TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Kristjan Tamm 242783IVCM

BYPASSING ANDROID'S SANDBOX SECURITY - A DEEP DIVE INTO PRIVILEGE ESCALATION EXPLOITS

Master's thesis

Supervisor: Matthew James Sorell

PhD

Tallinn 2026

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Kristjan Tamm 242783IVCM

ANDROIDI AEDIKU TURVALISUSEST MÖÖDUMINE - SÜVAUURING ÕIGUSTE VALLUTUSE EKSPLOITIDEST

Magistritöö

Juhendaja: Matthew James Sorell

PhD

Tallinn 2026

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Kristjan Tamm

17.05.2026

Abstract

This thesis examines how attackers bypass Android's sandbox through privilege escalation and why layered platform controls do not always hold in practice. Android combines User Identifier (UID) isolation, permission enforcement, Security-Enhanced Android (SEAndroid), Binder-mediated service boundaries, and kernel-level controls, yet exploit behaviour shows that these layers do not always form a coherent security boundary.

The study addresses how privileges are escalated on Android, how effectively existing defences respond, and how selected exploit paths behave across versions and configurations. It combines a systematic literature review, comparative analysis of documented exploit cases, and controlled experimental validation.

The findings show that Android privilege escalation is a cross-layer problem rather than a single vulnerability class. Recurrent routes include permission abuse, inter-component communication misuse, hidden service interfaces, policy divergence, user-interface-mediated abuse, and kernel or driver compromise. Conducted experimental results confirmed that Android defences reduce many attacks, but their effectiveness declines when enforcement is context-insensitive, inconsistent across layers, or weakened by configuration.

The thesis contributes a cross-layer analytical framework for evaluating Android privilege boundaries, supported by a structured exploit taxonomy and version-aware empirical design. Its central conclusion is that Android's main weakness lies less in absent defences than in incomplete coordination between them, motivating stronger policy coherence, vendor-aware evaluation, and post-exploitation containment.

The thesis is in English and contains 74 pages of text, 8 chapters, 20 figures, 3 tables.

Annotatsioon

ANDROIDI AEDIKU TURVALISUSEST MÖÖDUMINE - SÜVAUURING ÕIGUSTE VALLUTUSE EKSPLOITIDEST

Käesolev magistritöö uurib, kuidas ründajad mööduvad Androidi aedikust õiguste vallutuse abil ning miks platvormi mitmekihiline kaitse praktikas alati ei toimi. Android ühendab kasutajapõhise identifikaatori põhise eralduse, pääsuõiguste jõustamise, SEAndroidi, Binderi vahendatud teenusepiirid ja tuumataseme turvameetmed, kuid eksploitide käitumine näitab, et need kihid ei moodusta alati sidusat turvapiiri.

Uurimus käsitleb, kuidas Androidis õigusi vallutatakse, kui tõhusalt olemasolevad tõrjemehhanismid sellele vastavad ning kuidas valitud ründeteed käituvad eri versioonides ja konfiguratsioonides. Meetoditeks on süstemaatiline kirjandusülevaade, dokumenteeritud eksploitijuhtumite võrdlev analüüs ja kontrollitud katseline valideerimine.

Tulemused näitavad, et Androidi õiguste vallutus on kihtideülene probleem, mitte üksik nõrkusklass. Korduvad teed hõlmavad pääsuõiguste kuritarvitust, komponentidevahelise side väärkasutust, varjatud teenuse API-sid, poliitikate lahknevust, kasutajaliidese vahendatud kuritarvitust ning tuuma või draiveri kompromiteerimist. Läbiviidud katsed kinnitasid, et Androidi kaitsed vähendavad paljude rünnete mõju, kuid nende tõhusus kahaneb, kui jõustamine ei arvesta konteksti, on kihtide vahel ebaühtlane või sõltub nõrgast konfiguratsioonist.

Töö panus on kihtideülene raamistik Androidi pääsuõiguste piiride hindamiseks koos struktureeritud eksploitide taksonoomia ja versiooniteadliku empiirilise katsekujundusega. Põhijäreldus on, et Androidi nõrkus ei seisne niivõrd kaitsete puudumises, vaid nende puudulikus kooskõlas, mis eeldab terviklikumat poliitikakooskõla, tootjapõhist hindamist ja paremat vallutusjärgset ohjeldamist.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 74 leheküljel, 8 peatükki, 20 joonist, 3 tabelit.

List of abbreviations and terms

AOSP	Android Open Source Project
API	Application Programming Interface
adb	Android Debug Bridge
DAC	Discretionary Access Control
ICC	Inter-Component Communication
IPC	Inter-Process Communication
ION	Android memory management subsystem
OEM	Original Equipment Manufacturer
RQ	Research Question
SDK	Software Development Kit
SEAndroid	Security-Enhanced Android
SELinux	Security-Enhanced Linux
SELint	SEAndroid policy analysis tool
UI	User Interface
UID	User Identifier
logcat	Android logging utility

Table of Contents

List of figures 11

List of tables 13

1 Introduction 14

1.1 Motivation and Research Problem 15

1.2 Research Questions and Scope 16

1.3 Thesis Contribution 17

1.4 Thesis Structure 18

2 Background 20

2.1 Android Security Architecture 20

2.2 Application Components, Binder and IPC 21

2.3 Privilege Escalation on Android 22

2.4 Main Exploit Surfaces 23

2.5 Relevant Defensive Mechanisms 25

3 Literature Review 27

3.1 Review Protocol 27

3.2 Overview of Selected Studies 28

3.3 Theme 1: Kernel and Driver Exploitation 29

3.4 Theme 2: Permission Abuse and Custom Permissions 30

3.5 Theme 3: ICC Abuse and Delegated Privilege Misuse 31
3.6 Theme 4: Hidden APIs and Service-Layer Inconsistencies 32
3.7 Theme 5: Vendor Divergence and Policy Inconsistency 33
3.8 Theme 6: Defence Effectiveness and Containment 34
3.9 Research Gaps 35
4 Methodology 37
4.1 Research Design Overview 37
4.2 Phase 1: Systematic Literature Review 38
4.3 Phase 2: Comparative Analysis of Documented Cases 38
4.4 Phase 3: Experimental Validation 39
4.5 Ethics, Legality and Safety 40
4.6 Validity, Reliability and Limitations 41
5 Comparative Analysis of Documented Android Privilege Escalation Cases 43
5.1 Comparative Basis of Documented Cases 43
5.2 Exploit Families in Documented Cases 44
5.3 Temporal Development of Documented Cases 46
5.4 Targets, Preconditions, and Attack Construction 47
5.5 Encounters with Security Mechanisms 48
5.6 Version and Vendor Variation in Documented Cases 49
5.7 Interim Answer to RQ1 and RQ2 50
6 Experimental Validation 52

6.1 Testbed and instrumentation 52
6.2 Case selection rationale 55
6.3 Case studies 56
6.3.1 Case study A: Custom-permission mutation experiment 56
6.3.2 Case study B: ICC / delegated privilege misuse experiment 60
6.3.3 Case study C: Hidden API helper-bypass probe 64
6.3.4 Case study D: Overlay-mediated UI abuse and containment 68
6.4 Cross-case findings 75
6.5 Interim answer to RQ3 76
6.6 Experimental limitations 77
7 Discussion 78
7.1 Integrated answer to RQ1 78
7.2 Integrated answer to RQ2 79
7.3 Integrated answer to RQ3 80
7.4 Comparison with prior literature 81
7.5 Implications 82
7.6 Limitations of the combined findings 82
8 Conclusion 84
8.1 Summary of findings 84
8.2 Answers to the research questions 85
8.3 Contributions 85

8.4 Recommendations 86

8.5 Future work..... 87

8.6 Final remarks 87

References 88

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis 93

List of figures

Figure 1. Emulator-based validation testbed with Android Studio, Android Emulator, adb and filtered logcat. 54

Figure 2. ADB command sequence for the custom-permission mutation experiment... 57

Figure 3. Baseline custom-permission state before the Guardian update..... 58

Figure 4. Post-update state after replacing GuardianV1 with GuardianV2..... 58

Figure 5. Logcat evidence for the custom-permission mutation test..... 59

Figure 6. Runtime grant of READ_CONTACTS to the victim app. 61

Figure 7. Victim app confirming local access to contacts by reading the first contact.. 61

Figure 8. Attacker direct contact query denied by Android permissions. 62

Figure 9. Vulnerable ICC result returning contact data through the victim. 63

Figure 10. Fixed ICC result after removing the reachable deputy component..... 63

Figure 11. Logcat evidence for the vulnerable and fixed ICC configurations. 64

Figure 12. Cross-version HiddenProbe logcat evidence. 66

Figure 13. HiddenProbe result screen showing successful reflective probes and a StrictMode warning. 67

Figure 14. The OverlayLab application was granted permission to display over other apps. 69

Figure 15. Overlay interference on the unprotected TargetLab screen. 70

Figure 16. Filtered TargetLab screen rejecting obscured touch events. 70

Figure 17. Hide-overlay screen with the overlay visible on Android 10. 71

Figure 18. Android 10 overlay logcat evidence for unprotected, filtered, and hide-overlay screens. 72

Figure 19. Overlay containment on Android 12 using the hide-overlay window setting. 73

Figure 20. Android 12 overlay logcat evidence for unprotected, filtered, and hide-overlay screens. 74

List of tables

Table 1. Operational criteria for classifying experimental outcomes..... 54

Table 2. Case selection rationale and represented Android privilege boundaries..... 55

Table 3. Experimental outcomes across the four case studies..... 76

1 Introduction

Android's security model is designed as a layered system of privilege separation rather than as a single control. It combines Linux UID (User Identifier) isolation, permission-mediated API (Application Programming Interface) access, Binder-mediated service boundaries, SEAndroid (Security-Enhanced Android) mandatory access control, and kernel-level enforcement to confine untrusted applications (Armando et al., 2013; Chen et al., 2017; Reshetova et al., 2017). That architecture is technically significant because each layer is intended to limit the effect of compromise in a different execution context. Its practical strength, however, depends on whether those layers remain consistent when real applications, services, and vendor modifications interact.

This thesis examines whether Android's layered architecture mitigates privilege escalation in practice. Prior research shows that successful escalation does not always begin with kernel compromise. It may also arise from permission misuse, delegated privilege through inter-component communication, hidden service APIs, or policy divergence across OEM (Original Equipment Manufacturer) builds (Chin et al., 2011; He et al., 2023; Y.-T. Lee et al., 2023; Li et al., 2021). Android privilege escalation is therefore better understood as a cross-layer failure of privilege boundaries than as a single exploit class.

The study addresses that problem through a three-phase design. It combines a systematic literature review, comparative analysis of documented exploit cases, and controlled experimental validation. This structure allows the thesis to compare Android's intended defences with observed exploit behaviour across versions and configurations. The result is an evaluation of how privilege escalation is constructed, where Android defences succeed, and why some exploit paths still remain viable (He et al., 2023; Y.-T. Lee et al., 2023; Meng et al., 2018).

1.1 Motivation and Research Problem

Android privilege escalation warrants focused study because it directly tests the platform's core security promise. Android allows third-party code to execute on devices that hold authentication tokens, financial data, corporate credentials, and private communications. The platform therefore depends on sandbox integrity to prevent one application from reaching resources reserved for other applications or trusted system components (Armando et al., 2013; Meng et al., 2018). When privilege escalation succeeds, the failure is not local to one permission or one application. It undermines the assumptions on which higher-level security controls depend.

The persistence of Android escalation research shows that layered defences have not eliminated this problem. Real-world exploit surveys report that attackers continue to target kernels, drivers, privileged services, and vendor-specific subsystems to obtain elevated control (Meng et al., 2018; Zhang et al., 2015, 2016). At the same time, application- and framework-level studies show that escalation can also emerge through confused deputies, custom-permission flaws, or hidden service APIs without an immediate kernel exploit (Chin et al., 2011; He et al., 2023; Li et al., 2021; J. Wu et al., 2015). The technical implication is that Android privilege escalation is not explained adequately by a single attack surface.

A second motivation arises from the fragmented character of the existing evidence base. Much prior work evaluates permissions, inter-component communication, SEAndroid policy, hidden APIs, or kernel exploitation separately. That literature is valuable, but it often obscures how exploit chains traverse multiple layers and how defences behave when considered together (Chen et al., 2017; He et al., 2023; Y.-T. Lee et al., 2023). This fragmentation matters because Android security is enforced through interacting mechanisms, not isolated ones. A defence that appears sound within one layer may fail once an attacker crosses into another.

Vendor and version variation further deepen the research problem. Android hardening has reduced many older exploit paths, yet newer attacks increasingly depend on OEM-specific drivers, customised policies, and incomplete adoption of newer controls such as Scoped Storage (Y.-T. Lee et al., 2023; Meng et al., 2018; Yu et al., 2021). The same exploit family may therefore behave differently across devices and releases. This means

that Android privilege escalation cannot be assessed adequately through architecture alone. It must also be examined through observed exploit behaviour under different deployment conditions.

The research problem addressed in this thesis is therefore to determine how effectively Android security mechanisms mitigate privilege escalation in practice. The thesis analyses how attackers construct escalation paths, how those paths interact with Android's layered defences, and how exploit behaviour changes across versions and configurations. To address that problem, the study combines literature synthesis, empirical exploit analysis, and controlled validation. This approach is necessary because Android's main weakness appears not to be the absence of defensive mechanisms, but the incomplete coordination between them (He et al., 2023; Y.-T. Lee et al., 2023; Li et al., 2021; Park et al., 2012).

1.2 Research Questions and Scope

The thesis is guided by the following research questions (RQ):

- RQ1: How do attackers successfully escalate privileges on Android, and what techniques do they use?
- RQ2: How effective are Android's existing security mechanisms in mitigating these privilege escalation attacks?
- RQ3: How do selected privilege escalation exploits behave with respect to Android's security mechanisms across different versions and configurations?

Together, these questions reflect the literature's shift from isolated vulnerability analysis towards cross-layer evaluation of exploit construction and defence performance (He et al., 2023; Y.-T. Lee et al., 2023; Meng et al., 2018).

RQ1 focuses on exploit construction. Prior work shows that Android privilege escalation can emerge through kernel and driver flaws, inter-component communication misuse, permission-state manipulation, hidden service APIs, and OEM-specific policy weaknesses (Chin et al., 2011; He et al., 2023; Li et al., 2021; Yu et al., 2021). The question is therefore not whether Android contains vulnerabilities in general. It is how

attackers turn those weaknesses into usable escalation paths that cross intended privilege boundaries.

