ArkAnalyzer: The Static Analysis Framework for OpenHarmony

Haonan Chen, Daihang Chen, Yizhuo Yang
School of Software
Beihang University
China

Lingyun Xu, Liang Gao
CBG Software Engineering Department
Huawei
China

Mingyi Zhou, Chunming Hu, Li Li*
School of Software
Beihang University
China

Abstract—ArkTS is a new programming language dedicated to developing Apps for the emerging OpenHarmony mobile operating system. Like other programming languages (e.g., Typescripts) constantly suffering from performance-related code smells or vulnerabilities, the ArkTS programming language will likely encounter the same problems. The solution given by our research community is to invent static analyzers, which are often implemented on top of a common static analysis framework, to detect and subsequently repair those issues automatically. Unfortunately, such an essential framework is not available for the OpenHarmony community yet. Existing program analysis methods have several problems when handling the ArkTS code. To bridge the gap, we design and implement a framework named ArkAnalyzer and make it publicly available as an open-source project. Our ArkAnalyzer addresses the aforementioned problems and has already integrated a number of fundamental static analysis functions (e.g., control-flow graph constructions, call graph constructions, etc.) that are ready to be reused by developers to implement OpenHarmony App analyzers focusing on statically resolving dedicated issues such as performance bug detection, privacy leaks detection, compatibility issues detection, etc. Experiment results show that our ArkAnalyzer achieves both high analyzing efficiency and high effectiveness. In addition, we open-sourced the dataset that has numerous real-world ArkTS Apps.

I. Introduction

To support seamless interoperability among different devices, our community invents a new open-source mobile operating system called OpenHarmony, which is operated by the OpenAtom Foundation[1] in China. At the moment, the OpenHarmony ecosystem already has numerous applications [2]. Considering that OpenHarmony is still in its early development stage, it shows great potential and promising future market prospects[3], [4], [5]. However, as an independent all-scenario operating system, OpenHarmony features a brand-new application development paradigm and API that are not compatible with existing applications. Thus, a more user-friendly language, ArkTS, has been introduced to OpenHarmony ecosystem.

In this context, we expect that the various issues (e.g., security, compatibility, performance, etc.)[6], [7], [8] that have been previously encountered by the Android and iOS ecosystems will not be less for OpenHarmony. Hence, the various program analysis approaches proposed to address those issues will also need to be constructed for OpenHarmony. Taking Android as an example, there are numerous advanced program analysis tools that safeguard the Android ecosystem, and the majority of these tools are based on the static analysis framework Soot[9].

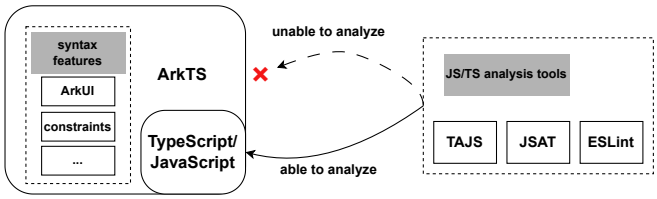


Fig. 1: Existing JS/TS Analysis Tools Inapplicable to ArkTS

Unfortunately, there is no such common static analysis framework available for the OpenHarmony community. As shown in Figure 1, although ArkTS originates from TypeScript, it has introduced a multitude of innovative features, most notably the declarative UI and its accompanying new syntax. The significant differences between ArkTS and TypeScript at the source code level lead to the addition of various new syntax nodes and structures in the Abstract Syntax Tree (AST) and will cause multiple kinds of errors in analyzing ArkTS codes by existing methods. In addition, although ArkTS and TS have some similar features, we do not consider to translate the ArkTS codes to TS codes to enable the program analysis for ArkTS. The rule-based translation approaches are not stable, and the learning-based methods do not have enough training data. As an emerging programming language, ArkTS is in rapid development and will have more and more unique features. Therefore, we need to invent an independent static analysis framework to analyze the ArkTS codes.

To bridge the gap, we present to the community a prototype tool called ArkAnalyzer[1] by providing support for the unique features of ArkTS in program analysis, which implements various software engineering approaches dedicated to scrutinizing OpenHarmony Apps. The imple-

*Corresponding author

[1]For the code related to ArkAnalyzer, please refer to https://gitee.com/OpenHarmony-sig/arkanalyzer

mentation of ArkAnalyzer is adapted to the new features of the OpenHarmony system and the emerging ArkTS language, achieving multi-dimensional analysis. Specifically, we propose our Code Representation module and Code Transformation module to address the definition mismatch and structure mismatch problems in existing program analysis methods. In addition, we collect the extra constraints of ArkTS and solve them in our ArkAnalyzer. The experiment results show that our method has high efficiency (within 10 seconds for analyzing a call graph of an App with thousands of lines of code) and effectiveness (93.75% of accuracy in CHA and 87.95% of accuracy in RTA).

The main contributions of our study are summarized as follows:

1) we provide a comprehensive analysis and empirical evaluations of ArkTS, the new programming language for developing native applications on HarmonyOS, to identify the challenge in analyzing the ArkTS by existing methods.
2) We propose and open-source a novel static analysis framework, ArkAnalyzer, which addresses the problem of analyzing the ArkTS program with unique syntax nodes and structures in AST.
3) We open-source a dataset of ArkTS Apps to the community, which was collected from three official OpenHarmony repositories [2]. We manually select high-quality Apps and repositories to improve the comprehensiveness and quality of evaluation results.
4) We conducted a comprehensive evaluation of ArkAnalyzer. It demonstrates that the Intermediate Representation (IR) generated by our method is highly readable and our tool is efficient and effective.

II. Background: ArkTS vs. TS

In order to benefit from the existing ecosystem of TypeScript (TS), which has gained a large number of libraries, ArkTS attempts to retain as many features as possible when extending the TypeScript language. Nevertheless, in order to support a high-performance experience that is essential for Apps running on mobile devices, ArkTS has to make some changes compared to its original design. Specifically, there are two major kinds of unique features: (1) adding new features required by mobile Apps like ArkUI, and (2) constraining the flexibility of TypeScript, primarily its dynamic features that could impact execution performance. We now detail these two types, respectively.

1) ArkUI: As a declarative UI framework, compared to the traditional procedural and imperative UI approaches, ArkUI focuses on the outcome of the UI description. It binds the UI to reactive data, which is more efficient as developers only need to concentrate on data management. Additionally, the declarative UI offers a declarative description akin to natural language, making it more intuitive. The industry has chosen declarative UI as the new generation model for application development and has undertaken a corresponding restructuring of the underlying UI component design to accommodate this paradigm shift.

