HARMOBRIDGE: Bridging ArkTS and C/C++ for Cross-Language Static Analysis on HarmonyOS

Jiale Wu*†, Jiapeng Deng*†, Yanjie Zhao§†, Li Li‡, and Haoyu Wang†
† Huazhong University of Science and Technology, Wuhan, China
jialewu@hust.edu.cn, jiapeng@hust.edu.cn, yanjie_zhao@hust.edu.cn, haoyuwang@hust.edu.cn
‡ Beihang University, Beijing, China, lilicoding@ieee.org

Abstract*—HarmonyOS is Huawei's distributed operating system designed for diverse smart devices, featuring ArkTS as its primary app development language. To enhance performance and leverage existing libraries, HarmonyOS apps can integrate native C/C++ modules through its Native Development Kit (NDK) mechanism. This creates significant challenges for static analysis, as critical data flows spanning ArkTS and native C/C++ boundaries remain invisible to existing single-language analyzers. Therefore, we present HARMOBRIDGE, the first cross-language static analysis system for HarmonyOS that bridges this gap through novel summary based SumIR abstraction and seamless ecosystem integration. Our approach extracts dataflow summaries from native code (supporting both binary and source code analysis) and translates these summaries into intermediate representations that integrate seamlessly with the existing HarmonyOS analysis infrastructure. HARMOBRIDGE introduces SumIR, a specialized intermediate representation that captures Node-API interaction semantics and converts them to ArkIR-compatible function bodies for downstream analysis tools. Also, we develop CROSSFLOWBENCH, a comprehensive benchmark covering representative Node-API interaction patterns, and identify potential security implications where established cross-language attack patterns widely prevalent in mobile ecosystems could adapt to HarmonyOS's architecture. Results demonstrate that HARMOBRIDGE achieves 81.0% accuracy in recovering cross-language data flows on CROSSFLOWBENCH, significantly outperforming baseline approaches that treat native calls as opaque operations, establishing a foundation for comprehensive cross-language analysis in the emerging HarmonyOS ecosystem.

***Index Terms*—HarmonyOS, Static Analysis, Mobile Security.**

I. INTRODUCTION

Modern mobile systems increasingly incorporate multi-language programming approaches to facilitate performance optimization and system-level access. HarmonyOS [1], Huawei's operating system for smart devices, provides the Native Development Kit (NDK) [2] which enables apps to adopt cross-language programming patterns by integrating native C/C++ modules with apps developed in ArkTS [3], a TypeScript-derived language designed for HarmonyOS. The cross-language interoperability is established through Node-API [4], an officially maintained interface that connects ArkTS/JavaScript environments with native code implementations. Such cross-language execution poses significant challenges for static analysis, as critical security issues like data leakage, privacy violations, and malicious behaviors often span both managed and native code boundaries.

In Android, the Java Native Interface (JNI) enables similar interactions. Recent research has revealed that malicious software developers increasingly leverage native code to conceal malicious behaviors, exploiting the opacity of cross-language boundaries to evade detection [5], [6], [7], [8]. This challenge has motivated extensive research on cross-language static analysis. These tools are essential for detecting cross-language data flows that could lead to sensitive information disclosure or serve as foundations for comprehensive static analysis of mobile apps. Prior works such as *JN-SAF* [9], *JuCify* [10], *μDep* [11], *JNFuzz-Droid* [12], and *NativeSummary* [13] each attempt to bridge Java/native gaps through heap modeling, summary generation, fuzzing, or binary interpretation.

However, while these approaches have demonstrated strong results for Android, they cannot be directly applied to HarmonyOS due to fundamental architectural differences. These tools are specifically designed around JNI semantics and Android's execution model, making them incompatible with HarmonyOS's Node-API-based interoperability mechanism and static analysis infrastructure. HarmonyOS presents unique cross-language analysis challenges that stem from its distinct language stack, compiler pipeline, and interop models, necessitating specialized analysis techniques. Therefore, in response to the increasingly vigorous development of HarmonyOS software engineering needs [14], we present HARMOBRIDGE, a comprehensive cross-language static analysis system tailored for HarmonyOS apps. Our novel summary-based modeling paradigm addresses the platform's distinctive execution environment and makes the following key contributions:

- **Dual-level native analysis framework**: A comprehensive analysis framework operating at both binary and source code levels. This dual approach supports flexible deployment across app lifecycle stages, from development-time security assessment to marketplace auditing.
- **SumIR generation with seamless ecosystem integration**: A unified SumIR representation (introduced in § III-B) capturing Node-API [15] cross-language dataflow semantics with automatic translation to existing static analysis

* Jiale Wu and Jiapeng Deng contributed equally to this work.
† The full name of the author's affiliation is Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology.
§ Yanjie Zhao (yanjie_zhao@hust.edu.cn) is the corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00261

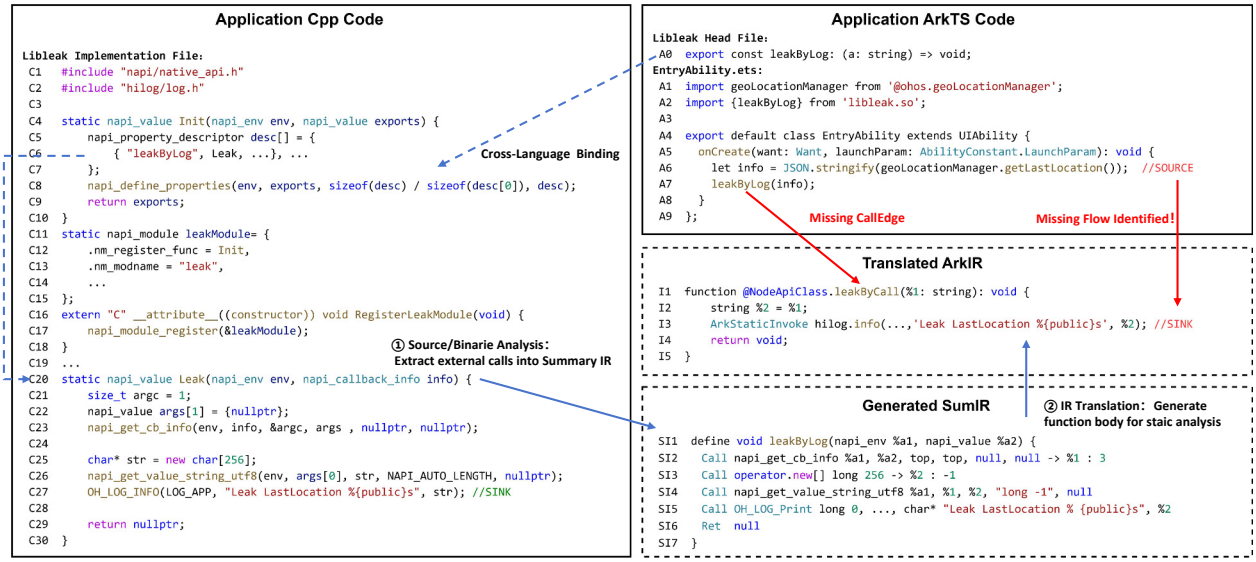

Fig. 1: The ArkTS code and native code of the motivation example, the generated SumIR and the translated ArkIR from native code.