RQ2 evaluates Android's defensive effectiveness. Android combines permissions, SEAndroid, kernel hardening, storage controls, and patching, yet the literature shows that these protections do not always align cleanly in practice (Chen et al., 2017; He et al., 2023; Y.-T. Lee et al., 2023). This question therefore examines whether Android blocks escalation, constrains it, or leaves exploitable gaps between layers. It also supports the thesis distinction between prevention and containment, which is necessary because some attacks remain technically possible even when their practical impact is reduced (Ho et al., 2014; Park et al., 2012).

RQ3 addresses variation across versions and configurations. Android is not a uniform platform, and exploitability often depends on release-specific hardening, application configuration, and OEM modification. Earlier survey work already shows that newer exploit paths increasingly depend on device-specific and vendor-specific conditions rather than on universal flaws (Meng et al., 2018; Zhang et al., 2015). This question therefore tests whether exploit behaviour changes materially when the platform version, policy state, or exposed application boundary changes.

The scope of the thesis is confined to Android privilege escalation and sandbox bypass. It focuses on attacks that move from a standard application context to protected resources, privileged services, or stronger execution domains. This includes kernel and driver exploitation, permission abuse, ICC (Inter-Component Communication) misuse, hidden APIs, and policy inconsistency, because these surfaces repeatedly appear in the literature as routes across Android privilege boundaries (He et al., 2023; Y.-T. Lee et al., 2023; Li et al., 2021; Meng et al., 2018). By contrast, threats such as phishing, ordinary social engineering, or vulnerabilities without escalatory effect fall outside the primary scope. This boundary is necessary because the thesis evaluates Android's privilege-separation architecture rather than the entire mobile threat landscape.

1.3 Thesis Contribution

This thesis makes three principal contributions. First, it provides a cross-layer explanation of Android privilege escalation that integrates application logic, service access, policy

enforcement, and lower-level compromise within one analytical framework. Prior studies often examine these surfaces separately, which limits understanding of how exploit chains traverse the full privilege boundary (Chen et al., 2017; He et al., 2023; Y.-T. Lee et al., 2023). The thesis instead argues that Android's main weakness lies in incomplete coordination between layers rather than in the total absence of defences.

Second, the thesis contributes a structured taxonomy of Android privilege escalation routes grounded in both prior literature and documented exploit behaviour. That taxonomy distinguishes kernel and driver exploitation, permission and custom-permission abuse, ICC misuse, hidden service APIs, policy divergence, and UI-mediated (User Interface mediated) abuse. This structure is analytically useful because it allows exploit families to be compared against the security mechanisms that should have mitigated them (He et al., 2023; Li et al., 2021; Meng et al., 2018). It also supports more precise reasoning about where Android privilege boundaries fail and where they remain effective.

Third, the thesis combines literature synthesis, comparative analysis of documented exploit cases, and controlled validation within one version-aware design. This methodological combination allows the study to distinguish between architectural expectation and observed platform behaviour. It also clarifies the difference between prevention and containment, which earlier defence-focused work identifies as practically important but insufficiently emphasised in mainstream Android evaluation (Ho et al., 2014; Park et al., 2012). The resulting contribution is therefore both explanatory and practical: it offers a clearer basis for assessing Android privilege boundaries and for prioritising future hardening efforts.

1.4 Thesis Structure

This thesis proceeds from technical foundation to empirical evaluation and then to integrated interpretation. Chapter 2 outlines Android's security architecture, its principal exploit surfaces, and the defensive mechanisms relevant to privilege escalation. Chapter 3 synthesises the literature on exploit construction, defence effectiveness, and ecosystem variation, and identifies the research gaps addressed by this study. Chapter 4 then presents the methodological design, including the literature review process, comparative analysis of documented exploit cases, and controlled validation strategy.

Chapter 5 applies the taxonomy to documented real-world exploit cases and analyses their temporal, technical, and defensive patterns. Chapter 6 validates selected exploit families in controlled environments in order to compare expected and observed security behaviour across versions and configurations. Chapter 7 integrates the findings and answers the research questions directly. Chapter 8 concludes the thesis by summarising the main findings, contributions, and implications for Android privilege escalation defence.

2 Background

This chapter outlines the Android security mechanisms most relevant to privilege escalation. It explains how sandboxing, permissions, Binder-mediated services, SEAndroid, and kernel protections define Android's privilege boundaries. These mechanisms matter because privilege escalation becomes possible when attackers cross one or more of those boundaries (He et al., 2023; Y.-T. Lee et al., 2023; Meng et al., 2018).

2.1 Android Security Architecture

Android security is a layered control system rather than a single barrier. It combines application isolation, permission mediation, SEAndroid mandatory access control, and kernel-level enforcement to constrain untrusted code. Privilege escalation becomes possible when attackers cross one or more of these intended boundaries (Armando et al., 2013; Meng et al., 2018).

The first of these boundaries is the application sandbox, which is rooted in Linux process isolation. Android assigns each installed application a distinct Linux UID, and the launched process inherits that identity, thereby separating code execution, files, and many resource accesses by default. Bhandari et al. (2017) note that Android augments Linux kernel isolation with sandboxing, and Armando et al. (2013) show that application launching binds the new process to the application's assigned UID. This arrangement creates strong default separation, although shared UIDs and inter-application interaction can still weaken practical isolation.

The permission model forms the second architectural layer by mediating access to sensitive APIs and resources beyond the basic sandbox. Felt et al. (2011) show that Android controls privacy- and security-relevant API access through application permissions, while Bhandari et al. (2017) describe the main protection levels as normal, dangerous, signature, and SignatureOrSystem. The security logic is therefore one of least privilege, because applications should receive only the permissions required for their

declared functions. However, the model depends on accurate declarations and consistent enforcement, and Li et al. (2021) demonstrate that design flaws in custom permissions can still create large numbers of successful privilege-elevation paths.

A third layer is provided by SELinux (Security-Enhanced Linux), implemented on Android as SEAndroid, which supplements discretionary controls with mandatory policy enforcement. Reshetova et al. (2017) explain that Android initially relied on permissions and Linux discretionary access control, but later adopted SEAndroid because DAC (Discretionary Access Control) alone could not contain powerful exploits. From Android 5.0 onward, each process is expected to execute inside a confined SEAndroid domain with defined access rules. This shift materially strengthened isolation, yet Chen et al. (2017) show that protection remains only as strong as the policy itself, and OEM policy customisation can introduce unintentional privilege assignments and other weaknesses.

Kernel hardening and memory protection constitute the lowest defensive layer, but they remain decisive because every higher control ultimately depends on kernel enforcement. Meng et al. (2018) show that modern Android exploits increasingly rely on memory corruption and vendor-specific driver targets as generic root exploits decline. Zhang et al. (2016) further show that customised subsystems such as ION (Android memory management subsystem) can expose protected memory, leak sensitive data, and even crash the operating system when heap interfaces are improperly exposed. These findings indicate that hardening raises attacker cost, yet driver complexity and vendor modification still preserve viable routes to sandbox escape and privilege escalation (Meng et al., 2018; Zhang et al., 2015, 2016).

These layers operate together. The sandbox limits the starting process, permissions regulate declared access, SEAndroid constrains runtime capabilities, and kernel protections enforce the final boundary. Android privilege escalation therefore depends not only on flaws within one layer, but also on unsafe interaction between them (Armando et al., 2013; He et al., 2023).

2.2 Application Components, Binder and IPC

Android's programming model is component-based, and its security properties follow from that design. Chin et al. (2011) and Bhandari et al. (2017) identify Activities,

Services, Broadcast Receivers, and Content Providers as the main execution components. Each exposes different reachability and lifecycle properties. Because these components may be invoked across application boundaries, Android must regulate both execution and capability transfer. Component structure therefore expands the sandbox question from process isolation to controlled interaction between principals.

Inter-process communication makes that component model operational. Chin et al. (2011) show that Intents and Binder enable cross-application invocation, while Feng and Shin (2016) treat Binder as a major attack surface rather than a neutral transport. He et al. (2023) further show that system services remain reachable through Binder-facing helper classes and hidden interfaces. Binder should therefore be analysed as a trust boundary whose enforcement quality directly shapes privilege exposure.

This communication model enables modularity, but it also enlarges the attack surface. Intents support explicit and implicit communication, and Binder exposes service functionality across process boundaries. As a result, security depends on whether components are exported, how Intents are resolved, and whether access checks are performed consistently at runtime (Chin et al., 2011; Feng & Shin, 2016).

ICC weaknesses can produce privilege escalation without kernel compromise. Exposed components, weak caller validation, and permission re-delegation can allow less-privileged code to trigger more-privileged behaviour through a deputy application (Chin et al., 2011; Felt et al., 2011; J. Wu et al., 2015).

IPC (Inter-Process Communication) also connects third-party code to privileged system services. For that reason, Binder and service helpers are relevant to privilege escalation whenever service-side validation is weaker than the exposed access path (Feng & Shin, 2016; He et al., 2023).

2.3 Privilege Escalation on Android

In Android, privilege escalation denotes a transition from a sandboxed application or constrained process to capabilities reserved for more trusted principals. Those principals include privileged system services, signature-level contexts, the system user, and in the most severe cases root or kernel control. Meng et al. (2018) describe vulnerability exploitation as a common route to higher privilege on Android. Park et al. (2012) further

show that root hijacking can bypass DAC, sandboxing, and permission enforcement, thereby undermining the platform's normal trust assumptions.

Privilege escalation on Android emerges through several routes with different preconditions and targets. These include application-layer abuse of permissions or exported components, service-layer access through Binder-facing interfaces, and kernel- or driver-assisted escalation to system or root control (He et al., 2023; Li et al., 2021; Meng et al., 2018).

Privilege escalation is often better understood as a chain than as a single bug. Attackers may combine an initial foothold with a capability transfer or enforcement mismatch to cross successive privilege boundaries (Armando et al., 2013; He et al., 2023; Li et al., 2021).

After escalation, attackers generally pursue control, persistence, surveillance, or defence evasion rather than privilege for its own sake. Once elevated privileges are obtained, an adversary may disable protections, execute unauthorised shell commands, replace system-critical files, or manipulate core infrastructure. Park et al. (2012) observed precisely these post-exploitation objectives when modelling root shell abuse and *core.jar* replacement. He et al. (2023) additionally show that hidden-service abuse can bypass user interactions and reach sensitive system functionality. These goals matter analytically because they show that the real impact of escalation lies in what elevated access enables afterwards.

2.4 Main Exploit Surfaces

Kernel and driver vulnerabilities remain one of the most consequential exploit surfaces because successful compromise at that layer can invalidate every higher Android control. The Linux kernel enforces process isolation, memory protection, and much of the platform's effective privilege boundary. Meng et al. (2018) show that modern privilege escalation increasingly relies on memory corruption in kernels, drivers, and closely coupled system components rather than on simple legacy root paths. Zhang et al. (2015) strengthen that conclusion by showing that commercial root providers actively adapt exploits to device-specific kernels and drivers, because generic kernel exploits no longer scale well across the fragmented Android ecosystem.

Permission misuse forms a second major exploit surface, although it is often less visible than kernel exploitation. Android's permission model is intended to enforce least privilege, yet several studies show that privilege elevation can arise from the framework's own design assumptions. Li et al. (2021) demonstrate that custom permissions can become escalation channels through dangling definitions, inconsistent protection-group mappings, and permission elevation during updates. Their CUPERFUZZER experiments produced 2384 successful escalation scenarios across 30 critical paths, which indicates that permission abuse is not merely theoretical. This surface shows that elevated behaviour can emerge without kernel compromise when permission definitions and state transitions are enforced inconsistently.

Inter-component communication creates a third exploit surface by enabling transitive privilege use between applications and components. Android's component model encourages reuse and delegation, yet that same flexibility can allow low-privileged code to trigger privileged operations indirectly. Mathew (2012) identifies this problem as a structural weakness in Android's access-control model, because developers are often left responsible for protecting exposed components correctly. The same concern appears in PaddyFrog, which models confused deputy exploitation through exposed components and privileged tasks (J. Wu et al., 2015). This matters because escalation may occur through trusted application relationships rather than low-level memory errors.

A fourth exploit surface exists at the service layer, where hidden or non-SDK (non Software Development Kit) APIs can bypass the controls implemented in public helpers. He et al. (2023) show that Android apps can reach Binder-accessible system services through hidden APIs loaded from *framework.jar*, even though developers are expected to use only public SDK interfaces. Their analysis found that service helpers and underlying service APIs often implement inconsistent checks, allowing attackers to bypass permission validation, forge identities, or invoke privileged operations directly.

Vendor customisation and policy divergence form a fifth exploit surface because Android security is implemented in an ecosystem rather than in one uniform platform build. Vendor-specific interfaces are an important exploit surface because OEM modifications often extend Android through customised drivers, storage controls, and policy rules. Zhang et al. (2015) show that commercial rooting increasingly targets vendor-specific drivers, while Y.-T. Lee et al. (2023) and Li et al. (2021) both expose differences in

permissions, mappings, and storage policies across devices and versions. This surface is analytically important because it explains why privilege escalation cannot be understood only as an AOSP (Android Open Source Project) problem. In practice, many exploitable conditions emerge when OEMs extend Android without preserving the intended equivalence of its security controls.

2.5 Relevant Defensive Mechanisms

Permissions remain Android's primary mechanism for mediating access to sensitive resources, and the later runtime consent model was intended to strengthen that control by tying dangerous access to user approval at execution time. In architectural terms, this model operationalises least privilege by requiring applications to request access only when needed. Permission enforcement remains vulnerable when permission semantics change, when custom permissions are defined inconsistently, or when applications chain loosely regulated capabilities (Li et al., 2021; Meng et al., 2018). Permissions therefore remain necessary, but they do not provide a complete defence against escalation.

SEAndroid provides a second defensive layer by enforcing mandatory access control independently of application-declared permissions. This matters because Android's earlier DAC-oriented model was insufficient against stronger exploit paths, especially once attackers reached privileged processes or lower system layers. Reshetova et al. (2017) explain that Android 5.0 required processes to run inside confined SEAndroid domains, thereby hardening the platform against broad privilege abuse. Yet Chen et al. (2017) show that protection is only as strong as the deployed policy, and their analysis across AOSP versions and seven manufacturers found several forms of unintentional privilege assignment. SEAndroid is therefore a critical control, but its effectiveness depends on policy correctness and OEM discipline.