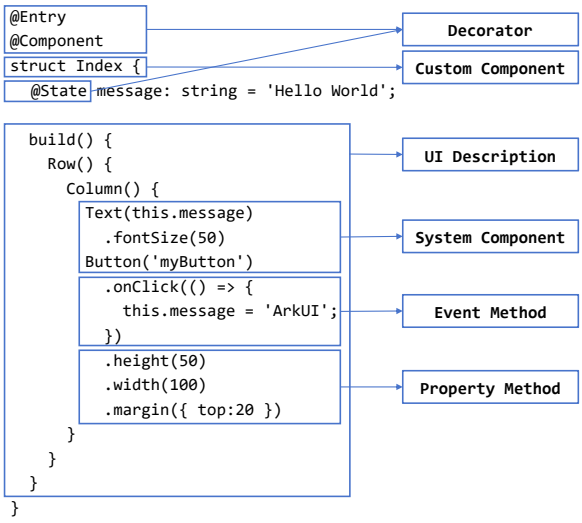


Fig. 2: ArkUI Code Example.

As previously mentioned, the new syntactic structures introduced by ArkUI are one of the primary reasons why traditional JS/TS analysis tools cannot effectively analyze ArkTS applications. In order to help understand, we illustrate the components of ArkUI through a simple ArkUI code example, as shown in Figure 2. Decorator features play a pivotal role, with elements like @Component marking custom components, @Entry specifying entry components, and @State indicating dynamic state variables that prompt UI updates upon modification. The UI Description is systematically defined within the build() method, detailing the UI's structural elements in a clear, declarative manner. Custom Component refers to reusable UI blocks, such as the Index structure, which can incorporate other elements and is designated by the @Component decorator. System Component includes fundamental and container components built into the framework, like Column, Text, Divider, and Button, offering readily accessible tools for developers. Property Method and Event Method allow for detailed customization and interaction handling within components; for instance, property methods like fontSize(), width(), height(), and backgroundColor() adjust visual aspects, while event methods such as onClick() facilitate user engagement strategies. This architecture not only simplifies the development process but also enhances the functionality and interactivity of the application interfaces.

2) Syntactic Constraints.: ArkTS specification imposes constraints on overly flexible features in TypeScript that can affect development correctness or introduce unnecessary overhead during runtime. Even though these con-

[2]The dataset is open-sourced on https://bhpan.buaa.edu.cn/anyshare/zh-cn/link/AA5F769683A23C43B0A4D384D70C7505EB?_tb=none

straints may not cause TypeScript analysis tools to throw errors when analyzing ArkTS code, as differences between the two languages, they are likely to affect the accuracy of the analysis results. For clearly presenting such Syntactic Constraints, here are some examples: (1) any Type. ArkTS mandates the use of static types to improve code clarity and performance. For example, it prohibits the use of the any type , encouraging explicit type definitions that can be analyzed at compile time for correctness. (2) Object Layout. ArkTS does not allow changes to an object's structure at runtime, such as adding or deleting properties, to optimize runtime performance and predictability. (3) Operator Semantics. ArkTS restricts certain operator semantics to encourage clearer code and avoid runtime overhead, such as disallowing the unary + operator on non-numeric types. (4)Structural Typing. Unlike TypeScript, which supports structural typing allowing objects with the same shape to be considered of the same type, ArkTS requires explicit declarations, enhancing type safety and consistency.

III. Preliminary Study

Recall that the ArkTS language is extended from the widely used TypeScript (hereinafter referred to as TS) programming language. When exploring the feasibility of static analysis for ArkTS, we would like to first explore if existing JS/TS static analysis tools can be directly applied to analyze ArkTS code. If so, there is no need to specifically develop a static analysis framework for the ArkTS language.

A. JS/TS Analyzers Identification

To delve into the aforementioned question, we first conduct an exploratory study to identify mainstream JS/TS static analysis tools. We choose three tools that have been widely used in academic papers or industry: TAJS, JSAI, and ESLint.

- TAJS (Type Analysis for JavaScript) [10] is a static program analysis tool designed to provide detailed and precise type information for JavaScript programs. TAJS not only detects common programming errors but also performs type inference and generates call graphs, among other analyses.
- JSAI [11] is another JavaScript static analysis platform that implements a range of analysis capabilities such as type inference, pointer analysis, control flow analysis, and constant propagation.
- ESLint[12], a powerful and highly pluggable JavaScript code-checking tool, is currently one of the most widely used JS/TS code analysis tools. ESLint supports modern JavaScript (ECMAScript) features and can integrate with various editors and build tools, thus enhancing development efficiency and code consistency.

B. Dataset

To support the preliminary study, we need to form a real-world dataset. We collect Apps from three official OpenHarmony organization repositories: the OpenHarmony repository [13], OpenHarmony-SIG [14], and OpenHarmony-TPC [15]. The main repository is the core codebase of the OpenHarmony project, containing the fundamental components of the operating system and serving as the primary interaction and contribution point for developers and contributors. The OpenHarmony-SIG repository supports specific interest groups (SIG), responsible for managing development in particular technical areas such as the graphics subsystem and the device driver subsystem. The OpenHarmony-TPC repository focuses on collecting and maintaining third-party open-source libraries, facilitating access for developers, and ensuring compliance with open-source standards.

It is important to note that the dataset used in this study does not include all applications, but rather underwent a selection process. We only select applications where the repository has more than 10 stars and the number of lines of ArkTS code exceeds 100, ensuring that the dataset consists of applications with a certain level of quality. As of April 10, 2024, the collected dataset includes 371 OpenHarmony repositories, 100 OpenHarmony-SIG repositories, and 147 OpenHarmony-TPC repositories. Ultimately, we collected 618 OpenHarmony applications.

C. Results

Our results indicate that the existing tools (i.e., TAJS, JSAI, and ESLint) are entirely incapable of analyzing ArkTS applications comprehensively. Among the three tools we tested, none were able to fully analyze any of the applications without encountering errors.

Upon reviewing the specific error descriptions, we observed that existing tools might be successful in analyzing code that does not deviate from standard TypeScript. However, when attempting to analyze code that incorporates new features unique to ArkTS, such as ArkUI and extra constraints, the tools produced "Parsing error" messages. Our collected applications have 7199 ArkTS code files. 3601 and 3138 files cannot be analyzed by TAJS and ESLint due to the ArkUI-related problems, respectively. 3585 and 2914 files cannot be analyzed by TAJS and ESLint due to the extra constraint problems, respectively. For JSAI, it even cannot successfully analyze any of the ArkTS code files, and its identical error messages prevent us from classifying the causes of the errors. Indeed, static analysis techniques are usually sensitive to programming languages, as different languages have different syntax rules, different semantics, and different language features.

