[21] Secure Software Engineering Group. 2026. DroidBench. https://github.com/secure-software-engineering/DroidBench.

[22] Mingshen Sun, Tao Wei, and John C.S. Lui. 2016. TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. doi:10.1145/2976749.2978343

[23] Team Counterpoint. 2026. Global Smartphone OS Market Share. https://counterpointresearch.com/en/insights/global-smartphone-os-market-share.

[24] Tree-sitter Contributors. 2026. Tree-sitter. https://tree-sitter.github.io/tree-sitter/.

[25] TS2 Tech. 2026. iOS vs.Android vs.OpenHarmony – Which Mobile OS Reigns Supreme? https://ts2.tech/en/ultimate-2025-showdown-ios-vs-android-vs-openharmony-which-mobile-os-reigns-supreme/. Accessed July 22, 2026.

[26] Luca Verderame, Davide Caputo, Andrea Romdhana, and Alessio Merlo. 2020. On the (Un)Reliability of Privacy Policies in Android Apps. In *2020 International Joint Conference on Neural Networks (IJCNN)*. doi:10.1109/IJCNN48605.2020.9206660

[27] Y. Wang, S. Chen, J. M. Li, W. J. Dang, R. C. Chai, Z. Y. Shi, and L. Li. 2025. HarmonyFlow: A Static Analysis Framework for HarmonyOS Applications Based on Ark Panda IR. *Journal of Software* (2025). doi:10.13328/j.cnki.jos.007477

[28] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security (CCS)*. doi:10.1145/2660267.2660357

[29] Wikipedia contributors. 2026. Addressing mode. https://en.wikipedia.org/wiki/Addressing_mode. Accessed July 22, 2026.

[30] Evan You and Vue.js Core Team. 2026. Vue.js. https://vuejs.org/. Accessed July 22, 2026.

[31] Yifei Zhang, Tian Tan, Yue Li, and Jingling Xue. 2018. Ripple: Reflection Analysis for Android Apps in Incomplete Information Environments. *Software: Practice and Experience* (2018). doi:10.1002/spe.2577

[32] Yukun Zhu, JiChao Guo, FengHua Xu, RuiDong Chen, XiaoSong Zhang, Shen Yi, and Jia Yu. 2023. HM-SAF: Cross-Layer Static Analysis Framework For HarmonyOS. In *2023 IEEE Smart World Congress (SWC)*. 1–10. doi:10.1109/SWC57546.2023.10449022

Received 2026-01-30; accepted 2026-04-16