Annotations: C1–C17: Native module registration via module descriptor; **C20–C29:** Native sink using Node-API routines; **A1–A9:** ArkTS invokes native with sensitive data; **I1–I4:** ArkIR lacks explicit native call edges; **SI1–SI7:** SumIR recovers the full cross-language flow.

frameworks' intermediate representations. This enables integration with HarmonyOS's existing analysis ecosystem (ArkAnalyzer [16]), allowing downstream tools to reason about cross-language results without modification.

- **Comprehensive benchmark suite for HarmonyOS**: CROSSFLOWBENCH [1], a systematically designed benchmark suite covering representative cross-language data flow patterns specific to HarmonyOS development practices. The benchmark suite provides a foundation for standardized evaluation and is open-sourced to support future research in cross-language analysis for HarmonyOS.

HARMOBRIDGE is implemented as a practical and extensible toolchain. Evaluation on synthetic benchmarks demonstrates its ability to recover complex data flows that traverse ArkTS and native boundaries—flows that remain invisible to current HarmonyOS static analyzers.

II. BACKGROUND AND MOTIVATION

A. Cross-language Analysis in the Mobile Ecosystem

Cross-language execution in mobile apps poses fundamental challenges for static security analysis. Critical behaviors such as permission misuse and privacy leaks often occur within native components that remain opaque to language-specific analyzers [17], [18]. Without proper cross-language reasoning, taint flows spanning managed and unmanaged code boundaries can silently escape detection. The Android ecosystem has motivated extensive research in this area. Dynamic approaches include Xue et al. [6] who use multi-layer monitoring and Valgrind-based [19] instrumentation to detect native behaviors, and several other works [5], [20], [21], [22] that propose different dynamic tracking mechanisms for native code interactions.

Meanwhile, researchers have also developed sophisticated static analysis tools that explicitly model cross-language interface semantics. Notable systems include *JN-SAF* [9] (heap-based abstraction for Java-native taint propagation), *JuCify* [10] (WALA [23] extension for cross-language control/data flow), *μDep* [11] (mutation-based fuzzing for cross-boundary dependencies), and *JNFuzz-Droid* [12] (hidden native entry point detection). Among these approaches, *NativeSummary* [13] represents a distinctive direction through abstract interpretation over native binaries, generating summaries that simulate native behavior for Java-level analysis. These techniques demonstrate the value of cross-language modeling, but they are fundamentally designed for Android's JNI-based architecture. HarmonyOS introduces distinct language stacks, compiler pipelines, and interop models that necessitate fundamentally different analysis approaches.

B. HarmonyOS Cross-language Context and Motivating Example

HarmonyOS enables cross-language interaction between ArkTS and native C/C++ code through Node-API, an officially supported interface that facilitates native extension. Unlike JNI, Node-API relies on a descriptor-based module system. Native functions are registered via `napi_module` descriptors and exposed to ArkTS through dynamic export without explicit function name mapping or signature matching. Additionally, while JNI explicitly passes data flows through parameters and return values, Node-API employs encapsulated object-centric value propagation across different API parameters. While this approach provides greater scalability and modu-

[1]The benchmark suite is open-sourced and accessible at https://github.com/security-pride/CrossFlowBench.

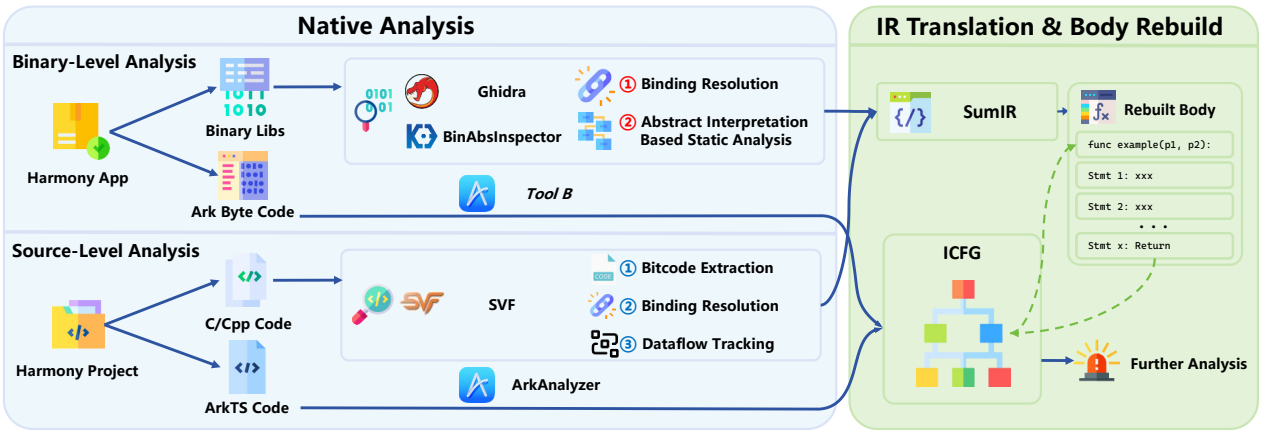

Fig. 2: Overview of HARMOBRIDGE.

larity in data interaction, it significantly complicates static analysis reasoning across language boundaries. To illustrate the analysis challenge posed by this design, Figure 1 presents a motivating example that shows a potential privacy leak across ArkTS and native layers. On the ArkTS side (A1–A9), the app retrieves sensitive location information and passes it to a native method `LeakByLog`. The native function (C20–C29), registered through the descriptor mechanism (C1–C17), extracts the sensitive data and leaks it through system logging.

From a static analysis perspective, this cross-language flow poses a fundamental problem. Static analyzers operating at the ArkTS level can observe the external invocation to `LeakByLog` but have no access to its native semantics—e.g., how the input argument is unpacked, processed, or propagated to system-level sinks. The analyzer treats native calls as opaque stubs, and no call edges or taint semantics are preserved across the boundary. As a result, analyzers fail to detect this end-to-end sensitive flow. This example is representative of a broader class of hidden flows in HarmonyOS, where security-relevant behaviors span ArkTS and native code. It motivates the need for an analysis framework that can recover and model native semantics in a way compatible with ArkIR reasoning. In our system, HARMOBRIDGE, we address this gap by statically analyzing native code (either at the binary or source code level) and synthesizing SumIR to represent cross-language semantics, which is then automatically translated into ArkIR-compatible function bodies. This enables downstream analyzers to detect full cross-language flows, as shown in the recovered IR in Figure 1 (SI1–SI7).

C. HarmonyOS Static Analysis Infrastructure

HarmonyOS provides a static analysis ecosystem centered on three key components. ArkAnalyzer [16] converts ArkTS source code into its intermediate representation ArkIR, for downstream analysis tools. Building upon this foundation, *Tool-F*[2] performs taint analysis on ArkIR with dataflow tracking capabilities similar to established Android tools like FlowDroid [24], IccTA [25], and Amandroid [26]. Complementing the source code analysis path, *Tool-B*[2] is an emerging bytecode analysis framework that converts compiled bytecode from HAP packages into ArkIR, paralleling the functionality of Soot [27] for Dalvik bytecode to Jimple IR conversion. This dual-path approach enables comprehensive static analysis coverage spanning both development-time security assessment and marketplace-level auditing.

