Context-Sensitive Pointer Analysis for ArkTS

Yizhuo Yang[1], Lingyun Xu[2], Mingyi Zhou[1*], Li Li[1*]
[1] *School of Software, Beihang University*
[2] *CBG Software Engineering Department, Huawei*

***Abstract*—Current call graph generation methods for ArkTS, a new programming language for OpenHarmony, exhibit precision limitations when supporting advanced static analysis tasks such as data flow analysis and vulnerability pattern detection, while the workflow of traditional JavaScript(JS)/TypeScript(TS) analysis tools fails to interpret ArkUI component tree semantics. The core technical bottleneck originates from the closure mechanisms inherent in TypeScript's dynamic language features and the interaction patterns involving OpenHarmony's framework APIs. Existing static analysis tools for ArkTS struggle to achieve effective tracking and precise deduction of object reference relationships, leading to topological fractures in call graph reachability and diminished analysis coverage. This technical limitation fundamentally constrains the implementation of advanced program analysis techniques.**

Therefore, in this paper, we propose a tool named ArkAnalyzer Pointer Analysis Kit (APAK), the first context-sensitive pointer analysis framework specifically designed for ArkTS. APAK addresses these challenges through a unique ArkTS heap object model and a highly extensible plugin architecture, ensuring future adaptability to the evolving OpenHarmony ecosystem. In the evaluation, we construct a dataset from 1,663 real-world applications in the OpenHarmony ecosystem to evaluate APAK, demonstrating APAK's superior performance over CHA/RTA approaches in critical metrics including valid edge coverage (e.g., a 7.1% reduction compared to CHA and a 34.2% increase over RTA). The improvement in edge coverage systematically reduces false positive rates from 20% to 2%, enabling future exploration of establishing more complex program analysis tools based on our framework. Our proposed APAK has been merged into the official static analysis framework ArkAnalyzer for OpenHarmony.

I. Introduction

The rapid development of OpenHarmony [1], [2], an open-source, all-scenario operating system, has fostered a diverse application ecosystem. As applications grow in complexity, ensuring their correctness and security has become progressively challenging [3], [4]. Static program analysis identifies potential issues without execution. Pointer analysis [5], central to many of these approaches, plays a vital role in understanding pointer usage within an application [6], aiding in the detection of bugs [7], memory leaks and vulnerabilities such as buffer overflows. Nevertheless, due to the inherent complexity of modern applications, especially OpenHarmony's integration of heterogeneous components and services, effective pointer analysis remains a non-trivial task. This paper explores pointer analysis challenges and methodologies for OpenHarmony static analysis.

However, existing static analysis tools for OpenHarmony focus solely on the static information from statements, leading to analysis accuracy issues. When the code contains statements that require dynamic execution for their full resolution, issues such as false positives or false negatives in call graph construction and incomplete type inference may arise. These issues adversely affect downstream OpenHarmony static analysis tools, like data flow analysis, which depend on accurate input.

Therefore, implementing pointer analysis within the existing OpenHarmony static analysis framework is urgently needed [8]. However, existing pointer analysis methods (e.g., Soot [9] and Doop [10], [11] for Java and SVF [12], [13] for C/C++) are not compatible with the ArkTS program. This difficulty stems from ArkTS's unique features, syntactic sugar, and the distinct architecture of the OpenHarmony framework. In addition, ArkTS introduces a unique declarative UI paradigm. OpenHarmony kernel's rich APIs present significant data flow modeling and tracking challenges.

To address these challenges, this paper introduces APAK, the first context-sensitive Andersen-style [5] pointer analysis framework. It is designed to tackle accuracy issues within ArkAnalyzer [14], an existing static analysis framework for OpenHarmony. A core design principle of APAK is its extensibility, realized through a modular plugin system. This architecture is critical for future-proofing the analysis against the rapid evolution of the OpenHarmony ecosystem, allowing new framework APIs and language constructs to be supported without altering the core engine. For OpenHarmony APIs, APAK adopts a specialized heap abstraction method. This method treats instantiated classes, function pointers, containers, and other relevant elements as heap objects [15], simplifying pointer passing and invocation. Furthermore, APAK differentiates between dynamic and static call statements to apply distinct context strategies. APAK applies different context strategies for various scenarios and constructing an on-the-fly call graph with callsite-sensitivity [16], [17] and function-sensitivity. This approach provides more accurate type and call graph information, particularly when statically analyzing dynamic language features. APAK has been integrated into the official ArkAnalyzer open-source project, providing more precise type and call graph information for OpenHarmony applications and supporting security analysis.

To validate APAK's effectiveness in call graph generation, we constructed a test set of 1,663 real-world, open-source OpenHarmony apps and conducted comparative experiments with the Class Hierarchy Analysis (CHA) and Rapid Type Analysis (RTA) algorithms [18] integrated within the ArkAnalyzer. The results indicate APAK achieves call edge set sizes comparable to CHA while demonstrating a

*Corresponding authors

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00269

34.2% improvement over RTA, all while maintaining efficient analytical performance. It also demonstrates call resolution accuracy improvements of 5.1% to 49.8% across 12 samples and reduces the false positive rate from 20% to 2%. These findings substantiate the practicality of the APAK framework in analyzing real-world OpenHarmony apps, and establish reliable foundational support for future program analysis tools.

The main contributions we made are summarized as follows:

1) We proposed the first context-sensitive pointer analysis framework APAK that is designed for ArkTS.
2) Our method customizes heap abstraction and context-sensitive strategies for OpenHarmony applications, addressing the need for accurate type inference and call graph construction.
3) We evaluated the proposed APAK, demonstrating its effectiveness, efficiency, and utility in deriving accurate types and generating precise call graphs in OpenHarmony apps. We open-sourced the collected real-world OpenHarmony apps to advance future research*.
4) We open-sourced our proposed APAK by merging it into the official static analysis framework ArkAnalyzer for OpenHarmony†.

II. Background

A. ArkTS and TS

As a superset of TypeScript, ArkTS maintains core language characteristics while implementing systematic adaptations for mobile high-performance scenarios. Key architectural evolutions manifest in: 1) Paradigm innovation of declarative UI framework ArkUI. 2) System-level extension support through OpenHarmony public SDK. Figure 1 shows an example of the unique features of ArkTS.

Key Differences: (1) ArkUI employs declarative syntax structures for UI description logic. By abstracting underlying rendering mechanisms (Virtual DOM diff algorithms, GPU instruction pipeline scheduling, etc.), it enables developers to focus on business logic construction. The framework automatically synchronizes interface state changes to rendering pipelines through DSL-layer semantic mapping. In the ArkUI framework, component declaration is achieved through the `struct` keyword, working in conjunction with decorators such as `@Entry` and `@Component` to complete component type annotation and enable diversified component definitions including entry components and custom components.

(2) The OpenHarmony Public SDK provides cross-device collaboration API collections through layered architecture design (including device abstraction layer, distributed capability middleware, atomic service interfaces [19]). Its standardized interface contracts support "develop once, deploy everywhere" paradigm, significantly reducing multi-device adaptation complexity. The OpenHarmony SDK establishes a runtime support infrastructure for componentized architectures through the definition of core component libraries (e.g., `Column`, `Text`), while creating cross-layer system service access channels. The `userFileManager` interface adopts an abstraction layer design for persistent data storage, providing developers with a programmable interface paradigm at the application layer.

```
import userFileManager from '@ohos.filemanagement.userFileManager';
@Entry
@Component
struct View {
  @State text: string = 'PTA';

  build() {
    Column() {
      Text(this.text)
        .fontSize(50)
      Divider()
      Button('click')
        .onClick(() => {
          this.text = 'ArkUI'
        })
        .height(50)
        .margin({top:20})
    }
  }
}
public getUserFile(context) {
  return userFileManager.getUserFileMgr(context);
}
```


Fig. 1: ArkTS code example.