Finding of the Preliminary Study

Existing JS/TS-based static analyzers cannot be applied to analyze OpenHarmony Apps. There is hence a strong need to design and implement dedicated static analyzers for OpenHarmony.

IV. Methodology

In this section, we will detail our solution ArkAnalyzer.

A. Motivation

According to the results in Section III-C, we need to invent new methods to address the problems in analyzing the ArkTS using existing analyzers. Before presenting our method, we first analyze the errors in existing analyzers:

- Failure by ArkUI - Definition Mismatch: As a new programming language, ArkTS defines new declaration keywords such as struct with a unique internal structure. Those structures do not exist in TypeScript, so analysis tools cannot recognize them. In addition, ArkUI allows using a number of decorators without prior definition (e.g., @Entry, @Component, @State in Figure 2), which is not permitted in TypeScript and would cause errors in the existing compilers and analysis tools. We call this kind of error Definition Mismatch. To address it, in the Code Representation module of our approach, ArkAnalyzer enable the modeling of mismatched definitions within ArkTS using a newly designed AST.
- Failure by ArkUI - Structure Mismatch: ArkUI contains nested system components (e.g., Row() and Column() in Figure 2), which are used in a way similar to the structure of function declarations, but without the 'function' keyword, and allow continuous built-in function calls at the end of the component. Therefore, the function structures of some ArkTS programs are not compatible with the existing analyzers which are designed for analyzing TypeScript's syntax rules. We call this kind of error Structure Mismatch. To this end, we propose a Code Transformation module to simplify the ArkTS code and support the analysis of the mismatched function structures.
- Extra Constraints: As we mentioned in the Background, compared with TypeScript, ArkTS uses extra constraints to optimize the development correctness and runtime overhead. These extra constraints such as Any Type, Object Layout, Operator Semantics, and Structural Typing (see Section II-2) are not compatible with existing program analysis methods for TypeScript. Because we fix the extra constraint errors in an ad-hoc manner, we omit them in this paper.

Therefore, we design and implement in this study a prototype tool called ArkAnalyzer, which aims at bridging the gap between the existing program analysis methods and ArkTS. Our method can remove the aforementioned errors while having high analyzing efficiency for ArkTS.

B. Overview of ArkAnalyzer

We now briefly introduce the core functions included in the ArkAnalyzer framework. As shown in Fig. 3, ArkAnalyzer by itself is a framework dedicated to facilitating the implementation of App analyzers such as tools for detecting the usages of sensitive APIs or characterizing Null-pointer issues. Inside ArkAnalyzer, the input App code will be handled in two layers, with the bottom layer responsible for basic analyses and the upper layer for more advanced analyses.

Specifically, in the bottom layer, ArkAnalyzer starts with the AST generated by the ArkTS compiler to model the application source code, and then transforms and augments the code to facilitate subsequent analysis. Then, in the upper layer, ArkAnalyzer leverages the outputs of the first layer to represent the App code with more advanced data structures (such as call graphs and inter-procedural data flows).

We now detail these modules to help readers better understand the design of ArkAnalyzer.

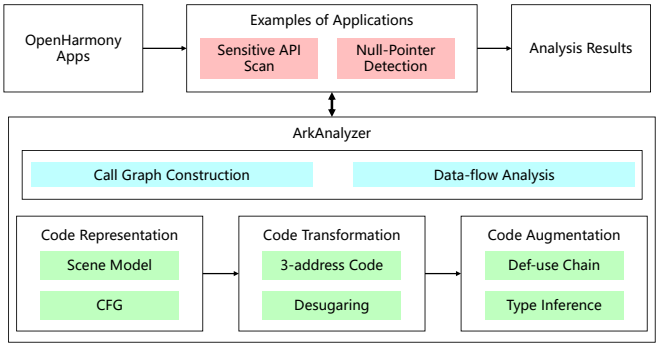


Fig. 3: Overview of the Design of ArkAnalyzer.

C. Code Representation

In ArkAnalyzer, We employ a newly designed AST specifically tailored for ArkTS analysis. This AST is a product of the ArkTS compiler, designed to accommodate the new features of ArkTS and support the modeling of ArkUI code segments. In each analysis, Scene serves as the entry point and contains comprehensive information about the project. It is designed to provide a unified context environment, enabling access to and manipulation of various program details during the analysis process. Figure 4 illustrates the core classes managed by the Scene model. We now detail the representative ones.

ArkFile represents each individual file, simplifying the management of project files. In the context of the ArkTS language, ArkNamespace object is designed to encapsulate the information and structure within a namespace. This facilitates access to and handling of classes and methods within the namespace scope, maintaining the logical organizational structure of the code. Given that ArkTS supports object-oriented programming, the analysis of object-oriented structures is essential. ArkClass object represents a class in the object-oriented paradigm, encapsulating internal structural information such as attributes and methods. Methods and Fields of a class are abstracted into ArkMethod and ArkField classes, respectively.

To address the definition mismatch problem, we abstract struct as ArkClass because it encompasses its own properties and functions, bearing similarities to class in

structure. The ArkClass corresponding to struct will have specific identifiers and special properties, such as viewTree, which represents its corresponding ArkUI component tree. Through the component tree, it is possible to deduce which components the struct uses and the composition relationships between them. Additionally, we have introduced an abstract class Decorators to correspond to the extensive use of decorators in ArkUI. Each namespace, class, method, and field can obtain their corresponding decorators through specified interfaces.

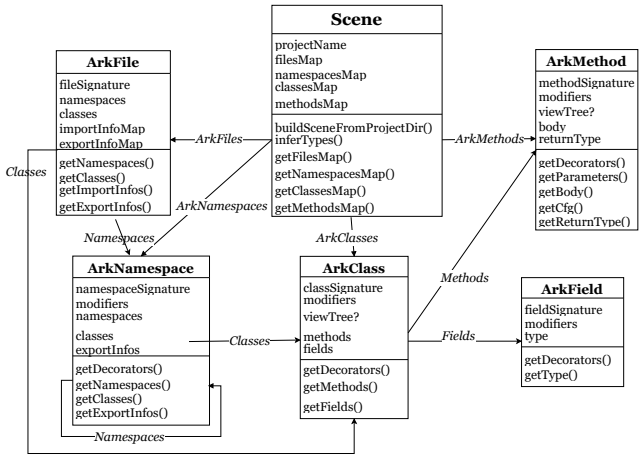


Fig. 4: The design of the core classes designed for representing code under analysis.

