SoK: History Doesn't Repeat Itself, but Android Design-Level Vulnerabilities Rhyme in OpenHarmony

Hongkai Chen[1], Yuqing Yang[2], Chao Wang[3], Arpit Nandi[1], Moritz Schloegel[2], Tiffany Bao[1], Ruoyu Wang[1], Adam Doupé[1], Zhiqiang Lin[3], Yan Shoshitaishvili[1]
[1]*Arizona State University,* [2]*CISPA Helmholtz Center for Information Security,* [3]*The Ohio State University*
{hongkai.chen, anandi5, tbao, fishw, doupe, yans}@asu.edu, {yuqing.yang, schloegel}@cispa.de, wang.15147@osu.edu, zlin@cse.ohio-state.edu

Abstract

The dominant market share of Android across the world has led to significant scrutiny by security researchers. Over the years, many security issues were identified and remedied not only in its implementation but also in its design. Interestingly, academia has not further systematized these design flaws. This is an important gap, especially as many phone vendors have recently begun developing their own mobile operating systems (OSes). While they do not necessarily reuse the code of Android, they may share much of its design as they often use Android as a starting point. A prime example is Huawei's new OS, OpenHarmony, which has been deployed on over one billion devices, but its security has not been studied in academia. We posit that the *Design-Level Vulnerabilities* in Android can affect these emerging mobile OSes, leaving their users at risk.

In this paper, we systematically study Design-Level Vulnerabilities in Android and how they affect emerging mobile OSes. First, we review 116 publications on Android from industry and academia, and we extract 56 unique vulnerabilities reported in Android's design. For each, we identify a reusable auditing methodology to enable testing of emerging mobile OSes. In a second step, we apply our auditing methodologies to OpenHarmony and demonstrate that OpenHarmony is vulnerable to 24 of the vulnerabilities, including ones that compromise user privacy, enable stealthy privilege escalation, and undermine overall system reliability.

1 Introduction

With more than 3 billion active devices and a worldwide market share of 72%, Android dominates the global market of mobile operating systems (OSes) [13]. It is a successful open-source project that attracts numerous mobile manufacturers who have adopted it for their devices. Over time, these vendors customized more and more components for their business purposes, thus *fragmenting* the Android ecosystem [11]—a well-studied phenomenon [74, 85, 127, 137, 179, 193, 199].

Recently, this fragmentation has evolved to a new stage. In 2019, due to various geopolitical, technological, and commercial factors, some Android vendors began to supersede their customizations of Android with implementations of their own mobile OSes. Many of these OSes are initially still based on Android: for example, Huawei's HarmonyOS up to version 4 was based on Android, but the developers had gradually replaced Android components with their own. Importantly, this gradual, Ship of Theseus-style replacement implies a similar design to maintain compatibility with the yet-untranslated Android components. As one illustration, an *Intent* is an object used in Android's Inter-Component Communication (ICC), such as launching an Activity from one app to another. HarmonyOS features *Wants*, which are ICC objects with very similar functionalities [133].

Huawei's efforts culminated in 2024 with the release of HarmonyOS NEXT, which no longer contains any Android code and no longer supports Android apps [82]. Despite being a new OS, HarmonyOS has already surpassed Apple's iOS to become the second best-selling mobile OS in China [81] and is deployed on over 1 billion devices [83]. Other Android vendors are following suit and developing their own OSes [166, 175], similarly starting from Android.

Surprisingly, the traction and rising popularity of these emerging OSes are not reflected in security research, with few researchers in academia or beyond paying attention to analyzing their privacy or security. Seeking to address this gap, we make the insight that the similar designs of emerging mobile OSes (such as HarmonyOS) to Android leave them at risk of similar design flaws to those found in Android by past research, regardless of implementation details. For example, in the HarmonyOS Want mechanism, we discovered the same implicit Intent hijacking vulnerability that had previously appeared in Android [92], *despite the implementations differing*. While this has been fixed in Android's design and implementation, the fix never made it to HarmonyOS. Thus, we hypothesize that other historical design-level flaws in Android may similarly exist in emerging mobile OSes, and potentially remain even when discovered and fixed in An-

droid. This makes them particularly dangerous, as attackers familiar with Android may already know these vulnerabilities and be well-prepared to exploit them.

To test this hypothesis, we first systematize Design-Level Vulnerabilities (DLVs) found in Android security literature into an actionable Systematized Auditing Model and then evaluate OpenHarmony, the open-source version of HarmonyOS, currently the only completed mobile OS reimplementation, against this model. We build the Systematized Auditing Model by conducting a thorough literature review of 450 Android security publications from computer security venues. While we primarily focus on papers published at top-tier system security venues, namely IEEE S&P, USENIX Security, ACM CCS, and ISOC NDSS, between 2010 and 2024, we also include Black Hat Briefings on Android security at Black Hat USA, Black Hat Europe, and Black Hat Asia between 2015 and 2024. For each of the 116 publications on Design-Level Vulnerabilities, we (i) extract the DLVs, (ii) analyze the root causes of the DLVs, and (iii) examine how Android mitigated the DLVs. For each DLV, we create an auditing method and a Proof-of-Concept (PoC) to verify whether it is fixed in practice. Overall, our systematization identifies 56 DLVs in the literature, 52 of which have been fixed in Android as of 2025.

In a second step, we apply our Systematized Auditing Model to OpenHarmony. Our work is the first to study the security of OpenHarmony. We find that 46 of the 56 DLVs are *applicable* to OpenHarmony, meaning analogous design or functionality exists in OpenHarmony. We create PoCs for these 46 DLVs and observe that OpenHarmony is vulnerable to 24. It is noteworthy that 43 of 46 DLVs have already been fixed in Android, while OpenHarmony remains vulnerable to many. The vulnerabilities we find in OpenHarmony pose serious security implications, including leaking sensitive system information, side-channel attacks to retrieve credentials, malicious Bluetooth manipulation, bypassing the permission control model, and provoking system-wide DoS. We responsibly disclosed all findings to OpenHarmony, who confirmed six before the submission of this paper (with the others under review) and awarded bug bounties of 25,000 CNY.

Contributions. Our key contributions are:

- We perform a systematic literature review of 116 publications on Android Design-Level Vulnerabilities (DLVs) and systematize 56 DLVs.
- We apply the systematized knowledge and conduct the first security study of OpenHarmony. We implement PoC apps to empirically test the existence of each DLV in OpenHarmony.
- Our analysis discovers 24 DLVs in OpenHarmony that pose serious security implications, such as cracking credentials or bypassing permission control, demonstrating that Android DLVs can translate to emerging mobile OSes with similar designs.

2 Study Scope

This paper focuses on Design-Level Vulnerabilities (DLVs), which stem from flaws in the intended logic of a complex system instead of mistakes in programming or configuration. In other words, a system containing DLVs behaves exactly as designed, but the design itself enables insecure or unintended behaviors. A representative example of a DLV can be found in Android's implicit Intent. Intents enable ICC between apps, and when declared as implicit, the system dynamically selects a receiver by matching Intent filters [12]. While this design enables flexible and decoupled communication among apps, it also introduces a security risk: Any app that registers a matching Intent filter can intercept or spoof communication [92]. As a result, sensitive data or actions may be inadvertently directed to malicious components. This flaw is not due to implementation errors but stems from the insecure logic underlying the Intent resolution design. Common testing techniques, for example, fuzzing or static analysis, are ill-equipped to identify such flaws that require a deeper understanding of the system's design semantics. Crucially, any *reimplementation* of this flawed design may likewise be affected; this does not hold for implementation bugs, which may or may not happen again after reimplementation.

There are two corner cases we consider to better capture vulnerabilities with real-world security impact: First, we also include app-level implementation issues if the OS can provide system-level mitigations to prevent them at scale. And second, we deviate from Android's threat model [119] for covert channel issues. Android does not consider them security-relevant, but we believe they are worthwhile to consider, as top-tier publications document their real-world security impact.

3 Systematization Methodology

To identify and systematize DLVs in Android, we conduct a comprehensive literature analysis. We focus on top-tier system security publications because they (i) capture the most impactful issues, (ii) are scientifically rigorous, and (iii) provide threat models, generalizable root-cause analysis, and attack processes to replicate on OpenHarmony and synthesize auditing methodologies.

Publication selection. We examine all papers mentioning "Android" in their title or abstract that have been published in top-tier system security conference proceedings[1] (IEEE S&P, USENIX Security, ACM CCS, and ISOC NDSS) between 2010 and 2024. This leads to 450 candidate papers in total that are related to Android. In a second step, we manually analyze each paper and retain it only if it focuses on Android DLVs. We exclude papers that focus on the security of apps, propose a new defense or mitigation (without introducing an attack), or focus on Android malware. Additionally, we do not

[1] ACM CCS proceedings include posters, which we include for completeness.

consider a paper if it focuses on implementation bugs such as memory corruption on Android. After our analysis, we obtain 102 papers introducing DLVs.

We additionally include Black Hat Briefings (Black Hat USA, Black Hat Europe, and Black Hat Asia) on Android security between 2015 and 2024 to capture notable industry research with substantial technical depth. We follow the same criteria to select relevant briefings and obtain a total of 14 for further analysis, leaving us with a total of 116 publications that we study in depth. Table 6 in Appendix A provides a breakdown of these publications.

Publication analysis. For each of the 116 publications, we extract the described DLVs and, if available, their exploitation methodology. We then identify and summarize the root causes of the DLVs and synthesize an auditing methodology. This is to keep the auditing methodology *Android-independent*, such that it can also be used to verify whether these DLVs exist in emerging mobile OSes. Next, we research the DLV to understand if and how it has been mitigated. As Android may have introduced mitigations after the papers were published, we also examine Android Security Bulletins [9], which list CVEs and fixing commits, to investigate the corresponding mitigations. In a last step, we *empirically* audit Android to assess the mitigation (or its absence) in practice, where we: (i) identify its related functionalities and components by referring to Android's documentation, (ii) generate a PoC app that implements the DLV, and (iii) test the vulnerability on an up-to-date Pixel 7a phone with Android 15 installed. For DLVs involving multiple components, our auditing method covers *all* applicable components and instances. Based on the results of our empirical verification, we denote whether Android is *not vulnerable*, *partially vulnerable*, or *fully vulnerable* to a DLV. *Partially vulnerable* indicates that the OS still exposes the design-level attack surface, but existing mitigations block some exploitation scenarios or reduce the impact. For example, Android's mitigation of DLV#45 only covered critical processes, leaving it partially vulnerable.

4 Systematized Auditing Model

After conducting literature analysis, we identify 56 DLVs, detailed in Table 1. We categorize the DLVs following Common Weakness Enumeration (CWE) [49], a hierarchical taxonomy of common software and hardware weaknesses with potential security implications. As Table 1 shows, our taxonomy covers five root CWEs and 12 child CWEs. We mark the child CWEs as N/A if they are unavailable. Table 1 also illustrates the security properties violated by each DLV, which correspond to Confidentiality, Integrity, and Availability (CIA). In the following, we discuss *in italics how to audit a mobile OS on a per-DLV basis*, the security implications, and how Android mitigated the DLV (if at all).

4.1 CWE-284: Improper Access Control

CWE-284 describes that a system fails to properly restrict access to a resource from an unauthorized actor [54]. Under this category, we group 36 DLVs into 8 child CWEs.

CWE-269: Improper Privilege Management

This CWE indicates a failure to correctly assign, update, or enforce privileges, thereby granting an actor an unintended sphere of control [53].

Audit-DLV#1 (C): *Third-party apps shall not invoke APIs whose outputs contain sensitive data.* In Android, malicious apps inferred the runtime state of other apps by exploiting vulnerable Android APIs in the GUI framework [140]. For example, an app could repeatedly invoke `getUidStats()` with another app's UID and observe changes in traffic counters, inferring when the victim app is active.

Mitigation: Android restricted app access to sensitive APIs. Vulnerable methods such as `getUidStats()` and `isAppForeground()` were limited to system apps.

Audit-DLV#2 (C): *Untrusted JS code in web interfaces shall not invoke sensitive native functions.* Android allowed JS code inside a WebView to interact with native Android Java methods [96]. Malicious JS code could execute sensitive operations, such as sending an SMS or reading contacts, by exploiting how the `addJavascriptInterface()` method binds Java objects to JS in a WebView [167, 181].

Mitigation: Android introduced JavaScriptSandbox to isolate untrusted JS and limit access to system APIs [98].

Audit-DLV#3–#7 (C): *Third-party apps shall not access non-resettable device IDs or the device's phone number and shall not obtain precise location, contacts, or SMS without user consent.* The lack of protection for sensitive information allowed apps to stealthily obtain and transmit privacy-infringing data to their servers, even if their core functionality did not require such data [121, 158, 184].

Mitigation: Since Android 10, only system apps can access non-resettable device IDs and must declare a special permission if doing so [143]. Stricter permission controls, such as runtime permission prompts, now restrict apps' access to device phone number, contacts, and SMS. Android 11 introduced one-time permissions (for location information, microphone, and camera), addressing concerns regarding persistent access [135]. Android also started resetting permissions for apps not used for a while [20].

Design principle #1: A mobile OS should explicitly specify and enforce privilege boundaries for interfaces and data-access capabilities to prevent untrusted actors from gaining unintended control, such as inferring system state or accessing sensitive operations, identifiers, and personal data.