III. OVERVIEW AND KEY IDEAS

The core insight of HARMOBRIDGE is that cross-language static analysis in HarmonyOS can be made precise and scalable by separating the problem into two orthogonal concerns: (1) recovering native semantics summaries from C/C++ code (either binary or source code), and (2) translating those semantics summaries into IRs that can be reused by existing static analyzers. HARMOBRIDGE adapts a summary-based modeling paradigm specifically designed for HarmonyOS's distinct interop architecture. In contrast to Android's JNI, HarmonyOS binds ArkTS and native modules via Node-API—a descriptor-driven mechanism that hides semantic connections behind runtime registration and opaque handles. These abstractions make the native layer invisible to ArkIR-based analyzers unless explicitly reconstructed.

A. Overview

To bridge the semantic gap, HARMOBRIDGE introduces a two-stage architecture as shown in Figure 2:

Native Analysis: HARMOBRIDGE statically analyzes native modules to extract precise cross-language interaction patterns, including parameter extraction sequences, data transformation operations, and external function invocations that affect taint propagation. It supports two forms of native input:

[2]Proprietary tools referred to as "F" (for Flow) and "B" (for Bytecode) are currently in development. **Names have been anonymized for compliance and confidentiality considerations.**

- *Binary-level analysis* handles packaged HarmonyOS apps where native code exists only in compiled ELF binaries. HARMOBRIDGE performs disassembly and abstract interpretation, with enhanced taint tracking for Node-API operations (e.g., `napi_get_cb_info`, `napi_get_value_string_utf8`). The analysis results are encoded in our custom intermediate representation, SumIR (detailed in § III-B).
- *Source code analysis* processes projects with available C/C++ sources. Using pointer analysis and value-flow graphs, HARMOBRIDGE tracks data propagation through complex program structures. Despite the different input format, the analysis similarly produces SumIR representations that capture native function behaviors.

Regardless of input mode, both paths generate a unified SumIR that abstracts each native function's behavior in a form suitable for integration.

IR Translation and Body Rebuilding: In this phase, HARMOBRIDGE converts the SumIR into ArkIR-compliant function bodies. These synthesized ArkIR fragments are injected into the existing analysis pipeline, replacing opaque native call stubs with semantically enriched models. This enables downstream analyzers to reason about cross-language flows without modification. Notably, HARMOBRIDGE is non-intrusive: it requires no changes to the ArkTS source code or the HarmonyOS build process. Its summaries can be precomputed and deployed independently, supporting both developer-side analysis and marketplace-level auditing.

B. SumIR

To bridge the semantic gap between native code execution and ArkIR-based analysis, HARMOBRIDGE introduces SumIR, a specialized intermediate representation that captures the essential cross-language semantics of native functions. The SumIR serves as the critical abstraction layer that transforms complex Node-API interactions into analyzable dataflow patterns compatible with HarmonyOS's analysis ecosystem.

Design Principles and Structure: The SumIR is designed as a lightweight, dataflow-centric representation that captures essential semantic behaviors without requiring full program state modeling. It focuses specifically on operations affecting cross-language data propagation.

The SumIR consists of three fundamental instruction types:

- **CALL instructions** represent invocations of Node-API functions (e.g., `napi_get_cb_info`, `napi_get_value_string_utf8`) and security-relevant C library functions (e.g., network operations, system calls). Each CALL instruction captures function signatures, input-output relationships. Later, they could be translated into corresponding operations according to the modeling of the function they carry.
- **PHI instructions** model control flow convergence points where data from different execution paths merge, preserving flow-sensitive taint analysis across conditional branches and loops.

```
Interface Binding Pattern #1:
  static napi_value Init(napi_env env, napi_value exports) {
      napi_property_descriptor desc[] = {
          { "exampleFunc", ExampleFunc, ...}, ...
      };   // binding with property descriptors
      napi_define_properties(env, exports, sizeof(desc) / sizeof(desc[0]), desc);
      return exports;
  }

Interface Binding Pattern #2:
  static napi_value Init(napi_env env, napi_value exports) {
      napi_value fn = nullptr;  // binding by directly setting properties
      napi_create_function(env, nullptr, 0, ExampleFunc, nullptr, &fn);
      napi_set_named_property(env, exports, "exampleFunc", fn);
      return exports;
  }
```

```
Function Parameter Acquisition and Result Return:
  ...
  static napi_value ExampleFunc(napi_env env, napi_callback_info info) {
      size_t argc = 2;
      napi_value args[2] = {nullptr, nullptr}; // get paraneters through array
      napi_get_cb_info(env, info, &argc, args , nullptr, nullptr);
      ...
      return ret;  // return napi_value typed value
  }
```

Fig. 3: Patterns of interface binding, parameter passing, and return in Node-API native code.

- **RET instructions** encode function return semantics.

The SumIR reconstructs native-side semantics by tracking value flow relationships between instructions. For each native function, it captures: (1) how ArkTS input parameters are extracted through Node-API calls, (2) transformations applied within native code, and (3) how results propagate back to ArkTS or external sinks. This dataflow-driven approach abstracts implementation details while preserving essential semantic relationships needed for cross-language analysis. Afterward, the SumIR is systematically translated to ArkIR through structured mapping of each instruction type to semantically equivalent ArkIR constructs. This translation preserves semantic fidelity while ensuring compatibility.

IV. DESIGN OF HARMOBRIDGE

This section presents HARMOBRIDGE's design across three main components: native analysis at binary level (§ IV-A), native analysis at source code level (§ IV-B), and IR translation (§ IV-C). The first two components focus on native code analysis using different input representations, while the third component handles the integration with existing HarmonyOS analysis infrastructure.

A. Native Analysis (Binary Level)

The binary analysis module statically analyzes native libraries to extract cross-language semantics for SumIR generation. Our approach addresses the challenges posed by Node-API's dynamic registration mechanism and opaque handle system. The analysis consists of three steps: binding resolution, abstract interpretation-based analysis and SumIR Generation.

1) Binding Resolution

Unlike JNI-based systems where name mangling provides a static mapping from high-level method names to native function symbols, HarmonyOS employs a dynamic, descriptor-driven registration model for binding ArkTS to native C/C++

code. As shown in Figure 3, native modules define a module structure containing a pointer to an initialization function. This function is automatically registered at load time via a global constructor. Within the initialization routine, native functions are registered through two common patterns: descriptor-based registration (Pattern #1) and function-based registration (Pattern #2). Both methods establish runtime bindings between exported method names and their corresponding native implementations. To recover such bindings from binaries, we scan the data section for module descriptor structures based on known layout signatures. For each structure, we resolve the registration function field to identify the associated initializer. Then, via abstract interpretation over the initializer's body, we detect invocations of binding functions. We symbolically evaluate the parameters of these calls using memory abstraction. For descriptor-based registration, we interpret the descriptor array to extract method names and corresponding function pointers. For function-based registration, we track function creation return values and correlate them with their symbolic use in subsequent property assignment calls. All recovered bindings are finally translated into structured descriptors of the following form:

$$\langle \texttt{ArkTS-visible name}, \texttt{ native function address} \rangle$$

These descriptors serve as the entry points for downstream function modeling and summary construction.

2) Abstract Interpretation Based Static Analysis

Having established the binding resolution process, we now describe how binary analysis generates the SumIR constructs through abstract interpretation. Our analysis pipeline extracts semantic summaries from native code using k-call-site-sensitive abstract interpretation over binary executables. This approach enables precise modeling of Node-API interactions while maintaining scalability for real-world apps. The analysis addresses three fundamental challenges: (i) establishing a faithful abstraction of Node-API's runtime semantics, (ii) tracking complex dataflow patterns through opaque handles and indirect parameter passing, and (iii) maintaining interprocedural precision despite assembly-level optimization artifacts. The output of this analysis directly populates the SumIR structures introduced earlier.

