Frontiers of Computer Science CN 10-1014/TP
https://doi.org/10.1007/s11704- ISSN 2095-2228
RESEARCH ARTICLE

HACMony: Automatically Detecting Hopping-related Audio-stream Conflict Issues on HarmonyOS

Jinlong He[1], Binru Huang[2,4], Changwei Xia[2,4], Hengqin Yang[2,4], Jiwei Yan[1], Jun Yan[1,2,3,4]

1. Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing, China
2. Hangzhou Institute for Advanced Study, University of Chinese Academy of Sciences, Hangzhou, China
3. University of Chinese Academy of Sciences, Beijing, China
4. Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

Front. Comput. Sci., **Just Accepted Manuscript •**
https://journal.hep.com.cn on

https://doi.org/10.1007/sxxxxx-yyy-zzzz-1

RESEARCH ARTICLE

HACMony: Automatically Detecting Hopping-related Audio-stream Conflict Issues on HarmonyOS

Jinlong He[1], Binru Huang[2,3], Changwei Xia[2,3], Hengqin Yang[2,3], Jiwei Yan[1]✉, Jun Yan[1,2,3,4]

1. Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing, China
2. Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
3. Hangzhou Institute for Advanced Study, University of Chinese Academy of Sciences, Hangzhou, China
4. University of Chinese Academy of Sciences, Beijing, China

Received month dd, yyyy; accepted month dd, yyyy

E-mail: {hejinlong,yanjiwei,yanjun}@otcaix.iscas.ac.cn, yanghq@ios.ac.cn, {huangbinru24,xiachangwei24}@mails.ucas.ac.cn.

Abstract

HarmonyOS is emerging as a popular distributed operating system for diverse mobile devices. One of its standout features is app-hopping, which allows users to switch apps seamlessly across different HarmonyOS devices. However, when apps play *audio-stream-hop* between different devices, they can easily trigger **H**opping-related **A**udio-stream **C**onflict (HAC) scenarios. Improper resolution of HAC will lead to significant HAC issues, which are hard to detect comprehensively due to the unclear semantics of HarmonyOS's app-hopping mechanism and the lack of effective multi-app hopping testing methods. To fill the gap, this paper introduces an automated and efficient approach to detecting HAC issues. We formalized the operational semantics of HarmonyOS's app-hopping mechanism for audio streams for the first time. Leveraging this formalization, we designed an **A**udio-stream-aware **S**tate **T**ransition **G**raph (ASTG) to model the behaviors of audio-streams during window transitions and proposed a model-based approach to detect HAC issues automatically. Our techniques were implemented in a tool, HACMony, and evaluated on 20 real-world HarmonyOS apps. Experimental results reveal that 12 of the 20 apps exhibit HAC issues. Among the 53 HAC issues detected, a total of 18 unique HAC issues were manually confirmed. Additionally, we summarized the detected issues into two typical types, namely MoD and MoR, and analyzed their characteristics to assist and guide both app and OS developers.

Key words

HarmonyOS, Audio-Stream Conflict, App-Hopping, Mobile Testing, Large Language model

1 Introduction

The use of audio-stream is prevalent in mobile applications, covering a range of use cases from simple music playing to complex audio processing and interaction. When more than one apps use audio streams on a single device, their audio streams may conflict and require proper handling. For example, users may launch a music app to play a song first and then switch to a movie app to play a video, both of which involve audio streams. However, if neither the music app nor the movie app handles the played audio streams according to the scenario, i.e., just let the two started audios play at the same time, users may feel confused and uncomfortable.

When there are conflicts on multiple audio-streams, there are specific coping solutions according to experience. In this example, users usually expect the music-playing can be paused automatically to ensure the normal playing of the newly launched video. To enhance users' experience, existing mobile systems typically offer an *audio-focus* feature to resolve such **A**udio-stream **C**onflicts (ACs). When an app attempts to play an audio, the system requests focus for the audio stream. Only the audio stream that gains the focus can be played, i.e., if the request is rejected, the audio stream cannot be played. If an audio stream is interrupted by another one, it loses audio focus and is expected to take actions like *pause*, *stop*, or *lower volume* to avoid unexpected AC-related issues.

As we can see, handling multiple audio app interaction scenarios on a single device is inherently complex. Fortunately, as apps undergo iterative updates, most app developers have made efforts to design proper and effective conflict-handling solutions for their apps. Nowadays, with the rise of multi-device distributed operating systems [1], applications can not only be used on a single device but can also be migrated to other devices through hopping operations. These emerging operating systems aim to enhance users' experience, but they also make audio app interactions scenarios much more complex. In such a context, the existing conflict-handling solutions designed for single-device scenarios often lose effectiveness. Thus, when developing apps working on distributed operating systems, the audio-stream conflict handling scenarios on multiple devices should be comprehensively

tested.

In recent years, as the most representative distributed mobile operating system, HarmonyOS has achieved remarkable success and is running on more than one billion devices [2]. Developed by Huawei, HarmonyOS is a distributed platform designed for seamless integration across smartphones, tablets, smart TVs, and more. A standout feature of HarmonyOS is *app-hopping* [3], a distributed operation mode that plays a fundamental role in its ecosystem. This functionality allows users to seamlessly transfer apps across different devices, enhancing convenience and flexibility. However, this innovation also complicates the resolution of ACs due to the increased interplay between devices. Through a preliminary investigation, we found that many users had complained about poor experiences caused by **Hopping-related Audio-stream Conflict** (HAC) issues [4, 5], where the audio-stream conflicts that occur during HarmonyOS's app-hopping are improperly handled. Given the significant disruption HAC issues cause to the user experience during app-hopping across multiple HarmonyOS devices, this paper focuses on how to detect HAC issues automatically and efficiently, alongside analyzing existing HAC issues to provide deeper insights.

To achieve that, it is crucial first to understand how HarmonyOS's app-hopping mechanism operates and design an efficient test generation approach tailored for app-hopping scenarios. **The first major challenge lies in the lack of semantics for the app-hopping mechanism.** Through an investigation of the official documentation, we found that existing materials focus on highlighting the benefits of app-hopping but lack detailed descriptions of the underlying mechanism. This lack of clarity significantly complicates the design of effective testing approaches for app-hopping. Specifically, it increases the difficulty of determining when and how to perform hopping operations that are more likely to trigger HAC issues. Moreover, this omission also impedes other hopping-related research efforts. **The second challenge is lacking hopping-specific models designed for efficient testing.** Although there are various models designed for mobile apps' GUI testing [6–18], there is no HAC-specific model, that describes the behaviors of audio streams of an app and can guide a compact test case generation. Without such a model, it would be difficult to design an effective testing strategy to detect HAC issues.

To address ***Challenge 1***, we meticulously picked several representative HarmonyOS native apps, designed and conducted a group of semantic experiments on app-hopping operations, and summarized the behaviors of app-hopping operations according to the experimental results. Based on that, we first present the formalized operational semantics of HarmonyOS's app-hopping mechanism in the aspect of audio stream. To address ***Challenge 2***, we propose an extended FSM [6] called **Audio-stream-aware State Transition Graph** (ASTG) to describe the behaviors of audio streams. Its node, **Audio-Stream-aware State** (ASS), denotes the window and its audio stream status in a running app; while its edge describes the transition rule between ASSs with the label of *GUI events*. To accurately and efficiently construct ASTG model, we propose an ASS-targeted and LLM-driven model exploration approach, which adopts LLM to identify and prioritize GUI events capable of triggering audio-stream interactions, then utilizes this information to guide the dynamic exploration of the app under test. As this exploration approach can only explore the audio-stream statuses without multiple apps' interaction, we also propose an ASS-guided enhancement approach to simulate the multi-app environment for extracting the extra ASSs and then construct a more precise ASTG. Based on both the operational semantics of HarmonyOS's app-hopping mechanism and the fine-grained ASTG model, we can finally generate targeted test cases and execute them to detect HAC issues.

We implemented our proposed techniques into a tool called HACMony (**H**opping-related **A**udio-stream **C**onflict issues for Har**Mony**OS) and evaluated it on 20 real-world popular HarmonyOS apps. The experimental results demonstrate that the proposed testing approach can efficiently detect HAC issues. Among the 20 apps, 12 were found to have HAC issues. Among the 53 HAC issues detected, there are 18 unique HAC issues, which are all manually confirmed. Through issue analysis, we categorized the identified HAC issues into two types: **Misuse of Device** (MoD) and **Misuse of Resolution** (MoR). We further summarized their characteristics and possible causes to provide deeper insights for both application and OS developers.

The main contributions of this work are summarized as follows:

- We present the first formal semantics of the HarmonyOS app-hopping mechanism, which serves as a foundation for HAC issue testing and could inspire further research.
- We design the ASTG model to describe the transitions of ASSs in HarmonyOS apps and propose a LLM-driven dynamic exploration approach to construct ASTG models. The approach is implemented into a tool HACMony [1], which is evaluated on 20 real-world apps and successfully discovered 18 unique HAC issues.
- We summarize two typical types of HAC issues, namely MoD and MoR, and analyze their possible causes. These findings can assist and guide both app and OS developers in improving the apps' quality on distributed mobile systems.

2 Background

This section introduces the basic concepts around HarmonyOS apps and audio streams. We also give a motivating example to illustrate the behavior of a real HAC issue.

2.1 HarmonyOS: Architecture and Application

HarmonyOS is designed with a layered architecture, which from bottom to top consists of the *kernel layer*, *system server layer*, *framework layer*, and *application layer*. Figure 1 illustrates the layered architecture of HarmonyOS [19, 20]. The application layer is composed of Android (AOSP) apps and HarmonyOS native (OpenHarmony) apps, which achieves binary compatibility. In the framework layer, the ABI-compliant Shim (application binary interface compliant layer) redirects Linux syscalls into IPCs, channeling them towards appropriate OS services. This mechanism effectively addresses compatibility issues with AOSP and OpenHarmony, as noted in [20]. The system service layer offers a comprehensive set of capabilities crucial for HarmonyOS to

[1] Available at `https://github.com/SQUARE-RG/hacmony`

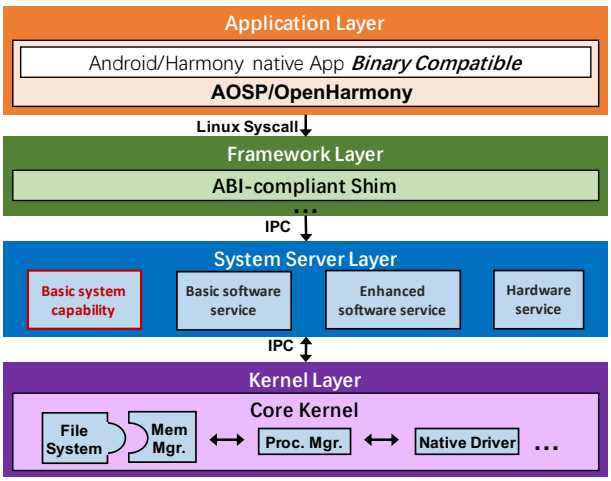


Fig. 1 HarmonyOS Architecture

provide services to applications. It consists of a basic system capability subsystem, a basic software service subsystem, an enhanced software service subsystem, and a hardware service subsystem. The kernel layer, through its core kernel, furnishes memory management, file system management, process management, and native driver functionality.