Table 1: Summary of Design-Level Vulnerabilities (DLVs) from analyzed papers by CWE. We indicate whether **And**roid 15 and **O**pen**H**armony 4.1/5.1 are vulnerable to this DLV, which property (**C**onfidentiality, **I**ntegrity, or **A**vailability) is violated, and **A**ndroid's **M**itigation **T**ime (Android version that fixed the DLV, if known).
○ *Not vulnerable;* ◐ *Partially vulnerable;* ● *Fully vulnerable;* – *DLV not applicable to OpenHarmony;* *Differences between the OSes are color-coded: Orange indicates that OpenHarmony is more exposed to a DLV than Android, blue the opposite.*

CWE Root	CWE Child	ID	Description of the DLV	Ad	OH	Property C	Property I	Property A	AMT
CWE-284: Improper Access Control	CWE-269: Improper Privilege Management	1	Vulnerable APIs can be exploited for state inference attacks [140]	○	◐	✗			10
		2	JS interfaces in WebView can invoke sensitive system functions [167, 181]	○	●	✗			
		3	Third-party apps can access non-resettable device IDs [121, 143, 158, 184]	○	●	✗			10
		4	Third-party apps can access device's phone number [184]	○	○	✗			6
		5	Third-party apps can access fine location with install-time permission [184]	○	◐	✗			11
		6	Third-party apps can access contacts without user consent [184]	○	○	✗			6
		7	Third-party apps can access SMS without user consent [184]	○	○	✗			6
	CWE-346: Origin Validation Error	8	Custom Tab component has critical security flaws such as cross-context state sharing [23]	○	–	✗			
		9	Exploiting autofill flaws for phishing attacks [17, 25]	○	–	✗			
		10	Deep Links allow malicious apps to hijack sensitive URLs [109, 111, 169]	○	○		✗		12
	CWE-923: Improper Restriction of Communication Channel to Intended Endpoints	11	Unix Domain Sockets can be misused due to weak security mechanisms [41, 157]	○	○	✗	✗		9
		12	Exploiting DICI mechanism for cache side-channel attacks [110]	●	–	✗			
		13	Content provider lacks access control [28, 104, 201]	○	◐	✗			
		14	Malicious apps can hijack unprotected components [191, 201]	○	○		✗		4
	CWE-300: Channel Accessible by Non-Endpoint	15	Apps can leak sensitive data by sending unprotected broadcasts [76]	○	○	✗			
		16	Implicit Intent hijacking [92, 102]	○	●		✗		11
		17	Insecure Intents can be exploited to control camera app [34]	○	○	✗	✗		
		18	Implicit PendingIntents can be hijacked for multiple attacks [32]	○	–		✗		
		19	System broadcasts lack access control, leaking sensitive information [194]	○	●	✗			12
		20	System apps' broadcasts are not protected [33]	○	○	✗			9
	CWE-295: Improper Certificate Validation	21	Improper SSL/TLS validation leads to MITM vulnerabilities [159]	○	○	✗	✗		7
		22	Apps trust any certificate by any trusted Certificate Authority [77, 134, 138]	○	●[1]	✗	✗		7
	CWE-1256: Improper Restriction of Software Interfaces to Hardware Features	23	Apps can exploit motion sensors to perform side-channel attacks [68, 118, 122, 128]	○	●	✗			12
		24	Apps can covertly access microphone in the background [197]	○	●	✗			12
		25	Third-party apps can stealthily pair Bluetooth devices [177]	○	●	✗	✗		10
		26	Third-party apps can stealthily disconnect Bluetooth devices [129]	○	●		✗		9
		27	Third-party apps can stealthily manipulate Bluetooth scan mode [29]	○	●		✗		12
		28	External Bluetooth devices can stealthily connect to the host device [31]	○	●	✗	✗		10
	CWE-552: Files or Directories Accessible to External Parties	29	Apps can exploit system files to bypass permissions and access sensitive data [148, 198]	○	◐	✗			12
		30	Apps can read packet counters to crack sequence number [145]	○	●	✗	✗		10
		31	Apps can read files in /proc to infer secrets [93, 198]	○	◐	✗			10
		32	Apps can read interruption data for inference attacks [69]	○	○	✗			10
		33	Apps can read power data for location tracking [123]	○	●	✗			9
	CWE-749: Exposed Dangerous Method or Function	34	Apps can trigger system crashes and reboots by exploiting unprotected components [156]	○	●			✗	
		35	Apps can exploit unprotected data storing operations to DoS system services [189]	○	○			✗	
		36	Apps can exploit synchronous callbacks to freeze and DoS system services [168]	○	○			✗	6
CWE-221: Information Loss or Omission	CWE-451: UI Misrepresentation of Critical Information	37	Malware can abuse accessibility services to observe and control GUI [78, 89, 94, 178]	○	○	✗	✗		13
		38	Exploiting overlay for clickjacking [139]	○	–		✗		
		39	Exploiting overlay for PHYjacking attacks [170]	○	–		✗		
		40	Exploiting overlay for GUI confusion attacks [37, 38]	○	–		✗		
		41	Exploiting overlay for phishing attacks [196]	○	–	✗			
		42	Apps can launch Activity from the background to hijack UI [44, 163]	○	◐		✗		10
		43	Picture-in-picture (PiP) can be exploited for UI hijacking attacks [27]	○	○		✗		11
		44	UI task stack flaws lead to task hijacking [149]	○	–		✗		
CWE-1038[2]	N/A	45	App process creation weakens ASLR by sharing memory layouts across processes [100]	◐	◐		✗		7
		46	Apps can exploit shared memory side channels to infer UI states [44]	○	○	✗			6
CWE-664: Improper Control of a Resource Through its Lifetime	N/A	47	System does not clean up data residues from uninstalled apps [192]	○	○	✗			6
	CWE-921: Storage of Sensitive Data in a Mechanism without Access Control	48	Apps can leak sensitive data to device logs, which other apps can access [76]	○	○	✗			4
		49	System apps can silently access global device logs [116]	○	●	✗			13
		50	Apps can access and modify other apps' data in external storage [76]	○	○	✗	✗		10
		51	Race conditions in external storage [72]	○	–		✗		
	CWE-514: Covert Channel[3]	52	Apps can covertly store identifiers in media files by appending bytes to images [71]	●	○	✗			
		53	Apps can exploit volume and vibration settings to stealthily transmit sensitive data [152]	●	○	✗			
		54	Apps can covertly send data to servers by launching other apps using ICC objects [172]	●	●	✗			
		55	Servers can covertly send commands to apps through timing-based network channels [172]	○	○	✗			9
CWE-311[4]	CWE-319[5]	56	Apps use insecure HTTP cleartext network communication [86, 134, 138]	◐	●	✗	✗		9

[1]: fixed in OpenHarmony 5.1
[2]: CWE-1038: Insecure Automated Optimizations
[3]: Covert channels are not within Android's threat model
[4]: CWE-311: Missing Encryption of Sensitive Data
[5]: CWE-319: Cleartext Transmission of Sensitive Information

CWE-346: Origin Validation Error

CWE-346 encompasses situations where a component accepts data or requests without correctly verifying that they originate from an authorized source [59].

Audit-DLV#8 (C): *Custom Tabs shall not share browser state with the host apps or expose detailed navigation callbacks.* Malicious apps leaked sensitive data, bypassed SameSite cookies, and injected HTTP headers in the cross-context between apps and web environments [23].
Mitigation: Android now sanitizes HTTP header values, prevents `SameSite=Strict` cookies from being sent in Custom Tabs, and disables scroll callbacks in sensitive contexts.

Audit-DLV#9 (C): *Autofill framework [8] must have secure target validation.* An insecure target validation allowed attackers to trick password managers into autofilling credentials into malicious apps and embedded WebViews [17, 25].
Mitigation: Android integrated Digital Asset Links–based domain verification into the autofill framework to return credentials only for verified request origins.

Audit-DLV#10 (I): *Deep links must be protected through strict origin-based verification.* Deep links redirect users directly into apps from links they tap [10]. Insufficient verification allowed malicious apps to register for and hijack legitimate deep links [109, 111, 169].
Mitigation: Android introduced the verification of App Links [7], which requires apps to prove domain ownership via Digital Asset Links (DAL) before automatically handling corresponding URLs.

> **Design principle #2:** A mobile OS should validate cross-app and cross-web origins to ensure actions and data are bound to verified, intended sources, and to prevent attackers from impersonating legitimate sources.

CWE-923: Improper Restriction of Communication Channel to Intended Endpoints

CWE-923 concerns cases where a privileged communication channel is established without ensuring the peer endpoint is the intended one [66].

Audit-DLV#11 (C+I): *Kernel-level and framework-level IPC mechanisms must have consistent security control.* Unprivileged apps could exploit kernel-level channels, such as unprotected Unix Domain Sockets, to bypass the Android permission model and communicate directly with privileged daemons or other apps, leading to privilege escalation, data leaks, and covert app collusion [41, 157].
Mitigation: Android integrated SELinux to enforce mandatory access control over socket creation and communication, restricting third-party apps from accessing specific sockets.

Audit-DLV#12 (C): *Third-party apps shall not map another app's code into their own memory space.* Attackers could perform fine-grained cache-based side-channel attacks without any permissions by abusing the Dynamic Inter-App Component Invocation (DICI) feature [110]. The insecure DICI mechanism enabled apps to dynamically load and execute other apps' code contexts (DEX, ODEX, and native libraries) into their own memory space using the `createPackageContext()` API. Even when the invocation fails, Android still loads the target app's code into the attacker's address space.
Mitigation: Android acknowledged the issue in 2024 but planned to patch it only in future releases.

Audit-DLV#13 (C): *Inter-app data sharing mechanisms must have sufficient validation and access control.* Using a content provider, an Android app can allow another app to access its own data through a Content URI, for example, `content://provider_authority/path` [46]. Insufficient access control allowed malicious apps to read sensitive private data (content leaks) or modify internal app databases and settings (content pollution) [28, 201]. Misconfigurations of content providers also exposed sensitive data, including user settings and personal information [104].
Mitigation: Android introduced custom permissions that app developers can define and specify protection levels, and only apps that declare the corresponding custom permissions can access the app's content providers.

Audit-DLV#14 (I): *Components in apps must be protected by default and should not accept external Intents without verifying their origin.* Exported components—such as Activities and Services in benign apps—were exploitable by malicious apps to perform unauthorized operations, including data exfiltration or privilege escalation [191, 201].
Mitigation: Android changed the default `android:exported` flag in components to `false` in Android 4, enforcing that app developers explicitly declare exported components.

CWE-300: Channel Accessible by Non-Endpoint

CWE-300 characterizes that an attacker who is not one of the intended endpoints can still interfere with the channel [56].

Audit-DLV#15 (C): *Broadcast Intents with sensitive data must be protected.* Apps leaked sensitive data (e.g., location and identifiers) by sending unprotected broadcast Intents, allowing any other app on the device to receive the data via interprocess communication (IPC) without any permissions [76].
Mitigation: Android introduced protected broadcasts to restrict system-level broadcasts from being intercepted and requires apps to declare custom permissions for them.

Audit-DLV#16 (I): *Third-party apps shall not hijack implicit Intents with critical Intent actions.* Unlike explicit Intents, which launch a particular component, implicit Intents just signal an action to be performed—any app can register to handle such Intents [12]. In Android, this allowed a malicious app to declare an action with a higher priority, e.g., `android:priority="1000"`, to hijack implicit Intents from other apps and perform malicious operations [92, 102].

Mitigation: Android mitigated this issue by restricting some critical Intent actions. Since Android 11, only pre-installed system camera apps can declare camera actions [125].

Audit-DLV#17 (C+I): *Third-party apps shall not trigger critical components in system apps without user consent.* Malicious apps could silently take photos, record videos, and access location data without requesting camera or storage permissions, by exploiting exported activities and implicit Intent actions [34].
Mitigation: Android revoked implicit exposure of exported camera components and enforced explicit Intent whitelisting and permission validation before camera actions.

Audit-DLV#18 (I): *Third-party apps shall not modify critical data in PendingIntents (tokens allowing other apps to perform an action [15]).* In Android, system apps (e.g., Settings, SystemUI, and Bluetooth) unintentionally exposed mutable PendingIntents with empty or implicit base Intents [32]. Attackers could retrieve these PendingIntents and modify their internal data to execute arbitrary actions.
Mitigation: Android 12 required developers to explicitly declare the mutability of every PendingIntent.

Audit-DLV#19 (C): *Sensitive system broadcasts must specify recipients or enforce required permissions on receivers.* In Android, system broadcasts lacked permission checks, leaking sensitive data (such as network information) to unauthorized apps without requiring any permissions [194].
Mitigation: Android required apps to declare specific permissions to receive such sensitive broadcasts.

Audit-DLV#20 (C): *All system apps must be able to declare protected broadcast actions.* Third-party apps could freely send spoofed system broadcasts, leading to privilege escalation and command execution with system privileges [33]. Due to a flaw in PackageManagerService, system apps outside privileged directories lost protection for their declared protected broadcasts.
Mitigation: Android modified PackageManagerService to explicitly clear all protected broadcast declarations for any non-system or non-privileged app, ensuring that only privileged system apps can register or send protected broadcasts.

Design principle #3: Recurring lessons (10 DLVs) indicate that maintaining communication channel integrity is a major challenge in Android, especially in various IPC mechanisms. Endpoint restriction and integrity protection must be enforced to prevent attackers from infiltrating these channels.

CWE-295: Improper Certificate Validation

CWE-295 concerns cases where a system accepts a certificate without performing correct validation, thereby undermining authentication of the remote party [55].

Audit-DLV#21 (C+I): *Apps must implement secure SSL/TLS certificate validation.* Insecure SSL/TLS certificate validation led to man-in-the-middle (MITM) attacks [159]. Developers often implemented custom SSL validation instead of using Android's, which led to mistakenly accepting all certificates or self-signed certificates without proper checks.
Mitigation: Android strengthened SSL enforcement, such as introducing the certificate pinning APIs and the Network Security Configuration feature, which allows developers to securely configure certificate validation in XML.