Filesystem defences, including Scoped Storage, are relevant because some escalation paths target shared storage and policy interaction rather than pure code execution. Although these controls reduce attack opportunities, they do not remove the need for coherent multi-policy enforcement across storage, permissions, and SELinux (Y.-T. Lee et al., 2023).

Patching and platform maintenance remain important because many exploit paths lose effectiveness once fixes are deployed. At the same time, patching is reactive, and its benefits depend on timely adoption and on whether fixes address root causes rather than isolated symptoms (He et al., 2023; Meng et al., 2018).

Detection and containment mechanisms are important because some privilege escalation attempts evade preventive controls. Prior work shows that runtime monitoring and post-exploitation restriction can still reduce attacker capability after privileged access is obtained (Ho et al., 2014; Park et al., 2012).

3 Literature Review

This chapter synthesises the literature on Android privilege escalation and evaluates how prior work explains the failure of Android's privilege boundaries. The aim is not to summarise individual papers in isolation, but to compare the main exploit routes, defensive approaches, and recurring structural weaknesses identified across the field. This focus is necessary because Android privilege escalation spans application logic, Binder-mediated services, policy enforcement, and lower system layers, and no single study captures that full path alone (He et al., 2023; Y.-T. Lee et al., 2023; Meng et al., 2018).

The literature is organised thematically rather than chronologically. This choice reflects the structure of the research problem. Existing work is dispersed across permissions, inter-component communication, hidden APIs, SEAndroid policy, storage controls, and kernel or driver exploitation, which makes a paper-by-paper narrative less useful than direct comparison of exploit surfaces and defensive assumptions (He et al., 2023; Li et al., 2021; Mathew, 2012). The chapter therefore moves from review protocol to study overview and then to thematic synthesis.

The chapter supports the thesis in three ways. First, it identifies how prior research explains the construction of privilege escalation attacks. Second, it evaluates how Android's existing controls are represented in the literature and where those controls are shown to fail. Third, it isolates the main research gaps that justify the later comparative analysis of documented exploit cases and controlled validation. These gaps include context-insensitive permission enforcement, incomplete cross-layer policy validation, vendor divergence, limited attention to containment, and underrepresentation of recent Android versions (He et al., 2023; Y.-T. Lee et al., 2023; Park et al., 2012).

3.1 Review Protocol

The review followed a structured protocol to keep study selection transparent and analytically focused. Searches were conducted in IEEE Xplore, ACM Digital Library, SpringerLink, ScienceDirect, Scopus, Google Scholar, and CiteSeerX using

combinations of Android, privilege escalation, sandbox, SELinux, SEAndroid, hidden API, and related exploit terms. The search process was supplemented through backward and forward snowballing, and duplicate records were removed before screening.

Studies were included when they were written in English, published between 2010 and 2025, focused on Android privilege escalation or sandbox bypass, and provided empirical or analytical evidence. Studies were excluded when Android was only incidental, when the work lacked methodological transparency, or when the source was non-peer-reviewed and did not contain technically verifiable evidence. This filtering was necessary because Android security research includes a mix of exploit demonstrations, architecture analyses, and tool papers with highly uneven empirical depth.

Study selection used title-and-abstract screening followed by full-text review. Accepted studies were then evaluated against five criteria: clarity of objective, technical completeness, appropriateness of validation method, reproducibility of claims, and venue credibility. Data were extracted into a structured form recording exploit vector, targeted component, affected version, required privileges, evaluated defences, and acknowledged limitations. The review was then synthesised thematically rather than paper by paper, because the thesis examines Android privilege escalation as a cross-layer problem rather than as a set of isolated mechanisms.

3.2 Overview of Selected Studies

The selected studies show that Android privilege escalation research has expanded from early permission and sandbox concerns into a broader literature on cross-layer enforcement failure. The corpus spans kernel and driver exploitation, permission misuse, inter-component communication abuse, hidden service APIs, policy inconsistency, and containment-oriented defences. This distribution is analytically important because it shows that escalation is not confined to one subsystem. Instead, the literature treats Android privilege boundaries as vulnerable wherever trust is delegated across application, service, policy, or kernel layers (He et al., 2023; Li et al., 2021; Meng et al., 2018).

Methodologically, the corpus is mixed but not shallow. It includes exploit surveys, static and dynamic analysis tools, policy-analysis frameworks, and controlled experimental evaluations. This diversity strengthens the review because different methods expose

different parts of the problem. Exploit surveys capture attacker practice, policy analyses expose structural inconsistencies, and tool papers test whether specific exploit paths can be detected or constrained. At the same time, the literature remains uneven. Offensive findings are often more concrete than defensive claims, and vendor-specific evidence is still more limited than AOSP-oriented analysis.

For this reason, the chapter treats the selected studies as a comparative evidence base rather than as a complete map of the Android ecosystem. Thematic synthesis is used to identify where studies converge, where they differ, and which questions remain underexplored. The result is a literature review oriented towards mechanism, limitation, and research gap, rather than toward descriptive coverage alone.

3.3 Theme 1: Kernel and Driver Exploitation

Kernel and driver exploitation remains central to Android privilege escalation because compromise at that layer can invalidate higher controls simultaneously. Android permissions, application sandboxing, and SEAndroid ultimately depend on kernel enforcement, so a successful kernel-level exploit can collapse multiple privilege boundaries at once (Chen et al., 2017; Meng et al., 2018). The literature therefore treats low-level exploitation not as one attack surface among many, but as the boundary whose failure has the widest systemic consequences.

Earlier work identified comparatively broad low-level weaknesses, including launch-flow flaws, symbolic-link abuse, and daemon misuse, which could be exercised across multiple devices and versions (Armando et al., 2013; Meng et al., 2018). Later studies show a marked shift towards memory corruption and vendor-specific driver targets. Meng et al. (2018) argue that memory corruption became the dominant exploitation route as Android hardened, whereas Zhang et al. (2015) show that commercial root providers increasingly relied on device-specific exploit variants. These studies converge on a clear trend: low-level privilege escalation has not disappeared, but it has become more fragmented and hardware dependent.

Driver customisation is especially important because OEM extensions repeatedly reintroduce privileged attack surfaces outside the AOSP baseline. Zhang et al. (2016) show that the ION memory subsystem exposed serious flaws, including denial of service

and memory disclosure, on recent Android devices. Zhang et al. (2015) similarly identify many unpublished driver exploits used by rooting providers, which indicates that vendor-specific low-level attack paths remain valuable in practice. The kernel literature therefore suggests that Android hardening has shifted attacker effort towards customised subsystems rather than eliminating low-level escalation.

The main limitation of this research area is that it often explains successful low-level compromise better than it explains ecosystem-wide defensive consistency. Exploit surveys reveal broad temporal trends, but they do not fully resolve how patching, driver variation, and vendor policy interact across the platform (Meng et al., 2018; Zhang et al., 2015). This limitation matters for the present thesis because it supports a cross-layer analysis: kernel exploitation remains important, yet it must be interpreted together with service, policy, and application-level weaknesses rather than in isolation.

3.4 Theme 2: Permission Abuse and Custom Permissions

The permission model is both a foundational Android defence and a recurring source of escalation weakness. Felt et al. (2011) show that many applications are overprivileged, which undermines least-privilege assumptions before exploitation begins. Fang et al. (2014) extend that critique by arguing that Android permissions remain coarse-grained and administratively difficult to manage. These studies suggest that permission-mediated security often fails through ordinary application design, not only through overtly malicious code.

Permission research also shows that static permission declarations are an incomplete model of practical privilege. PScout demonstrates that Android's permission specification is complex and context-sensitive, especially where enforcement occurs across processes and services rather than within a single API boundary (Au et al., 2012). Android Permissions Remystified reaches a complementary conclusion from runtime observation, showing that users often wish to block permission use in specific contexts rather than deny an application entirely (Wijesekera et al., 2015). Together, these studies indicate that Android permissions are formally expressive, yet still poorly aligned with runtime context and user expectation.

Later work shows that custom permissions can become direct escalation channels rather than mere documentation problems. Li et al. (2021) identify four design shortcomings in Android's custom-permission framework and demonstrate that those flaws can grant dangerous system permissions without user consent. Their findings differ from earlier overprivilege research in an important way. Felt et al. (2011) and Fang et al. (2014) mainly expose weak permission use, whereas Li et al. (2021) show that the framework itself can create privilege escalation through inconsistent state handling. This shift moves the problem from developer practice to platform design.

The main limitation of the permission literature is that it still explains permission abuse more effectively than permission interaction. Earlier studies establish overclaiming, incomplete documentation, and weak contextual integrity, while later work exposes framework-level inconsistency, yet few studies model how permissions combine with ICC, services, or UI capabilities in full exploit chains (Felt et al., 2011; Li et al., 2021; Wijesekera et al., 2015). This limitation is central to the thesis because it supports treating permission abuse as one part of a broader cross-layer escalation problem.

3.5 Theme 3: ICC Abuse and Delegated Privilege Misuse

Inter-component communication is a major Android exploit surface because it allows applications to trigger behaviour across package boundaries without receiving the callee's privileges directly. Chin et al. (2011) show that insecure message passing can expose confidentiality and integrity weaknesses, while Bhandari et al. (2017) argue that ICC remains a principal route for inter-app threats, including collusion and privilege escalation. The security problem is therefore structural: Android encourages delegation, but delegation can become a channel for unintended privilege transfer.

Confused deputy research shows that this risk is both common and technically tractable. Wu et al. (2015) identify exposed components and reachable privileged tasks as the key enabling conditions for Android confused deputy attacks and report large-scale evidence of exploitable cases. Demissie et al. (2016) reach a similar conclusion through combined static and dynamic analysis, showing that inadequate message validation allows unprivileged applications to exploit delegated privileged behaviour. These studies converge on the claim that ICC privilege escalation is not a fringe programming mistake, but a recurring consequence of Android's component model.

Defensive work in this area differs mainly in where it places control. Chin et al. (2011) focus on insecure communication patterns, whereas Y. K. Lee et al. (2017) propose SEALANT to combine static analysis of vulnerable ICC paths with runtime interception of suspicious intent exchanges. The contrast is important. Earlier work is stronger at exposing the deputy problem, while later work is stronger at constraining it operationally. Yet both lines of research assume that application-level structure is security-critical, which means Android's sandbox depends partly on correct component exposure and caller validation rather than on process isolation alone.

The main limitation of the ICC literature is that it remains more mature at detecting vulnerability patterns than at explaining how those patterns interact with service-layer and policy-layer weaknesses. ICC studies show clearly how privileged deputies are abused, but they often stop at the application boundary (Chin et al., 2011; Demissie et al., 2016; Y. K. Lee et al., 2017). This limitation matters for the present thesis because delegated misuse is one route through Android privilege boundaries, but not the only one. It must therefore be analysed together with permissions, hidden APIs, and policy inconsistency.

3.6 Theme 4: Hidden APIs and Service-Layer Inconsistencies

Hidden and non-SDK APIs expose a distinct Android privilege-escalation surface because they allow applications to bypass the public helper layer and interact more directly with privileged system services. He et al. (2023) show that many Android system services are accessed through helper classes that apply checks before issuing Binder calls, yet hidden APIs can still reach the underlying service interfaces. This matters because Android's public API restrictions do not automatically imply equivalent service-side enforcement.

The central contribution of this literature is its focus on inconsistent validation across the service path. He et al. (2023) show that hidden service APIs may omit permission checks, caller authentication, or input validation that appear in helper-side code. This produces a qualitatively different problem from earlier permission or ICC work. In permission studies, the question is often whether access was requested correctly. In hidden-API studies, the question is whether the trusted path and the reachable path enforce the same security assumptions. The latter is a deeper cross-layer consistency problem.

The literature also shows that Android's response has been only partially successful. Google's blacklist and greylist restrictions reduced some non-SDK access, yet He et al. (2023) still find inconsistently protected hidden APIs in newer versions and show that some restrictions can be bypassed. Their results therefore refine the interpretation of platform hardening. Later Android versions do not remove the hidden-API issue entirely. Instead, they narrow the conditions under which reflective or indirect service access becomes practically dangerous.

The main limitation of this area is that it remains heavily concentrated on service inconsistency rather than on its interaction with broader ecosystem variation. He et al. (2023) establish the technical importance of hidden APIs convincingly, but the literature is still less developed in showing how OEM modification, version drift, and service customisation alter that risk across the ecosystem. This limitation supports the thesis focus on exploit behaviour across configurations rather than on service-layer analysis alone.

3.7 Theme 5: Vendor Divergence and Policy Inconsistency

Vendor customisation is a recurring source of Android privilege-boundary weakness because OEMs extend the platform without preserving uniform security assumptions. Unlike AOSP-focused studies that analyse one platform baseline, later work shows that OEMs frequently modify drivers, SEAndroid rules, storage controls, and service exposure in ways that alter exploitability materially (Chen et al., 2017; Zhang et al., 2016; Yu et al., 2021). This matters because Android privilege escalation increasingly depends on device-specific conditions rather than on one universal flaw.

SEAndroid policy analysis provides the clearest evidence of this divergence. Chen et al. (2017) show that Android protection is only as strong as the deployed policy, and their analysis of AOSP versions and manufacturer builds identified several forms of unintentional privilege assignment. Yu et al. (2021) extend that argument at larger scale through SEPAL, which identified 7111 unregulated customised rules across 774 firmware images from 72 manufacturers. The proportion of problematic rules also increased in newer Android versions. These studies converge on a key point: policy customisation is not a secondary maintenance issue, but a direct source of privilege-boundary failure.

Storage policy research reveals the same pattern from a different angle. Y.-T. Lee et al. (2023) show that Scoped Storage reduced many attack operations on external storage, yet OEM systems benefited unevenly because adoption was partial and legacy behaviour remained available in important cases. Their analysis demonstrates that Android filesystem protection depends on the interaction between DAC, SEAndroid, Android permissions, and OEM-specific storage decisions rather than on one control alone. In that sense, storage hardening improved the baseline, but did not remove exploitability where policy interaction remained inconsistent (Y.-T. Lee et al., 2023).

Permission-state and mapping inconsistencies further reinforce the vendor-divergence problem. Li et al. (2021) show that permission definitions and mappings can create privilege-escalation paths when the framework handles state transitions or associations unsafely. Taken together, these studies suggest that Android privilege boundaries vary not only because OEMs add code, but also because they alter the meaning and coordination of existing controls (Y.-T. Lee et al., 2023; Li et al., 2021).