With this architecture, especially the design of *ABI-compliant shim*, HarmonyOS can support both AOSP [21] (for Android apps) and OpenHarmony [22] (for native apps). Notably, the distributed operation *app-hopping* is implemented within the *basic system capability* subsystem, which transports Android and HarmonyOS native apps to another HarmonyOS device through the distributed soft bus in the same way. In this paper, we take both of the two supported types of apps on HarmonyOS as *HarmonyOS apps*.

2.2 Audio Stream

Audio streaming is a technology that allows users to transmit and receive audio data in real-time over the internet. Audio streaming is commonly used in online music services, internet radio, podcasts, and other applications that require instant audio content transmission.

In general, an app typically has three *fundamental* audio-stream statuses when no other app is playing audio stream, i.e., STOP, PAUSE, and PLAY. In real-world scenarios, audio streams are prone to conflicts when apps interact. When such conflicts occur, the behavior of the audio playback in an app becomes more complex. To mitigate the interference impact of the conflict on users, the app will often temporarily lower the volume or pause the audio stream to avoid simultaneous playback. As a consequence, during the occurrence of these conflicts, an app has two *extra* audio-stream statuses, i.e., PLAY$^{\downarrow}$ and PLAY$^{\|}$. Specifically, when an app play together with another app, PLAY$^{\downarrow}$ signifies one app lower the volume, and PLAY$^{\|}$ signifies one app pause the playback and play again when the conflict disappears. As shown in Table 1, we consider the listed five audio stream statuses in this paper.

Furthermore, HarmonyOS adopt *audio focus* to manage audio streams from different apps to reconcile the audio-stream conflicts. When an audio stream requests or releases audio focus, the system manages focus for all streams based on predefined audio focus policies. These policies determine which audio stream can operate normally and which must be interrupted or subjected to other actions. The system's default audio focus policy primarily relies on the type of audio stream and the order in which the audio streams are initiated [23]. In HarmonyOS, "StreamUsage" is an enumeration type to define audio stream categories. It plays a crucial role in audio playback and management. The commonly used values include STREAM_USAGE_MUSIC (MUSIC), STREAM_USAGE_MOVIE (MOVIE), STREAM_USAGE_NAVIGATION (NAVIG), and STREAM_USAGE_VOICE_COMMUNICATION (COMMU) [24].

Table 1 Description of Audio Stream Statuses

Audio-stream Status	Description
STOP	stop the playback
PAUSE	pause the playback
PLAY	play the playback
PLAY$^{\downarrow}$	lower the volume, restore after conflict disappear
PLAY$^{\|}$	pause the playback, play after conflict disappear

Table 2 lists typical resolutions for solving ACs based on audio stream types by HarmonyOS, where app "pre" plays audio streams first and then app "post" plays at a later time. Although these resolutions are recommended ones, HarmonyOS also allows developers to handle conflicts on their own. This leads to different proper resolutions for solving conflicts for real-world apps in practice.

Table 2 Typical Resolutions for Solving the ACs, where ◑: app "pre" lowers the volume, after app "post" releases the audio focus, app "pre" restores the volume. ◐: app "post" lowers the volume, after app "pre" releases the audio focus, app "post" restores the volume. ▶: app "pre" and "post" play together. ||: app "pre" pauses the playback, after app "post" releases the audio focus, app "pre" plays again. □: app "pre" stops the playback.

		Type of app "post"			
		MUSIC	MOVIE	NAVIG	COMMU
Type of app "pre"	MUSIC	□	□	◑	||
	MOVIE	□	□	◑	||
	NAVIG	◐	◐	□	□
	COMMU	◐	◐	▶	||

2.3 Motivating Example

To show the motivation for this work, we use a navigation app, *AMap* [25], running on a phone, and a music app, *Kugou Music* [26], running on a tablet for illustration. As shown in Figure 2, the initial scenario is depicted in ①, where both apps, *AMap* and *Kugou Music*, play their audio streams at normal volume. When the user launches *Amap* on the tablet and navigating in ①, the app *AMap* plays the audio stream with the normal volume, but *Kugou Music* lowers the volume to avoid the audio-stream conflict, whose status is displayed in ②. When the user clicks the "recent" button on the phone in ①, the interface

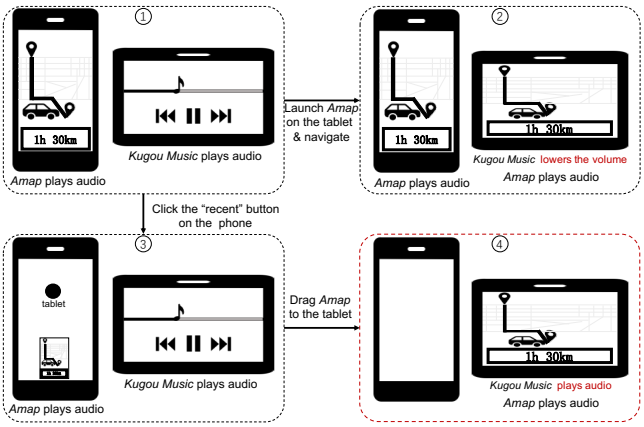


Fig. 2 Motivating Example

on the phone changes the interface for selecting the hopping app and target device, which is shown in ③. When the user drags the app *Amap* to the tablet on the phone in ③, the app *Amap* will be hopped to the tablet. However, in this situation, both *AMap* and *Kugou Music* play their audio streams at normal volume on the tablet, which is not expected. Since *Kugou Music* fails to lower its volume, users may have difficulty hearing navigation instructions from *AMap*. In the context of in-vehicle infotainment systems, such conflicts could even pose safety risks.

To uncover hidden vulnerabilities that can be triggered by the app-hopping operation on HarmonyOS, two key tasks must be accomplished. First, it is essential to understand the operational semantics of HarmonyOS's app-hopping mechanism. Besides, an efficient test generation approach tailored specifically for app-hopping scenarios should be designed.

3 The Operational Semantics of App-Hopping Mechanism on HarmonyOS

In this section, we describe the overview of the app-hopping mechanism and specify the mechanism as an operational semantics.

3.1 The Overview of App-Hopping

HarmonyOS provides the *Virtual Super Device* (Super Device) to integrate multiple physical devices and allow one device to share data and apps among devices with distributed communication capabilities. App-hopping is the fundamental feature of the Super Device to share the apps among devices [3].

When hopping an app a from device d to device d', the app a will seamlessly transfer from device d to d', i.e., it will be displayed on the screen of device d' only. Users could end a hop at any time when there is an app hop in the super device. Ending the hop of app a will let app a return to device d. To obtain HarmonyOS's hopping mechanism, We picked several representative HarmonyOS native apps to explore the behavior of app-hooping among multiple devices. By checking the official documents as well as conducting a group of experiments, we found that there is at most one app hop held in the super device in HarmonyOS. That is, if app a has been hopped from device d to device d' and the users hop another app a' in the super device, the hop of app a will be ended automatically. Furthermore, when considering the audio stream of apps, the behaviors of starting a hop and ending a hop will be more complicated. When starting a hop of app a that is playing music on device d to another device d', then app a will play music on device d'. If there is another app playing music on device d' before the hop of app a, the audio-stream conflict will occur on the device d', which should be carefully addressed to avoid HAC issue happen.

3.2 The operational Semantics of App-Hopping

According to our literal and experimental investigation, we first summarize the formal semantics of HarmonyOS's app-hopping mechanism, where the semantics of this mechanism have also been verified through the review of official materials. In this part, we specify its operational semantics to help users to better understand the app-hopping behaviors. Figure 3 defines domains, stacks, and operations to describe the operational semantics. We write a for an app name, and d for a device name. An app instance is a pair of its activity name, and audio stream status (a, μ). An app stack α is a sequence of app instances. A device instance is a pair of its device name and its app stack (d, α). A device stack β is a sequence of device instances. A hopping relation r is either a triple of source device name, app name and target device name (d, a, d'), or a dummy symbol ϵ representing no hop exists in the super device.