Audit-DLV#22 (C+I): *Apps shall not trust any valid certificate issued by any trusted Certificate Authority (CA).* Android apps trusted any certificate without verifying the domain name linked with the certificate, risking MITM attacks [77, 134, 138].
Mitigation: Android introduced the Network Security Configuration file and certificate pinning [131]. The former allows apps to customize network security settings globally in a declarative configuration file, and the latter restricts an app to accepting certain certificates.

Design principle #4: Relying on apps to use homegrown certificate validation leads to insecure implementations. A mobile OS should offer secure primitives, such as declarative validation [131], to prevent improper certificate validation.

CWE-1256: Improper Restriction of Software Interfaces to Hardware Features

CWE-1256 arises when software-accessible hardware controls are insufficiently constrained, allowing untrusted parties to abuse hardware state or physical side channels [51].

Audit-DLV#23 (C): *Third-party apps shall not access motion sensors [126] at a high rate without permissions or user consent.* Attackers could exploit sensitive motion sensors for multiple side-channel attacks. Accelerometer and gyroscope data sampled at 50–100 Hz could recover 4-digit PINs at a success rate up to 80% [120]. Gyroscopes sampled at 200 Hz can capture low-frequency components of speech [122]. An accelerometer could be exploited to eavesdrop on nearby keyboard input [118]. An app running on a phone placed close to a keyboard could capture subtle vibrations caused by keypresses and infer the typed text. Without GPS, Wi-Fi, or cellular data access, an app could still accurately infer a user's driving routes and locations using motion sensors [128]. In addition, ads embedded in apps could also access motion sensors through WebView and launch side-channel attacks [68].
Mitigation: Android 12 limits how often an app can access motion sensor data [154]. If an app uses the `SensorDirectChannel` class, Android limits the sensor sampling rate to `RATE_NORMAL` (usually about 50 Hz). If an app needs to access motion sensor data at a higher rate, it must obtain the `HIGH_SAMPLING_RATE_SENSORS` permission. When the user turns off microphone access in Android using device toggles, the motion sensors are always rate-limited.

Audit-DLV#24 (C): *Third-party apps shall not access the microphone in the background without user consent.* Malicious apps abused the microphone to perform acoustic side-channel attacks, for example, via emitting inaudible sound and using the microphone to capture acoustic reflections from a user's fingertip, enabling reconstruction of pattern locks [197].

Mitigation: Android 12 introduced privacy indicators, which indicate when apps access the microphone or camera [91]. Android also mandates that any apps needing background microphone access must register a foreground service that only the user can start. Background apps cannot access the microphone without user interaction [21].

Audit-DLV#25 (C+I): *Third-party apps shall not covertly pair Bluetooth devices without user consent.* Prior to Android 10, malicious apps could covertly pair Bluetooth peripherals without user interaction. This vulnerability (CVE-2019-2225 [47]) also allowed attackers to pair a benign device and switch to a malicious profile (e.g., Human Interface Device) to inject keyboard input [177].

Mitigation: Android displays a pop-up dialog and asks for users' consent when apps try to pair Bluetooth devices.

Audit-DLV#26 (I): *Third-party apps shall not disconnect a paired Bluetooth device without user consent.* Prior to Android 9, malicious apps could covertly hijack existing Bluetooth connections by disconnecting a paired Bluetooth device and then pairing to a malicious device [129].

Mitigation: Android deprecated the APIs that allowed apps to disconnect or unpair a Bluetooth device [70].

Audit-DLV#27 (I): *Third-party apps shall not manipulate Bluetooth scan mode without user consent.* Bluetooth scan mode determines if the device can be discovered and connected to by another device. In Android, insufficient permission enforcement allowed apps to change the Bluetooth scan mode, e.g., to discoverable and connectable mode without user consent [29].

Mitigation: This vulnerability (CVE-2022-20126 [48]) was mitigated in Android 12 by restricting access to these functions using the `BLUETOOTH_PRIVILEGED` permission, available only to system apps. As of Android 15, related APIs have been removed entirely, fully blocking app access.

Audit-DLV#28 (C+I): *External Bluetooth devices shall not connect to the phone without user consent.* Due to Android's flawed handling of Bluetooth pairing and profile authorization under the `Just Works` mode, external Bluetooth devices could bypass Android's user consent dialog by changing their capability to `NoInputNoOutput` [31]. When Bluetooth devices with `NoInputNoOutput` capability connect, Android falls back to `Just Works` mode, skipping user consent dialogs and forming temporary bonds without authentication. This allowed the device to pair and connect without user interaction, enabling attacks like accessing the phonebook or controlling calls.

Mitigation: Android enforced a user consent dialog for connection requests from all types of Bluetooth devices, including `NoInputNoOutput` devices.

> **Design principle #5:** A mobile OS must properly restrict high-fidelity hardware interfaces (e.g., microphone and motion sensors), because unrestricted access to them turns into powerful attack surfaces.

CWE-552: Files/Directories Accessible to External Parties

CWE-552 concerns cases where files or directories that should be restricted become accessible to unauthorized actors [62].

Audit-DLV#29–#32 (C, I): *Third-party apps shall not access system files with sensitive information in `/proc`, such as router MAC address, TCP packet counters, processes' statistics, and interrupt counts.* In Android, such system files allowed apps to infer sensitive user information [148, 198]. For example, apps can read `/proc/net/arp` to obtain the router MAC address to infer the user's geolocation. Android TCP packet counters were exposed through `/proc/net/netstat` and could be read by any app. This allows attackers to infer sequence numbers of active connections and, thus, enables fast off-path TCP injection and hijacking attacks [145]. An unprivileged app could also infer sensitive information, such as visited websites or keystroke timings, by monitoring changes in other processes' memory footprints and CPU scheduling statistics through `/proc` [93, 198]. In addition, an interrupt timing side-channel was discovered to allow apps to infer sensitive user information [69]. Android's kernel logs interrupt counts for hardware devices (touchscreen, display, etc.) in `/proc/interrupts`. By monitoring it, attackers could infer a user's unlock pattern by analyzing touchscreen interrupt timing and identifying the foreground app by analyzing display refresh interrupts.

Mitigation: Android hardened app access to system files in `/proc`. Android 10 restricted apps' access to `/proc/net/` [144]. Android also restricted visibility of other apps' `/proc/<pid>` entries, enforcing "same-UID only" access, and restricted app access to `/proc/interrupts`.

Audit-DLV#33 (C): *Third-party apps shall not access system files with power consumption statistics.* Unprivileged apps could access such files to track a user's location by analyzing the device's aggregate power consumption, even without GPS or location permissions [123].

Mitigation: Android removed access to `/sys` power files for third-party apps.

> **Design principle #6:** Beyond enforcing app permissions at the system API level, a mobile OS should carefully restrict apps' access to file systems such as `/proc` and `/sys`. Failing to do so enables attackers to bypass the intended permission model.

CWE-749: Exposed Dangerous Method or Function

CWE-749 captures interfaces that expose inherently dangerous functionality without sufficient restriction [64].

Audit-DLV#34 (A): *Third-party apps shall not trigger system crashes or soft reboots through unprotected components.* Android system components exposed unprotected or inconsistently protected interfaces that allowed third-party apps to trigger system crashes and soft reboots, leading to system-level DoS attacks [156].
Mitigation: Android added and aligned missing permission and UID checks in those unprotected components.

Audit-DLV#35 (A): *Data-storing operations in system services must be protected against DoS attacks.* Apps could exploit unprotected data-storing operations within system services to exhaust memory and trigger temporary or permanent Denial-of-Service (DoS) [189].
Mitigation: Android added memory size checks and limited the number of stored objects per app.

Audit-DLV#36 (A): *System services shall not use synchronous callbacks to apps without isolation or timeout checks.* Synchronous callbacks between privileged system services and untrusted apps led to DoS attacks [168] when these apps never returned or threw exceptions.
Mitigation: Android replaced synchronous callbacks with an asynchronous mechanism with timeout protection.

> **Design principle #7:** A mobile OS should avoid exposing dangerous system interfaces—especially crash-prone or resource-amplifying operations—to untrusted actors, to prevent system-level denial-of-service.

4.2 CWE-221: Information Loss or Omission

CWE-221 refers to a lack of security-relevant information that leads to incorrect decisions or hampers later analysis [52]. Under this root CWE, we group 8 DLVs into 1 child CWE.

CWE-451: User Interface (UI) Misrepresentation of Critical Information

CWE-451 concerns situations where the UI presents security-relevant information in a misleading way, making it possible to hide or impersonate the original content [60].

Audit-DLV#37 (C+I): *Third-party apps shall not utilize accessibility features to observe and manipulate confidential GUI interactions without sufficient user awareness.* Malicious apps used accessibility features to observe and control the GUI, breaking the app sandbox isolation [78, 89, 94, 178].
Mitigation: Android required explicit user consent for accessibility services on a per-app basis. This is complemented by additional warnings about the risk and by disabling accessibility features for apps installed from untrusted sources.

Audit-DLV#38–#41 (C, I): *Third-party apps shall not draw overlays without user awareness.* In Android, an overlay is an extra layer that sits on top of a View [14], which can be used by apps to display messages or alerts. This was exploited by malicious apps to perform clickjacking [139], PHYjacking [170], GUI confusion [37, 38], and phishing attacks [196].
Mitigation: To prevent non-system apps from drawing overlays on top of critical windows that handle sensitive operations, Android introduced the flag HIDE_NON_SYSTEM_OVERLAY_WINDOWS (HNSOW).

Audit-DLV#42 (I): *Third-party apps shall not launch Activities in the background.* Malicious apps used phishing attacks to hijack UIs by covertly launching their activities [44, 163].
Mitigation: Android prohibited all apps, including system apps, from launching Activities in the background [22].

Audit-DLV#43 (I): *Picture-in-picture (PiP) windows must enforce boundary checks, and apps shall not create tiny PiP windows.* Malicious apps could start tiny PiPs that were barely visible to keep themselves in the foreground, enabling Activity injection attacks even though launching activities from the background was removed [27].
Mitigation: Android enforced stricter PiP boundary checks to disallow background PiP transitions.

Audit-DLV#44 (I): *Activities from different apps shall not co-exist in the same UI task stack, such that apps can inject their own Activities into another app's task stack.* In Android, this led to task hijacking vulnerabilities, enabling UI spoofing without special permissions [149].
Mitigation: Android restricted cross-task activity behaviors to isolate app tasks and mitigate task hijacking attacks.

> **Design principle #8:** A mobile OS must prevent attackers from abusing UI features, such as accessibility, overlay, or PiP functionality, to impersonate trusted surfaces and actions to compromise UI integrity.

4.3 CWE-1038: Insecure Automated Optimizations

CWE-1038 refers to situations where system optimizations introduce side effects that break intended security assumptions [50]. We categorize 2 DLVs under this root CWE.

Audit-DLV#45 (I): *The app process creation mechanism shall not cause apps to share identical memory layouts.* In Android, Zygote—Android's app process creation model, where all apps are launched by forking from a single parent process—caused all apps and services to share identical memory layouts and weakened ASLR [100, 205].
Mitigation: Android now separately creates critical processes, e.g., Media Server, partly mitigating this threat. However, it still forks other processes from the same parent.

Audit-DLV#46 (C): *Third-party apps shall not access shared memory in the GUI framework.* Malicious apps inferred the runtime state of other apps via shared memory [44].
Mitigation: Apps can no longer access shared-memory files.

Design principle #9: Optimizations often create new sharing and determinism, which turns per-app uncertainty into system-wide predictability. A mobile OS should secure such optimizations by preserving per-process diversity and preventing cross-app observability.

4.4 CWE-664: Improper Control of a Resource Through its Lifetime

CWE-664 describes scenarios where a system incorrectly maintains control over a resource throughout its lifetime of creation, use, and release [63]. We categorize 1 DLV under this root CWE and 8 DLVs under its 2 child CWEs.

Audit-DLV#47 (C): *App data must be cleaned up completely after uninstallation.* Residual app data, for example, in the AccountManager, allowed newly installed apps to steal credentials, gain privileges, or access private data [192].
Mitigation: Android enforced proper data cleanup and unique app identifiers to prevent residual data reuse after app removal.

CWE-921: Storage of Sensitive Data in a Mechanism without Access Control

CWE-921 covers sensitive data being stored in a file system or device that does not have built-in access control [65].

Audit-DLV#48 (C): *Third-party apps shall not access global device logs, which can contain sensitive data such as credentials or personally identifiable information (e.g., GPS or IMEI) [113].* This led to privacy leaks [76], since any app holding the `READ_LOGS` permission could access global logs.
Mitigation: Since Android 4.1, only system apps have access to device logs by declaring the `READ_LOGS` permission. Third-party apps can now only read their own logs.

Audit-DLV#49 (C): *System apps shall not access global logs without user consent.* Preinstalled apps from manufacturers and carriers could misuse the `READ_LOGS` permission to access and leak system logs without user consent [116].
Mitigation: Android 13 enhanced the log access mechanism and restricted full-log access to privileged system apps. Android also displays a one-time user consent prompt before granting a system app access to global logs [114].

Audit-DLV#50–#51 (C+I): *Third-party apps shall not read or modify other apps' files.* In Android, the lack of protection on external storage (an unsandboxed partition shared by all apps) led to unauthorized data leakage and manipulation, since malicious apps could read from or modify another app's files [76]. Malicious apps could also manipulate a file just before a victim app accessed it [72].
Mitigation: Android introduced scoped storage and per-app sandboxed directories, which isolate each app's storage space and restrict arbitrary cross-app file access [153].