(a) Node-API Runtime Abstraction: Node-API introduces a fundamentally distinct execution paradigm compared to conventional interfaces. Instead of employing direct parameter passing mechanisms, native functions receive opaque runtime handles (`napi_env`, `napi_callback_info`) that encapsulate the execution context and argument metadata. Parameter acquisition requires a multi-step extraction process through specialized API calls, resulting in intricate indirect dataflow patterns that pose significant challenges for static analysis.

To address these challenges, we construct a comprehensive abstract interpretation model of the Node-API runtime environment. This model systematically formalizes the callback-based parameter extraction mechanism and the relationships between opaque handles. We develop function summaries for each Node-API function categorized by their semantic domains (as shown in Table I), with each summary precisely characterizing the function's input-output behavior and side effects. For example, our models for parameter extraction functions (e.g., `napi_get_cb_info`) capture the semantics of argument count retrieval, progressive population of argument vectors with abstract values representing ArkTS parameters, and precise tracking of auxiliary output parameters. This formal modeling approach enables accurate representation of Node-API's dataflow patterns, particularly the multi-return mechanism where a single function invocation produces multiple outputs through pointer parameters.

TABLE I: Node-API function categories modeled in HARMOBRIDGE.

Category	Number of APIs
Basic Data Types	15
Array Operations	8
Objects and Properties	10
Functions and Modules	4
Error Handling & Type Conversion	11
System Utilities	5

(b) Dataflow Tracking through Opaque Abstractions: Node-API employs a type system centered around opaque handles that abstract JavaScript values and runtime state. This design poses challenges for static analysis, as the semantic relationships between these handles and their underlying data are not directly observable in the binary code. Our solution introduces a systematic value categorization and tracking mechanism that addresses three distinct value categories:

- *Opaque Runtime Objects*: Entities like `napi_value`, `napi_env`, and `napi_callback_info` represent runtime state that cannot be directly examined. We model these within a specialized abstract address space, assigning each handle a unique identifier that retains its semantic origin and calling context. Our `NAPIValueManager` maintains bidirectional mappings between identifiers and their abstract representations. All values passed to or created within native code are categorized as `napi_value` without explicit type labeling—type recovery is deferred to the IR translation phase (detailed in § IV-C) where type inference links values back to their ArkTS origins.
- *Scalar Values and Status Codes*: For primitive types and API return codes, we employ taint-based abstraction that preserves dataflow dependencies without requiring concrete value resolution, accommodating the dynamic nature of JavaScript-to-native conversion.
- *Buffer and String Operations*: Memory-intensive operations require specialized heap modeling. We allocate abstract heap regions for buffer-returning APIs (e.g., `napi_get_value_string_utf8`) and track pointer relationships to maintain precise aliasing information. Also, for constant strings as arguments, we try to extract them to SumIR.

Our tracking mechanism captures key values transmitted through `napi_env`, recording their propagation paths and

semantic effects. These tracked values enable accurate reconstruction of Node-API's parameter-passing patterns and provide the foundation for subsequent SumIR generation, maintaining precise correspondence between call sites and their dataflow semantics across different value categories.

(c) Interprocedural Precision Maintenance: Binary-level abstract interpretation faces inherent challenges in maintaining precision across function boundaries, particularly when analyzing optimized native code. Our approach addresses critical precision-threatening scenarios through context-sensitive register preservation that maintains separate abstract states for caller and callee contexts, selective propagation of stack-based values using k-bounded calling context, and tail call detection logic that recognizes compiler optimizations and restores appropriate calling semantics. These techniques ensure that call-site sensitivity is preserved throughout the analysis while maintaining manageable complexity.

3) SumIR Generation

The final stage converts the collected abstract state into a structured SumIR through systematic extraction and encoding. Our process operates in three phases: (i) *call sequence extraction* identifies and orders all external function invocations encountered during analysis, capturing their calling contexts and control flow relationships; (ii) *argument resolution* queries the abstract interpreter to determine symbolic values and taint information passed to each external call, preserving dataflow dependencies; and (iii) *dataflow synthesis* generates SumIR instructions that encode the semantic effects of each native function, including CALL instructions for Node-API invocations, PHI instructions for control flow merges, and RET instructions for return value handling.

The generated SumIR directly implements the constructs described in § III-B, with opaque value references maintaining semantic origins, multi-output call instructions preserving complex parameter patterns, and taint propagation annotations enabling cross-boundary dataflow tracking. This structured representation serves as input to the subsequent IR translation stage, completing the native-to-ArkIR integration pipeline.

B. Native Analysis (Source Code Level)

While binary analysis targets packaged HarmonyOS apps, the rapidly evolving ecosystem requires integrated development-time security assessment tools. To address this need, we developed a source code analysis component within HARMOBRIDGE that enables developers to perform security self-assessment and compliance verification during the development process. This component analyzes native C/C++ source code by first generating complete LLVM bitcode representations [28] of the entire project, and then leveraging the Static Value-Flow Analysis (SVF) [29] framework to perform precise interprocedural Dataflow Graph Constructionedural data flow analysis. Through this approach, we extract value propagation behaviors and propagation paths, encoding them into the same SumIR format used by the binary analysis path, maintaining consistency in downstream processing. Source code analysis consists of four key steps: LLVM Bitcode Extraction, Native Interface Binding Recognition, interprocedural dataflow tracking and SumIR Generation.

1) LLVM Bitcode Extraction

To perform source code analysis of cross-language interactions, we extended the whole-program-llvm (WLLVM) [30] tool, which is a framework for building whole-program LLVM bitcode files from C/C++ source code packages, to accommodate HarmonyOS's cross-compilation environment and complex project structures. The process operates in two phases:

- **Compilation-time bitcode generation:** During compilation, we generate corresponding bitcode files and store their paths in dedicated sections (.llvm_bc) of object files, while customizing the process to handle HarmonyOS-specific compiler flags and toolchains.
- **Post-processing bitcode linking:** We parse library files to extract stored bitcode paths and link them into a complete whole-program bitcode file, with enhancements to accommodate HarmonyOS's build structures and prevent symbol conflicts.

This approach not only allows us to obtain a comprehensive project bitcode representation without interfering with the original build process but also handles the challenges posed by HarmonyOS's cross-compilation environment and custom toolchain, establishing a solid foundation for cross-language dataflow analysis and semantic interpretation.

2) Native Interface Binding Recognition

Our source code analysis follows a similar approach to the binary analysis (§ IV-A1), identifying ArkTS-native interface bindings by tracking module registration patterns. Working with LLVM bitcode, however, offers advantages over binary analysis: we can directly extract descriptor data structures from the intermediate representation and achieve more precise tracking of value flow relationships in function-based registration.

3) Interprocedural Dataflow Tracking

(a) Complete Dataflow Graph Construction: Following the extraction of complete LLVM bitcode, we address the challenge of generating SumIR from source code representations. Although LLVM bitcode preserves more symbolic information than binary files, Node-API functions remain opaque to static analysis, since they are declared but not defined, lacking function implementations and dataflow information that prevent SVF from analyzing their behavior.