The operational semantics are defined as the relation of the form $\langle \beta, r\rangle \overset{C}{\longmapsto} \langle \beta', r'\rangle$, where the current devices stack is β and the current hopping relation is r, the operation C resulting in the new devices stack β' and the new hopping relation r'. The typical behaviors of StartHop and EndHop operations are as follows:

$$\frac{\begin{array}{c}\beta = \beta_1 :: (d_s, \alpha_s) :: \beta_2 :: (d_t, \alpha_t) :: \beta_3 \quad \alpha_s = \alpha_1 :: (a, \mu) :: \alpha_2 \\ A = (a, \mu) \quad r = (d_s, a, d_t) \quad \alpha_s' = rmv(A, \alpha_s) \quad \alpha_t' = add(A, \alpha_t)\end{array}}{\langle \beta, \epsilon\rangle \overset{d_s.\mathsf{StartHop}(a,d_t)}{\longmapsto} \langle \beta_1 :: (d_s, \alpha_s') :: \beta_2 :: (d_t, \alpha_t') :: \beta_3, r\rangle}$$

$$\frac{\begin{array}{c}\beta = \beta_1 :: (d_s, \alpha_s) :: \beta_2 :: (d_t, \alpha_t) :: \beta_3 \quad \alpha_t = \alpha_1 :: (a, \mu) :: \alpha_2 \\ A = (a, \mu) \quad r = (d_s, a, d_t) \quad \alpha_s' = add(A, \alpha_s) \quad \alpha_t' = rmv(A, \alpha_t)\end{array}}{\langle \beta, r\rangle \overset{\mathsf{EndHop}}{\longmapsto} \langle \beta_1 :: (d_s, \alpha_s') :: \beta_2 :: (d_t, \alpha_t') :: \beta_3, \epsilon\rangle}$$

a	$\in$	App		*application name*
d	$\in$	Device		*device name*
μ	$\in$	Audio	$=$	$\{\mathsf{PLAY}, \mathsf{PAUSE}, \mathsf{STOP}, \mathsf{PLAY}^{\downarrow}, \mathsf{PLAY}^{\|}\}$
r	$\in$	HopRelation	$=$	$\mathsf{Device} \times \mathsf{App} \times \mathsf{Device} \cup \{\epsilon\}$
A	$\in$	AppInst	$=$	$\mathsf{App} \times \mathsf{Audio}$
D	$\in$	DeviceInst	$=$	$\mathsf{Device} \times \mathsf{AppStack}$
α	$::=$	$\epsilon \mid A :: \alpha$	$\in$	AppStack
β	$::=$	$\epsilon \mid D :: \beta$	$\in$	DeviceStack
C	$::=$	EndHop	$\mid$	d.StartHop(a, d)

Fig. 3 Domains, Stacks, and Operations

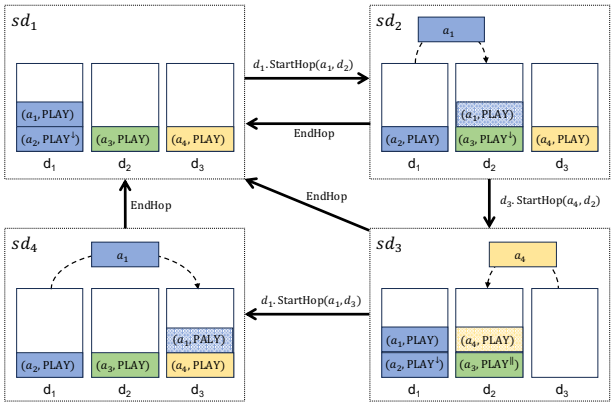


Fig. 4 Example of HarmonyOS's App-Hopping Mechanism

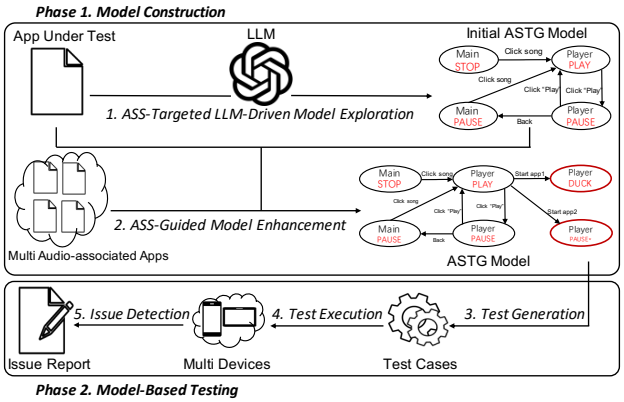


Fig. 5 HACMony's Workflow

The first specifies that if a user hops an app when there is no hop in the super device, the app will be moved to the target device, and the other apps in the source (resp. target) device will change the audio stream status according to the function *rmv* (resp. *add*). The second describes that if a user ends a hop of an app, the behavior of this operation is dual to that of the first.

Intuitively, the function $rmv(A, \alpha)$ (resp. $add(A, \alpha)$) indicates the behaviors of removing (resp. adding) an app instance A from (resp. into) a given app stack α. Moreover[2],

- if app instance A is in the status PLAY, and there exists another app instance A' in the status $\text{PLAY}^{\downarrow}$ or $\text{PLAY}^{\parallel}$, $rmv(A, \alpha)$ will let A' turn into PLAY,
- if A is in the status of $\{\text{PLAY}, \text{PLAY}^{\downarrow}, \text{PLAY}^{\parallel}\}$, and there exists another app instance A' in the status PLAY, $add(A, \alpha)$ will lead to the audio-stream conflict, the status of app instance A' will change according to the resolution to solve the conflicts (refer to Section 4.3.3).

Multiple-Device App-Hopping Example. In the following, we use an example to illustrate the operational semantics of the HarmonyOS app-hopping mechanism. Suppose that there are three devices d_1, d_2, d_3 in the super device, and four apps a_1, a_2, a_3, a_4 running on these devices. The types of audio streams used for each app are as follows, a_1 : NAVIG, a_2 : MOVIE, a_3 : MUSIC, and a_4 : COMMU. To simplify the complicated process, we suppose all the resolutions to solve audio stream conflicts following the typical resolutions listed in Table 2. As shown in Figure 4, there are four cases of the super device sd_1, sd_2, sd_3, sd_4. For each $i \in [1, 4]$, we let $sd_i = \langle \beta_i, r_i \rangle$ where $\beta_i = (d_1, \alpha_{i,1}) :: (d_2, \alpha_{i,2}) :: (d_3, \alpha_{i,3})$. For instance, $\alpha_{1,1} = (a_1, \text{PLAY}) :: (a_2, \text{PLAY}^{\downarrow})$ is the app stack of the device d_1 in the super device sd_1. The semantics of the app-hopping mechanism are illustrated by the following cases.

- When the operation d_1.StartHop(a_1, d_2) is applied to sd_1, the app instance (a_1, PLAY) will be removed from $\alpha_{1,1} = (a_1, \text{PLAY}) :: (a_2, \text{PLAY}^{\downarrow})$. Since the audio stream status of a_2 is $\text{PLAY}^{\downarrow}$, it will turn to PLAY , resulting in $\alpha_{2,1} = (a_2, \text{PLAY})$. Moreover, the app instance (a_1, PLAY) will be added into the device d_2, and request the audio focus of device d_2, then app instance of a_3 will turn to $\text{PLAY}^{\downarrow}$, resulting in $\alpha_{2,2} = (a_1, \text{PLAY}) :: (a_3, \text{PLAY}^{\downarrow})$.
- When the operation EndHop is applied to sd_2, since there is already a hop $r_2 = (d_1, a_1, d_2)$, the app instance (a_1, PLAY) will be moved back to device d_1 from d_2. Moreover, (a_1, PLAY) will be removed from $\alpha_{2,2} = (a_1, \text{PLAY}) :: (a_3, \text{PLAY}^{\downarrow})$. Since the audio stream status of a_3 is $\text{PLAY}^{\downarrow}$, it will then turn to PLAY, resulting in $\alpha_{1,2} = (a_3, \text{PLAY})$. Then the app instance (a_1, PLAY) will be added into the device d_1, and request the audio focus of device d_1, then app instance of a_2 will turn to $\text{PLAY}^{\downarrow}$, resulting in $\alpha_{1,1} = (a_1, \text{PLAY}) :: (a_2, \text{PLAY}^{\downarrow})$.
- When the operation d_3.StartHop(a_4, d_2) is applied to sd_2, since there is already a hop $r_2 = (d_1, a_1, d_2)$, it will end the previous hop first. That is, the case turns to sd_1. Then it will hop a_4 from device d_3 to device d_2. Since the audio stream conflict resolution for app pair (pre:a_3, post:a_1) is different from pair (pre:a_3, post:a_4) according to their types, the audio stream status of a_3 is $\text{PLAY}^{\parallel}$ instead of $\text{PLAY}^{\downarrow}$ in this case.
- When the operation d_1.StartHop(a_1, d_3) is applied to sd_2, it is similar to the previous case.

It shows that during the app-hopping, audio-stream conflicts may arise between the hopping app and audio-stream-using apps on both original device and target device, thereby altering their audio-stream statuses.

4 Model-based Testing Approach for HAC Issue Detection

In this section, we present the overview and design details of the model-based testing approach for automatically detecting HAC issues.

4.1 Approach Overview

Based on the knowledge of the HarmonyOS's app-hopping mechanism, we design a model-based automatic testing approach for HAC issue detection. Figure 5 presents an overview of HACMony's architecture and workflow, which has two key phases.

Phase 1: Model Construction. To obtain the GUI events that can influence the statuses of audio streams in further audio-directed testing, we designed a new model called **A**udio-stream-aware **S**tate **T**ransition **G**raph (ASTG). The node, Audio-Stream-aware State (ASS), of ASTG

[2]Due to the space limitation, we describe the remaining rules and helper functions in a companion report [27].

denotes a pair of the window and its associated audio-stream status. This binding arises from the fact that audio stream utilization is normally achieved through windows within an app, where multiple windows may coexist to manage audio streams. Therefore, we employ ASSs to explicitly define the audio-stream statuses of these windows, with the aim of testing HAC issues.

To construct ASTG model, We first employ a dynamic ASS-targeted app exploration strategy enhanced by large language model (LLM) analysis to identify and prioritize GUI events capable of triggering audio-stream interactions. Specifically, for each single app, the LLM-driven analysis examines its window components and corresponding event handlers to identify events that may activate audio streams (e.g., play buttons, volume sliders). This refined process of identifying events is then used to construct the initial ASTG model (step 1, details in Section 4.2.1). As the single-app exploration misses the audio-stream statuses (e.g., $\mathsf{PLAY}^{\downarrow}$ and $\mathsf{PLAY}^{\|}$) that happen during the interaction of multiple apps, we enhance the initial ASTG model by collaborating with multiple apps to explore extra ASSs (step 2, details in Section 4.2.2).

Phase 2: Model-Based Testing. To generate the compact test suite for audio-aware hopping behavior testing, we select the ASSs that are in the PLAY-like statuses (called $\mathsf{ASS}_{\mathsf{PLAY}}$) from the ASTG model constructed by Phase 1, and configure the devices according to different app-hopping operations (step 3, details in Section 4.3.1). Then we execute the test cases on multiple devices to detect whether there are HAC issues (step 4, details in Section 4.3.2). Finally, by analyzing the resolution to solve the audio-stream conflicts during hopping and checking whether it is consistent with the resolution on a single device, HACMony can automatically report HAC issues (step 5, details in Section 4.3.3).