As shown in Figure 5, the actual code of a given method (i.e., ArkMethod) will be recorded in a so-called ArkBody class, which is further represented via two Control Flow Graphs, namely OriginalCfg and Cfg. Cfg is the simplified version of the OriginalCfg, which represents the control-flow graph built based on the original code of the method. Each Cfg is composed of several BasicBlock, and each BasicBlock contains a series of sequentially executed lines of code (i.e., without branches). In this work, each line of code is recorded via a Stmt class.

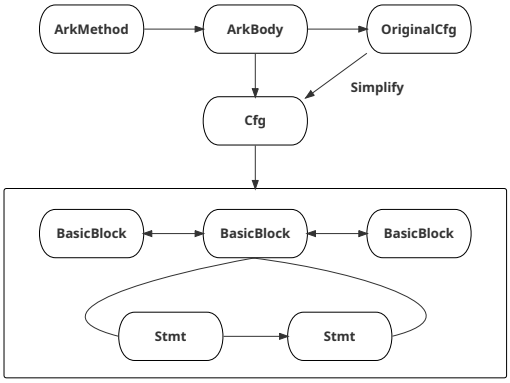


Fig. 5: The design of the ArkMethod class.

D. Code Transformation

After code representation, we leverage a code transformation step to mitigate the structure mismatch problem that may cause difficulties when performing existing analyzers to ArkTS. The ArkTS compiler directly transforms the source code into bytecode[16]. Although intermediate code, such as Panda IR, can be obtained through disassembly tools, it is more bytecode-oriented and lacks readability, which contradicts ArkAnalyzer's design philosophy of high readability and user-friendliness. Therefore, we need to design our own form of IR and establish the corresponding transformation rules.

Specifically, we take two approaches to transform the ArkTS code: (1) Change the code to align with the three-address form, and (2) Transform the code to mitigate certain features such as removing loops, naming anonymous classes or functions, transforming system components into regular code form, etc. We now detail these two approaches, respectively. Thus, our simplified code can be handled by the following analyzing steps (i.e., Section IV-E, IV-F).

1) Three-address Code.: Three-address code is a common intermediate code representation format, for which each line of code is ensured to have at most three operands (addresses). In addition to basic arithmetic expressions, syntactic constructs such as object property access, function calls, and array indexing also need to be converted into three-address code. Converting source code to three-address code has significant benefits for program analysis. Indeed, Its simple and uniform format simplifies the program structure, making it easier to handle and analyze.

In ArkAnalyzer, some of the representative conversion rules transforms source code to three-address format are highlighted in Table I. Given a complex statement, we will divide it into several simple statements. To do so, we will create temporary variables to bridge these simple statements (i.e., keep the same code semantics). Regarding complex function call expressions (including expressions as arguments, nested calls, and call chains), ArkAnalyzer will progressively break them down in the order of execution. The result of each step is stored in temporary variables. By applying these rules, complex code is transformed into a more manageable three-address code format, laying the groundwork for further optimization and analysis.

2) Code Desugaring: After representing the code to three-address format, we go one step deeper to further simplify the code by conducting a desugaring phase. Syntactic sugar refers to the addition of certain syntax features in a programming language that make the code more concise and readable without changing the language's functionality. These features typically simplify common programming patterns, allowing programmers to express the same logic in a more straightforward manner. However, those syntactic sugars, although being more friendly to developers, do make code analyses more complex. To that

TABLE I: Rules applied to achieve three-address code.

No	Rule	Before	After
1	Complex Expression	x = a.b + c.d	temp1 = a.b temp2 = c.d x = temp1 + temp2
2	Expression Parameter	x = fun(a + b)	temp1 = a + b x = fun(temp1)
3	Nested Function Call	x = funA(funB())	_ret = funB() x = funA(_ret)
4	Subscript Operation	x = funA()[1]	_ret = funA() x = _ret[1]
5	Call Chain Splitting	f1().f2().f3()	x = f1() y = x.f2() z = y.f3()

end, towards preventing syntactic sugar from hindering code analysis, we perform a code desugaring phase by transforming code using syntactic sugar into a semantically equivalent form.

Table II highlights some of the representative transformation rules adopted by ArkAnalyzer. First, increment operators (like i++) and compound assignments (like i /= 5) need to be converted to standard assignment operations. Template strings, using string interpolation, should be transformed into string concatenation operations. Arrow functions and Anonymous functions should be converted into regular function expressions. Object literals should be converted into explicit class definitions and instantiations. For better supporting the representation of control flows, we also take the opportunity to simplify the code by transforming if-else and loop statements into structures with explicit labels and jumps. These transformations standardize the code, making it easier for subsequent analyses.

The ninth row in the table demonstrates ArkAnalyzer's handling of nested system component code within ArkUI, which can cause the structure mismatch problem in existing analyzers.ArkAnalyzer maps each system component to the corresponding interface in the OpenHarmony SDK. First, each component is associated with its corresponding create function and pop function. Then, subsequent function calls on the component are applied to the temporary variable returned by the create function. In this way, the special function in ArkUI (the build function of the struct) is transformed into a regular code format, resolving the structure mismatch issue.

E. Code Augmentation

The code representation and transformation steps have greatly reduced the complexity of the code under analysis. However, there is still common information that is constantly required by follow-up analyzers but is not yet available in the current code representation. To further facilitate the implementation of App analyzers, we add another step to ArkAnalyzer to further augment the code. Specifically, we pre-calculate data-flow information for each method by building a def-use chain and the type information for local variables based on a set of pre-defined

TABLE II: Transformation rules applied to simplify code at the IR level.

No	Rule	Before	After
1	Increment Operators	i++	i = i + 1
2	Compound Assignment	i /= 5	i = i / 5
3	Template Strings	greet = !${name}!	temp1 = name + '!' greet = '!' + temp1
4	Arrow Function	fun = (x) => x + 1	def Anonymous_1(x) return x + 1 fun = Anonymous_1

5	Anonymous Function	set(fun() {
		  ...
		}, 1)
			def Anonymous_1():
			  ...
			set(Anonymous_1, 1)

6	Anonymous Class	let x = {name: 'a'};	class Anonymous_1{
			  name: string
			}
			x = new Anonymous_1()
			x.name = 'a'

7	Control Flow (if)	if (x > 0)
		  x++
		else
		  x--
			label1 :
			  if (x > 0)
			  goto label2 label3
			label2:
			  x++
			  goto label4
			label3:
			  x--
			  goto label4
			label4:
			  //following statements

8	Control Flow (while)	while (x > 0)
		  x--
		console.log(x)
			label1 :
			  if (x > 0)
			  goto label2 label3
			label2:
			  x--
			  goto label1
			label3:
			  console.log(x)

9	System Component	Row (){
		  Column (){
		  }.height(100)
		}
			temp0 = RowInterface.create()
			temp1 = ColumnInterface.create()
			temp1.height(100)
			ColumnInterface.pop()
			RowInterface.pop()

rules (more advanced.[3]) We now detail these two substeps, respectively.