Design principle #10: In addition to strict control over the data lifecycle, a mobile OS should also carefully handle other global data, especially that stored in insufficiently protected environments, such as globally readable logs and shared external storage.

CWE-514: Covert Channel

CWE-514 refers to a path that can be used to transfer information in a way not intended by the system's designers [61]. Although covert channels are typically considered app collusion, the mobile OS can mitigate them by controlling or blocking these channels. DLVs in this category (#52-#55) fall outside the Android threat model [119]. However, top-tier publications document their real-world security impact.

Audit-DLV#52 (C): *Third-party apps shall not store identifiers in media files.* In Android, this enabled cross-app user tracking and breached privacy protection policies, since apps could covertly store identifiers in media files by appending bytes to images without corrupting these files [71].
Mitigation: This vulnerability has *not* been mitigated, since public media files are not restricted by scoped storage. We verified that on Android 15, any Android app can store identifiers in an image file, and any apps with the `READ_MEDIA_IMAGES` permission can read images and extract these identifiers.

Audit-DLV#53 (C): *The vibration and volume levels shall not be globally observable and modifiable by third-party apps.* In Android, vibration and volume settings allow apps to stealthily transmit sensitive data [152]; for example, one app encodes data into volume levels while another reads these levels at the same frequency to recover the data.
Mitigation: Still viable on up-to-date Android 15 devices.

Audit-DLV#54 (C): *Third-party apps without the `INTERNET` permission shall not send data to remote servers.* Apps could covertly send data to remote servers by launching other apps (e.g., a browser) using Intents [172].
Mitigation: The latest version of Android is still vulnerable.

Audit-DLV#55 (C): *Third-party apps shall not access network statistics files.* In Android, this created a server-to-app covert channel, which allowed servers to encode commands in network packet timing intervals of legitimate traffic [172].
Mitigation: Android mitigated the server-to-app covert channel by restricting apps' access to network statistics files.

Design principle #11: Mitigating covert channels is challenging because they are often coupled to usability and interoperability. Even with strict app-level sandboxing, a mobile OS should minimize and mediate shared system states and IPC to prevent covert channels.

4.5 CWE-311: Missing Encryption of Sensitive Data

CWE-311 captures cases where a product does not encrypt sensitive information before storage or transmission [57]. We categorize 1 DLV under this root CWE.

CWE-319: Cleartext Transmission of Sensitive Information

CWE-319 concerns a system transmitting sensitive data in cleartext that can be sniffed by unauthorized actors [58].

Audit-DLV#56 (C+I): *Apps may not use cleartext HTTP for network communications.* In Android, cleartext HTTP communication posed significant security risks due to the lack of encryption or integrity checks [134, 138], enabling MITM attacks, session hijacking, or content injection.
Mitigation: Android enforces HTTPS connections by default [86]. An app sending an HTTP request will throw a runtime Exception, unless developers explicitly override this.

Design principle #12: Since the network communication of apps is implemented by their developers, a mobile OS should enforce encrypted transmission whenever possible to prevent unauthorized actors from sniffing network data.

Key insight #1: We identify 56 DLVs in Android from the 116 relevant publications. Empirical verification uncovers that four of them remain unfixed as of today. Interestingly, three of them are related to covert channels. DLVs primarily compromise confidentiality and integrity rather than availability (3 of 56), showing that their impact is more often stealthy and persistent than immediately disruptive. Notably, 36 out of the 56 DLVs are caused by improper access control. This suggests that, from the perspective of top-tier venue publications, access control is a major design challenge in Android.

5 Application to an Emerging Mobile OS

In the following, we apply our systematization of Android DLVs to an emerging mobile OS to examine whether Android DLVs also manifest in such systems. We choose OpenHarmony, the only completed mobile OS reimplementation to date, with others still in development [166, 175]. We introduce OpenHarmony and its relationship with Android, describe how we applied our auditing methodology, and discuss the Android DLVs that translated to OpenHarmony.

5.1 Background on OpenHarmony

Driven by various geopolitical, technological, and commercial factors, Huawei started to reimplement a custom mobile OS, HarmonyOS, in 2019, starting as a customized Android [84].

Table 2: Terminology mapping between Android and OpenHarmony for components mentioned in this paper.

Component	Android	OpenHarmony
App Development Language	Java/Kotlin	ArkTS
UI Component	Activity	Ability
Broadcast Component	Broadcast Receiver	Common Event
App Configuration File	AndroidManifest.xml	module.json5
App Process Creation Model	Zygote	Appspawn
App Data Sharing Mechanism	Content Provider	Data Ability
ICC Object	Intent	Want
Privileged App	system app	system_{basic,core}
System Logs Dumping Tool	logcat	hilog
Dynamic Permission Granting	Runtime permissions	user_grant permissions
IDE	Android Studio	DevEco Studio

Progressively, they replaced AOSP components with self-developed alternatives. By 2024, HarmonyOS NEXT eliminated all AOSP code and Android app compatibility [82], completing the transition from an Android customization to an independent mobile OS. Despite being new, HarmonyOS has already surpassed Apple's iOS to become the second best-selling mobile OS in China [81] and has been deployed on over 1 billion devices [83]. Its popularity is contrasted by a lack of security research: Li et al. [106] highlighted the limited research on OpenHarmony, the open-source counterpart to HarmonyOS NEXT (analogous to AOSP's relationship with commercial Android), with only 8 related papers published in software engineering venues compared to over 7,000 on Android, and none published in system security.

5.2 OpenHarmony vs. Android

Our paper's underlying hypothesis is that OpenHarmony and Android share similar designs, yet are different implementations. In a first step, we dissect the implementation differences of the two OSes. Publicly, news reports and Huawei claim that OpenHarmony is an independent OS and does not contain any Android code [81, 82, 90]. We download the Android Open Source Project (AOSP) from the `android16-qpr2-release` branch and the OpenHarmony source code from the `OpenHarmony-5.1.0-Release` branch, and we conduct a multi-layer analysis to empirically review their relationship:

Terminology mapping. Android and OpenHarmony use different terms to refer to similar concepts, see Table 2.

App runtime. Android apps are packaged as APKs and executed through the Android Runtime (ART), which runs DEX bytecode compiled from Java/Kotlin code. In contrast, OpenHarmony uses a different app model based on Ability components and HAP packages, with applications commonly developed using ArkTS/JS and compiled and executed through the Ark runtime and toolchain. This difference indicates that OpenHarmony does not reuse Android's app execution stack.

Shared file paths. AOSP contains 1,539,491 files and 302,998 directories, while OpenHarmony contains 1,149,148

Table 3: Code similarity between randomly selected modules outside the core framework in AOSP and OpenHarmony. NCDs closer to 0 indicate similarity; 1 indicates distinct code.

Module	AOSP Path	OpenHarmony Path	NCD
Hardware interfaces	hardware/interfaces	drivers/interface	0.9993
Storage management	system/vold/	foundation/filemanagement/storage_service/	0.9887
Media	frameworks/av/	foundation/multimedia/	0.9994
Network daemon	system/netd/	foundation/communication/netmanager_base/	0.9898

files and 291,002 directories. They share 0 file paths and only 5 directory paths, indicating structural differences.

The latest full codebases of both AOSP and OpenHarmony are around 100 GB in size, and they are difficult to directly compare due to their massive sizes. We treat their core frameworks and the rest of the source code separately.

Source code outside the core framework. Outside the core framework, AOSP and OpenHarmony both include substantial native-code components, such as native system services, hardware abstraction layers, system libraries, and other low-level OS modules. These components are largely implemented in C/C++ in both Android and OpenHarmony, making direct text-level code comparison meaningful. We apply Normalized Compression Distance (NCD) [107] to randomly selected modules outside the core framework to quantify their code similarity. To reduce compressor-window artifacts, we compute NCDs on code files for specific modules using `xz -lzma2=dict=1536MiB` (1.5GiB compressor-window). Values closer to 0 indicate similarity, while 1 means the code is distinct. Table 3 shows that the codebases differ significantly.

Core framework code. AOSP's core framework is largely implemented in Java/Kotlin, whereas OpenHarmony's core framework is primarily implemented in C/C++. This, together with the previous results, suggests that OpenHarmony is not a direct Android fork, but it does not rule out OpenHarmony reimplementing Android's core framework on a per-function basis. Thus, we further analyze language-agnostic features of the two core frameworks: We extract and compare function names, variable names, and interface names, which likely would be similar across function-by-function reimplementations. We extract these features from the Abstract Syntax Trees (ASTs) of randomly selected modules and compute the Jaccard similarity over exact names and cosine similarity over tokenized names after removing stop words (e.g., is, on, to, and from). As shown in Table 4, all Jaccard similarities are close to zero and cosine similarities remain modest, ranging from 0.0195 to 0.3036. These results further suggest that the two core frameworks share minimal similarity in the language-agnostic features.

We conclude that OpenHarmony is not a fork and not a function-by-function reimplementation of Android. It appears to be an independent reimplementation that follows similar high-level designs, as further supported by similar components in Table 2 and Huawei's description [90].

Table 4: Identifier-name similarity between randomly selected modules in the core frameworks of AOSP and OpenHarmony. Jaccard similarity is computed over exact names, while cosine similarity is computed over tokenized names. Values closer to 0 indicate limited shared naming patterns, whereas values closer to 1 indicate stronger naming similarity.

Module	Metric	Functions	Variables	Interfaces
Bluetooth	Jaccard	0.0042	0.0182	0.0053
	Cosine (tokenized)	0.2442	0.2579	0.0632
Telephony	Jaccard	0.0008	0.0317	0.0117
	Cosine (tokenized)	0.2620	0.2254	0.0993
Wifi	Jaccard	0.0022	0.0307	0.0196
	Cosine (tokenized)	0.3036	0.2722	0.1010
Location	Jaccard	0.0003	0.0240	0.0015
	Cosine (tokenized)	0.1659	0.0892	0.0195

5.3 Empirical Verification of DLVs

We apply our systematization from Section 4 to OpenHarmony, which is enabled by the implementation-independent auditing methodology and the similar designs, via these steps:
1) Applicability. We first determine if a DLV is applicable to OpenHarmony by finding the corresponding components and features. If a DLV in Android arises from a feature absent in OpenHarmony, we classify it as *inapplicable*.
2) Reconnaissance. Subsequently, we investigate the OpenHarmony documentation and locate the related functionalities and components, such as system APIs and system files.
3) Proof-of-Concept. Guided by the auditing methodology in Section 4, we create PoC apps and attempt to reproduce the attack described by the DLV.
Experimental setup and OpenHarmony version. For our experiments, we used stock OpenHarmony, installed without modifications or recompilation on an OHarmPi phone. We verified all DLVs against OpenHarmony 4.1 (released in March 2024) and OpenHarmony 5.1 (released in May 2025). Our systematized auditing model can be applied to future versions of OpenHarmony or other emerging mobile OSes.

5.4 Results

Out of the 56 Android DLVs, 46 are applicable to OpenHarmony. This emphasizes that a reimplemented OS shares many design decisions with its parent. Our empirical verification shows that OpenHarmony is vulnerable to 24 of the 46 DLVs. We now describe these in more detail.

CWE-284: Improper Access Control

DLV#1. Android limited the `getUidStats()` API to system apps in response to DLV#1 [140]. We identified two APIs in OpenHarmony, `statistics.getUidRxBytes()` and

`statistics.getUidTxBytes()`, that third-party apps can use to obtain network statistics [132], inferring when the victim app is active.

DLV#2. On OpenHarmony, apps can also expose native OpenHarmony methods to JavaScript code that runs inside a WebView using the `javaScriptProxy()` method [97]. Critically, OpenHarmony does not provide any sandboxing features to restrict JavaScript code running inside a WebView.

DLV#3, #5. OpenHarmony restricts apps' access to non-resettable device IDs. Apps must declare the `ACCESS_UDID`—a system-app only permission—before invoking the `deviceInfo.serial` API to get the device serial number [67]. However, all apps can obtain the device serial number by reading `/proc/bootdevice/cid`. OpenHarmony supports neither one-time permissions nor permission auto-reset, which means all permissions are persistent until users manually revoke them, leading to concerns of apps having persistent access to sensitive information [184].

DLV#13. In OpenHarmony, DataAbility (the equivalent of Android's Content Providers) enables apps to share internal data with other apps. The DataAbility configuration contains two optional fields, `readPermission` and `writePermission`. However, while apps set these fields, any other app invoking DataAbility without the permissions can still read or write the data, voiding the permission control. Additionally, OpenHarmony does not support custom permissions, making secure data sharing between apps unreliable.

```
1 export default class EntryAbility extends UIAbility {
2   onCreate(want: Want, launchParam: AbilityConstant.
        LaunchParam): void {
3     // Implicit Want usage
4     let wantInfo: Want = {
5       action: 'action.settings.app.info'
6     };
7     this.context.startAbility(wantInfo);
8     ...}}
```

Listing 1: Example of implicit Want usage in OpenHarmony.

DLV#16. OpenHarmony apps can use implicit Want objects to start an app component without specifying `abilityName`, which is the target component name, as shown in Listing 1. This mechanism is vulnerable to hijacking attacks: A malicious app can declare the action `action.settings.app.info` in its configuration file and hijack the system Settings app. Whenever an app uses an implicit Want to open the settings app, the malicious app gets launched instead. This applies to all system apps, as OpenHarmony provides no protection for critical system actions.