4.2 ASTG Model Construction

To generate the test case for detecting the HAC issues, we define an extended FSM, **Audio-stream-aware State Transition Graph** (ASTG), to represent the audio-stream-level behavior of an app. An ASTG model is a triple $G = (S, T, s_0)$, where

- S is a finite set of app's Audio-Stream-aware States (ASSs). A state $s \in S$ is a pair $\langle win, stat \rangle$ where win denotes the GUI window, which contains the screenshot as well as the *element hierarchical tree*, $stat \in$ Audio denotes audio-stream status, and $s_0 \in S$ is the initial ASS of the app.
- T denotes the set of transitions. An element $\tau \in T$ is a triple $\langle s, e, t \rangle$ representing the transition from the source ASS s to the destination ASS t caused by a GUI event e, e.g., click or drag.

4.2.1 ASS-Targeted and LLM-Driven Model Exploration

To conduct more effective ASS-targeted GUI exploration, we utilize an LLM-driven analysis to comprehensively understand the tested app to obtain the available audio-stream types, e.g. MUSIC, and the semantics of GUI components to pick the optimal event that can enable the app to play the audio stream corresponding to the available type. After identifying the optimal audio-related event in current window, we proceed to execute the event on the device and collect the information about the changes (e.g., GUI window changes). If the app has deviated from the exploration goal after previous event execution, the LLM will re-evaluate the identified event and select an alternative event that are more likely to lead the app towards playing the audio stream. This iterative process of event identification, execution, information collection, and verification continues until the app successfully plays the audio stream.

Algorithm 1 describes the ASS-targeted LLM-driven exploration approach. The input of the approach is the *tested_app* to be explored and the empty ASTG G, the output is the ASTG G of the *tested_app*. First, it initializes the variable *feedback*, which indicates the feedback of the event execution, as empty (line 1). It also obtains the audio-stream types *audios* available for the *tested_app* via UnderstandApp() function (line 2). Then, for each *audio* to explore from the available *audios*, it repeats the following process until the variable *feedback.terminated* becomes *True*, i.e., the app successfully play *audio* (lines 3-19).

1. Obtain the current ASS = $(win, stat)$ via the function GetASS(), and pass the current window win to the LLM for understanding, so as to obtain a set of *GUI elements* containing semantic information via UnderstandWin() function (lines 6-7).
2. Send the *audio* to explore, the current *GUI elements*, the current ASTG G, and the $feedback$ of the previous event execution, to the LLM. The LLM then selects the "optimal" event from all possible events based on the goal of enabling the app to play the audio stream (line 8). Then it execute the "optimal" event *event* (line 9).
3. Obtain the current ASS = $(win', stat')$ after executing the "optimal" event via GetASS() again (line 10), then update the ASTG G (lines 11-13).
4. Verify whether the "optimal" event deviates from the exploration goal and whether the current exploration can be terminated, and record such information in the feedback. If it does deviate from the goal, then restart the *test_app* to conduct a more target-directed exploration (lines 14-17).

LLM Prompt construction: The prompts used for interaction with the LLM in each exploration step are presented in Table 3, which will be elaborated as follows.

- UnderstandApp. The prompt directs the LLM to identify an application's supported audio stream types by analyzing its interface screenshots. This classification is governed by a set of heuristic rules that associate specific visual cues with four predefined categories, where: media controls such as play/pause buttons and playback sliders are indicative of the Music type; video thumbnails and player windows correspond to Video; map interfaces and route indicators signify Navigation; and call buttons or voice message icons suggest Communication.
- UnderstandWin. This prompt asks LLM to understand the semantics of each GUI element and specifies the output format. The LLM describes each element's function based on screenshots and individual element images. Each element is numbered to ensure descriptions are in order and no descriptions are missed, preventing parsing errors.
- GetOptimalEvent. This prompt asks the LLM to generate the

Algorithm 1: Exploration()

input : $G = (S, T, s_0), tested\_app$

```
1  feedback ← [];
2  audios ← UnderstandApp(tested_app);
3  for each audio in audios do
4      feedback.terminated ← False;
5      while feedback.terminated = False do
6          ⟨win, stat⟩ ← GetASS();
7          elements ← UnderstandWin(win);
8          event ← GetOptimalEvent(
             audio, elements, G, feedback);
9          event.execute();
10         ⟨win′, stat′⟩ ← GetASS();
11         S ← S ∪ {⟨win, stat⟩, ⟨win′, stat′⟩};
12         τ ← ⟨⟨win, stat⟩, event, ⟨win′, stat′⟩⟩;
13         T ← T ∪ {τ};
14         feedback ← Verify(stat′, event, win, win′);
15         if feedback.validity = False then
16             Restart the tested_app;
17         end
18     end
19 end
```

next event based on the current app state (GUI elements and the step for exploration goal). The LLM selects the optimal event, considering previous steps and feedback to avoid repetition.

- Verify. This prompt asks LLM to validate the executed event to checks if the event meets the exploration goal and causes GUI changes. It analyzes deviations, screen changes, and completion. It also suggests the next step to guide future event selection, and determine whether the audio-stream with the current audio type is played.

4.2.2 ASS-Guided Model Enhancement

As mentioned in Section 2.2, the audio stream statuses PLAY$^{\downarrow}$ and PLAY$^{\|}$ occur only when there is another app requesting the audio stream focus. To explore the extra audio stream statuses, we need to launch another app and execute specific events to make it use the audio stream and cause audio-stream conflicts. For different audio stream statuses, the collaborating apps may be different in general, so we select a set of representative apps that use different types of audio streams to explore these statuses. The principle of the collaborating apps selection is primarily based on the typical resolutions for solving the ACs (see Table 2). For example, the app with the type NAVIG (resp. COMMU) is more likely to be selected to explore PLAY$^{\downarrow}$ (resp. PLAY$^{\|}$) status for the app with MUSIC type.

Algorithm 2 describes the ASS-guided model enhancement approach. It takes the previously constructed ASTG $G = (S, T, s_0)$ by Algorithm 1, the previously tested app *tested_app* and an audio-associated app set *enhanced_apps* as inputs, and takes the ASTG G enhanced with extra ASSs as output. First, for the *tested_app*, it finds out all the ASSs in its ASTG where $stat$ = PLAY as the target ASSs. Then for each target ASS, according to the ASTG G, it obtains and executes the events to switch the *tested_app* to the ASS status (lines 1-3). For each *enhanced_app* in the audio-associated apps set *enhanced_apps*, it launches *enhanced_app* and switches *enhanced_app* to PLAY status to make its audio stream conflict with the *explored app* (lines 4-6). Finally, if the *target app* reaches a new ASS, we add the new ASS as well as the corresponding transition into the ASS set S and transition set T, respectively (lines 7-13).

Algorithm 2: Enhancement()

input : $G = (S, T, s_0), tested\_app, enhanced\_apps$

```
1  for each ⟨win, stat⟩ in S do
2      if stat = PLAY then
3          Switch to ⟨win, stat⟩;
4          for each enhanced_app in enhanced_apps do
5              Launch enhanced_app and switch
                 enhanced_app to PLAY status;
6              ⟨win′, stat′⟩ ← GetASS();
7              if stat ≠ stat′ then
8                  S ← S ∪ {⟨win′, stat′⟩};
9                  e ← launch enhanced_app and execute
                     enhanced_app;
10                 τ ← ⟨⟨win, stat⟩, e, ⟨win′, stat′⟩⟩;
11                 T ← T ∪ {τ};
12             end
13             End tested_app and switch back to ⟨win, stat⟩;
14         end
15     end
16 end
```

4.3 Model-Based HAC Issue Detection

Upon ASTG model construction, this section presents our model-based testing approach for detecting HAC issues[3].

4.3.1 HAC-Directed Test Generation

According to the operational semantics of HarmonyOS's app-hopping mechanism, we have two typical app-hopping commands StartHop and EndHop, hence two types of hopping-related test cases **Test$_{\text{StartHop}}$** and **Test$_{\text{EndHop}}$** should be generated for each tested app. The basic test generation idea is to select ASS$_{\text{PLAY}}$, the ASSs in the PLAY-like statuses in the ASTG model, and configure the devices according to different app-hopping operations. For an ASTG $G = (S, T, s_0)$, an ASS = $\langle win, stat\rangle \in S$ is an ASS$_{\text{PLAY}}$, if $stat \in \{\text{PLAY}, \text{PLAY}^{\downarrow}, \text{PLAY}^{\|}\}$. Intuitively, ASS$_{\text{PLAY}}$ indicates the audio stream status of the window is PLAY or will turn to PLAY after other apps release the focus.

Generate Test$_{\text{StartHop}}$. A test case **Test$_{\text{StartHop}}$** is to perform the process of hopping the tested app where the window is in the PLAY-like status to another device that is utilizing the audio stream. With a tested app a and an audio-associated app a', for each ASS$_{\text{PLAY}}$ s in the ASTG of app a, we can get the following test case, $E_s :: E_{a'} :: d_1.\text{StartHop}(a, d_2)$, where

- E_s is the event sequence that should be executed on device d_1 to let app a reach ASS$_{\text{PLAY}}$ s from the initial ASS s_0.

[3] We mainly consider the two-device hopping testing scenario as it is the most basic and common scenario and can cover many basic HAC issues.

Table 3 The LLM Prompt Templates for ASS-Targeted Exploration

UnderstandApp	UnderstandWin	GetOptimalEvent	Verify

Prompt:
Based on the information of the provided screenshots of a mobile app interface, **complete the task of** identifying which audio-stream types the app supports. **Note that**, return the answer as a Python list; base your identification on UI elements: use maps or routes as evidence for **Navigation**; media controls or thumbnails for **Music** or **Video**; and call icons or "Messages" tabs for **Communication**.
 	
Example outputs:
`["Navigation", "Music", "Video", "Communication"]`
	Prompt:
	Based on the information of the provided screenshot of a mobile app interface and images of clickable components, **complete the task of** analyzing each component image in order to describe its function. **Note that**, return the answer as a Python list; each description should be concise and functional; merge components with identical functions into a single, generalized description, while keeping main navigation tabs separate.
	
	Example outputs:
	`["Return button", "Search box", "Settings button", "Add device"]`
		Prompt:
		Based on the information of the target scenario, current screen, clickable elements, and previous feedback, **complete the task of** determining the next event. **Note that**, focus on functionality and adapt to the current screen; choose the most appropriate element with the same purpose; avoid repeating previous events; if the target element is not visible, your priority operation should be to swipe to reveal more content; respond only in JSON format.
		