The main limitation of this literature is that ecosystem-wide comparison remains incomplete. Public exploit reporting is uneven, firmware access varies by vendor, and many studies still focus on selected manufacturers or selected subsystems rather than on the ecosystem as a whole. As a result, the literature can show that vendor divergence matters, but it cannot yet measure that divergence comprehensively across all Android deployments. This limitation is important for the present thesis because it justifies the later emphasis on version-aware and configuration-aware analysis rather than on purely architectural claims (Y.-T. Lee et al., 2023; Meng et al., 2018; Yu et al., 2021).

3.8 Theme 6: Defence Effectiveness and Containment

The defence literature shows that Android privilege escalation cannot be addressed through one preventive mechanism alone. Early proposals such as Apex sought to refine permission enforcement by introducing selective grants and runtime constraints, thereby making the permission model more context-sensitive (Nauman et al., 2010). Later work shifted towards policy analysis and cross-layer triage. SELint (SEAndroid policy analysis tool) supports SEAndroid rule quality by identifying mistakes in OEM policy construction, whereas SEPAL shows at larger scale that customised SEAndroid rules frequently become unregulated and security-relevant (Reshetova et al., 2017; Yu et al.,

2021). These studies suggest that Android defence has expanded from access control design into policy correctness and ecosystem governance.

Recent policy-analysis work also shows that prevention depends on modelling the interaction between multiple access-control layers. Chen et al. (2017) analyse SEAndroid together with DAC and find unintentional privilege assignment, while Y.-T. Lee et al. (2023) show that Scoped Storage reduces many attack operations but does not remove risk where OEM adoption is partial. Unlike earlier permission-centric defences, these studies treat Android security as a combined policy problem. They therefore align closely with the thesis view that privilege escalation is sustained by inconsistent coordination between controls rather than by one missing barrier.

Containment research adds an important dimension that prevention-focused work often underplays. RGBDroid assumes that root privilege may already be obtained and seeks to reduce attacker capability through kernel-level whitelisting and protection of critical resources (Park et al., 2012). PREC similarly focuses on runtime detection and containment of root exploits through anomalous system-call behaviour in high-risk components (Ho et al., 2014). These approaches differ from earlier design-level or policy-level proposals because they accept that some exploit attempts will bypass initial controls. The practical implication is that Android defence should be judged not only by whether it prevents escalation, but also by whether it limits post-compromise damage.

The main limitation of the defence literature is its fragmentation. Apex improves permission flexibility, SELint and SEPAL improve policy quality, PolyScope improves multi-policy analysis, and RGBDroid and PREC improve containment, yet no single approach resolves privilege escalation across the full stack (Ho et al., 2014; Y.-T. Lee et al., 2023; Nauman et al., 2010; Park et al., 2012; Reshetova et al., 2017). This limitation is central to the present thesis. It motivates an integrated evaluation of Android privilege boundaries that distinguishes prevention from containment and compares exploit behaviour across layers, versions, and configurations.

3.9 Research Gaps

The first major gap concerns context-aware privilege enforcement. Earlier work showed overprivilege and permission-framework weakness, while later studies demonstrated that

benign-looking capability combinations can still produce high-impact escalation paths (Felt et al., 2011; Li et al., 2021; Meng et al., 2018). The literature therefore explains why Android's permission model is necessary, but not how it should reason about chained behaviour and execution context more effectively.

A second gap is the limited treatment of cross-layer policy coherence as a first-class security problem. Hidden service APIs, storage controls, and SEAndroid policy analyses all show that privilege escalation often emerges when neighbouring layers enforce different assumptions about identity, permission, or resource (Chen et al., 2017; He et al., 2023; Y.-T. Lee et al., 2023). Although these studies expose important inconsistencies, automated end-to-end validation across application, service, policy, and kernel layers remains uncommon.

A third gap concerns vendor divergence and incomplete policy adoption. The literature shows that OEM-specific drivers, customised SEAndroid rules, and uneven storage-policy deployment materially affect exploitability, yet ecosystem-wide empirical comparison remains limited (Y.-T. Lee et al., 2023; Zhang et al., 2015; Yu et al., 2021). This makes it difficult to determine whether Android privilege escalation is primarily an architectural problem, a vendor problem, or an interaction between both.

A fourth gap is the relative neglect of post-exploitation containment. Most studies focus on preventing initial escalation, while fewer examine what should happen after partial or temporary privilege theft. Yet containment-oriented work shows that runtime restriction and response can still reduce attacker capability even after compromise begins (Ho et al., 2014; Park et al., 2012). This suggests that Android defence evaluation remains too prevention-centric.

The fifth gap is the limited evaluation of newer Android versions. Much of the core literature remains concentrated on versions up to Android 10, 11, or 12, even though hidden-API restrictions, runtime permissions, storage policy, and sandbox behaviour continue to (He et al., 2023; Meng et al., 2018). The present thesis addresses that limitation by treating exploit behaviour across versions and configurations as an explicit research question rather than as a background assumption.

4 Methodology

This chapter explains the methodological framework used to investigate Android privilege escalation and sandbox bypass. The thesis uses a three-phase design consisting of a systematic literature review, an empirical analysis of documented exploit cases, and controlled experimental validation. These phases are used together because the thesis evaluates both how privilege escalation is constructed and how Android's defensive mechanisms behave in practice.

4.1 Research Design Overview

This thesis follows an interpretive-empirical design with a strong analytical orientation. It does not attempt to discover a new Android exploit. Instead, it examines how known privilege-escalation mechanisms operate across Android security layers and how effectively Android's controls respond to them. This design is appropriate because the research questions concern both technical attack construction and practical defensive performance.

The methodology consists of three connected phases. Phase 1 is a systematic literature review of Android privilege escalation, sandbox bypass, and relevant defence mechanisms. Phase 2 compares representative documented Android privilege-escalation cases drawn from public technical sources across the major exploit families identified in the literature. Phase 3 validates a purposive subset of exploit families in controlled environments using emulator-based experimentation and, where needed, device-aware reasoning.

This structure strengthens the credibility of the findings through triangulation. Prior literature provides conceptual and technical grounding, documented cases provide comparative empirical evidence, and controlled validation provides direct behavioural evidence. The methodology therefore supports the thesis argument that Android privilege escalation should be studied as a cross-layer interaction between exploit mechanisms,

defensive controls, and deployment conditions rather than as a set of isolated vulnerabilities.

4.2 Phase 1: Systematic Literature Review

Phase 1 gathered and synthesised prior research on Android privilege escalation and sandbox bypass. The review focused on studies that examined exploit vectors, defensive mechanisms, policy inconsistency, or platform behaviour relevant to privilege boundaries. The purpose of this phase was to identify recurring exploit mechanisms, compare defensive claims, and define the taxonomy used in the later empirical analysis.

Studies were selected through structured searching and screening. Searches were conducted across major academic databases using combinations of Android, privilege escalation, sandbox, SEAndroid, hidden API, and related exploit terms. Studies were included when they focused explicitly on Android privilege escalation or sandbox bypass and provided empirical or analytical evidence. Studies were excluded when Android was incidental, when the work lacked methodological transparency, or when the source did not contain technically verifiable findings.

Accepted studies were analysed using structured extraction and thematic synthesis. For each study, the review recorded attack vector, targeted component, affected version, required privileges, evaluated defences, and key limitations. The output of this phase was a literature-derived taxonomy of privilege-escalation routes together with a set of research gaps that informed exploit selection and experimental validation.

4.3 Phase 2: Comparative Analysis of Documented Cases

Phase 2 compared representative documented Android privilege-escalation cases drawn from public technical sources. Its purpose was to move beyond literature synthesis and examine how escalation is described in technically visible real-world cases. Sources included peer-reviewed studies, exploit surveys, technically credible case studies, and, where relevant, publicly documented vulnerability or advisory material. This source mix was necessary because Android privilege-escalation evidence is distributed unevenly across disclosure channels and varies substantially in technical depth.

Documented cases were selected when they described a clear Android privilege-escalation or sandbox-bypass path, identified the targeted mechanism or boundary, and provided enough technical detail to support comparison. Cases were excluded when they were anecdotal, duplicated another case without adding material technical insight, or did not identify a distinct escalation route. The aim of this phase was not to produce a statistically complete exploit inventory, but to compare structurally representative cases across the exploit families identified in Phase 1.

Each selected case was examined using a common comparison frame. The analysis recorded the exploit family, targeted component or layer, required preconditions, likely privilege outcome, and the Android security mechanisms that should have constrained the exploit under intended operation. This allowed the thesis to compare how different exploit families traverse Android privilege boundaries and how those paths encounter permissions, component restrictions, service checks, SEAndroid policy, storage controls, patching, or vendor-specific modifications.

The output of this phase was a comparative analysis of documented cases across exploit family, target layer, preconditions, defence encounter, and version or vendor context where visible. This comparative layer supported the interim answers to RQ1 and RQ2 and provided the basis for selecting representative exploit families for experimental validation in Chapter 6.

4.4 Phase 3: Experimental Validation

Phase 3 used controlled experimentation to validate selected Android privilege-escalation cases. The purpose of this phase was not to discover new vulnerabilities, but to observe whether representative exploit families still succeeded, failed, or became contained when confronted with real platform and application-level defences. This phase was necessary because literature findings and exploit records alone cannot fully show whether a documented exploit path remains effective under specific Android versions and configurations.

Exploit cases were selected purposively rather than randomly. Cases were chosen after completion of the comparative case analysis and were prioritised according to taxonomy coverage, technical clarity, identifiable prerequisites, legal feasibility, and expected

observable outcomes. The selected cases were intended to represent different escalation mechanisms, such as permission-state mutation, delegated privilege misuse, hidden-API reachability, and UI-mediated abuse, while remaining feasible to reproduce safely in controlled environments.

Validation was conducted in emulator-based testbeds. Android Studio, the Android Emulator, Android Debug Bridge (adb), and filtered logcat (Android logging utility) output were used to support repeatable setup, controlled execution, and structured observation. Emulator-based testing was selected because it enabled consistent comparison across Android versions and configurations. This thesis did not include physical-device testing, and hardware- or OEM-dependent exploit classes were therefore not selected for experimental reproduction.

The validation recorded whether the selected exploit path remained confined to the application sandbox, achieved the intended privileged effect, or was visibly constrained by Android's controls. Observed evidence included runtime permission state, application-visible behaviour, filtered logs, and screenshots of key outcomes. Success was defined as reproduction of the intended privileged effect, failure as inability to reproduce that effect, and containment as partial technical reachability without meaningful or durable privilege gain. The output of this phase was a set of observed exploit outcomes that directly supported the answer to RQ3.

4.5 Ethics, Legality and Safety

All experimental activity in this thesis was restricted to controlled environments created for research and validation. Exploit execution was confined to researcher-controlled emulator instances and, where necessary, device-aware reasoning about platform behaviour. This restriction was essential because the aim of the study was to observe Android defence behaviour under reproducible conditions rather than to target live systems or third-party infrastructure.

No unauthorised exploitation was performed. The thesis analyses documented exploit cases from public sources and validates only a purposive subset for which sufficient technical detail and legal feasibility exist. Cases lacking an ethically defensible or legally permissible validation path were excluded from experimentation. This boundary ensured

that the study remained focused on academic evaluation of Android privilege boundaries rather than operational exploitation.

Proof-of-concept artefacts, logs, and test materials were handled in a manner intended to minimise risk and prevent unintended spread or misuse. Experimental material was stored and executed only within the isolated research environment, and logs were retained only to the extent necessary for analysis and reproducibility. These precautions were proportionate to the aims of the study and consistent with the controlled-validation methodology adopted in this thesis.

4.6 Validity, Reliability and Limitations

Internal validity is limited by the possibility of misclassifying exploit mechanisms or misinterpreting defensive behaviour. Android privilege escalation often involves chained conditions, indirect enforcement paths, and interacting controls. To reduce this risk, the thesis used a literature-derived taxonomy, structured extraction variables, and triangulation across literature review, comparative documented-case analysis, and controlled validation. Even so, some ambiguity remains where public exploit descriptions are incomplete or where one case may plausibly fit more than one exploit family. (He et al., 2023; Meng et al., 2018)

External validity is constrained by Android ecosystem variation. Security behaviour differs across versions, OEM builds, firmware customisations, and policy configurations, and the thesis does not attempt exhaustive ecosystem coverage. Instead, it aims for analytically representative coverage across major exploit families and selected versions or configurations. Findings should therefore be interpreted as grounded comparative evidence rather than as universal claims about all Android devices. (Y.-T. Lee et al., 2023; Zhang et al., 2015; Yu et al., 2021)

Reliability is strengthened using structured review procedures, explicit exploit-selection criteria, and repeatable emulator-based validation. However, exploit research remains inherently difficult to reproduce fully because some cases depend on patch state, undocumented preconditions, vendor services, or hardware-dependent behaviour. The thesis therefore treats controlled validation as representative testing of selected exploit families rather than as full replication of every historical exploit.

A further limitation concerns data-source bias. Public exploit records vary in quality, and some high-impact exploit chains are only partially disclosed or never disclosed in enough detail for academic analysis. This creates a bias towards well-documented cases that can be classified and compared rigorously. The methodology mitigates this problem through multi-source collection and explicit exclusion criteria, but incomplete coverage remains an acknowledged limitation of the study.

5 Comparative Analysis of Documented Android Privilege Escalation Cases

This chapter compares representative documented Android privilege-escalation cases across the exploit families established in the literature review. It does not claim a statistically complete dataset. Instead, it uses documented cases with clear technical pathways to examine how escalation is constructed, which privilege boundaries are crossed, and which Android controls are expected to respond. This approach is appropriate because the thesis seeks to explain structural patterns in exploit behaviour rather than estimate ecosystem-wide prevalence.

The comparative analysis draws on six exploit families: kernel and driver exploitation, permission and custom-permission abuse, inter-component communication misuse, hidden or non-SDK service API exploitation, vendor and policy inconsistency, and UI-mediated abuse through overlay or accessibility channels. These families were selected because each captures a distinct route across Android privilege boundaries, yet none is sufficient on its own to explain privilege escalation as a whole.

The chapter addresses RQ1 and RQ2 at a comparative level. It first analyses how documented cases construct escalation paths across layers, preconditions, and attack surfaces. It then evaluates how the relevant Android controls are encountered, bypassed, or weakened in those cases. The chapter therefore functions as a bridge between the literature review and the controlled validation chapter by moving from thematic knowledge to structured case comparison.