DLV#19. OpenHarmony allows apps to subscribe to system Common Events [45], the equivalent of system broadcasts, to receive and monitor key system information. However, some Common Events expose data that typically needs privileged API access. Table 5 shows six system Common Events that violate the permission control model in OpenHarmony. For example, apps can subscribe to `wifi.HOTSPOT_STATE` to get Wi-Fi hotspot state changes. An OpenHarmony system API, `wifiManager.on('hotspotStateChange')`, yields the same information but requires a permission.

DLV#22. OpenHarmony did not support certificate pinning in version 4.1, but introduced it in version 5.1 [43]. However, apps can only set certificate pinning each time they make an HTTPS request. OpenHarmony does not provide configuration files for configuring the network security for the entire app, which is inconvenient for apps to follow network security best practices.

DLV#23. On OpenHarmony, apps can access motion sensor data using the `@ohos.sensor` APIs, including `on.ACCELEROMETER`, `on.GYROSCOPE`, and `on.ORIENTATION` [155]. OpenHarmony does not rate-limit these APIs. Apps can set the sampling rate to arbitrarily high values, which introduces risks of side-channel attacks.

DLV#24. Analogous to Android, OpenHarmony displays a privacy indicator when apps access the microphone. However, the indicator is difficult for users to notice without close inspection, even when using the default home screen background. Additionally, the privacy indicator does not appear on the lock screen, even when an app is actively using the microphone. This design fails to mitigate the risk of acoustic side-channel attacks. Apps (with the `MICROPHONE` permission) can also access the microphone from the background and record audio without user interaction [124]. This, combined with the hard-to-notice privacy indicator, makes OpenHarmony devices vulnerable to acoustic side-channel attacks [197].

DLV#25–#28. OpenHarmony apps can covertly pair Bluetooth devices using the `@ohos.bluetooth.connection` API module [40]. Invoking `connection.pairDevice()` does not require user interaction for pairing.

The `@ohos.bluetooth.ble` module provides an API for apps to disconnect Bluetooth Low Energy devices [39], but there is no API for apps to disconnect classic Bluetooth devices. However, we still found a technique for apps to covertly disconnect classic Bluetooth devices. Apps can invoke `access.disableBluetooth` and `access.enableBluetooth` to switch Bluetooth off and on without user consent [165]. In conjunction with the covert connection vulnerability, we launched a Bluetooth spoofing attack on the OpenHarmony device. Android is not vulnerable to this, as it deprecated the counterpart APIs in Android 13 [164].

`@ohos.bluetooth.connection` provides the `setBluetoothScanMode` API, which allows apps to change the device's Bluetooth scan mode [40]. We find that third-party apps can invoke this API without requiring user consent. An app can stealthily change the scan mode to connectable and discoverable, such that nearby devices can obtain the target device's Bluetooth MAC address for tracking or connecting to the device. Alternatively, an app can constantly set the scan mode to `SCAN_MODE_NONE`, namely non-discoverable by any device, to perform a DoS attack on the Bluetooth compo-

Table 5: Common Events in OpenHarmony that can be exploited by third-party apps to obtain sensitive information without any permission, violating the permission control model.

OpenHarmony Common Event	Obtained Sensitive Information	Normal System API	API Permission
`wifi.CONN_STATE`	Wi-Fi connection state changes	wifiManager.on('wifiStateChange')	GET_WIFI_INFO
`wifi.HOTSPOT_STATE`	Wi-Fi hotspot state changes	wifiManager.on('hotspotStateChange')	GET_WIFI_INFO
`PACKAGE_ADDED`	If a new app is installed, including sensitive information such as app's uid and accessTokenId	bundleManager.getBundleInfoForSelf can only get self uid	N/A
`NETWORK_STATE_CHANGED`	Detailed information of current network	radio.getNetworkState	GET_NETWORK_INFO
`CELLULAR_DATA_STATE_CHANGED`	Cellular data status	radio.getNetworkState	GET_NETWORK_INFO
`CONNECTIVITY_CHANGE`	Network connection state changes	wifiManager.on('wifiStateChange')	GET_WIFI_INFO

nent by continuously preventing the host device from being discoverable to other devices.

We also found that the Bluetooth component accepts pairing and connection requests from external devices. Specifically, when we set the capability of the external Bluetooth device to `NoInputNoOutput`, it pairs and connects to the OpenHarmony device, and no user consent is required. Afterwards, our device successfully redirects and intercepts the audio of the OpenHarmony device without user consent.

DLV#29–#31, DLV#33. On OpenHarmony, accessing the router's MAC address via `wifiManager.getLinkedInfo` requires the `GET_WIFI_INFO` permission [151], and obtaining the device's MAC address is restricted to system apps [112]. However, third-party apps on OpenHarmony can obtain MAC addresses in different ways without any permissions: Apps can read `/proc/net/arp` to get the router MAC address and `/sys/class/net/wlan0/address` to get the MAC address of the device, both of which violate the permission control model in OpenHarmony. Similar to Android, OpenHarmony restricts the visibility of `/proc/<pid>` entries, ensuring that an app can only access its own PID directory. However, we verified that apps can read `/proc/net/tcp` and `/proc/net/netstat` to obtain the TCP packet counters, which may result in TCP injection and hijacking attacks [145]. In addition, OpenHarmony apps can access `/sys/class/power_supply/`, introducing the risk of inferring a user's location through power consumption analysis [123]. To access these system files, we created a native C++ NAPI module and registered a function that ArkTS can invoke in a third-party OpenHarmony app. The function uses `popen()` to execute shell commands to read a file.

DLV#34. To identify potential unprotected components that allow apps to trigger system crashes or device reboots, we exercise relevant system files and APIs in OpenHarmony. We find that apps can read files in the DebugFS partition. We used NAPI to execute shell commands and access files in `/sys/kernel/debug`. An app with "normal" privilege declaring no permissions can still read `/sys/kernel/debug/usb/musb-hdrc.2.auto/regdump`, which panics the kernel and thus forces a reboot.

CWE-221: Information Loss or Omission

DLV#42. OpenHarmony also permits all system apps to launch UI Abilities from the background using the `START_ABILITIES_FROM_BACKGROUND` permission [160]. Third-party apps can also gain this permission via the Access Control List, which enables requests for high-privilege access [6]. This poses a potential UI hijacking risk.

CWE-1038: Insecure Automated Optimizations

DLV#45. OpenHarmony adopts appspawn, an app process creation model similar to Zygote. All but some critical processes (e.g., media_service) on OpenHarmony spawn from the same parent and share the same memory layout, weakening Address Space Layout Randomization (ASLR) [100, 205].

CWE-664: Improper Control of a Resource Through its Lifetime

DLV#49. OpenHarmony restricts third-party apps to accessing only their own logs. However, system apps (both system_basic and system_core apps) can access the global logs using the `@ohos.logLibrary` APIs [115]. A system app can invoke the `logLibrary.list` API to list all global log files in `/data/log/hilog` and the `logLibrary.copy` API to copy them to its own sandboxed directories. This poses privacy risks, as system apps can silently collect and transmit log data [116].

DLV#54. OpenHarmony apps without the `INTERNET` permission can covertly send data to remote servers by launching a browser with the `INTERNET` permission using a Want, creating an app-to-server covert channel [172].

CWE-311: Missing Encryption of Sensitive Data

DLV#56. OpenHarmony allows cleartext HTTP communications by default. Apps do not need any special permissions or declarations to send HTTP requests using `@ohos.net.http` [87]. This leads to various security risks, such as MITM attacks, due to the lack of encryption or integrity checks of the cleartext network traffic [134, 138].

Key insight #2: By analyzing the 46 applicable DLVs, we find that OpenHarmony is vulnerable to 24 of them. Studying historical Android DLVs is valuable since they can translate to emerging mobile OSes such as OpenHarmony, even if they are independently reimplemented—echoing the adage, "Those who don't learn from history are doomed to repeat it."

5.5 Result Analysis and Discussion

Overall, our analysis discovered 24 DLVs in OpenHarmony. More precisely, of the 46 applicable DLVs, OpenHarmony remains fully vulnerable to 17 and partially vulnerable to 7, as shown in Table 1. These vulnerabilities have serious security implications: they leak sensitive system information, enable side-channel attacks to crack credentials, allow malicious Bluetooth manipulation, bypass permission controls, and can DoS the entire system. These findings suggest that OpenHarmony fails to incorporate many well-documented Android security lessons. This is particularly concerning because these DLVs are publicly known, reducing the work for attackers to exploit them. Interestingly, OpenHarmony is not vulnerable to two DLVs where Android is vulnerable, both related to covert channels. For instance, OpenHarmony is not vulnerable to the volume-setting covert channels because it restricts third-party apps from setting volume levels.

Android mitigation time analysis. For all 46 applicable DLVs, we investigate their Android Mitigation Time (AMT)—that is, when Android mitigated the DLV—and document our best estimates in Table 1. Blank entries indicate that we could not find an estimated AMT. For this analysis, we focus on the 22 DLVs where OpenHarmony is vulnerable but Android is not. For 19 of them, we managed to identify the estimated AMT and plot the versions in Figure 1: Only one has an AMT earlier than Android 9, and only four have an AMT earlier than Android 10. This is no coincidence: Android 10 was released in September 2019 [16], while Huawei first launched HarmonyOS, the commercial version of OpenHarmony, in August 2019 [84]. Our analysis results imply that vulnerabilities discovered and mitigated in Android *after* the reimplementation are more likely to be overlooked by OpenHarmony. We speculate that Huawei reduced effort spent on their Android fork while working on the reimplementation.

HarmonyOS NEXT. We also validated the 24 DLVs affecting OpenHarmony on a commercial HarmonyOS NEXT device, and we confirmed that 20 of them work. HarmonyOS's filesystem hardening prevented DLV#31 and #33, and debugging restrictions (Huawei controls certain debugging access through vetting processes) have thus far prevented us from evaluating DLV#30 and #34.

Other emerging mobile OSes. Other mobile OS reimplementations exist, albeit at varying levels of completion. In 2023, Xiaomi announced its self-developed HyperOS [175]. Although HyperOS is still compatible with Android, Xiaomi has continuously added its proprietary components to it. Vivo also announced BlueOS, a self-developed OS written in Rust [166]. Although BlueOS has not yet been deployed on commercial mobile phones, both it and HyperOS indicate that Android vendors are accelerating the move toward mobile OS reimplementation. In this work, we apply our systematized auditing model to OpenHarmony, as it represents the first completed reimplementation of a mobile OS that we can systematically analyze. As these trends continue, the findings of this paper will become increasingly critical to the security of these emerging mobile OSes.

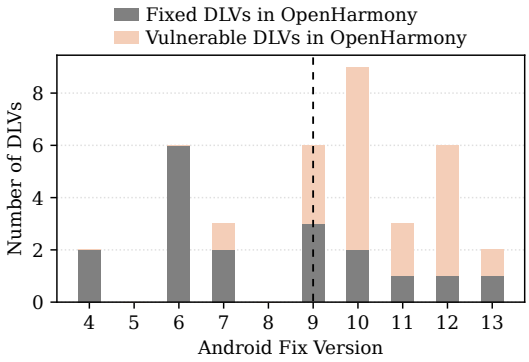

Figure 1: Stacked distribution of Android DLVs that are applicable to OpenHarmony, grouped by the Android version in which each DLV was fixed. The total bar height equals the number of applicable DLVs, partitioned into DLVs fixed in OpenHarmony (bottom) and those still vulnerable in OpenHarmony (top). The vertical dashed line is the estimated time when the OpenHarmony reimplementation started. The vast majority of DLVs that we found to affect OpenHarmony were fixed in Android *after* the start of the reimplementation project.

Limitations. Applying our auditing model to OpenHarmony required substantial manual effort. Although the model provides reusable auditing guidance, each test still required us to manually map Android concepts to OpenHarmony components, identify corresponding APIs or system resources, and implement OpenHarmony PoCs.

6 Related Work

Our paper investigates the next phenomenon after fragmentation, and uses existing literature to identify DLVs.

Android fragmentation security research. Researchers previously studied security issues caused by Android *fragmentation*, including analysis of driver customization in fragmented systems [199], bootloader vulnerabilities and security violations of Android OEM customizations [85, 137], and vulnerabilities in residual APIs of vendor-customized Android

ROMs [103]. While these works focused on customized Android OSes, our work examines whether vulnerabilities in the design translate to emerging mobile OSes using a similar design.

Literature-based security analysis. A prior study systematized research on Android security and privacy, compared the *appification* of software systems with traditional OSes and software ecosystems, and provided insights for future research [5]. Other work systematically characterized Android malware to investigate its evolution and detection techniques [200]. Our paper systematizes DLVs and evaluates whether they affect emerging mobile OSes with a similar design to Android. Prior work has proposed to study literature to uncover security vulnerabilities [204], augment cyber threat intelligence [146], and analyze malware [203]. Our paper adopts similar approaches for a systematic identification of security issues, which can be referred to by future mobile OS developers to avoid introducing DLVs.

7 Conclusion

In the context of the Android ecosystem's shift toward reimplementation, this work systematically studied historical Android Design-Level Vulnerabilities, which were then applied to OpenHarmony to examine its security. Through systematic analysis of 450 Android security publications, we identified 56 DLVs and tested each on both Android and OpenHarmony using PoC apps. Our findings reveal 24 unmitigated DLVs in OpenHarmony, demonstrating that reimplementation alone does not guarantee security improvement. Interestingly, 15 of the 19 DLVs with known mitigation dates were fixed in Android *after* OpenHarmony's reimplementation began, highlighting a critical gap in how reimplementers track evolving security knowledge. To address this systemic problem, we provide a comprehensive security checklist derived from our experience, offering concrete guidance for future mobile OS developers to avoid repeating these design flaws.