Our solution develops specialized function modeling techniques that provide precise implementations for each Node-API function, enabling SVF to recognize and understand their value flow behavior. This modeling approach resolves the opacity of external APIs and enables construction of complete interprocedural dataflow graphs. While LLVM bitcode's structural information simplifies some analysis challenges compared to binary analysis, Node-API's opaque handle system and indirect parameter passing still require specialized treatment to achieve accurate dataflow tracking.

(b) Interprocedural Value Flow Analysis: Our approach leverages SVF's pointer analysis to establish precise aliasing relationships between Node-API parameters across function boundaries. When opaque handles are passed through multiple call sites, alias analysis identifies which variables reference the same underlying objects, enabling accurate tracking of value propagation paths in the dataflow graph. The analysis examines pointer dereference operations to capture how Node-API functions modify values through output parameters and traces their subsequent usage across function boundaries.

For multi-output Node-API functions that return values through pointer parameters, we follow similar value categorization and tracking approaches as binary analysis, maintaining unique identifiers for opaque runtime objects and employing taint-based abstractions for scalar values.

(c) Precision Enhancement for Complex Operations: Node-API's string and array operations present precision challenges for SVF's standard analysis. SVF's default approach is inherently array-insensitive, treating entire C/C++ arrays as single monolithic objects without distinguishing individual elements. To overcome this limitation, we enhance SVF's memory model by leveraging its pointer analysis capabilities to identify array offsets and precisely track each array element, rather than relying solely on the array's base pointer. This fine-grained element-level tracking enables accurate taint propagation, facilitating precise correspondence between ArkTS data structures and their underlying C/C++ array representations.

For interprocedural precision maintenance, our approach leverages SVF's context-sensitive analysis capabilities while addressing Node-API specific challenges. We utilize SVF's call-site sensitivity to maintain precise tracking of opaque handle propagation across function boundaries, ensuring that the semantic relationships between Node-API parameters are preserved throughout the analysis. This approach handles multi-output parameter patterns effectively, where multiple values are returned through pointer parameters, by maintaining precise correspondence between call sites and their semantic effects in the resulting SumIR.

4) SumIR Generation

Similar to binary analysis, we convert dataflow analysis results into structured SumIR, extracting call sequences and argument dependencies to generate instructions that preserve Node-API semantic effects for subsequent IR translation.

C. IR Translation and Body Rebuild

The IR translation component implements the final stage of HARMOBRIDGE's analysis pipeline, converting SumIR into ArkIR-compatible function bodies that seamlessly integrate with HarmonyOS's existing analysis infrastructure.

Translation Pipeline: Given the SumIR generated from native analysis, the translation process operates through four stages:

- *Cross-language Call Site Identification*: The system analyzes ArkTS source code to locate cross-language function call sites by matching imported native module names and function identifiers with the binding descriptors extracted during native analysis.
- *Type Inference for Opaque Values*: At each identified call site, we extract function signatures and parameter types from the ArkTS context. This type information is propagated backward through the SumIR to resolve the concrete ArkTS types of previously opaque `napi_value` handles, enabling precise type-aware translation.
- *SumIR to ArkIR Translation*: Each SumIR instruction is systematically converted to semantically equivalent ArkIR operations. CALL instructions become ArkIR function invocations, PHI instructions translate to conditional assignments, and RET instructions map to return statements with appropriate type conversions.
- *CFG Integration and Call Edge Connection*: The translated ArkIR instructions are organized into basic blocks and integrated into the control flow graph. Call edges are established between ArkTS call sites and the newly generated native function bodies, completing the cross-language connection.

Key Technical Adaptations: The translation process addresses Node-API's execution model through specialized handling: opaque `napi_value` types are resolved to concrete ArkTS types using call site context, multi-output Node-API operations are decomposed into sequences of ArkIR assignments that preserve dataflow semantics, and taint propagation annotations are naturally encoded to ArkIR compatible with downstream analysis tools, which enables existing HarmonyOS analyzers to transparently process cross-language flows without requiring modifications to their implementations.

V. IMPLEMENTATION AND EVALUATION

A. Implementation

We have implemented HARMOBRIDGE using about 3.8K SLOC of Java for Native Binary Analysis (without our modification of BinAbsInspector), 4.7k SLOC of C++ for Native Source Code Analysis, and 3.9K SLOC of TypeScript for IR translation and body rebuilding.

B. Evaluation

We evaluate HARMOBRIDGE's effectiveness through experiments addressing the following research questions:

- **RQ1:** Can HARMOBRIDGE perform accurate cross-language static analysis?
- **RQ2:** How does HARMOBRIDGE scale to real-world HarmonyOS apps?

1) Precision (RQ1)

(a) Benchmark Design: To assess tool accuracy, we design CROSSFLOWBENCH, a comprehensive benchmark suite for cross-language data flow analysis in HarmonyOS. The suite systematically addresses HarmonyOS-specific patterns while incorporating insights from established benchmarks like NativeFlowBench [31], [32] and JuCify's benchmarks [33]. Our benchmark comprises 21 test cases spanning simple data

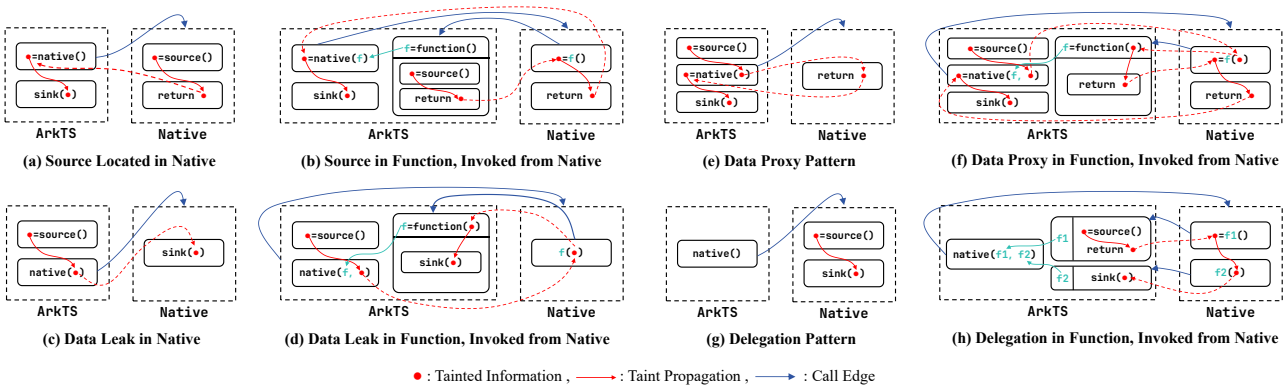

Fig. 4: Representative cross-language data flow patterns in CROSSFLOWBENCH.

access to multi-library interactions, providing systematic coverage of HarmonyOS-specific cross-language dataflow patterns (listed and detailed in Table II). Figure 4 illustrates eight representative patterns from our benchmark, demonstrating the complexity of cross-language interactions. The benchmark systematically covers representative interaction patterns across three categories:

- **Basic Cross-language Flows:** Fundamental source-sink patterns where sensitive data traverse language boundaries, which covers four combinations of source-sink placement across languages (T1-T4 as illustrated in Figure 4.a, c, e and g);
- **Advanced Interaction Patterns:** Complex scenarios involving bidirectional cross-language function invocations. These patterns test native code's ability to call back into ArkTS functions that contain sources or sinks, creating nested cross-language call chains that complicate dataflow tracking (T5-T8 as illustrated in Figure 4.b, d, f and h). Additionally, sophisticated multi-step interactions test scenarios where data undergoes multiple transformations across language boundaries (T9, T17-T18 and T21);
- **Data Structure Operations and Others:** Comprehensive coverage of Node-API's complex data manipulation capabilities, including object property access and modification (T10-T12), array element operations with precise indexing (T13-T15), and error handling patterns where exceptions carry sensitive data across boundaries (T16). Additional test cases validate PHI instruction of SumIR (T19-T20) and data encoding (T21).