5.1 Comparative Basis of Documented Cases

The documented cases examined in this chapter were selected because they make the exploit path technically visible. Each case identifies a targeted component, the boundary crossed, the required preconditions, and the privileged effect or security consequence that follows. This criterion excludes vague malware reporting and focuses instead on cases that reveal how Android privilege boundaries fail in practice. The result is a comparative

corpus of well-described escalation routes rather than a broad inventory of Android security incidents.

This comparative basis includes both attack-oriented and defence-oriented documentation. Some cases originate in exploit surveys, firmware analyses, or subsystem studies that expose vulnerabilities directly. Others arise from policy-analysis work or defensive studies that reveal attack opportunities by showing how Android controls are inconsistently enforced. This combination is useful because Android privilege escalation is often visible only when exploit construction and defensive expectation are analysed together.

The chapter does not treat all documented cases as equally representative of the whole ecosystem. Vendor reporting is uneven, proof-of-concept detail varies by source, and some exploit chains remain only partially disclosed. For that reason, the analysis makes structural comparisons rather than prevalence claims. Its empirical value lies in showing recurring exploit logic across well-documented cases, not in asserting exhaustive coverage of all Android privilege-escalation events.

5.2 Exploit Families in Documented Cases

Documented cases show that kernel and driver exploitation remain one of the most consequential routes to escalation because compromise at that layer can neutralise multiple higher controls simultaneously. The exploit survey by Meng et al. (2018) shows that earlier Android exploits often relied on broadly reusable local-root paths, whereas later cases increasingly targeted memory corruption in kernels, drivers, and closely coupled system components. Zhang et al. (2015) reinforce this pattern by showing that commercial root providers shifted towards device-specific exploit variants, and Zhang et al. (2016) demonstrate that customised memory-management subsystems such as ION create privileged attack surfaces outside the clean AOSP baseline. These cases show that low-level escalation remains powerful, but increasingly fragmented.

Permission and update-state cases show that escalation can also occur without immediate kernel compromise. Li et al. (2021) document how custom-permission state changes, inconsistent mappings, and definition errors can produce direct elevation paths. Xing et al. (2014) identify Pileup vulnerabilities in package management during OS updates,

where old permission declarations can become dangerous after version transitions. Earlier permission studies showed overprivilege and permission re-delegation, but these later cases show something more serious: the framework's own state transitions can become the attack surface when permission identity, grouping, or protection level changes unsafely (Felt et al., 2011; Li et al., 2021; Xing et al., 2014).

Inter-component communication cases reveal that Android privilege escalation frequently begins through delegated trust rather than code execution at a lower layer. Chin et al. (2011) show how insecure inter-application communication exposes paths for transitive privilege misuse. PaddyFrog later demonstrates that confused-deputy conditions remain common where exposed components and privileged tasks remain externally reachable (J. Wu et al., 2015). SEALANT and related defensive work confirm that the problem is not incidental. Android's component model repeatedly allows less-privileged callers to induce more-privileged applications to perform restricted operations on their behalf when export settings, caller checks, or ICC policies are weak (Bhandari et al., 2017; Y. K. Lee et al., 2017).

Hidden-API cases show that the service layer is itself a privileged exploit surface. He et al. (2023) demonstrate that hidden non-SDK service APIs can bypass helper-side checks and expose Binder-accessible privileged functionality directly. This differs from ICC misuse in an important way. The problem is not merely that a more-privileged application acts as deputy. The problem is that the trusted public path and the reachable internal path do not enforce the same security assumptions. Documented hidden-API cases therefore reveal a cross-layer inconsistency between API design and service-side enforcement (He et al., 2023).

Vendor and policy cases further expand the exploit landscape. L. Wu et al. (2013) show that vendor customisations materially alter Android's security posture, while Chen et al. (2017) and Yu et al. (2021) show that SEAndroid policy customisation frequently introduces unregulated or overly permissive rules. Y.-T. Lee et al. (2023) add that Scoped Storage reduced many attack operations, but that OEM adoption remained uneven and preserved exploitable conditions in practice. These cases indicate that privilege escalation increasingly depends on how OEMs extend and configure Android, not only on flaws in the AOSP reference design.

UI-mediated cases reveal that attackers can also cross privilege boundaries indirectly through high-impact interface control. Cloak and Dagger shows that two permissions can be combined to control the UI feedback loop and steer sensitive interaction without breaking the kernel boundary first (Fratantonio et al., 2017). Meng et al. (2018) show that screenshot and screen-recording applications use high-privilege interfaces to capture content and automate behaviour, while Chen et al. (2017) and Zhou et al. (2024) show that overlay-based abuse remains effective when application-level protections are not enabled consistently. These cases matter because they demonstrate that privilege escalation may be operational rather than purely nominal: attacker control over input, display, or interaction can create privilege-relevant effects even without direct root acquisition.

5.3 Temporal Development of Documented Cases

The documented cases show a clear temporal shift in Android privilege-escalation behaviour. Earlier cases more often relied on symbolic links, package-management flaws, weak component exposure, and broadly reusable local-root techniques that affected wide device ranges (Armando et al., 2013; Meng et al., 2018; Xing et al., 2014). These attacks exploited an Android ecosystem that was less hardened and less differentiated across devices.

Later cases are more fragmented and technically specialised. Driver exploitation, hidden service APIs, OEM policy drift, and filesystem-policy inconsistency become more prominent as Android hardened its baseline controls. Zhang et al. (2015) and Zhang et al. (2016) show that vendor-specific low-level targets grew in practical importance, whereas He et al. (2023) and Y.-T. Lee et al. (2023) show that later exploitability often depends on service inconsistency or multi-policy misalignment rather than on one broad root primitive. The change is therefore not simply from weak Android to strong Android. It is from generic exploitability towards more selective and layered exploit construction.

This temporal shift has an important interpretive consequence. Android hardening changed attacker strategy, but it did not eliminate privilege escalation. Instead, attackers increasingly exploit whichever layer remains weak in the current platform state, whether that layer is a custom driver, a hidden service API, an exposed deputy component, or a partially adopted storage policy. The documented cases therefore support a dynamic

interpretation of Android privilege escalation: exploit families evolve with the platform, and their persistence depends on cross-layer inconsistency rather than on one static weakness (He et al., 2023; Y.-T. Lee et al., 2023; Meng et al., 2018).

5.4 Targets, Preconditions, and Attack Construction

Across the documented cases, the most important targets are not confined to one layer. Some cases target kernels, drivers, or memory-management subsystems directly, because those layers can invalidate every higher control once compromised (Zhang et al., 2015, 2016). Other cases target framework helpers, package-management logic, or exposed components, where the escalation path depends on trust transfer rather than on direct low-level memory corruption (Chin et al., 2011; He et al., 2023; Xing et al., 2014). This distribution shows that Android privilege escalation is constructed across multiple layers of the stack, not only at the kernel boundary.

The preconditions for escalation are similarly varied. Some cases begin from a low-privileged application foothold with no special permissions beyond code execution. Others depend on already granted capabilities, such as custom permissions, overlay access, or accessibility services (Fratantonio et al., 2017; Li et al., 2021). Still others depend on update state, vendor firmware, or the presence of a reachable deputy or hidden service interface. The cases therefore show that Android privilege escalation does not have one standard starting condition. Its construction depends on which boundary is reachable and how much trust Android or the surrounding software delegates to that boundary.

User interaction is also a meaningful differentiator. Kernel and service-layer cases often require little more than application execution or control over a call path. By contrast, overlay and accessibility abuse frequently depend on persuading the user to grant dangerous capabilities or accept deceptive interface conditions (Fratantonio et al., 2017; Meng et al., 2018; Zhou et al., 2024). This does not make the latter category less relevant. It shows that privilege escalation on Android may be technical, socio-technical, or both, depending on whether the exploit path relies on architectural weakness alone or on a user-mediated transition into a more privileged interaction state.

The construction of exploit chains across these cases also follows a recurring logic. Attackers begin from a constrained foothold, identify a boundary where trust is delegated or weakly enforced, and then use that boundary to obtain stronger capability, persistence, or control. That pattern appears in permission-state mutation, confused deputies, hidden service APIs, and UI-mediated abuse, even though the technical mechanisms differ (Fratantonio et al., 2017; He et al., 2023; Li et al., 2021; J. Wu et al., 2015). The recurring structure of the cases therefore supports the thesis argument that privilege escalation is best understood as boundary traversal across interacting controls rather than as a collection of unrelated bugs.

5.5 Encounters with Security Mechanisms

The documented cases show that Android security mechanisms are meaningful but unevenly effective. Permissions, SEAndroid, storage controls, patching, exported-component restrictions, and hidden-API controls all appear in the documented cases as intended barriers. Yet the cases also show that these mechanisms often fail when enforcement is split across layers or depends on assumptions that no longer hold in practice (Chen et al., 2017; He et al., 2023; Y.-T. Lee et al., 2023). The relevant question is therefore not whether a mechanism exists, but whether it is placed at the boundary traversed by the exploit.

Permission-based controls are strongest when access is direct, stable, and explicitly mediated. They are weaker when privilege emerges from re-delegation, state transition, or capability combination. Felt et al. (2011) show that permission re-delegation allows a less-privileged application to trigger a more-privileged deputy, whereas Li et al. (2021) show that permission-state mutation can create escalation without a conventional low-level exploit. UI-mediated cases deepen the same point by showing that seemingly legitimate permissions can become escalatory when combined operationally (Fratantonio et al., 2017; Meng et al., 2018). These cases indicate that Android permission controls remain necessary, but they are not sufficient where the exploit path relies on interaction between permissions, services, and delegated trust.

SEAndroid and policy-layer controls are also effective only when policy remains coherent. Chen et al. (2017) show that Android protection is only as strong as the deployed policy, and Yu et al. (2021) show that large-scale OEM policy customisation

frequently introduces unregulated rules. Y.-T. Lee et al. (2023) further show that filesystem protection depends on the alignment of DAC, SELinux, and Android-specific storage controls. These cases suggest that policy mechanisms do not fail primarily because mandatory access control is conceptually weak. They fail when the deployed rule set no longer preserves the intended privilege boundary.

Patching and platform evolution clearly reduced many older exploit paths, but the documented cases show that patching is reactive and uneven. Meng et al. (2018) show that later devices resisted many legacy exploits more successfully than earlier devices, while He et al. (2023) show that hidden-API inconsistencies persisted across multiple Android versions despite blacklist and greylist restrictions. The practical implication is that Android's defensive progress is real, but incomplete. Hardening narrows exploitability, yet exploit paths remain where fixes are partial, delayed, or misaligned with the actual service or policy boundary.

The documented cases also support the distinction between prevention and containment. Most Android controls aim to prevent initial escalation, but some work shows that useful defence remains possible after compromise begins. RGBDroid and PREC both show that runtime restriction and post-exploitation control can reduce attacker capability even after elevated privilege has been reached or attempted (Ho et al., 2014; Park et al., 2012). This distinction matters for the documented cases because some exploit paths reach a technically privileged state without necessarily enabling durable or unrestricted attacker control. Defence effectiveness therefore needs to be interpreted across the full exploit path, not only at its entry point.

5.6 Version and Vendor Variation in Documented Cases

The documented cases show that Android privilege escalation is strongly version aware. Earlier cases such as launch-flow abuse, Pileup, and older permission or ICC weaknesses emerged in a platform state where privilege boundaries were less hardened and update transitions were less carefully constrained (Armando et al., 2013; Chin et al., 2011; Xing et al., 2014). Later cases more often exploit hidden APIs, vendor drivers, or multi-policy inconsistency, which indicates that platform hardening changed the point of attack rather than eliminating the problem (He et al., 2023; Y.-T. Lee et al., 2023; Zhang et al., 2015).

Vendor variation is equally important. L. Wu et al. (2013) show that vendor customisations can introduce security regressions, and Zhou et al. (2024) demonstrate that device-driver fragmentation creates security hazards in Android customisations. Later work on SEAndroid and storage policy shows that vendor divergence is not limited to code additions. It also includes rule expansion, inconsistent enforcement, and partial adoption of new protections such as Scoped Storage (Chen et al., 2017; Y.-T. Lee et al., 2023; Yu et al., 2021). The documented cases therefore support the claim that Android privilege escalation cannot be analysed adequately through AOSP architecture alone.

At the same time, the comparative evidence remains uneven. Some vendors, subsystems, and exploit classes are much more visible in public documentation than others. This means that documented cases can show that vendor divergence materially affects exploitability, but they cannot resolve the full scale of that effect across the ecosystem. Even so, the comparative pattern is clear: privilege escalation increasingly depends on specific version states, OEM extensions, and policy configurations. These make cross-version and cross-configuration validation necessary for any serious assessment of Android security behaviour. (Y.-T. Lee et al., 2023; Meng et al., 2018; Yu et al., 2021)

5.7 Interim Answer to RQ1 and RQ2

The comparative analysis of documented cases supports a clear interim answer to RQ1. Android privilege escalation is achieved through several recurring exploit families rather than through one dominant route. The representative cases show escalation through low-level compromise, permission-state abuse, delegated privilege misuse, hidden service APIs, vendor-specific policy inconsistency, and UI-mediated control paths. These cases differ in mechanism, but they share a common structure: attackers move from a constrained foothold to stronger privilege by exploiting a weak or misaligned boundary.

The documented cases also support an interim answer to RQ2. Android's existing security mechanisms are meaningful, but uneven in their practical effect. Permissions, SEAndroid, patching, storage controls, and component restrictions do block many exploit paths when they are applied directly and consistently. Yet the cases repeatedly show that exploit chains remain viable where controls are separated across layers, weakened by OEM customisation, or bypassed through delegated or indirect access paths. The analysis

therefore supports the thesis claim that Android's central weakness lies less in missing defences than in incomplete coordination between them.

These conclusions remain comparative rather than experimental. Documented cases show how escalation is constructed and which controls are implicated, but they do not fully resolve how those exploit families behave on the specific Android versions and configurations tested in this thesis. That question is addressed in the next chapter through controlled validation of selected representative cases.

6 Experimental Validation

The purpose of the chapter is not to discover new vulnerabilities, but to observe whether representative exploit families still succeed, fail, or become contained under current Android conditions. The chapter therefore addresses RQ3 directly by examining how exploit behaviour changes across versions and configurations.

All validation was conducted in an emulator-based testbed centred on Android Studio, the Android Emulator, Android Debug Bridge (adb), and filtered logcat output. This environment was used because it supported repeatability, controlled setup, and consistent observation across selected Android versions. The experiments recorded runtime permission state, application-visible behaviour, filtered logs, installation and launch commands, and screenshots of key outcomes. No physical-device experiments were done for this chapter.