1) Def-use Chain: Data flow analysis is used to track the path from the definition of a variable to its usage within a program. The primary technique frequently adopted by our community to record such data dependencies in the program is to build the so-called def-use chains. The chains are considered important for optimizing compilers, code refactoring, detecting potential errors, and identifying vulnerabilities. Analyzing these chains within a single program or function can help understand how local variables are initialized and utilized, thereby ensuring the correctness and efficiency of data flow.

2) Type Inference: Compared to TS, ArkTS imposes stricter type restrictions but still supports implicit type declarations. ArkAnalyzer has formulated a series of rules for analyzing code statements to infer the type information of variables and other syntactic elements in the code[17]. ArkAnalyzer conducts a comprehensive scan of code statements within a project, initially extracting type information from individual statements and assigning it to corresponding variables. If direct inference is not possible, type propagation will be carried out based on contextual information.

[3]At this stage, only lightweight analyses are considered for the sake of performance, i.e., data-flow analysis is limited within methods, type analysis is implemented without leveraging points-to analysis. More advanced analyses are also supported by ArkAnalyzer but are at later stages.

TABLE III: Rules applied to infer types.

No	Rule	Pattern	Type of x
1	Compare	x = a op b, op ∈ Eq, NotEq, Lt, LtE, Gt, GtE, Is, IsNot, In, NotIn	bool
2	BinOp	x = string * number x = bool * number	str bool
3	Heap Object Create	x = new ClassA()	ClassA
4	Return	x = func()	func's return type
5	Field Reference	x = ClassA.field	ClassA.field type

The specific rules leveraged in this step are listed in Table III. Calculations and comparisons between simpler primitive types can directly determine the result's type. We also determine the declared class based on the literal following the *new* keyword. Additionally, by referencing the declarations of corresponding classes, methods, and properties, we parse the types of the respective components within the statement.

F. Call Graph Construction

Call graph is a fundamental data structure that is required by many analysis tasks and is essential to support project-wide analyses. Call graph generally represents the relationships between method invocations within the program. In a given call graph, nodes represent methods, and directed edges signify the calling relationships initiated by the caller method pointed to the callee method. In this section, we will discuss the core algorithms adopted by ArkAnalyzer for call graph construction. We have implemented two algorithms: (1) Class Hierarchy Analysis (CHA) and (2) Rapid Type Analysis (RTA)[18], for which we will detail them respectively in this section.

1) CHA: Class Hierarchy Analysis: ArkTS is an object-oriented programming language. To support the object-oriented features, ArkAnalyzer organizes the project under analysis in the form of classes, and their inheritance relationships are recorded, which is referred to as a class hierarchy tree.

The CHA algorithm builds the call graph by parsing the invocation statements within the code to identify basic invocation relationships and form the call graph, e.g., building an edge from method m_1 to method m_2. It then augments the call graph by adding new edges based on the aforementioned hierarchy tree.

Listing 1: Example code snippet for demonstrating the principles of constructing call graphs.

```
1  function makeAnimalSound(animal: Animal) {
2      animal.sound();
3  }
4
5  function main() {
6      let dog = new Dog();
7      let cat = new Cat();
8      makeAnimalSound(dog);
9  }
```

Taking Listing 1 as an example, which illustrates a code snippet (omitting related class definitions), and the actual class hierarchy is presented in Figure 6a, where all classes have sound method. In this case, when Animal.sound is called, ArkAnalyzer will add three new edges (i.e., *makeAnimalSound* → *Dog.sound*, *makeAnimalSound* → *Cat.sound*) and *makeAnimalSound* → *Cow.sound*) to the call graph in 6b because these three edges could also be true due to the polymorphic characteristic, one of the core features adopted in the object-oriented concept. Observant readers may have already noticed that the CHA algorithm offers a call graph that is as comprehensive as possible, attempting to record all the possible calling relationships. This, however, will unavoidably introduce incorrect edges that subsequently would lead to false positive results for downstream analyzers.

2) RTA: Rapid Type Analysis: Building on the CHA, the RTA algorithm imposes certain constraints to filter potential calling relationships, thereby reducing the over-approximations inherent in CHA. During the construction of RTA, the actual creation of heap objects (i.e., instances created via the new keyword) is tracked and recorded. Upon encountering a method call statement and identifying potential call targets from the class hierarchy, RTA uses whether the class of the call target has been modeled as a criterion. It removes all methods from the call target that have not been modeled, ensuring that only relevant methods are considered. Given the same code snippets shown in Listing 1, since only classes Dog and Cat are instantiated (i.e., class Cow is not instantiated), the edge *makeAnimalSound* → *Cow.sound* will be removed by the RTA algorithm(cf. Figure 6c), resulting in preciser call graph compared to that built by the CHA algorithm.

V. Evaluation

We evaluate the efficiency and accuracy of ArkAnalyzer while exploring the readability of the IR[4] designed by ArkAnalyzer and the capability of supporting the implementation of advanced analyzers. Here are the details of the dataset and experiment environment:

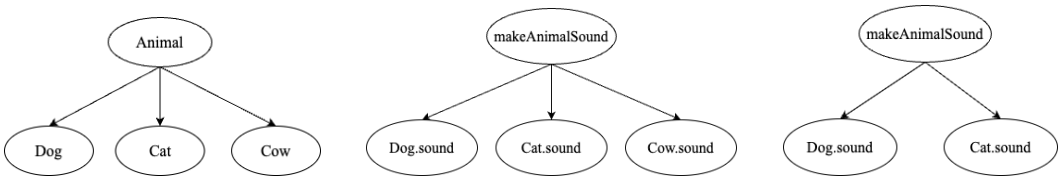
a) Dataset: Recall that we have formed a dataset of 618 Apps when performing the preliminary study, as discussed in Section III-B. In that dataset, we have endeavored to collect and select all the high-quality OpenHarmony Apps that are available to the public. In this section, we reuse this dataset for evaluation.

b) Experiment Environment: Our method is evaluated on a workstation with Intel(R) Core(TM) i7-14700KF CPU, 16GB of RAM and the 64-bit Windows 11 OS.

A. Efficiency of ArkAnalyzer

The performance of static code analysis tools is crucial, as they must provide feedback as quickly as possible while maintaining analytical accuracy[19], especially in Continuous Integration/Continuous Delivery (CI/CD) pipelines. As previously mentioned, the Scene is the core structure of

[4]Readability of static analyzer's IR is considered very important as developers often need to read the IR to debug the analyzer (e.g., to understand its behavior).