(b) Experimental Setup: We evaluate HARMOBRIDGE against multiple baseline configurations to assess its integration capabilities and accuracy improvements. The evaluation uses 5 configurations:

1) **Baseline Configurations:**
   - **B:** ArkAnalyzer + *Tool-F* (targeting source code)
   - **C:** *Tool-B* + *Tool-F* (targeting packed app)
2) **HARMOBRIDGE Integrated Configurations:**
   - **D:** *Tool-B* + HARMOBRIDGE (binary mode) + *Tool-F* (targeting packed app)
   - **E:** ArkAnalyzer + HARMOBRIDGE (binary mode) + *Tool-F* (utilizing ArkTS code of app and binary of native modules)
   - **F:** ArkAnalyzer + HARMOBRIDGE (source code mode) + *Tool-F* (targeting project source code)

Configuration E Rationale: Configuration **E** was introduced as a pragmatic solution to address evaluation limitations in our testing environment. As shown in Table II, configurations based on *Tool-B* (**C** and **D**) exhibited instability with few analysis failures. While it shows promise as an emerging bytecode analysis tool, its current implementation lacks robustness for complex cross-language scenarios. Configuration **E** combines ArkAnalyzer's stability with HARMOBRIDGE's binary analysis capabilities, allowing us to validate our cross-language analysis approach despite infrastructure limitations in the current bytecode analysis ecosystem.

(c) Results and Analysis: Table II presents our evaluation results, demonstrating HARMOBRIDGE's significant improvements over existing approaches. HARMOBRIDGE achieves 81.0% accuracy (22/26 correct flows), substantially outperforming baselines that achieve only 28.6% (ArkAnalyzer) and 23.8% (*Tool-B*) accuracy.

Baseline Performance Analysis: Baseline analyzers fail to properly model cross-language interactions. They treat native calls as opaque operations without understanding their internal behavior. Their few successes (T3, T9, T19-T20) are largely coincidental—occurring only when data flows directly through native functions without transformation, where conservative taint propagation policies accidentally preserve the correct flow. This approach inevitably fails when encountering real-world patterns with complex parameter handling or cross-language callbacks.

HARMOBRIDGE's Effectiveness: Our system demonstrates consistent performance across both binary analysis (Conf. **E**) and source code analysis (Conf. **F**), both achieving 81.0% accuracy. This validates our dual-level approach as effective regardless of input representation of native modules. The significant accuracy improvement comes from HARMO-

TABLE II: Descriptions of Test Cases in CROSSFLOWBENCH and Evaluation Results on Them

ID	Test Case	Description	Expected	Baselines		HARMOBRIDGE		
			A†	B‡	C§	D¶	E#	F*
T1	native_source	Source called in native (Figure 4.a)	●	○	○	●	●	●
T2	native_leak	Data leakage in native (Figure 4.c)	●	○	○	○	●	●
T3	native_proxy	Data propagated in native call (Figure 4.e)	●	●	●	●	●	●
T4	native_delegation	Source and sink both called in native (Figure 4.g)	●	○	×	○	●	●
T5	native_call_function_sink	Sink in an ArkTS function called in native (Figure 4.d)	●	○	×	○	●	●
T6	native_call_function_source	Source in an ArkTS function called in native (Figure 4.b)	●	○	×	●	●	●
T7	native_call_function_proxy	Data propagated in an ArkTS function called in native (Figure 4.f)	●	●	×	○	●	●
T8	native_call_function_delegation	Source and sink both called in native (Figure 4.h)	●	○	×	○	●	●
T9	native_proxy_copy	Data propagated in native calls (copied into buffer)	●	●	●	●	●	●
T10	native_call_function_object	Object method call in native	●	○	○	○	●	●
T11	native_complex_data	Accessing object and leaking data in native	●○	○○	○○	○○	●○	●○
T12	native_set_field	Writing data into field in native	●○	○○	○○	○○	●○	●○
T13	native_array_set	Setting array data in native	●○	○○	○○	○○	●●	●●
T14	native_array_get	Getting array data in native	●	○	○	○	●	●
T15	native_array_clean	Taint reassignment in native (string reallocation)	○	○	○	○	●	●
T16	native_error	Data leakage in error handling in native	●	○	○	○	○	○
T17	native_multiple_interaction	Multiple interactions of cross-language calls	●	○	×	×	●	●
T18	native_multiple_libraries	Multiple library interactions	●●●	●○○	×	×	●●●	●●●
T19	native_phi_branch	Conditional branching	●	●	●	●	●	●
T20	native_phi_concat	Data leakage of concatenated values	●	●	●	●	●	●
T21	native_encode	Base64 encoding in native	●	○	○	○	○	○
Total Detected Flows			**26/26**	**10/26**	**8/25**	**10/26**	**22/26**	**22/26**
Percentage (correct cases / total cases)			**100.0%**	**28.6%**	**23.8%**	**33.3%**	**81.0%**	**81.0%**

A†: Expected results B‡: ArkAnalyzer + *Tool-F* (baseline) C§:*Tool-B* + *Tool-F* (baseline)
D¶: *Tool-B* + HARMOBRIDGE(bin) + *Tool-F* E#:ArkAnalyzer + HARMOBRIDGE(bin) + *Tool-F* F*: ArkAnalyzer + HARMOBRIDGE(src) + *Tool-F*
●: Successfully detected taint flow (True Positive) ○: Failed to detect existing taint flow (False Negative) ×: Analysis failed (analysis tool crashed)
○: Correctly identified no taint flow (True Negative) ●: Incorrectly detected non-existent taint flow (False Positive)

BRIDGE's ability to extract precise semantic models from native code—capturing parameter access patterns, data transformations, and cross-boundary control flows that remain invisible to existing tools. Conf. **D**'s modest performance (33.3%) highlights the challenge of integrating with early-stage bytecode analyzers, yet still represents an improvement over the baseline.

Error Analysis: The few remaining limitations reflect inherent challenges in static analysis rather than HARMOBRIDGE-specific issues: T21 involves complex encodings that exceed standard data flow tracking capabilities; T16 requires modeling exceptional control flow paths through error handlers. False positives in T13 and T15 reflect our intentionally conservative approach to array operations.

Overall, the results demonstrate HARMOBRIDGE's ability to bridge the critical semantic gap between ArkTS and native code, enabling comprehensive cross-language static analysis that was previously impossible with existing HarmonyOS tools. This capability brings practical value for both developer-focused security assessment and marketplace-level app vetting in the emerging HarmonyOS ecosystem.

2) Performance on Real-World HarmonyOS Apps (RQ2)

The current state of the HarmonyOS ecosystem presents certain evaluation challenges for cross-language static analysis. While Android benefits from established open-source communities like F-Droid [34], the HarmonyOS app ecosystem is still evolving, with relatively few publicly available apps that make extensive use of Node-API. Furthermore, commercial apps in HarmonyOS markets often have access restrictions that limit comprehensive third-party analysis.