B. Call Graph

The call graph, as a core data structure in program analysis, formally characterizes invocation relationships between program units (nodes as entities, edges as potential flows). It is fundamental for code optimization [20], security defect detection [21], [22], and architectural maintainability analysis [23]. For compiler-driven optimizations, call graphs enable precise interprocedural analysis to guide decisions on function inlining, dead code elimination, and parallelization strategies. In security contexts, they underpin taint analysis by tracing vulnerability propagation paths across method boundaries. This is particularly crucial for identifying critical attack surfaces in Android applications, where incomplete call graph construction reportedly renders 58% of apps unanalyzable [24]. In this study, we will focus on leveraging pointer analysis to enhance the accuracy of call graph construction within OpenHarmony.

C. Pointer Analysis

Pointer analysis (or points-to analysis) [25], [26], [27] is a foundational static analysis technique that systematically determines the set of memory objects [15], [28] a pointer variable or expression may reference during program execution. It models relationships between pointers and their targets in memory, often by constructing a pointer assignment graph (PAG) where nodes are memory locations and edges denote possible references [29].

Modern pointer analysis tools exhibit distinct architectural characteristics and optimization strategies tailored for specific programming paradigms. SVF specializes in C program analysis by leveraging the LLVM intermediate representation to perform scalable and precise interprocedural value-flow analysis. It constructs value flow graphs [12] for security vulnerability detection utilizing optimizations such as sparse value flow graph construction to reduce memory consumption. Tai-e [30],

*The dataset is available at https://zenodo.org/records/15025243

†The repository is available at https://anonymous.4open.science/r/APAK-1E45

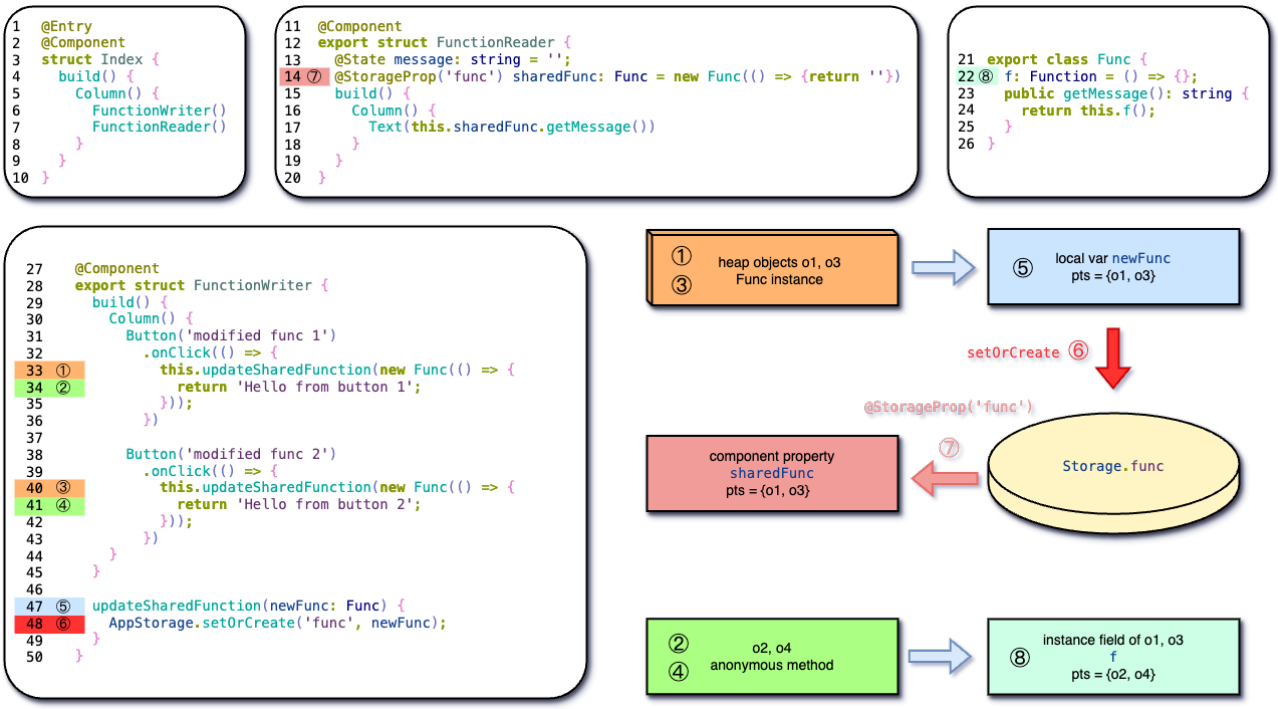


Fig. 2: ArkTS code with AppStorage and Conceptual PAG.

a developer-friendly framework designed for Java ecosystems, achieves higher precision in call graph construction compared to traditional tools. Doop [31] leverages Datalog-based context-sensitive analysis for Android application security vetting, while Rupta [6] implements pointer analysis through Rust MIR for call graph construction.

III. Motivating Example

Constructing precise call graphs for ArkTS applications is challenging when they leverage centralized reactive data stores for state management and intercomponent data synchronization. Common in modern UI frameworks, these stores allow disparate application parts to share and react to state changes without direct dependencies or explicit instance passing. Components typically write complex data, including object instances encapsulating functional logic, to a globally accessible location using a key. Other components can then subscribe to this key and reactively receive updates whenever the associated data changes.

As demonstrated in Figure 2, an architectural pattern is implemented using `AppStorage`, which provides a key-value mechanism at the system level for state management and synchronization throughout the application. When a button in the `FunctionWriter` component is activated, its `updateSharedFunction` method dynamically creates a new `Func` object instance, which contains a function property. This instance is then published to this centralized reactive store under the key `func`. Correspondingly, the `FunctionReader` component declares a reactive link to the `func` key in the store. This link ensures that `FunctionReader`'s local `sharedFunc` property is automatically updated to reference the latest `Func` instance dispatched by `FunctionWriter`. The displayed text is then derived from an invocation of `this.sharedFunc.getMessage()`, which executes the `this.f()` function property of the currently shared `Func` instance, reflecting the state managed through the centralized store.

Constructing a precise call graph for the scenario in Figure 2 presents a considerable challenge for existing static analysis tools within the OpenHarmony ecosystem. Conventional methods like CHA or RTA, the baseline algorithms in ArkAnalyzer, have fundamental limitations here. These methods typically struggle to accurately resolve the concrete object instance that the `sharedFunc` variable in `FunctionReader` points to when `getMessage()` is invoked. Because the instance of `Func` object is retrieved dynamically from the application's `AppStorage` (which is updated by button clicks in `FunctionWriter`), CHA/RTA lack the necessary pointer or data flow resolution capabilities to precisely track which specific `Func` instance (and therefore which encapsulated `f` function closure) is active. Consequently, they cannot reliably determine the precise target of the `this.f()` call within the `Func.getMessage()` method, leading to an imprecise or incomplete call graph where the true callees (the dynamically defined closures) are either missed or over-approximated.