		Example outputs:
		`{"event_type": "click", "id": 3}`
		`{"event_type": "input", "id": 2, "text": "music"}`
			Prompt:
			Based on the information of the target scenario, screenshots and elements changes before and after the event, **complete the task of** evaluating whether the event follows previous steps and progresses toward the exploration goal. **Note that**, check for unnecessary repetition of operations; verify if the played audio stream type matches the exploration type; assess whether to terminate based on the audio stream type and status; provide suggestions for the next step.
			
			Example outputs:
			`{"validity": true/false, "terminated": true/false, "suggestion": "Select a "Search" button"}`

- $E_{a'}$ is the event sequence that should be executed on device d_2 to let app a' reach an ASS whose status is PLAY from s_0.
- d_1.StartHop(a, d_2) is the event that hopping the tested app a from device d_1 to d_2.

Generate Test$_{\text{EndHop}}$. A test case **Test$_{\text{EndHop}}$** is to perform the process of ending a hop of the tested app where a window is in the PLAY-like status to another device that is utilizing the audio stream. Generating the test case **Test$_{\text{EndHop}}$** is more complicated than **Test$_{\text{StartHop}}$**, since before ending a hop, we need to construct a hop between these two devices. Similarly, a test case **Test$_{\text{EndHop}}$** generated can be formally defined as $E_s :: E_{a'} ::$ EndHop, where

- E_s could be divided into three parts: (1) the event that starts app a on the device d_1; (2) the StartHop event that transfers app a from the device d_1 to the device d_2; (3) the event sequence should be executed on device d_2 to let app a reach ASS$_{\text{PLAY}}$ s from the initial ASS s_0.
- $E_{a'}$ is the event sequence that should be executed on device d_1 to let app a' reach an ASS which status is PLAY from s_0.
- EndHop is the event that ending the hop of a between d_1 and d_2.

4.3.2 HAC-Directed Test Execution

After test generation, HACMony connects two devices d_1 and d_2 via HarmonyOS Device Connector [28] (HDC) or Android Debug Bridge [29] (ADB) to automatically execute the test cases for the target HarmonyOS app. For general click events or the EndHop operation, HACMony directly invokes the click command in HDC (or ADB) to execute the event. Note that, the EndHop operation can also be regarded as a click event. The StartHop operation could be regarded as a sequence of events, HACMony needs to click the "Recent" button, and then drag the current app to the target device. Finally, HACMony records the ASSs of the tested app a and the conflicting app a'.

4.3.3 HAC Issue Detection

After each test execution, HACMony will analyze the recorded ASSs, and report how the hopping-related audio-stream conflict between apps is resolved, i.e., the conflict resolution. To detect the HAC issues, our key idea is that the conflict resolutions that show up in the multiple-device scenario, i.e., the "hopping" resolutions, should be consistent with the ones in the single-device scenario, i.e., the "normal" resolutions. Thus, for each target app a and its collaborating app a' in app-hopping testing, for each ASS $s = \langle win, \text{PLAY} \rangle$ (resp. $s' = \langle win', \text{PLAY} \rangle$) in the ASTG, we perform the following operations to obtain the "normal" resolutions:

1. Start app a on the device, and execute it to the ASS s;
2. Start app a' on the device, and execute it to the ASS s';
3. Obtain the current ASSs for app a and a'.

We compare the "normal" resolutions with the "hopping" resolutions obtained by HACMony's test execution. If there is any inconsistency, a HAC issue will be reported.

5 Evaluation

To evaluate the effectiveness of our approach, we raise several research questions as follows:

- **RQ1 (ASTG Construction)** Is the ASTG construction process effective and efficient?
- **RQ2 (HAC Issue Detection)** To what extent can HACMony detect real-world HAC issues?
- **RQ3 (HAC Issue Analysis)** What are the categories and characteristics of the detected HAC issues?

5.1 Evaluation Setup

To answer these research questions, we collect 20 real-world HarmonyOS apps from Huawei AppGallery [30]. More specifically, we

take four categories associated with audio, i.e., Music, Video, Navigation, and Social, into account, which are respectively have the highest possibility of using the audio stream type MUSIC, MOVIE, NAVIG, and COMMU. For each type, we download its top five apps that support app-hopping as well as available for both phone and tablet versions. Table 4 lists the detailed information of these experimental apps. All of the following experiments are done on a phone HUAWEI P40 Pro and a tablet HUAWEI Matepad, both with HarmonyOS 4.2.

5.2 RQ1: ASTG Construction

Table 4 shows the results of the ASTG model of each experimental app constructed by HACMony. The fifth column gives the number of audio-stream types (**#Audio-Type**) that are analyzed by an LLM, i.e., Gemini-2.0-Flash [31] by Google. The sixth and seventh columns give the statistics of the ASS: the number of ASSs that are detected by exploration (**#ASS-Init**), and the number of the extra ASSs (**#ASS-Extra**) extracted by collaborating with multiple apps. Besides, the number of total ASSs, the edges in the model, and the dynamical exploration time are shown in the last three columns.

As we can see, HACMony can successfully explore all apps with an average of 1.4 audio-stream types and 6.1 ASSs in an average of 55 seconds. Additionally, 7 (35%) apps with Music or Social category have discovered 2 audio-stream types, and the types of these apps additionally found are all MOVIE type. We have manually verified that all audio-stream types detected by the LLM are accurate. Among all categories, the Navigation apps require more exploration time, which typically explore around 4 ASSs (GUI windows). This is because Navigation app usually involves complex events such as selecting a destination and means of transportation before starting navigation. This observation also demonstrates the effectiveness of the LLM-based exploration approach. Furthermore, the number of the extra ASSs extracted by the ASS-guided enhancement is twice the number of the audio-stream types in all apps with the Music or Video categories, indicating that these apps all discovered the $\text{PLAY}^{\downarrow}$ and $\text{PLAY}^{\|}$ statuses during the enhancement process.

The preceding results validate the effectiveness of HACMony for application exploration tasks. However, a critical consideration is whether the performance of HACMony is determined by a specific LLM. To assess the generalizability and scalability of HACMony across different models, we designed and conducted a comparative evaluation study.

For this study, we evaluated Gemini-2.0-Flash, alongside two other state-of-the-art multimodal LLMs: GPT-4o [32] by OpenAI and Qwen-VL-Plus [33] by Alibaba. We integrated each of the three models into HACMony and executed exploration tasks on the benchmark. As shown in Figure 6, all three LLMs successfully completed the tasks, which validates the broad model compatibility of HACMony. While the mean exploration times varied (Gemini-2.0-Flash: 55.0s, Qwen-VL-Plus: 67.9s, and GPT-4o: 127.6s)—likely due to differences in inference latency and response verbosity—their overall performance trends remained highly consistent. Crucially, exploration time correlated strongly with the application's intrinsic complexity rather than model-specific artifacts. All models required more time on complex apps (e.g., Xiaohongshu) and less on simpler ones (e.g., Kuaiyin). This low sensitivity to the choice of LLMs confirms that HACMony can operate effectively with various state-of-the-art LLMs, enhancing its generalizability and scalability for real-world deployment.

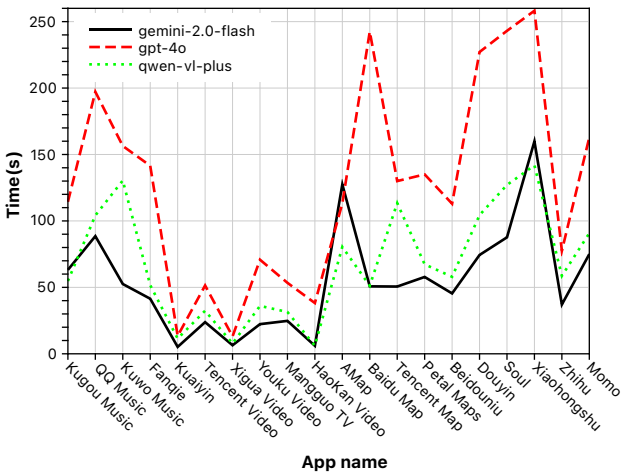


Fig. 6 LLM exploration time comparison on 20 apps

To further validate the effectiveness of LLM-based exploration of HACMony, we conducted a comparative evaluation against three widely-adopted and state-of-the-art exploration strategies: random search, depth-first search (DFS), and breadth-first search (BFS). We configured DroidBot [34], a lightweight exploration tool for Android applications, to employ three distinct exploration strategies, i.e., random, DFS, and BFS. We set the maximum exploration duration to 30 minutes and recorded both the number of audio-stream types discovered by DroidBot using these strategies and the corresponding exploration time.

As illustrated in Table 5, HACMony significantly outperforms random, DFS, and BFS strategy in almost all applications, achieving a 100% superiority over both DFS and BFS, and surpassing the random strategy in 95% of the cases. Notably, in navigation apps, the random, DFS, and BFS strategies failed to locate GUI corresponding to the audio-stream type within the 30-minute time limit, whereas HACMony succeeded on average in just 66.8 seconds. Furthermore, in apps with multiple audio-stream types (e.g., Kugou Music and Douyin), only HACMony managed to discover GUIs for every type within a short duration. In contrast, conventional strategies, i.e., random, DFS, and BFS perform poorly in audio-stream-related GUI exploration tasks, due to the inherent challenges of such complex GUI logic, deeply hidden targets, and scattered distributions of multi-type audio streams.

5.3 RQ2: HAC Issue Detection

Table 6 displays information of the real-world HAC issues detected by HACMony. Columns **#Test Cases** and **Avg. L** show the number of test cases and their average length. Columns **#HAC** and **#Unq. HAC** show the number of the total and unique HAC issues detected. And the column **Time** shows the time of testing.