Despite these challenges, we obtained approval to analyze binary packages from five popular real-world HarmonyOS apps [3] to evaluate HARMOBRIDGE's effectiveness in production environments. This effort aims to better promote the development of HarmonyOS app analysis toolchains and foster the co-evolution of our tool alongside the maturing HarmonyOS ecosystem. Table III presents our binary analysis results, focusing on HARMOBRIDGE's ability to extract SumIR from native libraries.

TABLE III: Binary Analysis Performance on Real-world HarmonyOS Apps

Metric/App	A	B	C	D	E
Node-API Libs (%)[1]	48.47	49.09	59.26	86.92	43.75
Timeout Libs (%)[2]	16.33	9.09	8.64	6.54	12.50
SumIR Generated Libs (%)[3]	32.65	34.55	49.38	33.64	31.25
Total Instructions[4]	6412	3434	3119	3095	137
Generation Rate (%)[5]	67.37	70.37	**83.33**	38.71	71.43

[1] **Node-API Libs(%)**: Proportion of libraries using Node-API for cross-language calls.
[2] **Timeout Libs(%)**: Proportion of libraries that triggered timeout.
[3] **SumIR Generated Libs(%)**: Proportion of libraries successfully generating SumIR.
[4] **Total Instructions**: Total number of SumIR instructions generated for the app.
[5] **Generation Rate (%)**: Generation Rate = $\frac{\text{SumIR Generated Libs}}{\text{Node-API Libs}} \times 100\%$.

Binary Analysis Performance: Our evaluation focused on the binary analysis component of HARMOBRIDGE, as source code for production apps is unavailable and current bytecode analysis tools remain in early stage. We analyzed each native library with a 300-second timeout to ensure practical performance in real-world scenarios. As shown in Table III, HARMOBRIDGE processed native libraries across different apps with varying

[3]Each with over **10 billion historically total downloads** globally across the mobile ecosystem. **Names have been anonymized for compliance and confidentiality considerations.**

success rates. Among libraries utilizing Node-API, binary analysis achieved SumIR generation rates up to 83.33%. The generated SumIR contained between 137 and 6412 instructions per app, representing recovered cross-language semantics that enable comprehensive dataflow analysis.

Scalability and Limitations: Our analysis demonstrates that HARMOBRIDGE can effectively process complex native libraries from real-world apps within reasonable time constraints. The analysis timeouts and absence of SumIR typically occurred in large, highly optimized libraries with complex control flow structures. Success rates for SumIR generation varied across apps, reflecting differences in Node-API usage patterns and implementation complexity.

These results provide encouraging evidence of HARMOBRIDGE's practical applicability while highlighting opportunities for improvement. As the HarmonyOS ecosystem continues to develop, we aim to expand our evaluation to include more apps and assess end-to-end analysis performance incorporating ArkTS components.

VI. DISCUSSION

A. Potential Cross-Language Security Scenarios

Malicious behaviors have been observed in mobile apps that leverage cross-language features by previous works [5], [35], [36], [37], [9], [10], [13]. we found their patterns easy to adapt to HarmonyOS's utilizing its cross-language mechanism. While such exploits haven't been found in HarmonyOS apps, our analysis indicates that these attack patterns would theoretically be detectable by HARMOBRIDGE. We identify the following key attack vectors:

Privacy Data Exfiltration: Sensitive data collected in ArkTS could be passed to native methods for covert transmission. Our benchmark cases in CROSSFLOWBENCH demonstrate HARMOBRIDGE's capability to track such cross-language flows.

Credential and Configuration Hiding: Security-sensitive elements such as server URLs, API keys, and configuration data could be stored in native code and accessed through Node-API calls, which might present challenges for analysis tools focused solely on ArkTS code. HARMOBRIDGE addresses this by resolving constants from native modules into SumIR, benefiting further and comprehensive assessment like characteristic-based malicious app detection.

Evasive Command Execution: Native code may perform privileged operations while maintaining a minimal ArkTS interface. HARMOBRIDGE can analyze external function calls in native code that might relate to system interactions. It is worth noting that HarmonyOS NDK implements sandbox policies [38] that restrict command execution in native, which represents a design that constrains this potential attack surface.

The evolution of the HarmonyOS ecosystem may introduce additional complexity in cross-language interactions. HARMOBRIDGE provides analysis capabilities that aim to address the semantic differences between ArkTS and native code to support security analysis across language boundaries.

B. Limitations And Outlook

Our approach has four principal limitations affecting the completeness and precision of cross-language analysis:

Summary-Based Analysis Limitations: The dataflow-focused approach of HARMOBRIDGE may not fully capture complex control flow semantics in native code, potentially missing vulnerabilities in cases where security-relevant behaviors depend on intricate conditional logic or state machines within native implementations.

Node-API Coverage Constraints: Our implementation covers 53 Node-API functions (Table I), but the complete Node-API [15] specification includes additional functions for multi-threading, asynchronous operations, and complex object binding mechanisms that could affect analysis completeness.

Binary Analysis Scope: Current analysis is limited to individual native libraries without cross-dynamic-library interaction analysis, potentially missing vulnerabilities that span multiple components or involve dynamic library loading.

Evaluation Dataset Constraints: The emerging state of the HarmonyOS ecosystem and restricted access to app markets limits our evaluation to primarily synthetic benchmarks. While CROSSFLOWBENCH covers representative patterns, real-world HarmonyOS app security characteristics remain to be fully documented and explored.

These limitations represent natural extension opportunities rather than fundamental barriers. The rapid evolution of HarmonyOS creates tangible prospects for continued improvement. Node-API function coverage can be systematically expanded through modular extensions as the specification evolves, particularly addressing multi-threading, asynchronous operations, and complex object binding mechanisms that would enhance analysis completeness. As the HarmonyOS ecosystem matures and app diversity increases, the growing availability of real-world apps will naturally enable more comprehensive testing and iterative tool refinement. We look forward to conducting broader testing and iteratively refining our analysis capabilities to better serve the evolving needs of HarmonyOS static analysis.

VII. CONCLUSION

We present HARMOBRIDGE, the first cross-language static analysis framework for HarmonyOS apps that bridges ArkTS and native C/C++ through novel SumIR abstraction. HARMOBRIDGE supports dual-mode analysis (binary/source) with seamless integration into existing HarmonyOS analysis infrastructure. Evaluation shows 81.0% accuracy in recovering cross-language data flows, significantly outperforming baselines treating native calls as opaque operations. We also contribute CROSSFLOWBENCH, an open-source benchmark capturing representative HarmonyOS cross-language patterns. HARMOBRIDGE establishes the foundation for comprehensive cross-language static analysis in the emerging HarmonyOS ecosystem, enabling transparent cross-language reasoning for existing analyzers and contributing to the security and reliability of HarmonyOS apps as the ecosystem evolves.

ACKNOWLEDGMENT

This work was supported in part by the National Natural Science Foundation of China (grants No. 62502168, 62572209), the Hubei Provincial Key Research and Development Program (grant No. 2025BAB057), and the CCF-Huawei Populus Grove Fund.