(a) Part of Class Hierarchy (b) CG By CHA (c) CG By RTA

Fig. 6: Call graph construction.

ArkAnalyzer, and all analyses depend on the construction of the Scene. Therefore, to evaluate its performance, we tested the time required to construct Scenes for all applications in the dataset. Additionally, we measured the time taken for call graph analysis to demonstrate the high performance of ArkAnalyzer.

Figure 7a shows the result of scene build time distribution. The majority of scene build times are concentrated between 0.4 and 0.5 seconds. A small portion of more complex scenes take longer to build, but all are built within 1 second. Figure 7b and Figure 7c demonstrate the performance of ArkAnalyzer during CHA and RTA analyses. Due to significant variations in some data, we applied a logarithmic transformation to both the x and y axes to enhance the clarity of the pictures. It turns out that whether using CHA or RTA, analysis of applications within a thousand of lines of code can be completed within 1 second, and analysis of applications with thousands of lines of code can be completed within 10 seconds. This result demonstrates the efficiency of call graph analysis.

These experimental results demonstrate that ArkAnalyzer exhibits excellent performance in application analysis, with extremely high code processing efficiency, fully reflecting its effectiveness and stability in handling applications of varying scales.

B. Accuracy of ArkAnalyzer

For the sake of simplicity, we validate the overall accuracy of ArkAnalyzer by testing the accuracy of the call graph module, which is considered one of the most crucial feature for program analysis tools.

TABLE IV: the accuracy of ArkAnalyzer in analyzing call graph

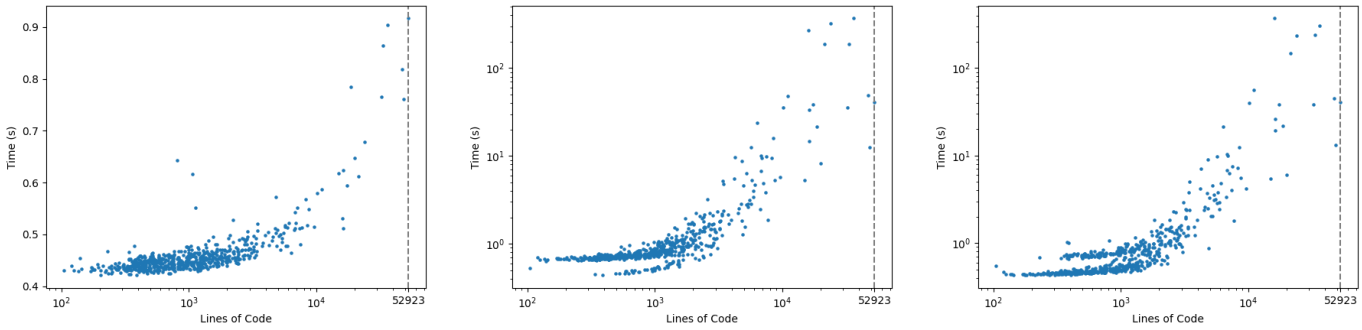
Dataset	Algorithm	TP	All	Precision	Recall
Benchmark	CHA	80	80	96.39%	100%
	RTA	78	80	100%	97.50%
Real Apps	CHA	351	375	99.72%	93.75%
	RTA	332	375	99.70%	87.95%

We conducted experiments on two datasets: one consisting of a series of benchmark test sets that we specified, and the other comprising randomly selected samples from the dataset of real HarmonyOS applications mentioned in Section III-B. For each dataset, we performed call graph analysis using both CHA and RTA. The tests were conducted at the level of call chains, where we compared the call chains obtained by ArkAnalyzer with those manually verified, calculating precision and recall. The results are shown in Table IV.

For the Benchmark dataset, the CHA algorithm achieved a precision of 96.39% and a recall of 100%, correctly identifying all 80 true positives (TP) out of 80 total calls. On the same dataset, the RTA algorithm yielded a precision of 100% and a recall of 97.50%, with 78 true positives correctly identified out of 80 calls. The CHA's strategy is to consider all methods with the same name in subclasses as potential call targets when the invoked method is identified within a calling statement and when the calling object has subclasses. This approach results in false positives in benchmark testing sets for the CHA algorithm, and RTA apply a stricter type check to avoid false positives. For the Real Apps dataset, CHA achieved a precision of 99.72% and a recall of 93.75%, identifying 351 true positives out of 375 total calls. The RTA algorithm on this dataset produced a precision of 99.70% but a slightly lower recall of 87.95%, correctly identifying 332 true positives out of 375 calls. Both CHA and RTA achieved high precision. However, in certain complex invocations or specific method calls(function pointers and rare instances of HarmonyOS SDK calls), the algorithm may fail to accurately locate method declarations, resulting in false negatives. Furthermore, the type checking employed by the RTA can lead to the incorrect exclusion of certain method calls. Overall, the results indicate that both algorithms performed well, with CHA showing slightly higher precision and recall results than RTA, particularly on the real applications dataset.

C. Readability of ArkAnalyzer-IR

To evaluate the readability of intermediate representation (IR) in ArkAnalyzer, we designed and implemented a questionnaire survey. We searched for participants based on their expertise in programming and ArkTS. The 17 participants' programming experience ranges from one year to eight years. Among them, six participants are experts of ArkTS while others only have few ArkTS skills. The questionnaire included six rating items focused on combination operations, conditional branches, arrays and loops, function calls, anonymous functions, and a composite score. The rating scale employed a five-point system, where 1 indicated "very difficult to understand" and 5 indicated "very easy to understand." Additionally, the questionnaire featured a non-mandatory open-ended question to gather specific feedback from respondents on



(a) Time of Scene Construction (b) Time of Scene and CHA Construction (c) Time of Scene and RTA Construction

Fig. 7: ArkAnalyzer Analysis Performance Scatter Plot

challenging aspects of the intermediate code. A total of 17 valid responses were collected, and the average scores for the rating items are presented in the Table V.

TABLE V: Average Scores for Readability of Intermediate Code.

Item	Average	Median	Min	Max
Combination Operations	4.82	5	4	5
Conditional Branches	4.37	5	1	5
Arrays and Loops	3.71	4	2	5
Function Calls	3.65	4	1	5
Anonymous Functions	4.00	5	2	5
Composite Score	4.06	4	3	5

The results reveal differences in readability among various types of intermediate code. Specifically, combination operations received the highest average score, indicating their clear structure and ease of understanding. In contrast, arrays and loops, as well as function calls, had lower scores, respectively, likely due to their complexity and the inclusion of extraneous and redundant information, as further confirmed by the open-ended responses. Both the average and median of the composite scores reached 4, suggesting that most intermediate code performs well in terms of readability.