Acknowledgments

We would like to thank our anonymous shepherd and reviewers for their valuable feedback. Further, we thank Kyle Zeng and Yue Zhao for their assistance in obtaining the testing devices. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-22-C-4026, Advanced Research Projects Agency for Health (ARPA-H) under Contract No. SP4701-23-C-0074, and the National Science Foundation (NSF) under Award No. 2232915 and No. 2146568. Additionally, we acknowledge the generous support of the US Department of Defense (DOD). The authors at The Ohio State University were supported in part by NSF under Award No. 2330264. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA, NIWC Pacific, ARPA-H, NSF, or DOD.

Ethical Considerations

Stakeholder analysis and impact. We consider all possible stakeholders that may be impacted by our research.

OpenHarmony (project maintainers): Our findings may require design-level changes, security hardening, and coordinated patching across releases and downstream forks. Publicizing systemic weaknesses can increase attack pressure before fixes are widely deployed. So we promptly and responsibly disclosed all 24 identified DLVs to OpenHarmony, and we are actively collaborating with OpenHarmony to fix these vulnerabilities. We follow a standard timeline of 90 days before public disclosure.

Developers in the OpenHarmony ecosystem (such as app developers): Security recommendations can impose engineering and audit costs. However, our research will help improve the security of the OpenHarmony ecosystem. It will protect a large number of OpenHarmony apps from attacks that exploit these vulnerabilities, thereby safeguarding the interests of app developers.

OpenHarmony users: If the DLVs are disclosed without timely mitigation, users may face risks of being attacked—especially on devices that are slow to update. That said, our research will help improve the security of OpenHarmony and protect its users from attacks that exploit these vulnerabilities, and will reduce security risks, such as privacy leakage.

Attackers: Attackers could adapt our analysis into exploitation strategies. As our paper essentially builds on already publicly available information, the risk already exists. We have systematically studied all design-level vulnerabilities in Android and systematically applied them to OpenHarmony, responsibly disclosing any findings, thereby preventing attackers from exploiting them in the future.

Other considerations: This work does not involve any human subjects. Our research does not involve any experiments that could harm individuals, including the research team, directly.

Mitigations. To minimize the potential harms mentioned above, we promptly and responsibly disclosed all 24 identified design-level vulnerabilities to OpenHarmony, who has confirmed receipt of all our reports and acknowledged six vulnerabilities, while the others are still under review. So far, we have received bug bounties of 25,000 CNY. We are actively collaborating with OpenHarmony to fix these vulnerabilities. We plan to publicly release the artifacts only after OpenHarmony publishes the fixes, committing to allowing them at least 90 days (the standard timeline for bug disclosure). However, even without exploit code, publishing the

paper may help attackers prioritize targets or recognize DLVs across OpenHarmony. Thus, we completed the responsible disclosure of all 24 DLVs to OpenHarmony in June 2025, well over 90 days before submission of the paper.

After careful consideration of all the above factors, we decided to publish this work. We believe that publishing this paper provides greater value to the community, showing that emerging OSes can indeed suffer from publicly known DLVs in Android, and our research will help OpenHarmony to secure its platform and users.

Open Science

In this work, we create PoCs to verify the 56 identified DLVs in Android, and we create PoCs to verify the 46 DLVs that are applicable to OpenHarmony. All artifacts can be accessed at https://doi.org/10.5281/zenodo.20655878.

References

[1] Y. Aafer, W. You, Y. Sun, Y. Shi, X. Zhang, and H. Yin. Android SmartTVs Vulnerability Discovery via Log-Guided Fuzzing. In *USENIX Security Symposium*, 2021.

[2] Y. Aafer, N. Zhang, Z. Zhang, X. Zhang, K. Chen, X. Wang, X. Zhou, W. Du, and M. Grace. Hare Hunting in the Wild Android: A Study on the Threat of Hanging Attribute References. In *ACM Conference on Computer and Communications Security*, 2015.

[3] Y. Aafer, X. Zhang, and W. Du. Harvesting Inconsistent Security Configurations in Custom Android ROMs via Differential Analysis. In *USENIX Security Symposium*, 2016.

[4] A. Acar, G. S. Tuncay, E. Luques, H. Oz, A. Aris, and S. Uluagac. 50 Shades of Support: A Device-Centric Analysis of Android Security Updates. In *Symposium on Network and Distributed System Security*, 2024.

[5] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith. SoK: Lessons Learned from Android Security Research for Appified Software Platforms. In *IEEE Symposium on Security and Privacy*, 2016.

[6] Requesting Restricted Permissions. https://docs.openharmony.cn/pages/v5.1.0/en/application-dev/security/AccessToken/declare-permissions-in-acl.md.

[7] Verify App Links. https://developer.android.com/training/app-links/verify-applinks.

[8] Autofill Framework. https://developer.android.com/identity/autofill.

[9] Android Security and Update Bulletins. https://source.android.com/docs/security/bulletin.

[10] About Deep Links. https://developer.android.com/training/app-links.

[11] A Complete Guide To Android Fragmentation & How to Deal With It. https://testlio.com/blog/what-is-android-fragmentation/.

[12] Intents and Intent Filters | Android Developers. https://developer.android.com/guide/components/intents-filters.

[13] iPhone vs Android Statistics. https://backlinko.com/iphone-vs-android-statistics.

[14] ViewOverlay. https://developer.android.com/reference/android/view/ViewOverlay.

[15] Pending Intents. https://developer.android.com/privacy-and-security/risks/pending-intent.

[16] Android OS. https://endoflife.date/android.

[17] S. Aonzo, A. Merlo, G. Tavella, and Y. Fratantonio. Phishing Attacks on Modern Android. In *ACM Conference on Computer and Communications Security*, 2018.

[18] I. Arkalakis, M. Diamantaris, S. Moustakas, S. Ioannidis, J. Polakis, and P. Ilia. Abandon All Hope Ye Who Enter Here: A Dynamic, Longitudinal Investigation of Android's Data Safety Section. In *USENIX Security Symposium*, 2024.

[19] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Symposium on Network and Distributed System Security*, 2014.

[20] Auto-reset Permissions from Unused Apps. https://developer.android.com/about/versions/11/privacy/permissions#auto-reset.

[21] Foreground Service Types | Background Work | Android Developers. https://developer.android.com/develop/background-work/services/fgs/service-types#microphone.

[22] Restrictions on Starting Activities from the Background. https://developer.android.com/guide/components/activities/background-starts.

[23] P. Beer, M. Squarcina, L. Veronese, and M. Lindorfer. Tabbed Out: Subverting the Android Custom Tab Security Model. In *IEEE Symposium on Security and Privacy*, 2024.

[24] Stagefright: Scary Code in the Heart of Android. https://www.blackhat.com/us-15/briefings.html#stagefright-scary-code-in-the-heart-of-android.

[25] AutoSpill: Zero Effort Credential Stealing from Mobile Password Managers. https://www.blackhat.com/eu-23/briefings/schedule/#autospill-zero-effort-credential-stealing-from-mobile-password-managers-34420.

[26] Attacking Debug Modules In The Android Ecosystem. https://www.blackhat.com/asia-24/briefings/schedule/#attacking-debug-modules-in-the-android-ecosystem-36451.

[27] SystemUI As EvilPiP: The Hijacking Attacks on Modern Mobile Devices. https://www.blackhat.com/asia-24/briefings/schedule/#systemui-as-evilpip-the-hijacking-attacks-on-modern-mobile-devices-36260.

[28] Dirty Stream Attack, Turning Android Share Targets Into Attack Vectors. https://www.blackhat.com/asia-23/briefings/schedule/#dirty-stream-attack-turning-android-share-targets-into-attack-vectors-30234.

[29] Deep into Android Bluetooth Bug Hunting: New Attack Surfaces and Weak Code Patterns. https://www.blackhat.com/eu-22/briefings/schedule/#deep-into-android-bluetooth-bug-hunting-new-attack-surfaces-and-weak-code-patterns-28496.

[30] ExplosION: The Hidden Mines in the Android ION Driver. https://www.blackhat.com/asia-22/briefings/schedule/#explosion-the-hidden-mines-in-the-android-ion-driver-25848.

[31] Stealthily Access Your Android Phones: Bypass the Bluetooth Authentication. https://www.blackhat.com/us-20/briefings/schedule/#stealthily-access-your-android-phones-bypass-the-bluetooth-authentication-19993.

[32] Re-route Your Intent for Privilege Escalation: A Universal Way to Exploit Android PendingIntents in High-profile and System Apps. https://www.blackhat.com/eu-21/briefings/schedule/#re-route-your-intent-for-privilege-escalation-a-universal-way-to-exploit-android-pendingintents-in-high-profile-and-system-apps-24340.

[33] (Un)protected Broadcasts in Android 9 and 10. https://www.blackhat.com/asia-21/briefings/schedule/#unprotected-broadcasts-in-android--and--22378.

[34] Hey Google, Activate Spyware! When Google Assistant Uses a Vulnerability as a Feature. https://www.blackhat.com/asia-20/briefings/schedule/#hey-google-activate-spyware--when-google-assistant-uses-a-vulnerability-as-a-feature-18486.

[35] Securing the System: A Deep Dive into Reversing Android Pre-Installed Apps. https://www.blackhat.com/us-19/briefings/schedule/#securing-the-system-a-deep-dive-into-reversing-android-pre-installed-apps-16040.

[36] Simple Spyware: Android's Invisible Foreground Services and How to (Ab)use Them. https://www.blackhat.com/eu-19/briefings/schedule/#simple-spyware-androids-invisible-foreground-services-and-how-to-abuse-them-17738.

[37] Cloak & Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. https://www.blackhat.com/us-17/briefings.html#cloak-and-dagger-from-two-permissions-to-complete-control-of-the-ui-feedback-loop.

[38] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the App is That? Deception and Countermeasures in the Android User Interface. In *IEEE Symposium on Security and Privacy*, 2015.

[39] @ohos.bluetooth.ble (Bluetooth BLE Module). https://docs.openharmony.cn/pages/v5.1.0/en/application-dev/reference/apis-connectivity-kit/js-apis-bluetooth-ble.md.

[40] @ohos.bluetooth.connection (Bluetooth Connection Module). https://docs.openharmony.cn/pages/v5.1.0/en/application-dev/reference/apis-connectivity-kit/js-apis-bluetooth-connection.md.

[41] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *Symposium on Network and Distributed System Security*, 2012.

[42] W. Cao, C. Xia, S. T. Peddinti, D. Lie, N. Taft, and L. M. Austin. A Large Scale Study of User Behavior, Expectations and Engagement with Android Permissions. In *USENIX Security Symposium*, 2021.

[43] @ohos.net.http (Data Request). https://docs.openharmony.cn/pages/v5.1.0/en/application-dev/reference/apis-network-kit/js-apis-http.md.

[44] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your App without Actually Seeing it: UI State Inference and Novel Android Attacks. In *USENIX Security Symposium*, 2014.

[45] Introduction to Common Events. https://docs.openharmony.cn/pages/v5.1.0/en/application-dev/basic-services/common-event/common-event-overview.md.

[46] Content Providers. https://developer.android.com/guide/topics/providers/content-providers.

[47] NVD - CVE-2019-2225. https://nvd.nist.gov/vuln/detail/CVE-2019-2225.

[48] NVD - CVE-2022-20126. https://nvd.nist.gov/vuln/detail/CVE-2022-20126.

[49] CWE - About CWE. https://cwe.mitre.org/about/index.html.

[50] CWE - CWE-1038: Insecure Automated Optimizations (4.19.1). https://cwe.mitre.org/data/definitions/1038.html.

[51] CWE - CWE-1256: Improper Restriction of Software Interfaces to Hardware Features (4.19.1). https://cwe.mitre.org/data/definitions/1256.html.

[52] CWE - CWE-221: Information Loss or Omission (4.19.1). https://cwe.mitre.org/data/definitions/221.html.

[53] CWE - CWE-269: Improper Privilege Management (4.19.1). https://cwe.mitre.org/data/definitions/269.html.

[54] CWE - CWE-284: Improper Access Control (4.19.1). https://cwe.mitre.org/data/definitions/284.html.

[55] CWE - CWE-295: Improper Certificate Validation (4.19.1). https://cwe.mitre.org/data/definitions/295.html.

[56] CWE - CWE-300: Channel Accessible by Non-Endpoint (4.19.1). https://cwe.mitre.org/data/definitions/300.html.

[57] CWE - CWE-311: Missing Encryption of Sensitive Data (4.19.1). https://cwe.mitre.org/data/definitions/311.html.

[58] CWE - CWE-319: Cleartext Transmission of Sensitive Information (4.19.1). https://cwe.mitre.org/data/definitions/319.html.

[59] CWE - CWE-346: Origin Validation Error (4.19.1). https://cwe.mitre.org/data/definitions/346.html.

[60] CWE - CWE-451: User Interface (UI) Misrepresentation of Critical Information (4.19.1). https://cwe.mitre.org/data/definitions/451.html.

[61] CWE - CWE-514: Covert Channel (4.19.1). https://cwe.mitre.org/data/definitions/514.html.

[62] CWE - CWE-552: Files or Directories Accessible to External Parties (4.19.1). https://cwe.mitre.org/data/definitions/552.html.

[63] CWE - CWE-664: Improper Control of a Resource Through its Lifetime (4.19.1). https://cwe.mitre.org/data/definitions/664.html.

[64] CWE - CWE-749: Exposed Dangerous Method or Function (4.19.1). https://cwe.mitre.org/data/definitions/749.html.

[65] CWE - CWE-921: Storage of Sensitive Data in a Mechanism without Access Control (4.19.1). https://cwe.mitre.org/data/definitions/921.html.