①, ③ are `Func` object instances created in `FunctionWriter`. APAK abstracts them as distinct

heap objects (denoted as o_1 and o_3).

②, ④ are lambda expressions in lines 33 and 40. They are also abstracted as heap objects (denoted as o_2 and o_4), and then passed to these `Func` constructors. These anonymous function objects become targets of the `f` field within their respective `Func` instances. APAK establishes that the points-to set of $o_1.f$ is o_2 (i.e., $pts(o_1.f) = \{o_2\}$), and similarly, $pts(o_3.f) = \{o_4\}$.

⑤ represents the parameter `newFunc` in the method `updateSharedFunction`. When `updateSharedFunction` is called, the parameter receives a reference to one of these `Func` instances. $pts(newFunc)$ can be considered as potentially containing $\{o_1, o_3\}$ with different potential executions initiated by button clicks.

⑥ represents the crucial step that occurs with `AppStorage.setOrCreate`. APAK models the `func` key within `AppStorage` as a specific storage location. The pointer analysis rule for `setOrCreate` propagates the points-to set of `newFunc` to this storage location. Thus, $pts(AppStorage.func)$ becomes $\{o_1, o_3\}$

⑦ represents the `@StorageProp('func')` decorator on the `sharedFunc` property in the `FunctionReader` component, meaning the variable is synchronized with `AppStorage.func`. APAK establishes a pointer-flow relationship from `AppStorage.func` to `sharedFunc`. This means $pts(sharedFunc)$ will also be $\{o_1, o_3\}$.

This precise tracking of pointer propagation through heap abstractions for class instances and functions, along with specialized modeling of `AppStorage` interactions, allows APAK to identify the correct potential callees (o_2 or o_4) for the indirect function call, thus constructing an accurate call graph where traditional methods would falter.

In essence, the motivating example underscores the core difficulties for static analysis in ArkTS: precisely tracking dynamically created objects and their encapsulated logic (like closures) as they propagate indirectly through framework-specific reactive data stores such as AppStorage, and subsequently, accurately resolving indirect call targets that depend on these dynamically managed objects.

IV. APAK: Pointer Analysis for ArkTS

A. Overview

To address the technical challenges discussed in previous sections, we propose APAK – the first pointer analysis framework for ArkTS. Grounded in the formal semantics of the ArkTS language specifications, APAK employs a constraint-driven analysis algorithm augmented with incremental rule extensions to achieve precise modeling of OpenHarmony application characteristics.

APAK's core innovation lies in (1) a highly extensible plugin architecture: APAK's design allows for the integration of custom, rule-based plugins to analyze new or specialized APIs, ensuring future-proof adaptability. (2) semantical model of the features of ArkTS (e.g., the store mechanism), which allows APAK to understand and precisely model the behavior of OpenHarmony applications. This chapter will sequentially present the language model of ArkTS programs (Section IV-B), the core algorithm for pointer propagation (Section IV-C), the extension mechanism of the analysis algorithm (Section IV-D), and the context-sensitive processing mechanism (Section IV-E).

TABLE I: Simplified language model of ArkTS.

$t \in$ Type	::=	`number` | `string` | `boolean` | `T` | `(t|t)` | ...
$e \in$ Expr	::=	`e.x` | `new x({e})` | `({e}) => e`
$s \in$ Stmt	::=	`let x: t = e` | `const x = e` | `e.method({e})` | `x({e})` | `e.x := e` | `x := e` | `return e`
$f \in$ Function	::=	`function x({e}): t {{s}}`
$c \in$ Class	::=	`{d} class x({e}) {{m}}`
$st \in$ Struct	::=	`{d} struct x({e}) {{m}}`
$m \in$ Method	::=	`method x({e}): t {{s}}`
$n \in$ Namespace	::=	m | `{s | f | c}`
$d \in$ Decorator	::=	`@Component` | `@Entry` | ...

B. ArkTS Language Model

APAK adopts ArkIR, an intermediate representation (IR) of ArkTS source code implemented within ArkAnalyzer [14], as its input. Serving as an intermediate abstraction layer, ArkIR performs semantic normalization of source code through decoupling and desugaring processes. Table I illustrates a simplified syntactic specification of ArkIR along with the formal description of its type system. The model intentionally excludes the control flow statements (e.g., conditional branches, loop structures) to focus on core semantic features for pointer analysis.

The language model adopts a hierarchical architecture organized as follows:

- Declaration Layer: Includes declaration units of namespaces, functions, classes, and their methods.
- Statement Layer: Composes basic execution units, with focused modeling on statement types critical to pointer analysis: `AssignmentStmt` (Drives propagation of pointer aliasing relationships), `PropertyAccessStmt` (Involves pointer dereferencing on object fields) and `CallStmt` (Handling interprocedural propagation).
- Expression Layer: At the statement granularity level, expressions are decomposed into atomic operational units, `NewExpr` and `LambdaExpr` serve as the core semantics for heap object allocation.

ArkTS builds upon TypeScript's structural typing by introducing mandatory type declarations, which explicitly constrain the type boundaries for variables and parameters, particularly those used as pointers. This enhances the foundation for type inference in static analysis. In addition, ArkTS extends support for ArkUI features, such as struct components and decorators. Struct components serve as state containers for UI elements, and their memory layouts have a significant impact on pointer reachability analysis. Meanwhile, decorators, which inject lifecycle hooks through metaprogramming, necessitate special handling of their implicit pointer operations.

The language model demonstrates comprehensive coverage of ArkIR while effectively filtering out non-pointer-related statements. Building upon this foundation, we constructed the core rule set for interprocedural pointer propagation.

TABLE II: Pointer analysis notation system and domain definitions. We use these symbols to formalize the propagation rules introduced in this paper.

Syntactic Element	Notation	Domain
local	x, y	V
object	o_i, o_j	O
method	m	M
function	$func$	$Func$
field	f	F
pointer	p	P
pointer set	pts	$\mathcal{P}(O)$

C. Propagation Algorithm

APAK models set constraints of the ArkIR through a PAG. Its symbolic notations are defined in Table II. The pointer domain P in the program comprises both program variables and object field pointers, formally defined as $P = V \bigcup (O \times F)$. V contains all program variables (including local variables, global variables, and formal parameters). O denotes the set of runtime-allocated heap objects, and F represents the set of object fields. Each pointer $p \in P$ is associated with a points-to set $pts(p)$, which statically records all abstract heap objects potentially referenced by pointer p during program execution, whereby any object $o \in O$ becomes reachable through pointer p.

The graph structure contains two node categories: (1) program pointer nodes (including variables, formal parameters, etc.) whose points-to sets characterize potential heap object references, and (2) heap object nodes generated via heap abstraction, visually distinguished using double-circle notation. Directed edges represent inclusion constraints between pointers, formalized as $\forall(p_i, p_j) \in E$, $pts(p_i) \subseteq pts(p_j)$ to ensure semantic correctness in value propagation.

The edge generation process is implemented through semantic parsing of ArkIR instructions. ArkIR opcode classifier categorizes program statements into eight distinct pointer operation patterns, followed by the propagation rule set defined in Table III to generate corresponding constraint edges under different context environment (denoted as c). Through iterative solving of the constraint system's reachability properties, the pointer analysis converges to a fixed-point solution, ultimately establishing the complete PAG. Building upon this, Section IV-D will present a plugin framework of ArkTS's advanced language features. Section IV-E will elaborate on hybrid context-sensitivity strategies.