Cases were selected purposively rather than randomly. Selection prioritised exploit-family coverage, technical clarity, identifiable prerequisites, legal feasibility, and expected observable outcomes. The final set covers four representative paths: custom-permission mutation, delegated privilege misuse through ICC, hidden-API reachability, and overlay-mediated UI abuse. Across all cases, outcomes are classified as success, failure, or containment. Success denotes reproduction of the intended privileged effect. Failure denotes inability to reproduce that effect. Containment denotes partial technical reachability without meaningful or durable privilege gain.

6.1 Testbed and instrumentation

The core experimental environment for this chapter was emulator-based. This follows the methodology established earlier in the thesis, which defines the emulator as the primary validation setting because it provides repeatability, controllability, and instrumentation across multiple Android versions. The practical setup centred on Android Studio, the Android Emulator, adb, and filtered logcat output. Proof-of-concept applications were developed as small Android projects and installed through controlled command-line

deployment so that each run could be reset and reproduced cleanly. This design supports consistent comparison between expected and observed exploit behaviour across selected cases and Android versions.

No physical-device experiments are reported in the present set of case studies. The methodology allows physical devices when emulator-based reproduction is insufficient, especially for hardware-dependent or vendor-specific cases, but the experiments implemented here were selected because they could be executed safely and reproducibly in emulated environments. This keeps the focus on application-layer and framework-layer exploit behaviour while preserving methodological consistency with the emulator-first strategy defined in Chapter 4.

The tested environments covered multiple Android versions in order to support version-aware comparison. The custom-permission mutation case was executed across selected emulator configurations to observe whether application update and permission-state mutation produced different outcomes under older and newer platform conditions. The ICC / delegated privilege misuse case was validated on Android 12 and Android 17 emulator configurations to determine whether the same vulnerable deputy path remained effective as the platform version changed. These version choices were intended to provide a manageable but meaningful comparison rather than exhaustive ecosystem coverage, consistent with the thesis scope and methodology.

Logging, monitoring, and success criteria were defined in advance. Each experiment recorded application-visible state, runtime permission state, filtered logcat output, installation and launch commands, and screenshots of key outcomes. The outcome categories used in the case studies are defined in Table 1. This classification reflects the methodological distinction between prevention and containment adopted throughout the thesis and allows each case to be evaluated in terms of whether Android blocked the exploit path entirely, limited it, or allowed it to succeed.

The emulator-based validation environment is shown in Figure 1. It was used to build, deploy, execute, and monitor all proof-of-concept applications under controlled conditions.

Table 1. Operational criteria for classifying experimental outcomes.

Outcome	**Operational criterion**
Success	The tested path produced the intended privileged effect, such as unauthorised access to protected data or accepted unsafe interaction.
Prevention	The attempted path was blocked before the protected operation occurred.
Failure to reproduce	The expected escalatory effect did not occur under the tested setup, even though the experiment executed correctly.
Containment	A technically relevant path remained reachable, but Android or the application prevented meaningful privileged gain.
Partial containment	Some exposed behaviour remained possible, but specific protections reduced or removed the exploit value on protected paths.

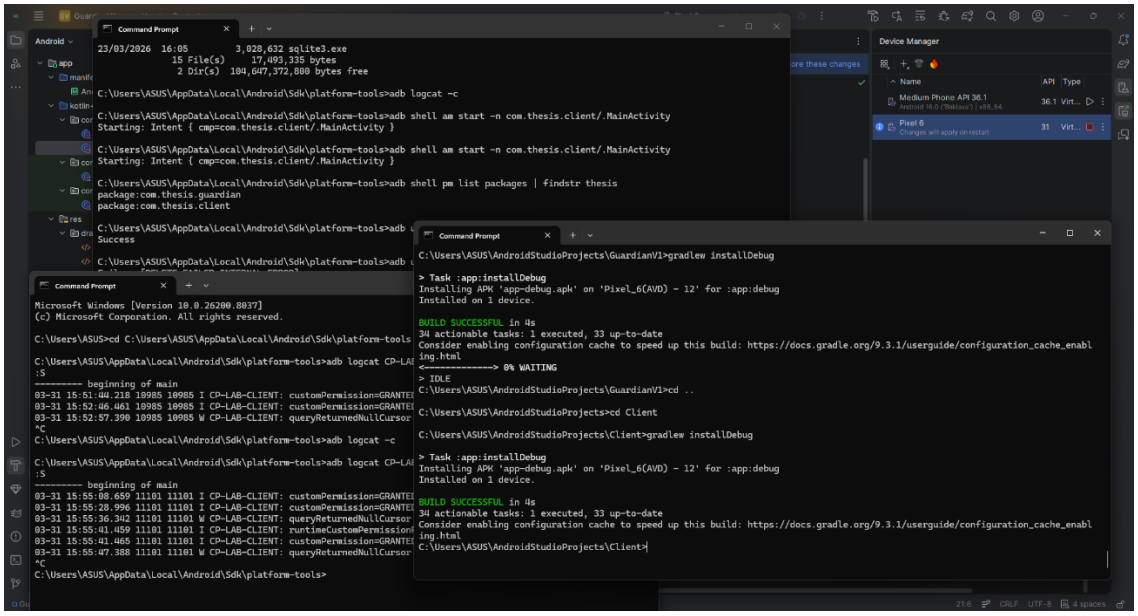

Figure 1. Emulator-based validation testbed with Android Studio, Android Emulator, adb and filtered logcat.

6.2 Case selection rationale

The selected cases were chosen because each represents a distinct exploit family identified in earlier chapters and each can produce an observable result in a controlled emulator environment. The set therefore supports comparison across permission-state change, delegated privilege, hidden service access, and UI-mediated abuse without searching for new vulnerabilities. Prior literature is cited only to anchor each case to its exploit family, whereas the core of the chapter is the observed evidence.

The cases are analytically representative rather than statistically representative. They were selected because each corresponds to a major exploit family identified in the literature review and each tests a different Android privilege boundary. Table 2 summarises the case-selection rationale.

Table 2. Case selection rationale and represented Android privilege boundaries.

Case	**Exploit family represented**	**Android boundary tested**	**Representation**
A	Permission-state / custom-permission abuse	Permission framework and update state	Represents escalation through permission definition inconsistency
B	ICC / confused deputy	Application component boundary	Represents delegated privilege misuse without kernel compromise
C	Hidden API / service-layer access	Framework helper and Binder-facing service boundary	Represents service-layer inconsistency and non-SDK reachability
D	Overlay-mediated UI abuse	UI trust boundary	Represents privilege-relevant interaction abuse through overlay control

6.3 Case studies

Each case study in this chapter follows the same structure. First, the exploit family and targeted component are introduced. Second, the expected preconditions and expected defence interaction are stated using the literature and the case-specific setup. Third, the practical test procedure is described. Fourth, the observed result is presented using screenshots, commands, and filtered log output. Fifth, the case is interpreted as succeeded, prevented, or contained considering both the observed evidence and the broader thesis argument about Android privilege boundaries. Using the same structure for all cases makes the experiments directly comparable and helps ensure that each contributes clearly to the final answer to RQ3.

6.3.1 Case study A: Custom-permission mutation experiment

Case purpose

This case tested whether a controlled custom-permission mutation path would yield an escalatory effect after application update. It was selected because prior work shows that Android custom permissions can become escalation channels when permission ownership, protection level, or group mapping changes are handled unsafely (Li et al., 2021).

Setup and preconditions

The experiment used two applications. Guardian V1 defined com.thesis.guardian.PERMISSION_USE_SECRET as a normal custom permission and exposed a content provider protected by that permission. Client requested the custom permission and also declared CALL_PHONE so that any unsafe transition into a dangerous system permission would be visible. Guardian V2 preserved the package identity, signing context, permission name, and provider authority, but changed the custom permission to dangerous and associated it with the PHONE group.

Procedure

Guardian V1 and Client were installed first. Baseline state was then recorded by checking three things: whether Client held the Guardian custom permission, whether CALL_PHONE was granted or denied, and whether Client could query the protected

provider successfully. Guardian V2 was then installed over Guardian V1, after which the same checks were repeated. The Client application also triggered explicit interaction with the custom permission through its user interface so that any delayed or implicit permission-state change would become visible. The installation, update, reset, and logcat sequence used for this experiment is shown in Figure 2.

```
adb install -r -d
"C:\Users\ASUS\AndroidStudioProjects\GuardianV1\app\build\outputs\
apk\debug\app-debug.apk"
adb install -r
"C:\Users\ASUS\AndroidStudioProjects\Client\app\build\outputs\apk\
debug\app-debug.apk"
adb logcat -c
adb shell am start -n com.thesis.client/.MainActivity
adb logcat CP-LAB-CLIENT:I CP-LAB-GUARD:I AndroidRuntime:E *:S
adb shell am force-stop com.thesis.guardian
adb shell am force-stop com.thesis.client
adb logcat -c
adb install -r -d
"C:\Users\ASUS\AndroidStudioProjects\GuardianV2\app\build\outputs\
apk\debug\app-debug.apk"
adb shell am start -n com.thesis.client/.MainActivity
adb logcat CP-LAB-CLIENT:I CP-LAB-GUARD:I AndroidRuntime:E *:S
```

Figure 2. ADB command sequence for the custom-permission mutation experiment.

Observed result

In the baseline state, Client held the Guardian custom permission, CALL_PHONE remained denied, and the provider query returned TOP_SECRET_V1 value as shown in Figure 3. After the Guardian V2 update, the provider remained reachable and returned TOP_SECRET_V2 value as shown in Figure 4, confirming that the updated application was active and that the custom-permission-protected resource remained accessible. However, CALL_PHONE remained denied before and after the update, and it also remained denied after explicit interaction through the Client application. No silent grant of the dangerous system permission was observed. The logcat output in Figure 5 confirms the same result at runtime: provider access remains available after the update, but CALL_PHONE is still denied.

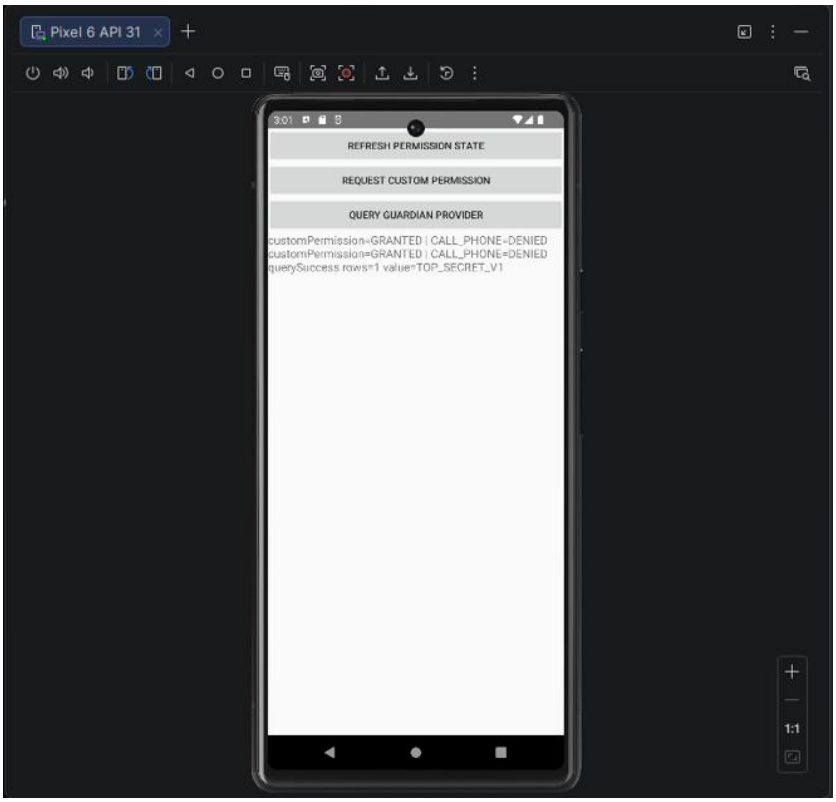

Figure 3. Baseline custom-permission state before the Guardian update.

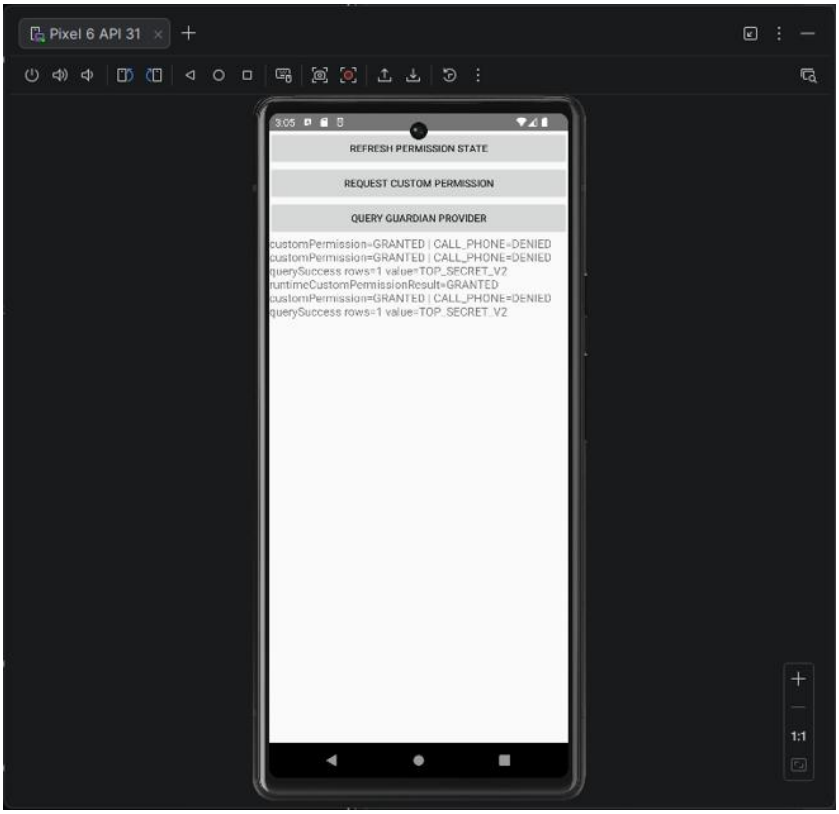

Figure 4. Post-update state after replacing GuardianV1 with GuardianV2.

```
04-01 15:01:10.689 16314 16314 I CP-LAB-CLIENT:
customPermission=GRANTED | CALL_PHONE=DENIED
04-01 15:01:19.570 16353 16353 I CP-LAB-GUARD: SecretProvider
onCreate()
04-01 15:01:19.585 16353 16369 I CP-LAB-GUARD: query() entered
uri=content://com.thesis.guardian.secretprovider/secret
callerUid=10154
04-01 15:01:19.586 16353 16369 I CP-LAB-GUARD: query() returning 1
row
04-01 15:01:19.592 16314 16314 I CP-LAB-CLIENT: querySuccess rows=1
value=TOP_SECRET_V1
04-01 15:05:15.622 16458 16458 I CP-LAB-CLIENT:
customPermission=GRANTED | CALL_PHONE=DENIED
04-01 15:05:25.376 16501 16501 I CP-LAB-GUARD: SecretProvider
onCreate()
04-01 15:05:25.391 16501 16516 I CP-LAB-GUARD: query() entered
uri=content://com.thesis.guardian.secretprovider/secret
callerUid=10154
04-01 15:05:25.401 16501 16516 I CP-LAB-GUARD: query() returning 1
row
04-01 15:05:25.418 16458 16458 I CP-LAB-CLIENT: querySuccess rows=1
value=TOP_SECRET_V2
04-01 15:05:32.892 16458 16458 I CP-LAB-CLIENT:
runtimeCustomPermissionResult=GRANTED
04-01 15:05:32.901 16458 16458 I CP-LAB-CLIENT:
customPermission=GRANTED | CALL_PHONE=DENIED
04-01 15:05:41.813 16501 16516 I CP-LAB-GUARD: query() entered
uri=content://com.thesis.guardian.secretprovider/secret
callerUid=10154
04-01 15:05:41.814 16501 16516 I CP-LAB-GUARD: query() returning 1
```

Figure 5. Logcat evidence for the custom-permission mutation test.