D. Capability of ArkAnalyzer

To demonstrate the the practical utility of ArkAnalyzer, we now present two concrete App analyzers that are implemented on top of ArkAnalyzer. These two examples are selected because of their simplicity (with only a few lines of code). The ArkAnalyzer by itself is designed to be as generic as possible and thereby it should be able to support the implementation of as many App analyzers (including ones involving complicated logic) as possible.

1) Sensitive API Scan: Scanning sensitive API in the code is crucial for ensuring the security, performance, and privacy of software. Sensitive APIs may involve accessing personal information or system-level resources of users[20].

ArkAnalyzer enables precise and convenient API scanning. As previously mentioned, ArkAnalyzer provides various call graph analysis algorithms that allow developers to accurately identify specific functions in projects even with extensive usage of advanced object-oriented features such as inheritance and polymorphism.

Listing 2 provides an example of scanning code for locating log invocations. Given a scene and an array of MethodSignature as entry points, we can obtain the corresponding project call graph. The returned result is a map, with the key being the caller and the value being the callee. By traversing the map, it is very easy to find out which functions call the target function.

Listing 2: Code snippet to locate the usage of a given API.

```
function scanLog(scene:Scene, entryPoints: MethodSignature[],
        targetMethodSig: MethodSignature) {
    let callGraph = scene.makeCallGraphCHA(entryPoints);
    let calls = callGraph.getDynEdges();

    calls.forEach((callees: Set<MethodSignature>, caller:
            MethodSignature) => {
        if (callees.has(targetMethodSig)) {
            console.log(caller.toString());
        }
    });
}
```

This example demonstrates the usefulness of ArkAnalyzer, i.e., with the help of ArkAnalyzer, one only needs to write a few lines of code in order to implement a concrete program analysis task.

2) Null-pointer Analysis: Null pointer errors are a common type of error in programming practices, which occur when uninitialized pointers are used. These errors not only cause program crashes during runtime, severely affecting user experience, but may also lead to data loss or inconsistencies in program state[21]. Therefore, there is a need to automatically detect and thereby mitigate these errors before releasing the code to public. However, it is non-trivial to automatically locate this kind of error, as it involves field-aware inter-procedural data flow analysis.

Listing 3: Sample code with an Null-pointer error.

```
class Property{ pp=1; }
class T{
    p: property;
    printP(){ console.log(this.p.pp); }
}
function Main(){
    let t1 = new T();
    t1.printP(); // null pointer error
}
```

For example, Listing 3 illustrates an interprocedural null pointer error. In the Main function, t1.p.pp will be utilized. But in reality, t1.p is undefined at this point, which will cause the program to crash.

To facilitate the implementation of inter-procedural data-flow analyses, we have implemented in ArkAnalyzer the famous IFDS (Interprocedural Finite Distributive Subset) algorithm[22], [23], which provides a flexible framework that allows developers to define data flow facts and transfer functions as needed. Taking the aforementioned null-pointer error detection as an example, one only needs to extend the given IFDS framework to define how will the data propagate. Figure 8 illustrates the handling process of the example code in Listing 3.

Generally, the data propagation between statements is divided into four types of edges: Normal Edge, Call Edge, ReturnToExit Edge, and CallToReturn Edge. Each type of edge has a different data flow processing function. Ultimately, ArkAnalyzer will accurately detect which line of code will cause a null pointer exception.

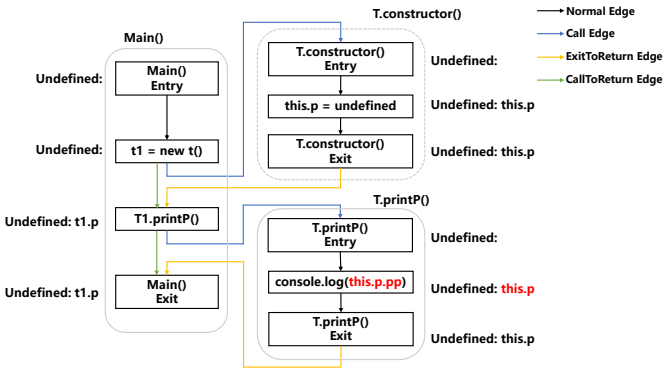


Fig. 8: The process to implement Null-pointer detectors.

This example further demonstrates the usefulness of ArkAnalyzer, being able to be leveraged to implement automated null pointer error detector.

VI. Threats to Validity

a) Internal threats to validity: The efficiency evaluation results (Section V-A) may be affected by other services running in the experimental environments (i.e., 64-bit Windows 11). Besides, in the accuracy evaluation(Section V-B), we employed a sampling approach to manually verify the invocation edges. This inherently carries the potential for inaccuracies.

b) External threats to validity: Considering that OpenHarmony is still in its early development stage, the features of ArkTS may change a lot. Our proposed ArkAnalyzer needs continuous updates in the future. Moreover, the dataset of OpenHarmony applications in our study was conducted up to April 2024. Given the rapid development pace of the HarmonyOS ecosystem, it is anticipated that the number of applications has significantly increased since then, and some applications in our dataset may have been updated.

VII. Related Work

Static analysis has been regarded as one of the most important techniques in the field of software engineering[24], [25], assisting developers and researchers in security analysis[26], [27], [28], vulnerability detection[29], [30], [31], and so on. To facilitate the development of static analysis approaches, our fellow researchers and practitioners have proposed to our community various static analysis frameworks. Table VI summarizes some of the representative frameworks grouped based on their targeted programming languages, such as Java [9], [32], [33], [34], [35], C/C++[36], [37], [38], [39], JavaScript and Typescript[10], [11], [12], Python[40], Swift[41], and Rust[42].

a) Program analysis tools: A large number of static analysis tools have emerged based on static analysis frameworks, such as FlowDroid for detecting sensitive data-flows[26], CiD for detecting API-induced compatibility issues[43], IccTA for inter-component data flow analysis[27], etc. These tools each focus on specific areas of code analysis, helping developers improve code quality and security.

TABLE VI: The list of representative static analysis frameworks.