REFERENCES

[1] "Harmonyos - the next generation distributed operating system," Huawei Technologies Co., Ltd., 2025, (Accessed 2025-07-08). [Online]. Available: https://www.harmonyos.com/en/

[2] "Harmonyos ndk development overview," Huawei Technologies Co., Ltd., 2025, (Accessed 2025-07-08). [Online]. Available: https://developer.huawei.com/consumer/en/doc/harmonyos-guides/ndk-development-overview

[3] L. Huawei Technologies Co., "Arkts: The application development language for harmonyos," 2025, (Accessed 2025-07-08). [Online]. Available: https://developer.huawei.com/consumer/en/arkts/

[4] "Harmonyos node-api overview," Huawei Technologies Co., Ltd., 2025, (Accessed 2025-07-08). [Online]. Available: https://developer.huawei.com/consumer/en/doc/harmonyos-guides/napi-introduction

[5] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. De Geus, C. Kruegel, G. Vigna *et al.*, "Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy," in *The Network and Distributed System Security Symposium 2016*, 2016, pp. 1–15.

[6] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu, "Malton: Towards {On-Device}{Non-Invasive} mobile malware analysis for {ART}," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 289–306.

[7] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. Van Der Veen, and C. Platzer, "Andrubis–1,000,000 apps later: A view on current android malware behaviors," in *2014 third international workshop on building analysis datasets and gathering experience returns for security (BADGERS)*. IEEE, 2014, pp. 3–17.

[8] K. Tam, A. Fattori, S. Khan, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors," in *NDSS Symposium 2015*, 2015, pp. 1–15.

[9] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang, "Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1137–1150. [Online]. Available: https://doi.org/10.1145/3243734.3243835

[10] J. Samhi, J. Gao, N. Daoudi, P. Graux, H. Hoyez, X. Sun, K. Allix, T. F. Bissyandé, and J. Klein, "Jucify: A step towards android code unification for enhanced static analysis," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1232–1244. [Online]. Available: https://doi.org/10.1145/3510003.3512766

[11] C. Sun, Y. Ma, D. Zeng, G. Tan, S. Ma, and Y. Wu, "dep: Mutation-based dependency generation for precise taint analysis on android native code," *IEEE Trans. Dependable Secur. Comput.*, vol. 20, no. 2, p. 1461–1475, Mar. 2023. [Online]. Available: https://doi.org/10.1109/TDSC.2022.3155693

[12] J. Cao, F. Guo, and Y. Qu, "Jnfuzz-droid: a lightweight fuzzing and taint analysis framework for native code of android applications," *Empir. Softw. Eng.*, vol. 30, no. 5, p. 113, 2025. [Online]. Available: https://doi.org/10.1007/s10664-025-10671-9

[13] J. Wang and H. Wang, "Nativesummary: Summarizing native binary code for inter-language static analysis of android apps," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, M. Christakis and M. Pradel, Eds. ACM, 2024, pp. 971–982. [Online]. Available: https://doi.org/10.1145/3650212.3680335

[14] L. Li, X. Gao, H. Sun, C. Hu, X. Sun, H. Wang, H. Cai, T. Su, X. Luo, T. F. Bissyandé, J. Klein, J. C. Grundy, T. Xie, H. Chen, and H. Wang, "Software engineering for openharmony: A research roadmap," *CoRR*, vol. abs/2311.01311, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2311.01311

[15] "Node-api-standard libraries - huawei developers," https://developer.huawei.com/consumer/en/doc/harmonyos-references/napi, 2025.

[16] H. Chen, D. Chen, Y. Yang, L. Xu, L. Gao, M. Zhou, C. Hu, and L. Li, "Arkanalyzer: The static analysis framework for openharmony," 2025. [Online]. Available: https://arxiv.org/abs/2501.05798

[17] A. Ruggia, A. Possemato, S. Dambra, A. Merlo, S. Aonzo, and D. Balzarotti, "The dark side of native code on android," *ACM Trans. Priv. Secur.*, vol. 28, no. 2, Feb. 2025. [Online]. Available: https://doi.org/10.1145/3712308

[18] H. Zhou, H. Wang, S. Wu, X. Luo, Y. Zhou, T. Chen, and T. Wang, "Finding the missing piece: Permission specification analysis for android ndk," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 505–516.

[19] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[20] M. Sun, T. Wei, and J. C. Lui, "Taintart: A practical multi-level information-flow tracking system for android runtime," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 331–342.

[21] L. Xue, C. Qian, H. Zhou, X. Luo, Y. Zhou, Y. Shao, and A. T. Chan, "Ndroid: Toward tracking information flows across multiple android contexts," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 814–828, 2019.

[22] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi, "Droidnative: Automating and optimizing detection of android native code malware variants," *computers & security*, vol. 65, pp. 230–246, 2017.

[23] wala, "Wala, the t. j. watson libraries for analysis," 2025, (Accessed 2025-07-08). [Online]. Available: https://github.com/arguslab/NativeFlowBench

[24] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, M. F. P. O'Boyle and K. Pingali, Eds. ACM, 2014, pp. 259–269. [Online]. Available: https://doi.org/10.1145/2594291.2594299

[25] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. D. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds. IEEE Computer Society, 2015, pp. 280–291. [Online]. Available: https://doi.org/10.1109/ICSE.2015.48

[26] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," *ACM Trans. Priv. Secur.*, vol. 21, no. 3, pp. 14:1–14:32, 2018. [Online]. Available: https://doi.org/10.1145/3183575

[27] R. Vallée-Rai, P. Lam, C. Verbrugge, P. Pominville, and F. Qian, "Soot (poster session): a java bytecode optimization and annotation framework," in *Addendum to the 2000 Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2000, Minneapolis, MN, USA, October 15-19, 2000*, J. Haungs, Ed. ACM, 2000, pp. 113–114. [Online]. Available: https://doi.org/10.1145/367845.368008

[28] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[29] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*, 2016, pp. 265–266.

[30] "Github - travitch/whole-program-llvm: A wrapper script to build whole-program llvm bitcode files," https://github.com/travitch/whole-program-llvm, 2025.

[31] ArgusLab, "Nativeflowbench: Benchmark apps for static analyzing native world of android applications." 2025, (Accessed 2025-07-30). [Online]. Available: https://github.com/arguslab/NativeFlowBench

[32] NativeSummary, "Nativeflowbenchextende," 2025, (Accessed 2025-07-30). [Online]. Available: https://github.com/NativeSummary/NativeFlowBenchExtended

[33] J. Samhi, "Jucify benchapps: Benchmark applications for hybrid program analysis," 2025, (Accessed 2025-07-30). [Online]. Available: https://github.com/JordanSamhi/JuCify/tree/master/benchApps

[34] "F-droid - free and open source android app repository," https://f-droid.org/, 2023.
[35] C. Li, X. Chen, R. Sun, M. Xue, S. Wen, M. E. Ahmed, S. Camtepe, and Y. Xiang, "Cross-language android permission specification," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 772–783.
[36] S. Lee, H. Lee, and S. Ryu, "Broadening horizons of multilingual static analysis: Semantic summary extraction from c code for jni program analysis," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 127–137.
[37] J. Park, S. Lee, J. Hong, and S. Ryu, "Static analysis of jni programs via binary decompilation," *IEEE Transactions on Software Engineering*, vol. 49, no. 5, pp. 3089–3105, 2023.
[38] "Application sandbox-application files-core file kit-application framework - huawei developers," https://developer.huawei.com/consumer/en/doc/harmonyos-guides-V5/app-sandbox-directory-V5, 2025.