The alloc rule uniformly handles memory allocation operations in programs. It covers two scenarios: object instantiation and function pointer management. For `new` expressions involving classes/interfaces, the type inference module extracts the explicit type T from the statement and generates a corresponding heap object o_i (i is the code line number) . APAK establishes the reference relationship between variables and heap objects through the assignment constraint $o_i \rightarrow v$. In the case of lambda expressions, ArkIR represents them as anonymous method variables (e.g., `anonymous_method_1`). The type system annotates these variables with `FunctionType` while creating special function objects o_i during heap abstraction. When parsing assignments in the form of `let v = anonymous_method_1`, it not only establishes the pointing constraint $o_i \rightarrow v$ but also records the corresponding IR definition node in o_i. This implementation enables bidirectional data flow tracking between lambda expression declarations and their call targets. Such dual-processing mechanisms ensure static analysis can accurately capture memory behavior characteristics under both object-oriented and functional programming paradigms.

Algorithm 1 Propagation algorithm

Require:
`entries` $\leftarrow$ ability and component lifecycle methods collected from project
Ensure:
▷ PAG
▷ CG
1: **procedure** START(workList = $[entries]$)
2: **while** workList not empty **do**
3: INITWORKITEM
4: SOLVECONSTRAINTS
5: SOLVEDYNAMICCALL
6: SOLVEFUNCTIONPOINTERCALL
7: **end while**
8: **end procedure**

The assign rule, the core mechanism for resolving assignment semantics, establishes dataflow edges within the PAG. When processing assignment statements of the form `y = x`, this rule establishes a directed dataflow edge from the right side pointer p_x to the left side pointer p_y (denoted as $p_x \rightarrow p_y$) within the PAG. Concurrently, it enforces pointer set propagation through the set inclusion relationship $pts(x) \subseteq pts(y)$. The load/store rules govern the dynamic resolution of object field access operations. When processing field access statements of the form `x = y.f`, the rules first retrieve the $pts(y)$ of the base pointer y through traversal of the PAG. For each heap object instance $o_i \in pts(y)$, it dynamically generates a corresponding field-sensitive pointer $o_i.f$. This process ultimately constructs dataflow propagation paths from the field-sensitive pointers $o_i.f$ to the left-hand side variable x.

The call rule adopts a phased analysis strategy that employs distinct resolution strategies for three invocation types mentioned before. Static invocations are resolved through direct symbol resolution to determine the target method body. Dynamic invocations `o.m()` dynamically compute the $pts(o)$ of receiver objects during analysis iterations, subsequently resolving concrete instance methods via virtual method table (VMT)[32] lookups. Function pointer invocations require inspection of points-to set, where anonymous functions are retrieved from contained `FunctionType` objects. Algorithm 1 demonstrates how the pointer analysis framework handles these call types through stage-specific resolution logic in APAK.

The pointer analysis propagation algorithm implements an iterative analysis framework, which is organized based on

TABLE III: Basic Pointer analysis propagation rules.

Kind	Operation	Statement	Propagation Rule	PAG Edge
alloc	create object	i: `let v = new T()`	$c : o_i \in pts(c : v)$	
	create function pointer	i: `let v = () => {}`		
assign	variable assign	y = x	$c : o_i \in pts(c : x) \rightarrow c' : o_i \in pts(c' : y)$	$x \rightarrow y$
store	field store	`y.f = x`	$c : o_i \in pts(c : x), c' : o_j \in pts(c' : y) \rightarrow c'' : o_i \in pts(c'' : o_j.f)$	$x \rightarrow o_j.f$
load	field load	`x = y.f`	$c : o_i \in pts(c : y), c' : o_j \in pts(c' : o_i.f) \rightarrow c'' : o_j \in pts(c'' : x)$	$o_i.f \rightarrow x$
call	static call	`y = func(v)`	$f = Dispatch(), c' = [f :: c]$	$v \rightarrow v_{param}$
	dynamic call	`y = x.method(v)`	$c : o_i \in pts(c : v), c' : o_j \in pts(c' : v_{return}) \rightarrow$	$v_{return} \rightarrow y$
	function pointer call	`y = p()`	$c'' : o_i \in pts(c'' : v_{param}), c''' : o_j \in pts(c''' : y)$	

method-level processing units. During initialization, ArkAnalyzer collects lifecycle methods of all abilities and components in `DummyMain` [33] as the project entry method. The algorithm enqueues the project's entry method into the `worklist` as the initial processing target. The analysis subsequently enters a multi-iteration processing loop. For the method m in the worklist:

1) `initWorkItem`: Generates method-local PAG while deferring processing of dynamic method invocations and function pointer invocations and storing them in a pending call queue. Static calls are immediately resolved through symbol resolution to establish interprocedural edges.
2) `solveConstraints`: Iterates over assignment and load-/store statements within the method. It applies pointer propagation rules to transfer and merge points-to sets. The monotonicity of set inclusion relationships guarantees rapid convergence to local fixed-point solutions.
3) `solveDynamicCall`, `solveFunctionPointerCall` : For dynamic method invocations, retrieves VMT through receiver objects' points-to sets to derive concrete target methods. For function pointer invocations, matches executable functions via type signature mapping tables under current context constraints.

Newly resolved methods from call sites are injected back into the `workList`, driving iterative advancement of cross-procedural analysis. When the worklist becomes empty, the iteration terminates and returns the completed PAG and CG.

D. Extensibility for ArkTS Language Features

A core design principle of APAK is its extensibility, which is achieved through a highly modular plugin system in Figure 3. This architecture is crucial to keeping up with the rapidly evolving OpenHarmony ecosystem. Instead of a monolithic design, APAK allows new language features and framework APIs to be supported by adding new plugins, without altering the core analysis engine.

The plugin system is designed to handle API calls to both external libraries (e.g., the OpenHarmony SDK) and TypeScript's built-in libraries. The architecture is illustrated in Figure 3. During the `solveDynamicCall` and `solveFunctionPointerCall` stage, the plugin manager extracts the signature of a call statement and checks for a registered plugin capable of handling that API type. The corresponding plugin then invokes the PAG and CG interfaces

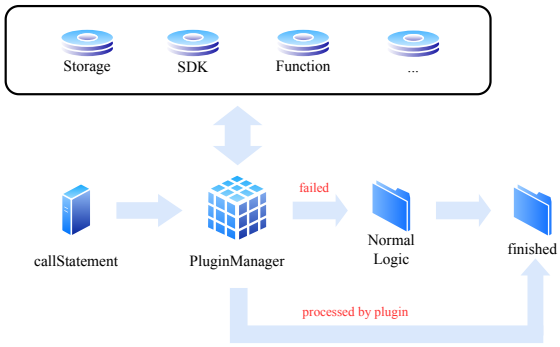


Fig. 3: Plugin System Framework.

according to its specific rules. Finally, it returns the result to the manager, allowing the analysis to proceed to the subsequent statements.