Interpretation

This case is best interpreted as failure to reproduce privilege escalation under the tested conditions. The experiment confirms that custom-permission state change is a meaningful validation surface, but the tested application-update path did not reproduce dangerous-system-permission escalation. The result should be read as a non-reproduction of the stronger escalation path described by Li et al. (2021), because the present case examined an application-update mutation sequence rather than the more specific operating-system update transition discussed in the literature. The evidence therefore suggests that exploitability in this family depends on exact trigger conditions, version state, and update sequence rather than on custom-permission mutation alone. This case contributes primarily to RQ3 by showing that a documented permission-escalation family does not necessarily reproduce under a simplified application-update path.

6.3.2 Case study B: ICC / delegated privilege misuse experiment

Case purpose

This case tested whether a low-privileged application could induce a more-privileged application to perform a restricted contact read through an exported component. It was selected because the confused-deputy literature treats delegated privilege misuse as a recurring Android escalation path (Chin et al., 2011; Y. K. Lee et al., 2017; J. Wu et al., 2015).

Setup and preconditions

The Victim application held READ_CONTACTS and exposed an activity, LeakContactActivity, that read the first available contact and returned it to the caller. The Attacker application held no READ_CONTACTS permission. Two configurations were tested: a vulnerable configuration in which the deputy component remained externally reachable, and a fixed configuration in which that reachability was removed.

Procedure

Victim first allowed local contact access as shown in Figure 6 and confirmed access by reading the first contact as shown in Figure 7. Attacker then confirmed that its own READ_CONTACTS state was denied and attempted a direct query to the Contacts provider. Finally, Attacker launched Victim's LeakContactActivity through an explicit intent and recorded whether contact data were returned. The same procedure was then repeated after the Victim manifest was changed so that the deputy component was no longer externally reachable.

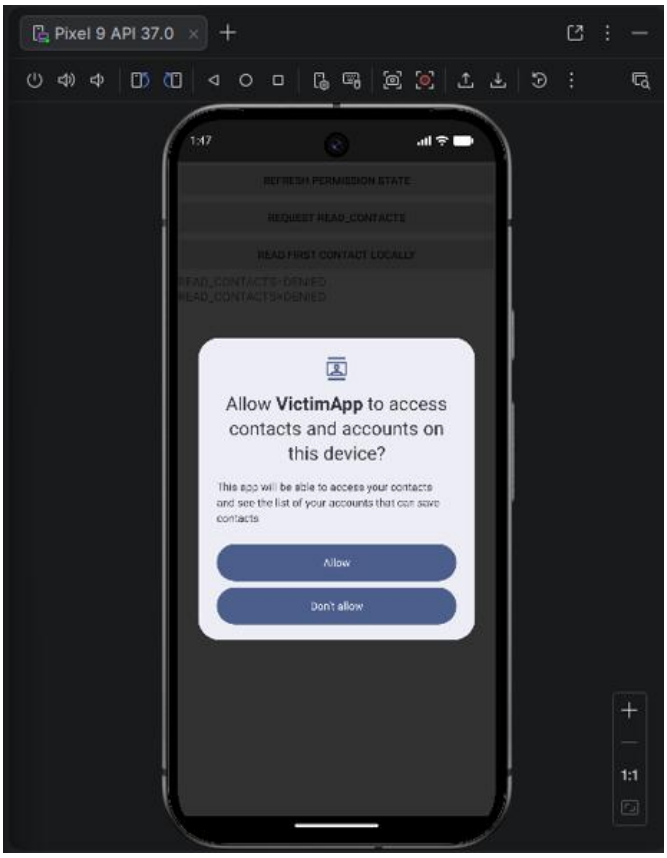

Figure 6. Runtime grant of READ_CONTACTS to the victim app.

Figure 7. Victim app confirming local access to contacts by reading the first contact.

Observed result

In the vulnerable configuration, Attacker had READ_CONTACTS=DENIED, and its direct query failed with a SecurityException as shown in Figure 8. Despite that Figure 9 shows that the exported Victim activity was triggered and returned contact data to Attacker through the deputy path. In the fixed configuration, Attacker could not read contacts directly and could not trigger the deputy component, and no contact data were returned as presented in Figure 10.

Figure 11 provides a clear vulnerable-versus-fixed evidence for the ICC case. Direct contact access is denied in both configurations, but only the vulnerable configuration returns contact data through the victim.

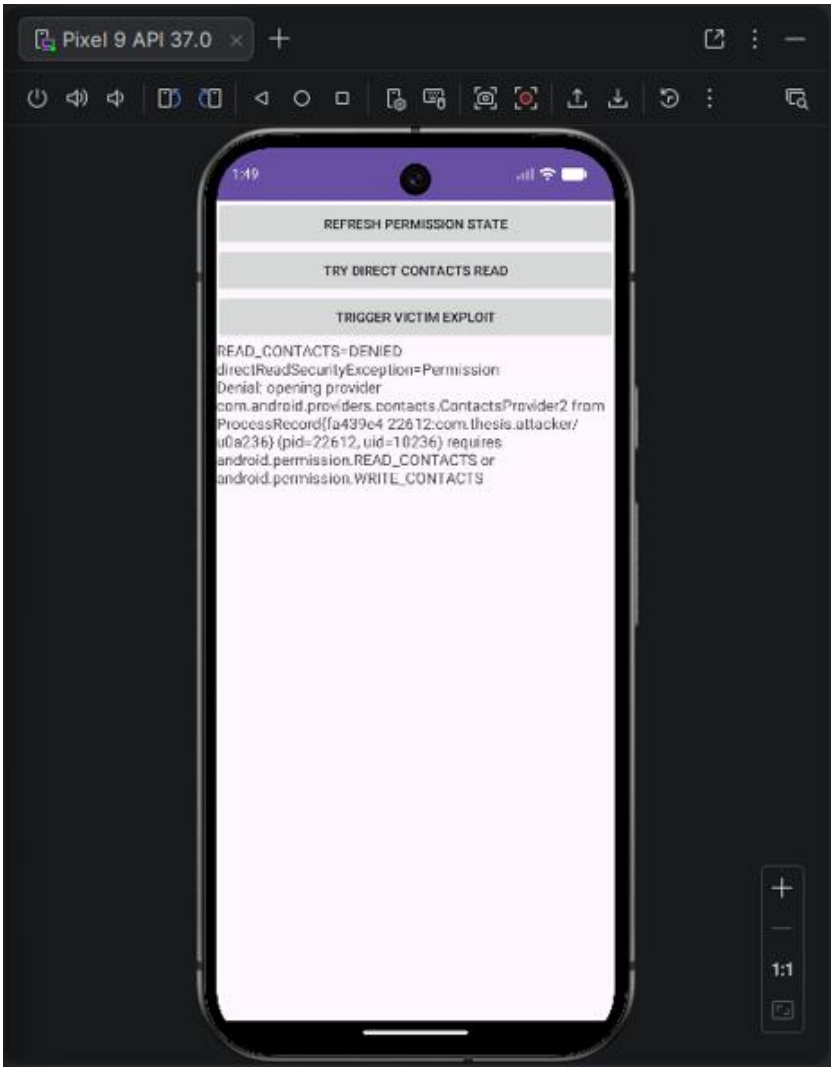

Figure 8. Attacker direct contact query denied by Android permissions.

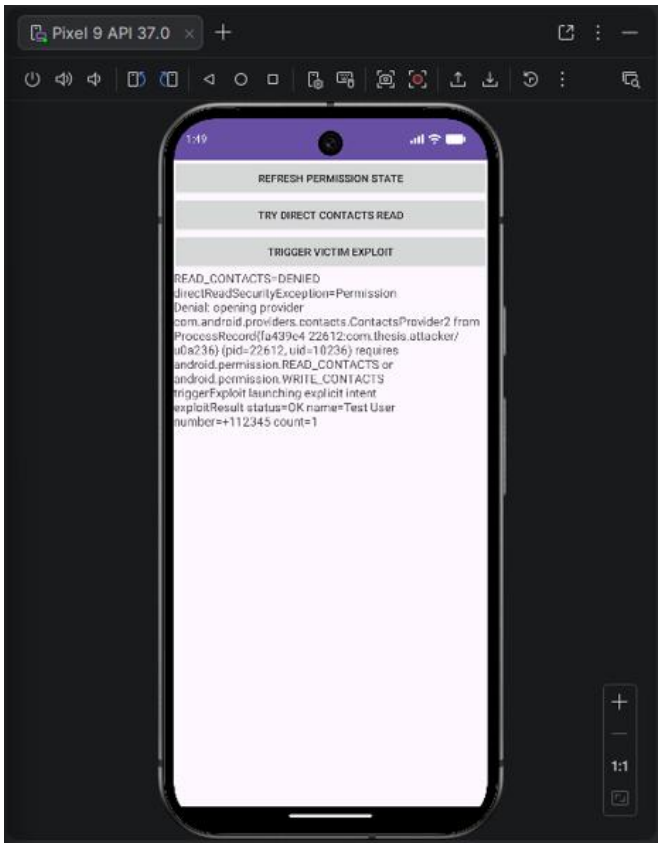

Figure 9. Vulnerable ICC result returning contact data through the victim.

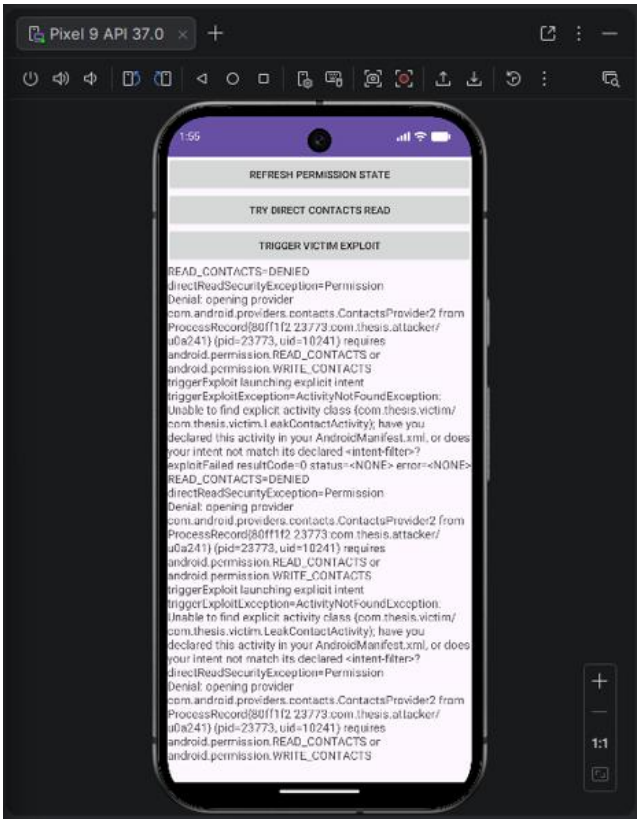

Figure 10. Fixed ICC result after removing the reachable deputy component.

```
04-03 13:48:57.869 22612 22612 I CP-LAB-ATTACKER:
READ_CONTACTS=DENIED
04-03 13:49:14.666 22612 22612 E CP-LAB-ATTACKER:
directReadSecurityException=Permission Denial: opening provider
com.android.providers.contacts.ContactsProvider2 from
ProcessRecord{fa439e4 22612:com.thesis.attacker/u0a236} (pid=22612,
uid=10236) requires android.permission.READ_CONTACTS or
android.permission.WRITE_CONTACTS
04-03 13:49:24.731 22612 22612 I CP-LAB-ATTACKER: triggerExploit
launching explicit intent
04-03 13:49:26.150 22669 22669 I CP-LAB-VICTIM: LeakContactActivity
invoked requested_by=com.thesis.attacker
04-03 13:49:26.218 22669 22669 I CP-LAB-VICTIM: leakSuccess name=Test
User number=+112345 count=1
04-03 13:49:27.415 22612 22612 I CP-LAB-ATTACKER: exploitResult
status=OK name=Test User number=+112345 count=1

04-03 13:52:46.155 23451 23451 I CP-LAB-ATTACKER:
READ_CONTACTS=DENIED
04-03 13:52:59.577 23451 23451 E CP-LAB-ATTACKER:
directReadSecurityException=Permission Denial: opening provider
com.android.providers.contacts.ContactsProvider2 from
ProcessRecord{9c46b14 23451:com.thesis.attacker/u0a241} (pid=23451,
uid=10241) requires android.permission.READ_CONTACTS or
android.permission.WRITE_CONTACTS
04-03 13:53:03.597 23451 23451 I CP-LAB-ATTACKER: triggerExploit
launching explicit intent
04-03 13:53:03.609 23451 23451 E CP-LAB-ATTACKER:
triggerExploitException=ActivityNotFoundException: Unable to find
explicit activity class
{com.thesis.victim/com.thesis.victim.LeakContactActivity}
```

Figure 11. Logcat evidence for the vulnerable and fixed ICC configurations.