[66] CWE - CWE-923: Improper Restriction of Communication Channel to Intended Endpoints (4.19.1). https://cwe.mitre.org/data/definitions/923.html.

[67] @ohos.deviceInfo (Device Information). https://docs.openharmony.cn/pages/v5.1.0/en/application-dev/reference/apis-basic-services-kit/js-apis-device-info.md.

[68] M. Diamantaris, S. Moustakas, L. Sun, S. Ioannidis, and J. Polakis. This Sneaky Piggy Went to the Android Ad Market: Misusing Mobile Sensors for Stealthy Data Exfiltration. In *ACM Conference on Computer and Communications Security*, 2021.

[69] W. Diao, X. Liu, Z. Li, and K. Zhang. No Pardon for the Interruption: New Inference Attacks on Android Through Interrupt Timing Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[70] core/java/android/bluetooth/BluetoothDevice.java - platform/frameworks/base - Git at Google. https://android.googlesource.com/platform/frameworks/base/+/b1dc1757071ba46ee653d68f331486e86778b8e4/core/java/android/bluetooth/BluetoothDevice.java#948.

[71] Z. Dong, T. Liu, J. Deng, H. Wang, L. Li, M. Yang, M. Wang, G. Xu, and G. Xu. Exploring Covert Third-party Identifiers through External Storage in the Android New Era. In *USENIX Security Symposium*, 2024.

[72] S. Du, X. Liu, G. Lai, and X. Luo. Watch Out for Race Condition Attacks When Using Android External Storage. In *ACM Conference on Computer and Communications Security*, 2022.

[73] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *ACM Conference on Computer and Communications Security*, 2013.

[74] Z. El-Rewini and Y. Aafer. Dissecting Residual APIs in Custom Android ROMs. In *ACM Conference on Computer and Communications Security*, 2021.

[75] M. Elsabagh, R. Johnson, A. Stavrou, C. Zuo, Q. Zhao, and Z. Lin. FIRMSCOPE: Automatic Uncovering of Privilege-Escalation Vulnerabilities in Pre-Installed Apps in Android Firmware. In *USENIX Security Symposium*, 2020.

[76] W. Enck, D. Octeau, P. D. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *USENIX Security Symposium*, 2011.

[77] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security. In *ACM Conference on Computer and Communications Security*, 2012.

[78] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee. Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. In *IEEE Symposium on Security and Privacy*, 2017.

[79] I. Gasparis, Z. Qian, C. Song, and S. V. Krishnamurthy. Detecting Android Root Exploits by Learning from Root Providers. In *USENIX Security Symposium*, 2017.

[80] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Symposium on Network and Distributed System Security*, 2012.

[81] OpenHarmony OS Continues to Advance in China. https://www.taipeitimes.com/News/biz/archives/2024/06/29/2003820035.

[82] HarmonyOS NEXT Comes without a Single Line of Android Code. https://gizchina.net/en/2024/01/19/harmonyos-next-predstavleno-bez-zhodnoho-ryadka-kodu-android/.

[83] HarmonyOS NEXT Becomes the World's Third Major Mobile OS, Joining Android and iOS. https://technode.com/2024/10/23/harmonyos-next-becomes-the-worlds-third-major-mobile-os-joining-android-and-ios/.

[84] HarmonyOS. https://www.harmonyos.com/en/.

[85] R. Hay. fastboot oem vuln: Android Bootloader Vulnerabilities in Vendor Customizations. In *USENIX Workshop on Offensive Technologies*, 2017.

[86] Cleartext Communications | Security | Android Developers. https://developer.android.com/privacy-and-security/risks/cleartext-communications.

[87] @ohos.net.http (Data Request)-ArkTS APIs-Network Kit-Network-System - HUAWEI Developers. https://developer.huawei.com/consumer/en/doc/harmonyos-references-V5/js-apis-http-V5.

[88] H. Huang, S. Zhu, K. Chen, and P. Liu. From System Services Freezing to System Server Shutdown in Android: All You Need Is a Loop in an App. In *ACM Conference on Computer and Communications Security*, 2015.

[89] J. Huang, M. Backes, and S. Bugiel. A11y and Privacy Don't Have to be Mutually Exclusive: Constraining Accessibility Service Misuse on Android. In *USENIX Security Symposium*, 2021.

[90] What are the Differences between HarmonyOS and Android? What Scenarios are They Each Suitable for? https://developer.huawei.com/consumer/cn/blog/topic/03207049550171162.

[91] Privacy Indicators. https://source.android.com/docs/core/permissions/privacy-indicators.

[92] Implicit Intent Hijacking | App Quality | Android Developers. https://developer.android.com/privacy-and-security/risks/implicit-intent-hijacking.

[93] S. Jana and V. Shmatikov. Memento: Learning Secrets from Process Footprints. In *IEEE Symposium on Security and Privacy*, 2012.

[94] Y. Jang, C. Song, S. P. Chung, T. Wang, and W. Lee. A11y Attacks: Exploiting Accessibility in Operating Systems. In *ACM Conference on Computer and Communications Security*, 2014.

[95] Y. Ji, M. Elsabagh, R. Johnson, and A. Stavrou. DEFInit: An Analysis of Exposed Android Init Routines. In *USENIX Security Symposium*, 2021.

[96] JavascriptInterface. https://developer.android.com/reference/android/webkit/JavascriptInterface.

[97] Web. https://docs.openharmony.cn/pages/v5.1.0/en/application-dev/reference/apis-arkweb/ts-basic-components-web.md.

[98] JavaScriptSandbox. https://developer.android.com/reference/androidx/javascriptengine/JavaScriptSandbox.

[99] B. Kondracki, B. A. Azad, N. Miramirkhani, and N. Nikiforakis. The Droid is in the Details: Environment-aware Evasion of Android Sandboxes. In *Symposium on Network and Distributed System Security*, 2022.

[100] B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee. From Zygote to Morula: Fortifying Weakened ASLR on Android. In *IEEE Symposium on Security and Privacy*, 2014.

[101] J. Lee and D. S. Wallach. Removing Secrets from Android's TLS. In *Symposium on Network and Distributed System Security*, 2018.

[102] L. Lei, Y. He, K. Sun, J. Jing, Y. Wang, Q. Li, and J. Weng. Vulnerable Implicit Service: A Revisit. In *ACM Conference on Computer and Communications Security*, 2017.

[103] Z. Lei, G. S. Tuncay, B. C. Williem, Z. B. Celik, and A. Bianchi. ScopeVerif: Analyzing the Security of Android's Scoped Storage via Differential Analysis. In *Symposium on Network and Distributed System Security*, 2025.

[104] C. Lenk and J. Kinder. Poster: Privacy Risks from Misconfigured Android Content Providers. In *ACM Conference on Computer and Communications Security*, 2023.

[105] J. Li, H. Zhou, S. Wu, X. Luo, T. Wang, X. Zhan, and X. Ma. FOAP: Fine-Grained Open-World Android App Fingerprinting. In *USENIX Security Symposium*, 2022.

[106] L. Li, X. Gao, H. Sun, C. Hu, C. 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. *ACM Computing Surveys*, 58:34:1–34:36, 2026.

[107] M. Li, X. Chen, X. Li, B. Ma, and P. M. Vitányi. The Similarity Metric. *IEEE Transactions on Information Theory*, 50:3250–3264, 2004.

[108] R. Li, W. Diao, S. Yang, X. Liu, S. Guo, and K. Zhang. Lost in Conversion: Exploit Data Structure Conversion with Attribute Loss to Break Android Systems. In *USENIX Security Symposium*, 2023.

[109] T. Li, X. Wang, M. Zha, K. Chen, X. Wang, L. Xing, X. Bai, N. Zhang, and X. Han. Unleashing the Walking Dead: Understanding Cross-App Remote Infections on Mobile WebViews. In *ACM Conference on Computer and Communications Security*, 2017.

[110] Y. Lin, J. Wong, X. Li, H. Ma, and D. Gao. Peep With A Mirror: Breaking The Integrity of Android App Sandboxing via Unprivileged Cache Side Channel. In *USENIX Security Symposium*, 2024.

[111] F. Liu, C. Wang, A. Pico, D. Yao, and G. Wang. Measuring the Insecurity of Mobile Deep Links of Android. In *USENIX Security Symposium*, 2017.

[112] @ohos.wifiManager (WLAN) (System API). https://docs.openharmony.cn/pages/v5.1.0/en/application-dev/reference/apis-connectivity-kit/js-apis-wifiManager-sys.md#wifimanagergetdevicemacaddress9.

[113] Log Info Disclosure. https://developer.android.com/privacy-and-security/risks/log-info-disclosure.

[114] Understand Logging | Android Open Source Project. https://source.android.com/docs/core/tests/debug/understanding-logging#device-logs.

[115] @ohos.logLibrary (Log Library) (System API). https://docs.openharmony.cn/pages/v5.1.0/en/application-dev/reference/apis-performance-analysis-kit/js-apis-loglibrary-sys.md.

[116] A. Lyons, J. Gamba, A. Shawaga, J. Reardon, J. Tapiador, S. Egelman, and N. Vallina-Rodríguez. Log: It's Big, It's Heavy, It's Filled with Personal Data! Measuring the Logging of Sensitive Information in the Android Ecosystem. In *USENIX Security Symposium*, 2023.

[117] L. Maar, F. Draschbacher, L. Lamster, and S. Mangard. Defects-in-Depth: Analyzing the Integration of Effective Defenses against One-Day Exploits in Android Kernels. In *USENIX Security Symposium*, 2024.

[118] P. Marquardt, A. Verma, H. Carter, and P. Traynor. (sp)iPhone: Decoding Vibrations From Nearby Keyboards Using Mobile Phone Accelerometers. In *ACM Conference on Computer and Communications Security*, 2011.

[119] R. Mayrhofer, J. V. Stoep, C. Brubaker, and N. Kralevich. The Android Platform Security Model. *ACM Transactions on Privacy and Security*, 24:19:1–19:35, 2021.

[120] M. Mehrnezhad, E. Toreini, S. F. Shahandashti, and F. Hao. Stealing PINs via Mobile Sensors: Actual Risk versus User Perception. *International Journal of Information Security*, 17:291–313, 2018.

[121] M. H. Meng, Q. Zhang, G. Xia, Y. Zheng, Y. Zhang, G. Bai, Z. Liu, S. G. Teo, and J. S. Dong. Post-GDPR Threat Hunting on Android Phones: Dissecting OS-level Safeguards of User-unresettable Identifiers. In *Symposium on Network and Distributed System Security*, 2023.

[122] Y. Michalevsky, D. Boneh, and G. Nakibly. Gyrophone: Recognizing Speech from Gyroscope Signals. In *USENIX Security Symposium*, 2014.

[123] Y. Michalevsky, A. Schulman, G. A. Veerapandian, D. Boneh, and G. Nakibly. PowerSpy: Location Tracking Using Mobile Device Power Analysis. In *USENIX Security Symposium*, 2015.

[124] @ohos.multimedia.audio (Audio Management). https://docs.openharmony.cn/pages/v5.1.0/en/application-dev/reference/apis-audio-kit/js-apis-audio.md.

[125] Behavior Changes: Apps Targeting Android 11 | Android Developers. https://developer.android.com/about/versions/11/behavior-changes-11.

[126] Your Smartphone is a Science Lab: 50 Physics Measures you can do with your Mobile. https://www.fizziq.org/en/post/turn-your-smartphone-into-a-scientific-instruments.

[127] P. Mutchler, Y. Safaei, A. Doupé, and J. Mitchell. Target Fragmentation in Android Apps. In *Workshop on Mobile Security Technologies*, 2016.

[128] S. Narain, T. D. Vo-Huu, K. Block, and G. Noubir. Inferring User Routes and Locations using Zero-Permission Mobile Sensors. In *IEEE Symposium on Security and Privacy*, 2016.

[129] M. Naveed, X.-y. Zhou, S. Demetriou, X. Wang, and C. A. Gunter. Inside Job: Understanding and Mitigating the Threat of External Device Mis-Binding on Android. In *Symposium on Network and Distributed System Security*, 2014.

[130] T. T. Nguyen, M. Backes, and B. Stock. Freely Given Consent? Studying Consent Notice of Third-Party Tracking and Its Violations of GDPR in Android Apps. In *ACM Conference on Computer and Communications Security*, 2022.

[131] Network Security Configuration. https://developer.android.com/privacy-and-security/security-config.

[132] @ohos.net.statistics (Traffic Management). https://docs.openharmony.cn/pages/v5.1.0/en/application-dev/reference/apis-network-kit/js-apis-net-statistics.md.

[133] Want Overview. https://gitee.com/openharmony/docs/blob/master/en/application-dev/application-models/want-overview.md.

[134] M. Oltrogge, N. Huaman, S. Klivan, Y. Acar, M. Backes, and S. Fahl. Why Eve and Mallory Still Love Android: Revisiting TLS (In)Security in Android Applications. In *USENIX Security Symposium*, 2021.

[135] One-time Permissions. https://developer.android.com/about/versions/11/privacy/permissions#one-time.

[136] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Symposium on Network and Distributed System Security*, 2014.

[137] A. Possemato, S. Aonzo, D. Balzarotti, and Y. Fratantonio. Trust, but Verify: A Longitudinal Analysis of Android OEM Compliance and Customization. In *IEEE Symposium on Security and Privacy*, 2021.

[138] A. Possemato and Y. Fratantonio. Towards HTTPS Everywhere on Android: We Are Not There Yet. In *USENIX Security Symposium*, 2020.

[139] A. Possemato, A. Lanzi, S. P. H. Chung, W. Lee, and Y. Fratantonio. ClickShield: Are You Hiding Something? Towards Eradicating Clickjacking on Android. In *ACM Conference on Computer and Communications Security*, 2018.