1) AppStorage & LocalStorage Plugin: `AppStorage` and `LocalStorage` are built-in storage mechanisms in ArkTS, offering cross-component data access and state synchronization, albeit with different scopes. `AppStorage` provides an application-level singleton for UI state, while `LocalStorage` is typically scoped more locally (e.g., to a specific component or custom class instance). Despite these scope differences, both employ a similar set of APIs and underlying principles for property storage, as exemplified by `AppStorage` in Listing 1. Developers can use these APIs (like `setOrCreate`, `get`, `Link`, and `Prop`, which are common to both) to register variable instances, function pointers, and other objects into their respective storage contexts. The combination of direct writability to these storage objects and the propagation of references they hold introduces significant challenges for pointer analysis.

Listing 1: AppStorage Usage.

```
function set() {
    AppStorage.setOrCreate(x, 1);
}
function getOrSynchronize() {
    let x = AppStorage.get(x); // x = 1
    let link1 = AppStorage.Link('x');
    let link2 = AppStorage.Link('x');
    let prop = AppStorage.Prop('x');

    link1.set(2); // two-way sync: link1=link2=prop=2
    prop.set(3); // one-way sync: link1=link2=2, prop=3
    link2.set(4); // two-way sync: link1=link2=prop=4
```

```
13      }
```

The *StoragePlugin* employs signature matching to identify storage-related API invocation patterns, activating distinct processing strategies. For storage creation operations, it generates field nodes with isolated storage domains, whose points-to sets maintain the references to initialized objects. To implement differentiated synchronization between `prop` and `link` mechanisms, the system innovatively deploys backflow edge injection. This strategy involves selectively adding dataflow edges to the PAG that run in the opposite direction of typical value propagation (i.e., from a local variable back to a storage location) to model specific data binding semantics, as detailed for `Link` and `Prop` below.

For `link` synchronization, APAK establishes bidirectional edges between parameter variable `p` and `AppStorage.x`, forming Strongly Connected Components (SCC) in the PAG (denoted as $SCC(N, E)$, N represents node set and E edge set). This ensures mutual reference synchronization. For `prop` synchronization, the base strategy is retained. Without backflow edges in the topology, its points-to set only supports unidirectional data flow, preventing local modifications from propagating back to source attribute nodes.

2) OpenHarmony SDK Invocation Plugin: The OpenHarmony Software Development Kit (SDK)[19], as a standardized capability collection for OpenHarmony native applications and meta-service development, provides modular development interfaces covering application frameworks, distributed services, system kernels, multimedia processing, and artificial intelligence. When developers implement business logic by integrating domain-specific development kits (Kits), the actual implementation code of SDK APIs is encapsulated in type declaration files (.d.ets/.d.ts), presenting black-box characteristics to developers. This design prevents program static analysis from constructing call graph models for API internal method bodies through conventional interprocedural analysis, and also hinders the establishment of precise cross-layer invocation models. However, as the core carrier for system-level capability invocation, if SDK APIs are ignored during pointer analysis, pointer propagation chains will break at system boundaries, leading to premature termination of analysis paths. Therefore, *SDKPlugin* is designed to maintain analysis path continuity through predefined pointer propagation rules with specialized object modeling mechanisms.

When *SDKPlugin* detects SDK API interfaces as call targets, it executes return value type verification: when interfaces return types are non-constant references (including class instances, interface types, and other objects with pointer propagation capabilities), these return values serve as potential starting points for subsequent pointer propagation paths. For this purpose, *SDKPlugin* instantiates abstract heap object nodes matching declared types as logical return value proxies for API calls. Although this modeling strategy may introduce abstraction deviations between return values and runtime actual objects, it effectively maintains the integrity of pointer flow propagation chains across system boundaries, ensuring that pointer outflow edges at SDK call points remain valid

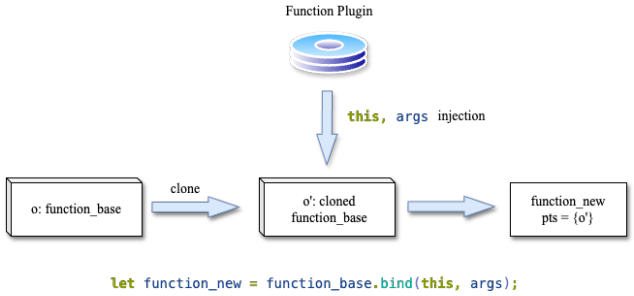


Fig. 4: Context and param injection via Function Plugin

connections.

3) Function Plugin: In TypeScript, `Function` is a built-in global type representing the set of all functions. It can be considered the "top type" for functions, analogous to how any serves as the top type for all other types. Consequently, any function value can be assigned to a variable of type Function. This type provides three core methods—apply, call, and bind, which are instrumental for handling execution contexts (`this`), dynamic invocation, and metaprogramming by allowing for the explicit specification of `this` context and arguments. While this design offers significant flexibility, it introduces substantial challenges for pointer analysis. Therefore, the central task of the *FunctionPlugin* is to accurately model the behavior of these methods and correctly reflect their effects on the PAG.

APAK's standard approach abstracts variables pointing to functions into a specific PAG node type, storing the function as an object model in the node's points-to set. The `Function` type is handled similarly in *FunctionPlugin*. However, the specification of this context and arguments via its methods acts as a form of "secondary processing" on the original function. To address this, the *FunctionPlugin* clones the original function's object model and injects the new contextual information into the clone(Figure 4. This strategy enables customized behavior without altering the original function node. This new, context-aware function node is then immediately invoked for call and apply, or propagated to a new variable in the case of bind for subsequent invocation.

E. Extensibility for Context Sensitivity

During the dynamic execution of a program, the same program point may exist in different runtime context environments [34], [35], [36]. These contextual differences lead to multi-version evolution of data flow states at that program point. To precisely analyze data flow across these varying contextual paths, APAK already implements function-level sensitivity and call-site sensitivity [16], [17], with object sensitivity [37] currently under development.

Furthermore, APAK is designed for the straightforward extension of its context-sensitivity policies. Through the `ContextManager`, users can define custom strategies for generating context elements on demand. The framework's base interfaces support the implementation of custom contexts

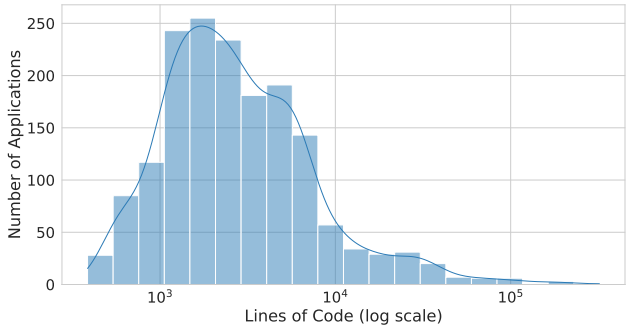


Fig. 5: Distribution of collected Apps in our test set.

with a depth of up to 5, significantly enhancing the overall extensibility of the analysis.

For specific semantic scenarios, APAK uses selective context-sensitive strategies [38], [39], [40]. Taking the global object `globalThis` access scenario as an example: As a cross-context shared global namespace carrier, regardless of how the call stack's context evolves, property access operations on `globalThis` must maintain a consistent view of global state. If context identifiers are forcibly injected at such access points, it would incorrectly split the data flow of the same global property into multiple contextual versions. Therefore, APAK selectively suppresses context injection at special semantic nodes like global object access and singleton pattern invocations through static scope determination rules. This mechanism ensures the analyzer can both maintain context sensitivity for regular object access and properly handle data flow homogeneity requirements under global scope.