In total, with the ASTG model, HACMony generates an average

Table 4 Experimental Apps and Model Size

App name	Categories	Size(MB)	Version	#Audio-Type	#ASS-Init	#ASS-Extra	#ASS	#Edge	Time(s)
Kugou Music	Music	156.6	12.4.2	2	4	4	8	7	64
QQ Music	Music	188.7	13.9.0.8	2	5	4	9	8	89
Kuwo Music	Music	181.4	11.0.0.0	2	4	4	8	7	53
Fanqie	Music	71.5	7.41.18	1	2	2	4	3	42
Kuaiyin	Music	75.8	5.57.11	1	2	2	4	3	6
Tencent Video	Video	145.9	8.11.71	1	2	2	4	3	24
Xigua Video	Video	65.6	8.8.6	1	1	2	3	2	6
Youku Video	Video	123.5	11.0.99	1	2	2	4	3	23
Mangguo TV	Video	133.2	8.13.0	1	2	2	4	3	25
HaoKan Video	Video	49.5	7.64.0.10	1	2	2	4	3	7
AMap	Navigation	254.9	15.01.0	1	4	1	5	4	128
Baidu Map	Navigation	171.5	20.7.30	1	4	1	5	4	51
Tencent Map	Navigation	162	10.11.1	1	4	1	5	4	51
Petal Maps	Navigation	83.9	4.5.0.303	1	5	1	6	5	58
Beidouniu	Navigation	59.1	3.3.1	1	4	1	5	4	46
Douyin	Social	271.9	31.4.0	2	6	3	9	8	75
Soul	Social	158.5	5.40.0	2	7	3	10	9	88
Xiaohongshu	Social	164	8.52.0	2	7	3	10	9	160
Zhihu	Social	87.8	10.22.0	1	3	2	5	4	38
Momo	Social	127	9.13.10	2	6	3	9	8	76
Avg./Max.	-	**136.6/271.9**	-	**1.4/2**	**3.8/7**	**2.3/4**	**6.1/10**	**5.1/9**	**55/160**

Table 5 Exploration performance comparison of different input generation strategies

App name	Audio-stream types / Time(s)			
	HACMony	**Random**	**DFS**	**BFS**
Kugou Music	**2 / 64**	1 / 1800	1 / 1800	1 / 1800
QQ Music	**2 / 89**	1 / 1800	1 / 1800	0 / 1800
Kuwo Music	**2 / 53**	1 / 1800	1 / 1800	0 / 1800
Fanqie	1 / 42	**1 / 11**	1 / 28	1 / 23
Kuaiyin	**1 / 6**	1 / 12	1 / 52	1 / 51
Tencent Video	**1 / 24**	1 / 63	1 / 242	1 / 356
Xigua Video	**1 / 6**	1 / 12	1 / 34	1 / 33
Youku Video	**1 / 23**	1 / 31	0 / 1800	0 / 1800
Mangguo TV	**1 / 25**	1 / 128	1 / 87	1 / 221
HaoKan Video	**1 / 7**	1 / 12	1 / 37	1 / 24
AMap	**1 / 128**	0 / 1800	0 / 1800	0 / 1800
Baidu Map	**1 / 51**	0 / 1800	0 / 1800	0 / 1800
Tencent Map	**1 / 51**	0 / 1800	0 / 1800	0 / 1800
Petal Maps	**1 / 58**	0 / 1800	0 / 1800	0 / 1800
Beidouniu	**1 / 46**	0 / 1800	0 / 1800	0 / 1800
Douyin	**2 / 75**	1 / 1800	1 / 1800	1 / 1800
Soul	**2 / 88**	1 / 1800	1 / 1800	0 / 1800
Xiaohongshu	**2 / 160**	1 / 1800	1 / 1800	0 / 1800
Zhihu	**1 / 38**	0 / 1800	0 / 1800	0 / 1800
Momo	**2 / 76**	1 / 1800	1 / 1800	1 / 1800

of 137 test cases for each app, with an average length of 6.1 events. There are 12 out of 20 (60%) apps detected to have HAC issues, which involve a total of 18 unique HAC issues out of 53 HAC issues. This indicates that HAC issues are relatively likely to occur during the HarmonyOS app-hopping. The video demonstrations of HAC issues found by HACMony can be viewed [35].

Recall that during the exploration phase, we leverage LLMs to explore multiple audio-stream types (MT) within apps, while in the enhancement phase, we utilize other apps to explore multiple audio-stream statues (MS), e.g., $\mathsf{PLAY}^{\downarrow}$ and $\mathsf{PLAY}^{\|}$, with the aim of discovering as many PLAY-like ASSs as possible to detect more HAC issues. To validate the effectiveness of MT and MS in our approach, we conducted additional experiments, as shown in Table 7. The second to fifth columns shows the number of HAC issues and unique HAC issues of each configuration, specifically,

- The **Base** column represents the baseline configuration without using either MS or MT.
- The **MT** column denotes the configuration using only multi audio-type (without MS).
- The **MS** column denotes the configuration using only multi audio-status (without MT).
- The **MT+MS** column represents the full configuration that combines both MS and MT.

The results demonstrate that the full configuration (MT+MS) outperforms all other configurations, with a 35.9% increase in detection of the HAC issues and a 12.5% increase in detection of the unique HAC issues compared to the baseline. Individually, MT and MS improve the detection of the HAC issues by 25.6% and 7.7% over the baseline, respectively, but show limited improvement in the detection of the unique HAC issues (only one new unique HAC issue each). This reflects that both the MT- and MS-aware exploration are important in achieving comprehensive HAC issues detecting. Testers should focus their efforts on generating test cases for different audio-types and

audio-statuses to detect HAC issues.

Table 6 Detected HAC Issues by HACMony

App name	#Test Cases	Avg. L	#HAC	#Unq. HAC	Time(s)
Kugou Music	228	5.7	0	0	2891
QQ Music	228	6.7	3	1	3402
Kuwo Music	228	6.2	5	1	3202
Fanqie	114	5.2	0	0	1407
Kuaiyin	114	6.2	0	0	3221
Tencent Video	114	5.2	5	2	1337
Xigua Video	114	6.2	0	0	1634
Youku Video	114	5.2	5	1	1367
Mangguo TV	114	5.4	0	0	1417
HaoKan Video	114	5.2	4	1	2793
AMap	76	7.3	7	3	1241
Baidu Map	76	7.3	5	2	1375
Tencent Map	76	8.4	2	2	1187
Petal Maps	76	7.3	5	2	1793
Beidouniu	76	7.1	3	1	1857
Douyin	190	5.6	0	0	1450
Soul	190	5.4	0	0	1415
Xiaohongshu	190	5.4	3	1	1453
Zhihu	114	5.4	0	0	1417
Momo	190	6.2	5	1	1572
Avg.	**137**	**6.1**	**2.7**	**0.9**	**1872**

Table 7 Impact of Multi Audio-Type (MT) and Multi Audio-Status (MS) on HAC Detection

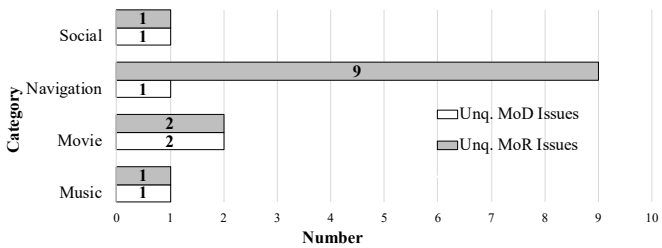
App name	#HAC (Unq.)			
	Base	MT	MS	MT+MS
QQ Music	2 (1)	2 (1)	**3 (1)**	3 (1)
Kuwo Music	5 (1)	5 (1)	5 (1)	5 (1)
Tencent Video	3 (1)	3 (1)	**5 (2)**	5 (2)
Youku Video	5 (1)	5 (1)	5 (1)	5 (1)
HaoKan Video	4 (1)	4 (1)	4 (1)	4 (1)
AMap	4 (3)	**7 (3)**	4 (3)	7 (3)
Baidu Map	3 (2)	**5 (2)**	3 (2)	5 (2)
Tencent Map	2 (2)	2 (2)	2 (2)	2 (2)
Petal Maps	3 (2)	**5 (2)**	3 (2)	5 (2)
Beidouniu	3 (1)	3 (1)	3 (1)	3 (1)
Xiaohongshu	0 (0)	**3 (1)**	0 (0)	3 (1)
Momo	5 (1)	5 (1)	5 (1)	5 (1)
Sum.	**39 (16)**	**49 (17)**	**42 (17)**	**53 (18)**

5.4 RQ3: HAC Issue Analysis

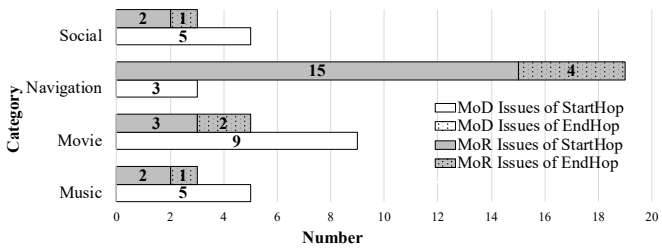
To assist both the developer of Harmony apps and OS better understanding the real-world HAC issues. We category issues and perform case studies to investigate their characteristics.

First, we summarize the specific behaviors of the apps where HAC issues occur and category issues into two types, **Misuse of Device (MoD)** and **Misuse of Resolution (MoR)**. MoD issue refers to the situation where, during the hopping of an app, the usage of the audio streams fails to be transferred to the target device along with the app. The MoR issue refers to the situation where, during the hopping of an app, an audio-stream conflict occurs on the target device, but the "normal" resolution to solve the conflict is not applied. In our experiments, HACMony detected four apps with MoD issues and nine apps with MoR issues.

Then, we count the number of HAC issues of different app categories. As shown in Figure 7(a), the MoD issues are more likely to occur in the Video applications, while the MoR issues are more likely to occur in the Navigation applications. Furthermore, we count the number of HAC issues of different types of test cases. As shown in Figure 7(b), all the MoD issues are detected through the test cases in the form of $\mathbf{Test_{StartHop}}$, and few (26%) MoR issues are detected through the test cases in the form of $\mathbf{Test_{EndHop}}$. Although most of the HAC issues are detected through the $\mathbf{Test_{StartHop}}$ test cases, there are still some issues identified by the $\mathbf{Test_{EndHop}}$ test cases, this indicates that it is necessary to consider different operations when generating test cases (See Section 4.3.1). Therefore, Navigation apps trigger more HAC issues than all other types. They suffer severe MoR issues, especially in the process of StartHop operation. Besides, Video apps are easier to trigger MoD issues. Testers and developers can perform testing/developing according to the type of the target app.



(a) Issue-related apps and unique issues



(b) Issues triggered with different ops

Fig. 7 Number of HAC Issues

To further study the characteristics of HAC issues, we analyze the two types of HAC issues with case studies, respectively.