Language	Framework	Paper Title Or GitHub Page
Java	Soot/SootUp	Soot: A Java bytecode optimization framework
	WALA	https://github.com/wala/WALA
	Doop	Strictly declarative specification of sophisticated points-to analyses
	Tai-e	Tai-e: A developer-friendly static analysis framework for Java by harnessing the good designs of classics
C/C++	SVF	SVF: interprocedural static value-flow analysis in LLVM
	PhASAR	Phasar: An inter-procedural static analysis framework for c/c++
JS/TS	TAJS	Type Analysis for JavaScript
	JSAI	JSAI: a static analysis platform for JavaScript
	ESLint	https://github.com/eslint/eslint
Python	Scalpel	Scalpel: The python static analysis framework
Swift	Swan	Swan: A static analysis framework for swift
Rust	RUPTA	A Context-Sensitive Pointer Analysis Framework for Rust and Its Application to Call Graph Construction

VIII. Conclusion

In this work, we present the first static analysis framework ArkAnalyzer for OpenHarmony Apps to the community. ArkAnalyzer addresses the problems of existing program analysis methods and has a set of common features (e.g., call graph construction) that are recurrently required when implementing in-depth static analyzers such as privacy leak detectors and compatibility issue detectors. We have collected and open-sourced a HarmonyOS native application dataset and conducted a series of evaluations on ArkAnalyzer, confirming its high performance and accuracy, intermediate representation (IR) readability, and ease of use. As for our future work, we commit to keep improving the ArkAnalyzer framework so as to support our fellow researchers in implementing efficient tools to resolve realistic App analysis problems.

References

[1] Openatom foundation. https://www.openatom.cn/. Accessed: July 9, 2024.

[2] Li Li, Xiang Gao, Hailong Sun, Chunming Hu, Xiaoyu Sun, Haoyu Wang, Haipeng Cai, Ting Su, Xiapu Luo, Tegawendé F Bissyandé, et al. Software engineering for openharmony: A research roadmap. arXiv preprint arXiv:2311.01311, 2023.

[3] Runlin Liu, Yuhang Lin, Yunge Hu, Zhe Zhang, and Xiang Gao. Llm-based java concurrent program to arkts converter. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, pages 2403–2406, 2024.

[4] Yige Chen, Sinan Wang, Yida Tao, and Yepang Liu. Model-based gui testing for harmonyos apps. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, pages 2411–2414, 2024.

[5] Kalok Sie. The analysis of smartphones' operating system and customers' purchasing decision: application to harmonyos and other smartphone companies. In 2022 7th International Conference on Financial Innovation and Economic Development (ICFIED 2022), pages 417–421. Atlantis Press, 2022.

[6] Neville Grech, Kyriakos Georgiou, James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. Static analysis of energy consumption for llvm ir programs. In Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems, pages 12–21, 2015.

[7] Ya Pan, Xiuting Ge, Chunrong Fang, and Yong Fan. A systematic literature review of android malware detection using static analysis. IEEE Access, 8:116363–116379, 2020.

[8] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C Gall, and Andy Zaidman. How developers engage with static analysis tools in different contexts. Empirical Software Engineering, 25:1419–1457, 2020.

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

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

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

[12] Find and fix problems in your javascript code. Accessed: July 9, 2024.

[13] OpenAtom 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.

[14] OpenHarmony SIG. Openharmony sig organization and pr command support list. https://gitee.com/openharmony-sig, 2024. Accessed on: 2024-06-07.

[15] OpenHarmony Third Party Components SIG. Openharmony-tpc: Third party components repository for openharmony applications. https://gitee.com/openharmony-tpc, 2024. Accessed on: 2024-04-10.

[16] Overview of the arkts compilation toolchain. https://developer.huawei.com/consumer/cn/doc/harmonyos-guides-V5/compilation-tool-chain-overview-V5/. Accessed: Augest 20, 2024.

[17] Yiwen Dong, Tianxiao Gu, Yongqiang Tian, and Chengnian Sun. Snr: constraint-based type inference for incomplete java code snippets. ICSE '22, page 1982–1993, New York, NY, USA, 2022. Association for Computing Machinery.

[18] David Grove and Craig Chambers. A framework for call graph construction algorithms. ACM Trans. Program. Lang. Syst., 23(6):685–746, nov 2001.

[19] Danilo Nikolić, Darko Stefanović, Dušanka Dakić, Srđan Sladojević, and Sonja Ristić. Analysis of the tools for static code analysis. In 2021 20th International Symposium INFOTEH-JAHORINA (INFOTEH), pages 1–6, 2021.

[20] Hadi Asemi. A study on api security pentesting. 2023.

[21] Hao Zhang, Ji Luo, Mengze Hu, Jun Yan, Jian Zhang, and Zongyan Qiu. Detecting exception handling bugs in c++ programs. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pages 1084–1095. IEEE, 2023.

[22] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 49–61, 1995.

[23] Nomair A Naeem, Ondřej Lhoták, and Jonathan Rodriguez. Practical extensions to the ifds algorithm. In Compiler Construction: 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings 19, pages 124–144. Springer, 2010.

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

[25] Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. Racerd: compositional static race detection. Proceedings of the ACM on Programming Languages, 2(OOPSLA):1–28, 2018.

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

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

[28] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng 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, pages 2455–2472, 2019.

[29] Li Li, Kevin Allix, Daoyuan Li, Alexandre Bartel, Tegawendé F Bissyandé, and Jacques Klein. Potential component leaks in android apps: An investigation into a new feature set for malware detection. In 2015 IEEE International Conference on Software Quality, Reliability and Security, pages 195–200. IEEE, 2015.

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

[31] Xiaoyu Sun, Xiao Chen, Kui Liu, Sheng Wen, Li Li, and John Grundy. Characterizing sensor leaks in android apps. In 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE), pages 498–509. IEEE, 2021.

[32] Kadiray Karakaya, Stefan Schott, Jonas Klauke, Eric Bodden, Markus Schmidt, Linghui Luo, and Dongjie He. Sootup: A redesign of the soot static analysis framework. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 229–247. Springer, 2024.

[33] Wala. https://github.com/wala/WALA. Accessed: July 9, 2024.

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

[35] Tian Tan and Yue 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, pages 1093–1105, 2023.
[36] Clang static analyzer. https://clang-analyzer.llvm.org/. Accessed: July 2, 2024.
[37] Chris Lattner. Llvm and clang: Next generation compiler technology. In The BSD conference, volume 5, pages 1–20, 2008.
[38] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. Phasar: An inter-procedural static analysis framework for c/c++. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 393–410. Springer, 2019.
[39] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In Proceedings of the 25th international conference on compiler construction, pages 265–266, 2016.
[40] Li Li, Jiawei Wang, and Haowei Quan. Scalpel: The python static analysis framework. arXiv preprint arXiv:2202.11840, 2022.
[41] Daniil Tiganov, Jeff Cho, Karim Ali, and Julian Dolby. Swan: A static analysis framework for swift. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 1640–1644, 2020.
[42] Wei Li, Dongjie He, Yujiang Gui, Wenguang Chen, and Jingling 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, pages 60–72, 2024.
[43] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. Cid: Automating the detection of api-related compatibility issues in android apps. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 153–163, 2018.