V. Evaluation

We conducted comprehensive evaluation of APAK using our self-built ArkTS application dataset, with experimental design focusing on three core research questions to systematically validate APAK's capabilities across effectiveness, efficiency, and practical dimensions:

- RQ1: How precise is APAK in call graph construction?
- RQ2: How efficient is APAK in analysis?
- RQ3: How practical is APAK in resolving real-world problems?

App Dataset: The dataset comprises 1,672 production-grade ArkTS applications collected from the OpenHarmony open-source community, covering applications spanning API versions 9 to 12. It incorporates representative application scenarios including UI component development, cross-device management, and distributed data synchronization, ensuring the evaluation results reflect authentic industrial-scale code characteristics. Its scale is shown in Figure 5.

Experimental Environment: The experimental platform configuration is as follows: Ubuntu 20.04.6 LTS, Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz, 64GB RAM, Node v23.9.0.

A. RQ1: Accuracy

We conducted systematic validation on the dataset using a context length of k=2. Through random sampling, 12 application samples (500-7000 LOC) were selected for manual verification to perform quantitative evaluation of the generated call graphs. The validation was conducted by two software engineering experts. We first established a comprehensive annotation protocol, complete with precise definitions, rules for handling complex call types, and illustrative examples to ensure consistency. Following this protocol, each expert independently inspected the source code and meticulously labeled every call graph edge. Subsequently, their findings were merged, and any initial disagreements were resolved through a brief joint discussion to reach a final, verified consensus.

The validation was performed by two software engineering experts following a systematic protocol.

Experimental data in Table IV demonstrates that APAK achieves breakthrough precision performance: at the call statement resolution level, its average precision metric consistently approaches the theoretical maximum of 100%, showing significant false positive reduction compared to the CHA/RTA methods. Simultaneously, APAK achieves an absolute recall rate improvement exceeding 10%. This enhancement primarily originates from its precise modeling capability for ArkTS-specific programming paradigms such as dynamic closure binding and asynchronous callback chains, enabling successful capture of call relationships missed by the baseline methods and consequent expansion of coverage scope.

Further experimental evidence confirms that APAK's precision advantage directly correlates with enhancements in its type constraint system. The implemented inference rule set, combined with ArkTS's semantic constraints, significantly improves object flow resolution accuracy, particularly in handling complex scenarios including polymorphic dispatch and cross-component communication patterns.

We also conducted full-dataset call edge statistics to compare scale characteristics of call graphs constructed by APAK, the CHA, and RTA methods (as shown in Figure 6). Experimental data reveal that the APAK framework achieves 34.2% absolute increases in total call edges compared to RTA, demonstrating its superior recall in identifying call relationships that RTA misses. Conversely, compared to the sound but imprecise CHA, APAK generates a 7.1% smaller graph. This reduction is a sign of higher precision, as APAK's powerful pointer analysis effectively prunes a significant number of false-positive edges that CHA incorrectly reports.

> These empirical evidences quantitatively demonstrate APAK's completeness advantage in modeling ArkTS language features and validates its effectiveness in call graph construction.

B. RQ2: Efficiency

We implemented multi-dimensional monitoring of runtime performance metrics in APAK through code instrumentation. Figure 7 presents the core performance parameters measured across the application dataset, including per-instance execution time (7a), peak memory consumption (7b), node scale of PAG

TABLE IV: Comparison between APAK and existing static analysis methods for the accuracy of call graph construction.

Application	APAK		CHA		RTA	
	precision	recall	precision	recall	precision	recall
AACommandpackage	**100.0%**	**78.5%**	**100.0%**	70.8%	**100.0%**	52.3%
ActsCleanTempFilesRely	**100.0%**	**80.7%**	91.2%	74.4%	94.2%	68.1%
ActsRegisterJsErrorRely	**100.0%**	**85.9%**	89.3%	51.0%	93.0%	42.1%
AdaptiveServiceWidget	**100.0%**	**84.9%**	75.2%	75.2%	75.3%	56.5%
btmanager_errorcode401	**100.0%**	**92.2%**	97.6%	78.4%	97.4%	72.5%
CustomCommonEventRely	**100.0%**	**94.3%**	94.3%	89.2%	95.9%	78.0%
formsupplyapplicationC	**100.0%**	**92.2%**	94.3%	52.5%	97.1%	43.2%
NdkOpenGL	**100.0%**	63.3%	93.5%	**73.5%**	91.5%	47.8%
PixelConversion	**100.0%**	**75.9%**	95.4%	52.3%	93.9%	39.2%
rpcserver	**98.7%**	**79.2%**	96.1%	51.6%	95.8%	47.4%
server	**100.0%**	**93.6%**	97.1%	43.0%	97.5%	43.8%
TestExtensionAbility_001	**100.0%**	**86.7%**	97.9%	76.7%	97.1%	56.7%

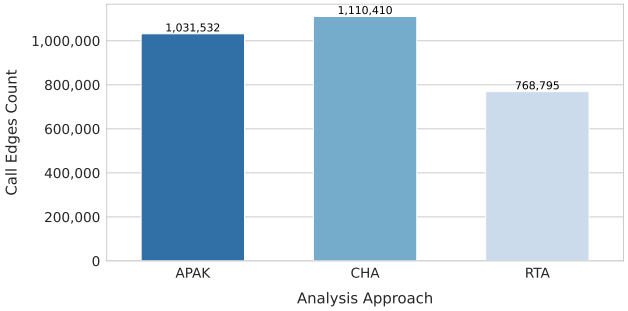


Fig. 6: Call edges scale of different approaches.

(7c), analysis iterator times (7d, reflect the dynamic call depth) and analysis success rate (7e) with k=2 callsite context limit. Within a benchmark suite comprising 1,672 industrial ArkTS applications, the APAK demonstrates a 99.4% analysis success rate (1,662/1,672, $p < 0.001$), confirming its engineering applicability in large-scale codebases.

Experimental results reveal superior runtime characteristics for successfully analyzed instances:

1) Median analysis execution time: 53 milliseconds (IQR: 33-160 ms)
2) 95th percentile peak memory consumption: 284.56 MB
3) PAG node count distribution exhibits stable patterns with median 225 nodes (IQR: 137-1114)
4) 95th percentile pointer analysis iterator times: 11

Experimental observations reveal a discrepancy between theoretical expectations and empirical measurements regarding the impact of codebase size on memory consumption, execution time, and PAG scale. The data demonstrates no statistically significant positive correlation between these performance metrics and application scale. Through reverse engineering and call chain analysis, we identify code structure complexity – specifically manifested in call depth and dynamic invocation frequency – as the primary determinant of framework runtime performance.

We conducted an in-depth investigation into the observed bimodal distribution phenomenon of the analysis time metric. Experimental data reveals that in the first peak region with a time threshold of $<$ 10ms, the code complexity of the sample set is relatively low, and there exists a significant proportion of static analysis failure scenarios (including literal expressions and unreachable paths with specific syntactic structures). In the secondary peak region ($<$ 100ms interval), the call graph structures across samples demonstrate concentration (call depth 2-5), validating a certain correlation between control flow complexity and static analysis latency.