5.4.1 Misuse of Device: MoD issues

Recall that MoD issues are more likely to occur in the Video applications, hence we pick a Video app to investigate the characteristic of MoD issue.

Case study 1: MoD. When *Youku Video* [36] is playing a video normally on the mobile phone, if the user hop it to the tablet, the video continues to play on the tablet, but the audio is still playing on the mobile phone. It leads to the audio-visual inconsistency problem

which makes it difficult for the users to focus on the video content and affects users' understanding and enjoyment of the video.

Analysis: We noticed that, the MoD issues may **not** occur in all test cases of the same app, i.e., sometimes the issue do not occur. Thus, we infer that such sporadic issues may be caused by the lack of synchronization of commands and data between devices, which prevents the new device from taking over audio playback in a timely manner, so the audio playback on the original device does not stop. Moreover, since the MoD issues are only detected though the $\textbf{Test}_{\textbf{StartHop}}$ test cases, it indicates that the handling process between StartHop operation and EndHop operation is different. EndHop operation may force all resources related to the hopping app in the target device to be transferred back to the original device.

To further analyze the cause of the MoD issue, we conducted an analysis by decompiling the apk file and manually auditing the code. For the MoD issue, there were a total of 5 applications, and 4 of them were successfully decompiled. Inspection of the decompiled code suggests that the most likely cause of the reason for the MoD issue is that when the application is hopped from the original device to the new one, the application on the original device fails to release the audio focus in a timely and correct manner. If the application on the original device does not release the focus, the system will not granting the focus to the application on the new device, which will cause the audio to continue playing on the original device. The occasional occurrence of MoD in the same application also indicates the complexity of its causes, which may be related to complex audio focus management strategies, timing issues, or specific application states.

Moreover, in *Kuwo Music* [37], the core issue is its shared audio focus management based on the listener list, which only releases the focus when all listeners have been removed. If there are other listeners during the hopping, the release will fail and the MoD issue will occur. *Momo* [38], on the other hand, has introduced a strong binding with the Activity lifecycle in "AudioFocusManager", which may cause conflicts during hopping. When the app hopping starts, the system will change the lifecycle state of the Activity on the original device. At this time, the application may release the focus as expected. However, during the complex process of hopping, the scheduling sequence of the Activity lifecycle by the system might precisely trigger the refocus logic in "onActivityResumed", which is equivalent to regain the audio focus that is about to be transferred to the new device.

5.4.2 Misuse of Resolution: MoR issues

As MoR issues involve more statuses, we categorize them into three sub-types according to the status changes, namely PLAY$^{\downarrow}$→PLAY, PLAY$^{\downarrow}$→STOP, and STOP→PLAY. The first (resp. second) issue refers to the situation where the "hopping" resolution for solving audio-stream conflict changes from lowering the volume to playing (resp. stopping) compared to the "normal" one. The third issue refers to the situation where the "hopping" resolution for solving audio-stream conflict changes from stopping to playing. Table 8 shows the sub-types of MoR issues HACMony have detected. Next, we pick two Navigation apps and a Video app as the case study hopping apps to investigate the characteristic of MoR issue.

Table 8 Sub-types of the App that Detected MoR Issues

App name	#PLAY$^{\downarrow}$→PLAY	#PLAY$^{\downarrow}$→STOP	#STOP→PLAY
QQ Music			★
Tencent Video			★
AMap	★	★	★
Baidu Map	★	★	
Tencent Map	★	★	
Petal Maps	★	★	
Xiaohongshu			★

Case study 2: PLAY$^{\downarrow}$→PLAY type MoR. When *Baidu Map* [39] is running on the mobile phone and navigating, the user hops it to the tablet for further navigation, on which *Kuaiyin* [40] is playing music. The expected behaviour is that *Kuaiyin* lower the volume. However, in this situation, both *Baidu Map* and *Kuaiyin* play their audio streams at normal volume on the tablet. As a result, it makes difficult for users to clearly hear the navigation instructions or information from *Baidu Map*, which brings inconvenience or safety risks to their travels.

Case study 3: PLAY$^{\downarrow}$→STOP type MoR. When *Petal Map* [41] is running on the mobile phone and navigating, the user hops it to the tablet for further navigation, on which *QQ Music* [42] is playing music. The expected behaviour is that *QQ Music* lower the volume. However, *QQ Music* stops its audio stream. On the one hand, it ruins the user's immersive music-listening experience, where the sudden interruption breaks the continuity of the music. On the other hand, the unexpected stop of the music may force the user to interrupt other ongoing operations to check and resume the music playback, distracting the user's attention from using *Petal Map* for navigation or other tasks.

Case study 4: STOP→PLAY type MoR. When *Tencent Video* [43] is playing the video on the mobile phone, the user hops it to the tablet for further playing, on which *Kugou Music* is playing music. The expected behaviour is that *Kugou Music* stops playing. However, both *Tencent Video* and *Kugou Music* play their audio streams at normal volume on the tablet. As a result, users can't clearly distinguish the dialogue in the video from the music, leading to extreme auditory discomfort and ruining the original audio-visual enjoyment.

Analysis: After conducting all the experiments, we observed that while *Kuaiyin* and *Kugou Music* exhibit MoR issues as the "pre" apps in hopping, no HAC issues were detected when they served as the "post" apps, i.e., the hopped apps. Although an STOP→PLAY issue was detected in *QQ Music* as shown in Table 8, it occurred in the audio-stream conflict with *Kugou Music*, not with *Tencent Video*. This shows that MoR issues are generally asymmetric, meaning that a change in the order of audio-stream conflict can influence the occurrence of MoR issues.

Similarly, to further analyze the causes of the MoR issue, we conducted the analysis by decompiling the apk file and manually auditing the code. For the MoR issue, there were a total of 10 applications, and all of them were successfully decompiled. Inspection of the decompiled code indicates that the MoR issue most likely stems from the abnormal transmission and processing of the audio focus state during the cross-device hopping process, which leads to its behavior

being inconsistent with the standard audio focus strategy within a single device. After analysis, we concluded that the happening of MoR issue is due to the relevant applications have not fully adapted to the cross-device hopping function of HarmonyOS. The cross-device hopping feature of HarmonyOS requires applications to proactively and accurately manage their audio focus states across different devices. When an application is transferred to a new device and starts playing audio, it needs to handle the application and release of audio focus in accordance with the specific mechanism of HarmonyOS hopping.

For instance, *Petal Maps* should request focus type AUDIOFOCUS_GAIN_TRANSIENT_CAN_DUCK during regular playback. However, in the hopping scenario, due to the application's failure to properly handle the state changes brought about by the hopping, its focus request was wrongly identified as AUDIOFOCUS_GAIN_TRANSIENT, thereby triggering conflicts with audio playback in other applications (PLAY$^{\downarrow}\rightarrow$STOP issue). Similarly, the *AMap* application also changed its focus requests due to the lack of adaptation to the hopping, and eventually played simultaneously with the *Kugou Music*. The MoR issue has exposed that during the development process of related applications, they failed to follow HarmonyOS's development norms for cross-device hopping and did not make corresponding adaptations and adjustments to the management of audio focus. It is precisely because the application layer code lacks compatibility with this new system feature that in the specific scenarios of hopping, the application fails to correctly respond to the native audio focus strategy, ultimately presenting audio behavior that does not meet expectations.

While code-level inconsistencies in audio focus management are the primary cause, potential network-level factors could also exacerbate or trigger the MoR issue. For example, network latency during cross-device hopping might delay the transmission of audio focus state messages between devices, leading to temporary desynchronization in focus management. Similarly, out-of-order delivery of critical state-update messages could further disrupt the application's ability to align with native audio focus strategy.

This indicates that the MoR issues are related to the resolution of conflicts between two apps, which are generally asymmetric. Testers should not design test cases merely based on the conventional symmetric assumption.

6 Discussion

This section primarily discusses the threats to the validity (including limited generalizability due to restricted app sampling and version dependency on HarmonyOS) and proposes future research directions (testing in multi-device scenarios, generating test cases with complex hopping operations, and root-cause analysis combining static analysis).

6.1 Threats to Validity

There are two main threats to the validity of our study.

★ The representativeness of selected benchmarks can affect the fidelity of our conclusions. To mitigate this threat, we have selected 20 real-world apps from Huawei AppGallery, which are: (1) Highly popular (Ranked within the top 5 in their respective categories); (2) Diverse in categories (Specifically aligned with audio-stream types, e.g., music players, video platforms, live streaming apps); (3) Large-scale (An avarage of 136.6 MB size). Future work could expand the app sample to include diverse categories and low-download apps, ensuring a more comprehensive assessment of the approach's robustness.

★ The version of HarmonyOS, e.g., *3.1*, *4.2*, *Next*, etc., may affect the semantics of app-hopping mechanism. Besides, the latest HarmonyOS version currently faces challenges in experimental validation due to limited device support and a scarcity of available apps, which restricts our ability to assess the framework's compatibility with emerging system architectures. To mitigate this, we selected the version HarmonyOS 4.2, which is the most widely used version of HarmonyOS up to now, as well as has a large number of available HarmonyOS apps.

6.2 Directions for Further Research

According to the previous investigation, we will provide several directions for further researches.

★ **Testing hopping behaviours by generating more complex test cases.** In this paper, the test cases designed are restricted to incorporating only a single hopping operation. However, users may frequently perform multiple consecutive hopping operations. To account for this real-world behavior, more complex test cases should be generated in the future, aiming to more comprehensively detect HAC issues.

★ **Combining static analysis technique to make in-depth root cause analysis.** In this paper, the cause of HAC issues are analyzed solely based on their phenomena. However, to uncover the root causes of issues, in-depth analysis of the application is required using static analysis techniques to figure out the audio stream related code patterns. Future research works could combine static analysis technique, e.g., data-flow, control-flow, to analyze the root cause of HAC issues.

7 Related work

This section introduces the research works related to HarmonyOS and model-based testing.

7.1 Analysis and Testing for HarmonyOS

Since HarmonyOS is an emerging system, there are few research works of analysis and testing for it. Ma et al. [44] are the first to provide an overview of HarmonyOS API evolution to measure the scope of situations where compatibility issues might emerge in the HarmonyOS ecosystem. Zhu et al. [45] propose the HM-SAF framework, a cross-layer static analysis framework specifically designed for HarmonyOS applications. The framework analyzes HarmonyOS applications to identify potential malicious behaviors in a stream and context-sensitive manner. Chen et al. [46] design a framework ArkAnalyzer for OpenHarmony Apps. ArkAnalyzer addresses a number of fundamental static analysis functions that could 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. These works are all static

analyses of HarmonyOS apps and do not focus on the ACs studied in this paper.