[140] A. Possemato, D. Nisi, and Y. Fratantonio. Preventing and Detecting State Inference Attacks on Android. In *Symposium on Network and Distributed System Security*, 2021.

[141] S. Pourali, N. Samarasinghe, and M. Mannan. Hidden in Plain Sight: Exploring Encrypted Channels in Android Apps. In *ACM Conference on Computer and Communications Security*, 2022.

[142] S. Pourali, X. Yu, L. Zhao, M. Mannan, and A. Youssef. Racing for TLS Certificate Validation: A Hijacker's Guide to the Android TLS Galaxy. In *USENIX Security Symposium*, 2024.

[143] Privacy Checklist. https://developer.android.com/privacy-and-security/about.

[144] Restriction on Access to /proc/net Filesystem. https://developer.android.com/about/versions/10/privacy/changes#proc-net-filesystem.

[145] Z. Qian, Z. M. Mao, and Y. Xie. Collaborative TCP Sequence Number Inference Attack – How to Crack Sequence Number Under A Second. In *ACM Conference on Computer and Communications Security*, 2012.

[146] M. R. Rahman, R. Mahdavi-Hezaveh, and L. Williams. A Literature Review on Mining Cyberthreat Intelligence from Unstructured Texts. In *International Conference on Data Mining Workshops*, 2020.

[147] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques. In *Symposium on Network and Distributed System Security*, 2016.

[148] J. Reardon, Á. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez, and S. Egelman. 50 Ways to Leak your Data: An Exploration of Apps' Circumvention of the Android Permissions System. In *USENIX Security Symposium*, 2019.

[149] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu. Towards Discovering and Understanding Task Hijacking in Android. In *USENIX Security Symposium*, 2015.

[150] D. Reynaud, D. X. Song, T. R. Magrino, E. X. Wu, and E. C. R. Shin. FreeMarket: Shopping for Free in Android Applications. In *Symposium on Network and Distributed System Security*, 2012.

[151] @ohos.wifiManager (WLAN). https://docs.openharmony.cn/pages/v5.1.0/en/application-dev/reference/apis-connectivity-kit/js-apis-wifiManager.md#wifimanagergetlinkedinfo9.

[152] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *Symposium on Network and Distributed System Security*, 2011.

[153] Scoped Storage. https://source.android.com/docs/core/storage/scoped.

[154] Sensor Rate-Limiting. https://developer.android.com/develop/sensors-and-location/sensors/sensors_overview#sensors-rate-limiting.

[155] @ohos.sensor (Sensor). https://docs.openharmony.cn/pages/v5.1.0/en/application-dev/reference/apis-sensor-service-kit/js-apis-sensor.md.

[156] Y. Shao, Q. A. Chen, Z. M. Mao, J. Ott, and Z. Qian. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In *Symposium on Network and Distributed System Security*, 2016.

[157] Y. Shao, J. Ott, Y. J. Jia, Z. Qian, and Z. M. Mao. The Misuse of Android Unix Domain Sockets and Security Implications. In *ACM Conference on Computer and Communications Security*, 2016.

[158] S. Son, D. Kim, and V. Shmatikov. What Mobile Ads Know About Mobile Users. In *Symposium on Network and Distributed System Security*, 2016.

[159] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan. SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *Symposium on Network and Distributed System Security*, 2014.

[160] Permissions Available for System Applications via ACL. https://docs.openharmony.cn/pages/v5.1.0/en/application-dev/security/AccessToken/permissions-for-system-apps.md.

[161] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Symposium on Network and Distributed System Security*, 2015.

[162] D. J. Tian, G. Hernandez, J. I. Choi, V. Frost, C. Raules, P. Traynor, H. Vijayakumar, L. Harrison, A. Rahmati, M. Grace, and K. R. B. Butler. ATtention Spanned: Comprehensive Vulnerability Analysis of AT Commands Within the Android Ecosystem. In *USENIX Security Symposium*, 2018.

[163] G. S. Tuncay, J. Qian, and C. A. Gunter. See No Evil: Phishing for Permissions with False Transparency. In *USENIX Security Symposium*, 2020.

[164] BluetoothAdapter. https://developer.android.com/reference/android/bluetooth/BluetoothAdapter#disable().

[165] @ohos.bluetooth.access (Bluetooth Access Module). https://docs.openharmony.cn/pages/v5.1.0/en/application-dev/reference/apis-connectivity-kit/js-apis-bluetooth-access.md.

[166] After Huawei and Xiaomi, Vivo Announced BlueOS its Own Mobile Operating System. https://www.huaweicentral.com/after-huawei-and-xiaomi-vivo-announced-blueos-its-own-mobile-operating-system/.

[167] D. Wang, S. Dai, Y. Ding, T. Li, and X. Han. Poster: AdHoneyDroid–Capture Malicious Android Advertisements. In *ACM Conference on Computer and Communications Security*, 2014.

[168] K. Wang, Y. Zhang, and P. Liu. Call Me Back! Attacks on System Server and System Apps in Android through Synchronous Callback. In *ACM Conference on Computer and Communications Security*, 2016.

[169] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized Origin Crossing on Mobile Platforms: Threats and Mitigation. In *ACM Conference on Computer and Communications Security*, 2013.

[170] X. Wang, S. Shi, Y. Chen, and W. C. Lau. PHYjacking: Physical Input Hijacking for Zero-Permission Authorization Attacks on Android. In *Symposium on Network and Distributed System Security*, 2022.

[171] X. Wang, Y. Zhang, X. Wang, Y. Jia, and L. Xing. Union under Duress: Understanding Hazards of Duplicate Resource Mismediation in Android Software Supply Chain. In *USENIX Security Symposium*, 2023.

[172] J. Wu, Y. Wu, M. Yang, Z. Wu, T. Luo, and Y. Wang. Poster: biTheft: Stealing your Secrets by Bidirectional Covert Channel Communication with Zero-permission Android Application. In *ACM Conference on Computer and Communications Security*, 2015.

[173] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The Impact of Vendor Customizations on Android Security. In *ACM Conference on Computer and Communications Security*, 2013.

[174] X. Xiang, R. Zhang, H. Wen, X. Gong, and B. Liu. Ghost in the Binder: Binder Transaction Redirection Attacks in Android System Services. In *ACM Conference on Computer and Communications Security*, 2021.

[175] Xiaomi Developing Own Operating System Compatible with Android, Aims to Compete with Huawei's HarmonyOS and Eventually Google. https://www.gizmochina.com/2023/08/23/xiaomi-developing-smartphone-operating-system/.

[176] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang. Upgrading Your Android, Elevating My Malware: Privilege Escalation Through Mobile OS Updating. In *IEEE Symposium on Security and Privacy*, 2014.

[177] F. Xu, W. Diao, Z. Li, J. Chen, and K. Zhang. Badbluetooth: Breaking Android Security Mechanisms via Malicious Bluetooth Peripherals. In *Symposium on Network and Distributed System Security*, 2019.

[178] H. Xu, M. Yao, R. Zhang, M. M. Dawoud, J. Park, and B. Saltaformaggio. DVa: Extracting Victims and Abuse Vectors from Android Accessibility Malware. In *USENIX Security Symposium*, 2024.

[179] Z. Xu, Y. Zhang, L. Zheng, L. Xia, C. Bao, Z. Wang, and Y. Liu. Automatic Hot Patch Generation for Android Kernels. In *USENIX Security Symposium*, 2020.

[180] C. Yang, V. Yegneswaran, P. A. Porras, and G. Gu. Poster: Detecting Money-Stealing Apps in Alternative Android Markets. In *ACM Conference on Computer and Communications Security*, 2012.

[181] G. Yang, J. Huang, and G. Gu. Iframes/Popups Are Dangerous in Mobile WebView: Studying and Mitigating Differential Context Vulnerabilities. In *USENIX Security Symposium*, 2019.

[182] R. Yang, J. Cai, and X. Han. Poster: TaintGrep: A Static Analysis Tool for Detecting Vulnerabilities of Android Apps Supporting User-defined Rules. In *ACM Conference on Computer and Communications Security*, 2022.

[183] Y. Yang, M. Elsabagh, C. Zuo, R. Johnson, A. Stavrou, and Z. Lin. Detecting and Measuring Misconfigured Manifests in Android Apps. In *ACM Conference on Computer and Communications Security*, 2022.

[184] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In *ACM Conference on Computer and Communications Security*, 2013.

[185] G. Ye, Z. Tang, D. Fang, X. Chen, K. I. Kim, B. Taylor, and Z. Wang. Cracking Android Pattern Lock in Five Attempts. In *Symposium on Network and Distributed System Security*, 2017.

[186] C. Yue, C. Zhong, K. Chen, Z. Zhang, and Y. Lee. DARKFLEECE: Probing the Dark Side of Android Subscription Apps. In *USENIX Security Symposium*, 2024.

[187] H. Zhang, D. She, and Z. Qian. Android Root and its Providers: A Double-Edged Sword. In *ACM Conference on Computer and Communications Security*, 2015.

[188] H. Zhang, D. She, and Z. Qian. Android ION Hazard: The Curse of Customizable Memory Management System. In *ACM Conference on Computer and Communications Security*, 2016.

[189] L. Zhang, K. Lian, H. Xiao, Z. Zhang, P. Liu, Y. Zhang, M. Yang, and H. Duan. Exploit the Last Straw That

Breaks Android Systems. In *IEEE Symposium on Security and Privacy*, 2022.

[190] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *ACM Conference on Computer and Communications Security*, 2014.

[191] M. Zhang and H. Yin. Appsealer: Automatic Generation of Vulnerability-specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *Symposium on Network and Distributed System Security*, 2014.

[192] X. Zhang, K. Ying, Y. Aafer, Z. Qiu, and W. Du. Life after App Uninstallation: Are the Data Still Alive? Data Residue Attacks on Android. In *Symposium on Network and Distributed System Security*, 2016.

[193] X. Zhang, Y. Zhang, J. Li, Y. Hu, H. Li, and D. Gu. Embroidery: Patching Vulnerable Binary Code of Fragmentized Android Devices. In *IEEE International Conference on Software Maintenance and Evolution*, 2017.

[194] H. Zhou, X. Luo, H. Wang, and H. Cai. Uncovering Intent based Leak of Sensitive Data in Android Framework. In *ACM Conference on Computer and Communications Security*, 2022.

[195] H. Zhou, H. Wang, X. Luo, T. Chen, Y. Zhou, and T. Wang. Uncovering Cross-Context Inconsistent Access Control Enforcement in Android. In *Symposium on Network and Distributed System Security*, 2022.

[196] H. Zhou, S. Wu, C. Qian, X. Luo, H. Cai, and C. Zhang. Beyond the Surface: Uncovering the Unprotected Components of Android Against Overlay Attack. In *Symposium on Network and Distributed System Security*, 2024.

[197] M. Zhou, Q. Wang, J. Yang, Q. Li, F. Xiao, Z. Wang, and X. Chen. Patternlistener: Cracking Android Pattern Lock using Acoustic Signals. In *ACM Conference on Computer and Communications Security*, 2018.

[198] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, Location, Disease and More: Inferring Your Secrets from Android Public Resources. In *ACM Conference on Computer and Communications Security*, 2013.

[199] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *IEEE Symposium on Security and Privacy*, 2014.

[200] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *IEEE Symposium on Security and Privacy*, 2012.

[201] Y. Zhou and X. Jiang. Detecting Passive Content Leaks and Pollution in Android Applications. In *Symposium on Network and Distributed System Security*, 2013.

[202] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Symposium on Network and Distributed System Security*, 2012.

[203] Z. Zhu and T. Dumitraş. FeatureSmith: Automatically Engineering Features for Malware Detection by Mining the Security Literature. In *ACM Conference on Computer and Communications Security*, 2016.

[204] F. Zuo and J. Rhee. Vulnerability Discovery based on Source Code Patch Commit Mining: A Systematic Literature Review. *International Journal of Information Security*, 23:1513–1526, 2024.

[205] About the Zygote Processes. https://source.android.com/docs/core/runtime/zygote.

A Appendix

We summarize all 116 publications analyzed in Table 6.

Table 6: Overview of all analyzed publications across venues.

Year	Venue	Publications	Count
2022-2024	S&P	[23], [189]	32
	USENIX	[178], [18], [71], [117], [142], [110], [186], [116], [171], [108], [105]	
	CCS	[104], [194], [141], [72], [183], [130], [182]	
	NDSS	[196], [4], [99], [170], [195], [121]	
	BlackHat	[27], [26], [28], [25], [29], [30]	
2019-2021	S&P	[137]	22
	USENIX	[89], [95], [42], [1], [134], [75], [138], [148], [181], [163]	
	CCS	[174], [68], [74]	
	NDSS	[177], [140]	
	BlackHat	[33], [32], [31], [34], [36], [35]	
2016-2018	S&P	[5], [128], [69], [78]	22
	USENIX	[162], [111], [3]	
	CCS	[139], [197], [17], [157], [188], [168], [109], [102]	
	NDSS	[101], [185], [192], [156], [147], [158]	
	BlackHat	[37]	
2013-2015	S&P	[38], [176], [100], [199]	29
	USENIX	[79], [149], [44], [122], [123]	
	CCS	[172], [187], [88], [2], [167], [190], [73], [184], [173], [198], [169], [94]	
	NDSS	[161], [129], [191], [159], [19], [136], [201]	
	BlackHat	[24]	
2010-2012	S&P	[200], [93]	11
	USENIX	[76]	
	CCS	[180], [77], [118], [145]	
	NDSS	[41], [80], [202], [150]	
Total #publications analyzed			116