Notably, 67.9% of applications in the test suite exhibit shallow call characteristics (mean call depth: 2.8 ± 0.7 layers). This structural characteristic results in the limited effectiveness of context-sensitive rules within APAK, particularly reducing the precision in polymorphic method resolution and propagation of heap objects.

Furthermore, of the 10 applications that failed during the analysis, 9 failed due to type errors encountered in ArkIR. The remaining application was excluded because its large scale caused the analysis to exceed our 20-minute timeout threshold.

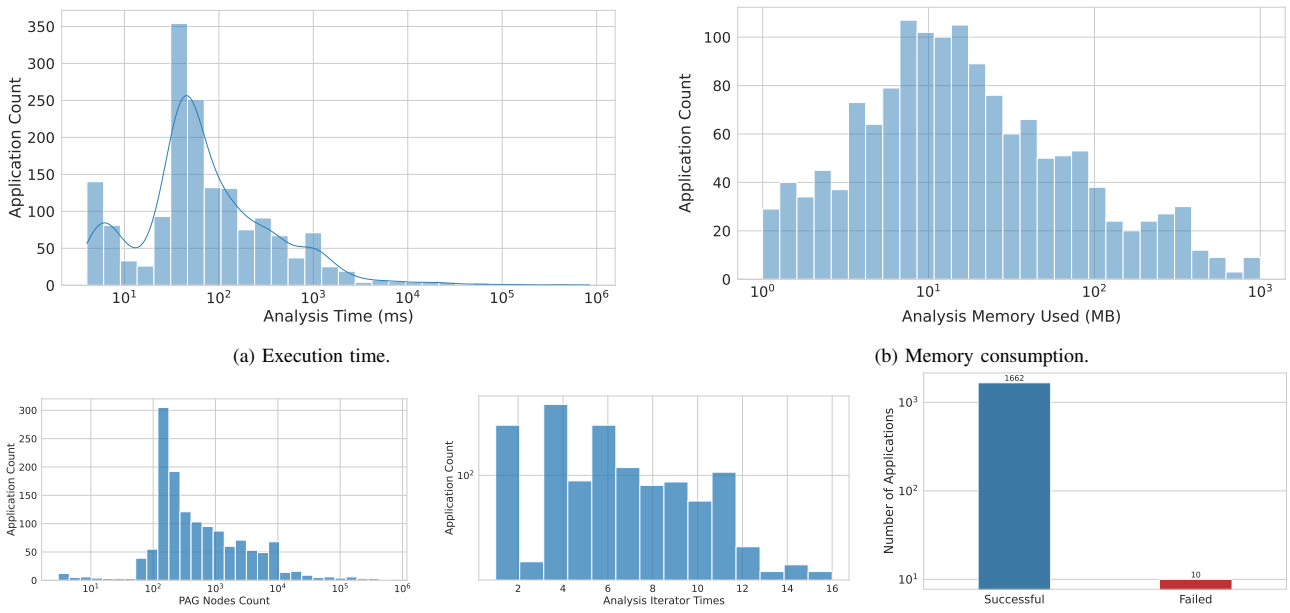
> Experimental results validate APAK's efficiency in industrial-scale analysis and demonstrate stable runtime performance. This confirms its viability as an efficient static analysis framework for real-world OpenHarmony applications.

C. RQ3: Practicality

In this part, we will show the practicality of our proposed APAK by evaluating our method on a large-scale commercial app (1.2 million lines of code).

To further validate the efficiency and effectiveness of APAK, we used a complex commercial application with codebase exceeding 1,206,000 lines as a study case, which can help us evaluate the utility of our proposed APAK in real-world applications. Since we do not have the permission to make this application open-source, we cannot provide any code examples.

Experimental results show that as the callsite context-sensitivity parameter k is incrementally expanded to $k = 3$, the single-analysis time remains within the 370-second to 720-second range with gradual increases, while memory consumption exhibits a growing trend (5.4 GB–6.9 GB), and the PAG node scale demonstrates progressive expansion with configurations of 588,281, 823,122, and 1,067,595 units respectively. Compared to CHA and RTA, the call edge set size shows substantial increases of 40.0% and 54.2% ($k = 2$ limit), respectively. As this app is closed-source and has an exceptionally large scale of call graphs (nodes $\geq 8.2 \times 10^5$ and edges $\geq 2.8 \times 10^5$), we cannot manually verify the accuracy between our method and existing static analyzers. However,



(a) Execution time.

(b) Memory consumption.

(c) Node scales of PAG.

(d) Analysis iterator times.

(e) Analysis success rate.

Fig. 7: Analysis efficiency of APAK on execution times, memory consumption, node scales of PAG, analysis iterator times, and success rates.

the results still demonstrate that our method can achieve a more comprehensive analysis than the existing static analysis methods for OpenHarmony.

Experimental results demonstrate that APAK exhibits promising feasibility in resolving real-world problems. APAK can effectively address practical challenges in OpenHarmony app data flow analysis, achieving cross-procedural data flow propagation that traditional static type systems cannot handle.

VI. Threats to Validity

a) Internal Threats to Validity: The manual verification process for call edge statistics in RQ1 may introduce human validation bias, particularly in higher-order closure invocation scenarios. The experimental design of RQ2 faces inherent constraints due to the execution environment limitations of commercial closed source systems. The exclusive availability of proprietary applications may lead to potential deviations in performance evaluation metrics.

b) External Threats to Validity: The pointer analysis technology stack exhibits deep technical dependencies on the completeness of ArkIR. With the continuous evolution of the OpenHarmony ecosystem and ongoing iterative development of ArkAnalyzer, APAK requires persistent updates to maintain compatibility with rapidly changing technical infrastructures.

VII. Related Work

In C++ ecosystems, the SVF [41], [12] framework specifically designed for LLVM-based languages provides comprehensive support for interprocedural pointer analysis with value flow tracking capabilities. Within Java environments, frameworks including SPARK [42], DooP [10], [11], Qilin [43], and Tai-e [30] establish a complete analytical ecosystem, where DooP further extends its applicability to Android application analysis. For JS/TS analysis, TAJS [44] and Jelly [45] employ pattern matching to optimize call graph construction for JavaScript/TypeScript, achieving improvement in vulnerability detection accuracy compared to conventional approaches. Notably, the ArkAnalyzer [14] targeting OpenHarmony ecosystem implements standardized processing of ArkTS IR, whose architecture integrates modules like call graph construction.

VIII. Conclusion

This paper addresses the accuracy issue of static analysis for ArkTS, a new programming language, by proposing APAK. APAK is the first highly extensible context-sensitive pointer analysis framework specifically designed for OpenHarmony, which achieves higher precision through dual innovation in heap abstraction and API call resolution. Experiments demonstrate that APAK effectively addresses analysis failures and virtual call resolution challenges stemming from ArkUI declarative syntax while maintaining manageable overhead. Future work will focus on enhancing ArkIR parsing capabilities, improving modeling precision for OpenHarmony distributed architectures, and exploring synergistic optimization mechanisms between pointer analysis and taint analysis.

Acknowledgements

This work was partially supported by the National Key Research and Development Program of China (No.2024YFB4506300) and the National Natural Science Foundation of China (No.62572024, No.62502021).

REFERENCES

[1] O. Foundation, "Openharmony: A comprehensive open source project for all-scenario, fully-connected, and intelligent era," https://gitee.com/openharmony, 2024, accessed on: 2024-04-10.