7.2 Model-Based Testing of GUI

Model-based testing (MBT) technique is commonly used in automated GUI testing for applications. Existing woks mainly extract models through static analysis, dynamic analysis and hybrid analysis. FSM [6] is the first to model the GUI behaviors of Android apps using static analysis for MBT. WTG [7], an extension of FSM with back stack and window transition information, is a relatively classic model in MBT. Based on WTG, some models [8–10, 16–18] which can be considered as a finer-grained WTG, are built by dynamic analysis. There are also some works [11–14, 47] that extend the WTG through a hybrid technique of static and dynamic analysis. With the rise of large language models (LLMs), the GUI exploration methods based on LLMs are capable of extracting the WTG more quickly and accurately. This new approach leverages the powerful language understanding and generation capabilities of LLMs, which can effectively analyze the complex interactions and transitions within the GUI [48, 49]. However, the models proposed in these works are almost used to describe the transitions of GUIs. They do not take into account information related to audio streams, nor do they consider the interactions among multiple applications. These two factors are the key points that ASTG takes into account.

8 Conclusion

Hopping-related audio-stream conflict (HAC) issues are common on the distributed operating system HarmonyOS. To test them automatically and efficiently, we design the Audio Service Transition Graph (ASTG) model and propose a model-based testing approach. To support it, we also present the first formal semantics of the HarmonyOS's app-hopping mechanism. The experimental results show that, with the help of the formal semantics of the app-hopping mechanism and the ASTG model, the HACMony can detect real-world HAC issues effectively and efficiently. For the detected issues, we also analyze their characteristics to help app and OS developers improve apps' quality on distributed mobile systems.

Acknowledgement

This project was partially funded by the Strategic Priority Research Program of the Chinese Academy of Sciences (Grant No. XDA0320102), National Natural Science Foundation of China (Grant No. 62132020), and Major Project of ISCAS (ISCAS-ZD-202302).

References

[1] Community H. Huawei's HarmonyOS Gains Market Share. See consumer.huawei.com/en/community/details/topicId-225051/ website, 2024

[2] Times G. China's first fully home-grown mobile operating system HarmonyOS NEXT launched. See www.globaltimes.cn/page/202410/1321670.shtml website, 2024

[3] Huawei . Hopping Overview. See developer.huawei.com/consumer/en/doc/design-guides-V1/service-hop-overview-0000001089296748-V1 website, 2024

[4] HarmonyOS Developer Issue. See developer.huawei.com/consumer/cn/forum/topic/0202700699545450014?fid=0101587866109860105 website, 2021

[5] HarmonyOS Developer Issue. See developer.huawei.com/consumer/cn/forum/topic/0202646978991840491?fid=0101591351254000314 website, 2021

[6] Yang W, Prasad M R, Xie T. A grey-box approach for automated gui-model generation of mobile applications. In: Proceedings of Fundamental Approaches to Software Engineering. 2013, 250–265

[7] Yang S, Zhang H, Wu H, Wang Y, Yan D, Rountev A. Static window transition graphs for android (T). In: Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering. 2015, 658–668

[8] Gu T, Sun C, Ma X, Cao C, Xu C, Yao Y, Zhang Q, Lu J, Su Z. Practical GUI testing of android applications via model abstraction and refinement. In: Proceedings of the 41st International Conference on Software Engineering. 2019, 269–280

[9] Ma Y, Huang Y, Hu Z, Xiao X, Liu X. Paladin: Automated generation of reproducible test cases for android apps. In: Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications. 2019, 99–104

[10] Su T, Meng G, Chen Y, Wu K, Yang W, Yao Y, Pu G, Liu Y, Su Z. Guided, stochastic model-based GUI testing of android apps. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering,. 2017, 245–256

[11] Yan J, Wu T, Yan J, Zhang J. Widget-sensitive and back-stack-aware GUI exploration for testing android apps. In: Proceedings of 2017 IEEE International Conference on Software Quality, Reliability and Security. 2017, 42–53

[12] Yan J, Liu H, Pan L, Yan J, Zhang J, Liang B. Multiple-entry testing of android applications by constructing activity launching contexts. In: Proceedings of the 42nd International Conference on Software Engineering. 2020, 457–468

[13] Liu C, Wang H, Liu T, Gu D, Ma Y, Wang H, Xiao X. PROMAL: precise window transition graphs for android via synergy of program analysis and machine learning. In: Proceedings of 44th IEEE/ACM 44th International Conference on Software Engineering. 2022, 1755–1767

[14] Azim T, Neamtiu I. Targeted and depth-first exploration for systematic testing of android apps. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. 2013, 641–660

[15] Wu T, Deng X, Yan J, Zhang J. Analyses for specific defects in android applications: a survey. Journal of Frontiers Comput. Sci., 2019, 13(6): 1210–1227

[16] Chen T, He J, Song F, Wang G, Wu Z, Yan J. Android stack machine. In: Proceedings of Computer Aided Verification. 2018, 487–504

[17] He J, Chen T, Wang P, Wu Z, Yan J. Android multitasking mechanism: Formal semantics and static analysis of apps. In: Proceedings of the 17th Asian Symposium on Programming Languages and Systems. 2019, 291–312

[18] He J, Wu Z, Chen T. Formalization of android activity-fragment multitasking mechanism and static analysis of mobile apps. Journal of Form. Asp. Comput., 2025, 37(2): 1–86

[19] Huawei . About HarmonyOS. See developer.huawei.com/consumer/en/doc/harmonyos-guides-V3/harmonyos-overview-0000000000011903-V3 website, 2025

[20] Chen H, Miao X, Jia N, Wang N, Li Y, Liu N, Liu Y, Wang F, Huang Q, Li K, Yang H, Wang H, Yin J, Peng Y, Xu F. Microkernel goes general: Performance and compatibility in the hongmeng production microkernel. In: Proceedings of 18th USENIX Symposium on Operating Systems Design and Implementation. 2024, 465–485

[21] Google . Android Open Source Project. See source.android.com website, 2025

[22] Foundation O. OpenHarmony Project. See gitee.com/openharmony/docs/blob/master/en/OpenHarmony-Overview.md website, 2025

[23] Huawei . Processing Audio Interruption Events. See developer.huawei.com/consumer/en/doc/harmonyos-guides-V5/audio-playback-concurrency-V5 website, 2025

[24] Huawei . StreamUsage. See developer.huawei.com/consumer/en/doc/harmonyos-references-V13/js-apis-audio-V13#streamusage website, 2025

[25] AMap. See url.cloud.huawei.com/tXaf6tZ5sY website, 2025

[26] Kugou Music. See url.cloud.huawei.com/tXafXtrfyM website, 2025

[27] HACMony . Operational Semantics of App-Hopping Mechanism on HarmonyOS. See github.com/SQUARE-RG/hacmony/blob/main/Semantics_of_HarmonyOS_App_Hopping.pdf website, 2025

[28] Huawei . hdc. See developer.huawei.com/consumer/en/doc/harmonyos-guides-V5/hdc-V5 website, 2025

[29] Google . Android Debug Bridge (adb). See developer.android.com/tools/adb website, 2024

[30] Huawei . Huawei Appgallery. See consumer.huawei.com/en/mobileservices/appgallery/ website, 2025

[31] Google . Gemini 2.0 Flash. See cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-0-flash website, 2024

[32] OpenAI . GPT-4o. See openai.com/index/hello-gpt-4o/ website, 2024

[33] Alibaba . Qwen-VL-Plus. See github.com/QwenLM/Qwen-VL website, 2024

[34] Li Y, Yang Z, Guo Y, Chen X. Droidbot: a lightweight ui-guided test input generator for android. In: Proceedings of 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). 2017, 23–26

[35] HACMony . HAC Issues Detected by HACMony. See www.youtube.com/playlist?list=PL9InyCjzL53mWIbPP5ixylr7Qwd-kzUTa website, 2025

[36] Youku Video. See url.cloud.huawei.com/tXLQZi7oZi website, 2025

[37] Kuwo Music. See url.cloud.huawei.com/x7rkqpzQ1W website, 2025

[38] Momo. See url.cloud.huawei.com/x7rZfOT2I8 website, 2025

[39] Baidu Map. See url.cloud.huawei.com/tXQg34wJXy website, 2025

[40] Kuaiyin. See url.cloud.huawei.com/u2T5hQKLjW website, 2025

[41] Petal Map. See url.cloud.huawei.com/tXRdtLucnu website, 2025

[42] QQ Music. See url.cloud.huawei.com/tXRhftsDPW website, 2025

[43] Tencent Video. See url.cloud.huawei.com/tXRhNDqDWo website, 2025

[44] Ma T, Zhao Y, Li L, Liu L. Cid4hmos: A solution to harmonyos compatibility issues. In: Proceedings of 38th IEEE/ACM International Conference on Automated Software Engineering. 2023, 2006–2017

[45] Zhu Y, Guo J, Xu F, Chen R, Zhang X, Yi S, Yu J. Hm-saf: Cross-layer static analysis framework for harmonyos. In: Proceedings of 2023 IEEE Smart World Congress (SWC). 2023, 1–10

[46] Chen H, Chen D, Yang Y, Xu L, Gao L, Zhou M, Hu C, Li L. Arkanalyzer: The static analysis framework for openharmony. In: Proceedings of 47th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice. 2025, 136–147

[47] Chen Z, Liu J, Hu Y, Wu L, Zhou Y, He Y, Liao X, Wang K, Li J, Qin Z. Deuedroid: Detecting underground economy apps based on UTG similarity. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2023, 223–235

[48] Liu Z, Chen C, Wang J, Chen M, Wu B, Huang Y, Hu J, Wang Q. Unblind text inputs: Predicting hint-text of text input in mobile apps via LLM. In: Proceedings of the CHI Conference on Human Factors in Computing Systems. 2024, 51:1–51:20

[49] Liu Z, Chen C, Wang J, Chen M, Wu B, Che X, Wang D, Wang Q. Make LLM a testing expert: Bringing human-like interaction to mobile GUI testing via functionality-aware decisions. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. 2024, 100:1–100:13