[2] L. Li, X. Gao, H. Sun, C. Hu, X. Sun, H. Wang, H. Cai, T. Su, X. Luo, T. Bissyande, J. Klein, J. Grundy, T. Xie, H. Chen, and H. Wang, "Software engineering for openharmony: A research roadmap," *ACM Comput. Surv.*, Feb. 2025, just Accepted. [Online]. Available: https://doi.org/10.1145/3720538

[3] Y. Chen, S. Wang, Y. Tao, and Y. Liu, "Model-based gui testing for harmonyos apps," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 2411–2414. [Online]. Available: https://doi.org/10.1145/3691620.3695364

[4] T. Ma, Y. Zhao, L. Li, and L. Liu, "Cid4hmos: A solution to harmonyos compatibility issues," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '23. IEEE Press, 2024, p. 2006–2017. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00134

[5] L. ANDERSEN, "Program analysis and specialization for the c programming language," *PhD Thesis, University of Copenhagen*, 1994.

[6] W. Li, D. He, Y. Gui, W. Chen, and J. Xue, "A context-sensitive pointer analysis framework for rust and its application to call graph construction," in *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 60–72. [Online]. Available: https://doi.org/10.1145/3640537.3641574

[7] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, and D. D. Yao, "Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2455–2472. [Online]. Available: https://doi.org/10.1145/3319535.3345659

[8] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and Y. Le Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, 2017.

[9] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[10] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, 2009, pp. 243–262.

[11] Plast-Lab, "Doop: The official repo of doop, the declarative pointer analysis framework." 2025, accessed: 2025-03-06. [Online]. Available: https://github.com/plast-lab/doop

[12] Y. Sui, D. Ye, and J. Xue, "Static memory leak detection using full-sparse value-flow analysis," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: Association for Computing Machinery, 2012, p. 254–264. [Online]. Available: https://doi.org/10.1145/2338965.2336784

[13] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 265–266. [Online]. Available: https://doi.org/10.1145/2892208.2892235

[14] H. Chen, D. Chen, Y. Yang, L. Xu, L. Gao, M. Zhou, C. Hu, and L. Li, "Arkanalyzer: The static analysis framework for openharmony," *arXiv preprint arXiv:2501.05798*, 2025.

[15] V. Kanvar and U. P. Khedker, "Heap abstractions for static analysis," *ACM Comput. Surv.*, vol. 49, no. 2, Jun. 2016. [Online]. Available: https://doi.org/10.1145/2931098

[16] M. Pnueli and M. Sharir, "Two approaches to interprocedural data flow analysis," *Program flow analysis: theory and applications*, pp. 189–234, 1981.

[17] M. Might, Y. Smaragdakis, and D. Van Horn, "Resolving and exploiting the k-cfa paradox: illuminating functional vs. object-oriented program analysis," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 305–315. [Online]. Available: https://doi.org/10.1145/1806596.1806631

[18] D. F. Bacon and P. F. Sweeney, "Fast static analysis of c++ virtual function calls," in *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 324–341. [Online]. Available: https://doi.org/10.1145/236337.236371

[19] HUAWEI, "Hms core," https://developer.huawei.com/consumer/en/hms, 2025, accessed: 2025-3-13.

[20] C. Chambers, J. Dean, and D. Grove, "Whole-program optimization of object-oriented languages," *University of Washington Seattle, Technical Report 96-06*, vol. 2, 1996.

[21] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 49–61. [Online]. Available: https://doi.org/10.1145/199448.199462

[22] M. Someya, Y. Otsubo, and A. Otsuka, "Graph neural network based function call graph embedding for malware classification," *Journal of Surveillance, Security and Safety*, vol. 4, no. 2, 2023. [Online]. Available: https://www.oaepublish.com/articles/jsss.2022.26

[23] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for javascript ide services," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, p. 752–761.

[24] J. Samhi, R. Just, T. F. Bissyandé, M. D. Ernst, and J. Klein, "Call graph soundness in android static analysis," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 945–957. [Online]. Available: https://doi.org/10.1145/3650212.3680333

[25] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav, "Alias analysis for object-oriented programs," *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pp. 196–232, 2013.

[26] W. E. Weihl, "Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables," in *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '80. New York, NY, USA: Association for Computing Machinery, 1980, p. 83–94. [Online]. Available: https://doi.org/10.1145/567446.567455

[27] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Foundations and Trends® in Programming Languages*, vol. 2, no. 1, pp. 1–69, 2015. [Online]. Available: http://dx.doi.org/10.1561/2500000014

[28] T. Tan, Y. Li, and J. Xue, "Efficient and precise points-to analysis: modeling the heap by merging equivalent automata," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 278–291. [Online]. Available: https://doi.org/10.1145/3062341.3062360

[29] F. M. Q. Pereira and D. Berlin, "Wave propagation and deep propagation for pointer analysis," in *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '09. USA: IEEE Computer Society, 2009, p. 126–135. [Online]. Available: https://doi.org/10.1109/CGO.2009.9

[30] T. Tan and Y. Li, "Tai-e: A developer-friendly static analysis framework for java by harnessing the good designs of classics," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1093–1105.

[31] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Foundations and Trends® in Programming Languages*, vol. 2, no. 1, pp. 1–69, 2015. [Online]. Available: http://dx.doi.org/10.1561/2500000014

[32] K. Driesen and U. Hölzle, "The direct cost of virtual function calls in c++," in *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 306–323. [Online]. Available: https://doi.org/10.1145/236337.236369

[33] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 259–269. [Online]. Available: https://doi.org/10.1145/2594291.2594299

[34] M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," *ACM SIGPLAN Notices*, vol. 29, no. 6, pp. 242–256, 1994.

[35] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, ser. PLDI '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 131–144. [Online]. Available: https://doi.org/10.1145/996841.996859

[36] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "Precision-guided context sensitivity for pointer analysis," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276511

[37] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for java," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 1, p. 1–41, Jan. 2005. [Online]. Available: https://doi.org/10.1145/1044834.1044835

[38] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, "Introspective analysis: context-sensitivity, across the board," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 485–495. [Online]. Available: https://doi.org/10.1145/2594291.2594320

[39] J. Lu and J. Xue, "Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360574

[40] J. Lu, D. He, and J. Xue, "Eagle: Cfl-reachability-based precision-preserving acceleration of object-sensitive pointer analysis with partial context sensitivity," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 4, Jul. 2021. [Online]. Available: https://doi.org/10.1145/3450492

[41] SVF-tools, "Svf: Static value-flow analysis framework for source code." 2025, accessed: 2025-03-06. [Online]. Available: https://github.com/SVF-tools/SVF

[42] O. Lhoták and L. Hendren, "Scaling java points-to analysis using spark," in *Compiler Construction*, G. Hedin, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 153–169.

[43] D. He, J. Lu, and J. Xue, "Qilin: A New Framework For Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis," in *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), K. Ali and J. Vitek, Eds., vol. 222. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 30:1–30:29. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2022.30

[44] Cs-Au-Dk, "Tajs: Type analyzer for javascript." 2025, accessed: 2025-03-06. [Online]. Available: https://github.com/cs-au-dk/TAJS

[45] ——, "Jelly: Javascript/typescript static analyzer for call graph construction, library usage pattern matching, and vulnerability exposure analysis." 2025, accessed: 2025-03-06. [Online]. Available: https://github.com/cs-au-dk/jelly