Interpretation

This case is best interpreted as success in the vulnerable configuration and prevention in the fixed configuration. Android correctly blocked direct contact access, but the privileged deputy still executed the protected operation when the exposed component remained reachable. Once the component was no longer externally reachable, the exploit path disappeared. The result confirms that delegated privilege misuse depends primarily on application-side exposure and caller validation rather than on bypass of the underlying permission model itself (Y. K. Lee et al., 2017; J. Wu et al., 2015). This case contributes to RQ3 by showing that delegated privilege misuse depends directly on component reachability and application-side access control.

6.3.3 Case study C: Hidden API helper-bypass probe

Case purpose

This case tested whether a third-party application could still reach selected hidden non-SDK APIs and thereby approach the service-layer inconsistency problem identified in the hidden-API literature. It was selected because earlier work shows that helper-side and service-side checks may diverge in Android system services (He et al., 2023).

Setup and preconditions

The experiment used a debuggable probe application, HiddenProbe, with StrictMode.detectNonSdkApiUsage() enabled. The application provided a public baseline and two reflective probes: android.os.SystemProperties.get and android.os.ServiceManager.getService. The case was executed on Android 10, Android 12, and Android 17 emulator configurations.

Procedure

For each tested Android version, the public baseline was executed first. The SystemProperties and ServiceManager reflective probes were then triggered in sequence. HiddenProbe was launched under filtered logcat monitoring, and the cross-version output is shown in Figure 12. The logs recorded whether reflection succeeded, whether Android emitted StrictMode non-SDK API warnings, and whether the result indicated any meaningful privileged system effect.

```
05-17 12:03:37.815  6218  6218 I CP-LAB-HIDDEN:
strictModeNonSdkApiUsage=ENABLED sdk=29
05-17 12:04:33.527  6218  6218 I CP-LAB-HIDDEN: publicBaseline
sdk=29 release=10 debug=true
05-17 12:04:33.534  6218  6218 I CP-LAB-HIDDEN: probeSuccess
api=android.os.SystemProperties.get value=10
05-17 12:04:33.545  6218  6218 I CP-LAB-HIDDEN: probeSuccess
api=android.os.ServiceManager.getService
binderClass=android.os.BinderProxy
05-17 12:04:33.552  6218  6290 W CP-LAB-HIDDEN:
strictModeNonSdkViolation=NonSdkApiUsedViolation:
Landroid/os/ServiceManager;-
>getService(Ljava/lang/String;)Landroid/os/IBinder;

05-17 12:09:54.800  6561  6561 I CP-LAB-HIDDEN:
strictModeNonSdkApiUsage=ENABLED sdk=31
05-17 12:10:09.392  6561  6561 I CP-LAB-HIDDEN: publicBaseline
sdk=31 release=12 debug=true
05-17 12:10:09.404  6561  6561 I CP-LAB-HIDDEN: probeSuccess
api=android.os.SystemProperties.get value=12
05-17 12:10:09.477  6561  6561 I CP-LAB-HIDDEN: probeSuccess
api=android.os.ServiceManager.getService
binderClass=android.os.BinderProxy
05-17 12:10:09.475  6561  6592 W CP-LAB-HIDDEN:
strictModeNonSdkViolation=NonSdkApiUsedViolation:
Landroid/os/ServiceManager;-
>getService(Ljava/lang/String;)Landroid/os/IBinder;

05-17 12:19:02.019 21124 21124 I CP-LAB-HIDDEN:
strictModeNonSdkApiUsage=ENABLED sdk=37
05-17 12:19:12.395 21124 21124 I CP-LAB-HIDDEN: publicBaseline
sdk=37 release=17 debug=true
05-17 12:19:12.500 21124 21124 I CP-LAB-HIDDEN: probeSuccess
api=android.os.SystemProperties.get value=17
05-17 12:19:12.559 21124 21124 I CP-LAB-HIDDEN: probeSuccess
api=android.os.ServiceManager.getService
binderClass=android.os.BinderProxy
05-17 12:19:12.700 21124 21228 W CP-LAB-HIDDEN:
strictModeNonSdkViolation=NonSdkApiUsedViolation:
Landroid/os/ServiceManager;-
>getService(Ljava/lang/String;)Landroid/os/IBinder;
```

Figure 12. Cross-version HiddenProbe logcat evidence.

Observed result

Across the tested configurations, the public API baseline executed normally and both reflective probes succeeded. SystemProperties.get returned the platform release value on each run, while ServiceManager.getService returned android.os.BinderProxy. However, the ServiceManager probe also triggered a StrictMode NonSdkApiUsedViolation. The repeated runs after setting hidden_api_policy to value 1 produced the same observable pattern, so the experiment demonstrates reflective reachability but not a stronger

privileged service-side effect. A representative Android 17 HiddenProbe run is shown in Figure 13.

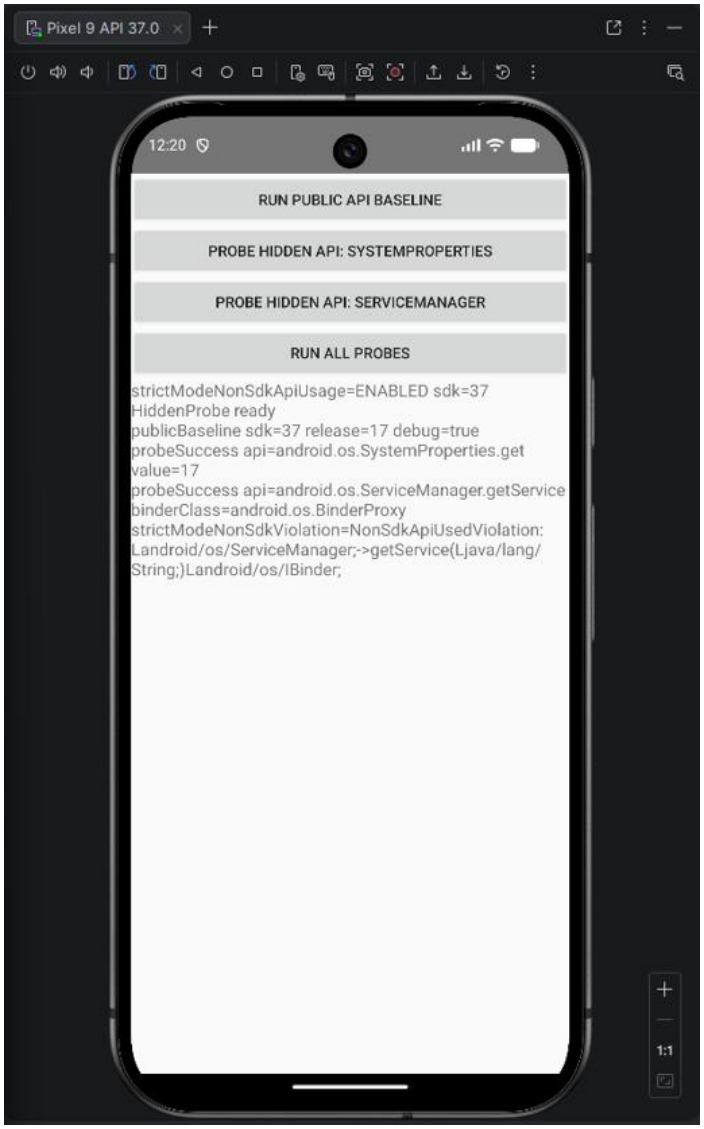

Figure 13. HiddenProbe result screen showing successful reflective probes and a StrictMode warning.

Interpretation

This case is best interpreted as containment rather than full success. Selected hidden APIs remained reachable through reflection, but the probe did not produce the stronger privileged effect associated with helper-bypass exploitation in the literature. The result therefore suggests that hidden-API reachability persisted in the tested configurations, while the practical severity of the service-layer inconsistency was reduced relative to earlier documented cases (He et al., 2023). This case contributes to RQ3 by distinguishing hidden-API reachability from practical privileged effect.

6.3.4 Case study D: Overlay-mediated UI abuse and containment

Case purpose

This case tested whether a visible third-party overlay could still interfere with user interaction on newer Android versions and whether current controls contained that behaviour. It was selected because overlay-mediated abuse remains a documented route to privilege-relevant UI control when sensitive interaction surfaces are left insufficiently protected (Chen et al., 2017; Fratantonio et al., 2017; Zhou et al., 2024).

Setup and preconditions

The experiment used two proof-of-concept applications. OverlayLab requested overlay permission, as shown in Figure 14, and displayed a visible, non-touchable overlay strip using TYPE_APPLICATION_OVERLAY. TargetLab exposed three screens: an unprotected action screen, a screen with obscured-touch filtering, and a screen that invoked overlay hiding for protected windows. The case was executed on Android 10, Android 12, and Android 17.

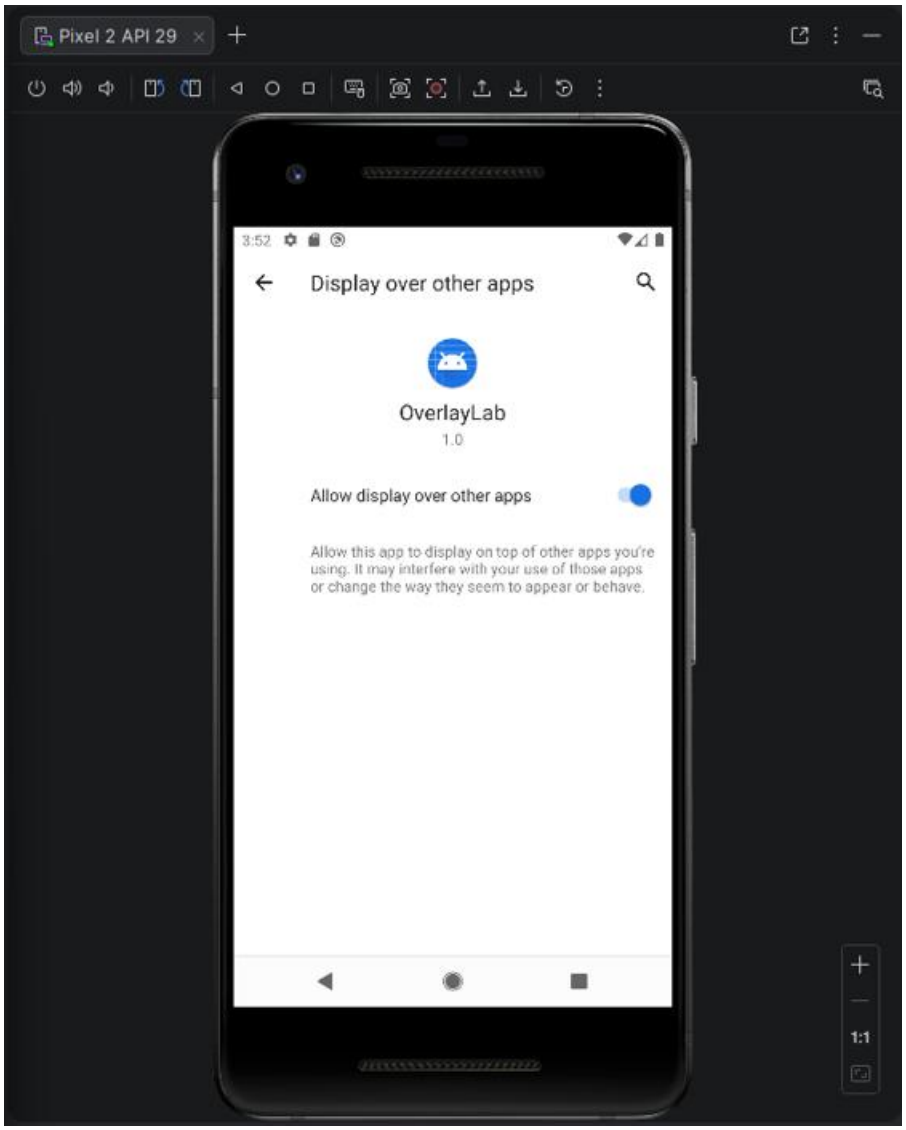

Figure 14. The OverlayLab application was granted permission to display over other apps.

Procedure

Overlay permission was granted to OverlayLab, after which the overlay was displayed over each TargetLab screen. The experiment first tested the unprotected screen shown in Figure 15, then the filtered screen shown in Figure 16, and finally the hide-overlay screen shown in Figure 17. For Android 12 and Android 17, the overlay was tested in both broader and reduced forms to distinguish complete occlusion from partial obscuration more clearly.

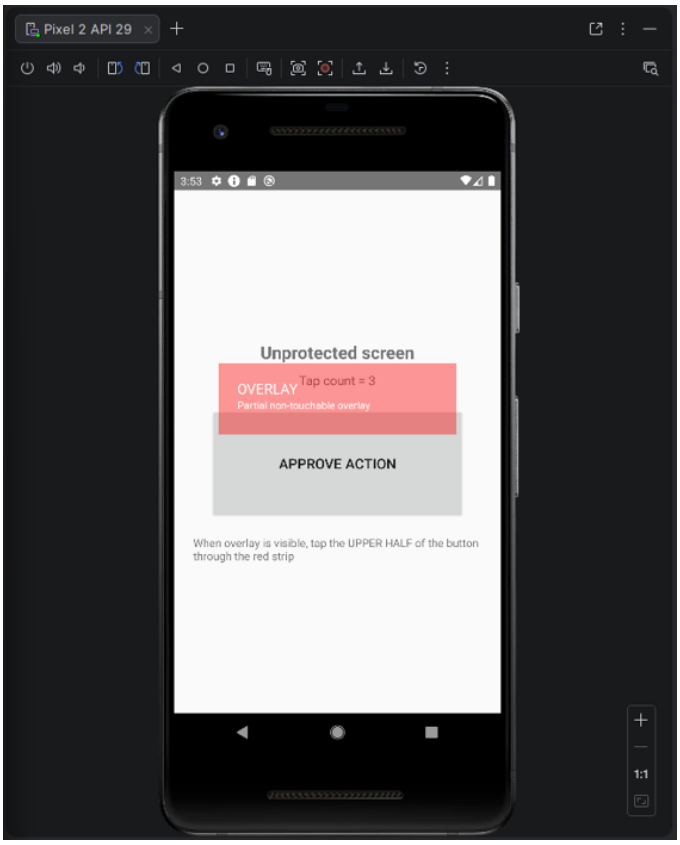

Figure 15. Overlay interference on the unprotected TargetLab screen.

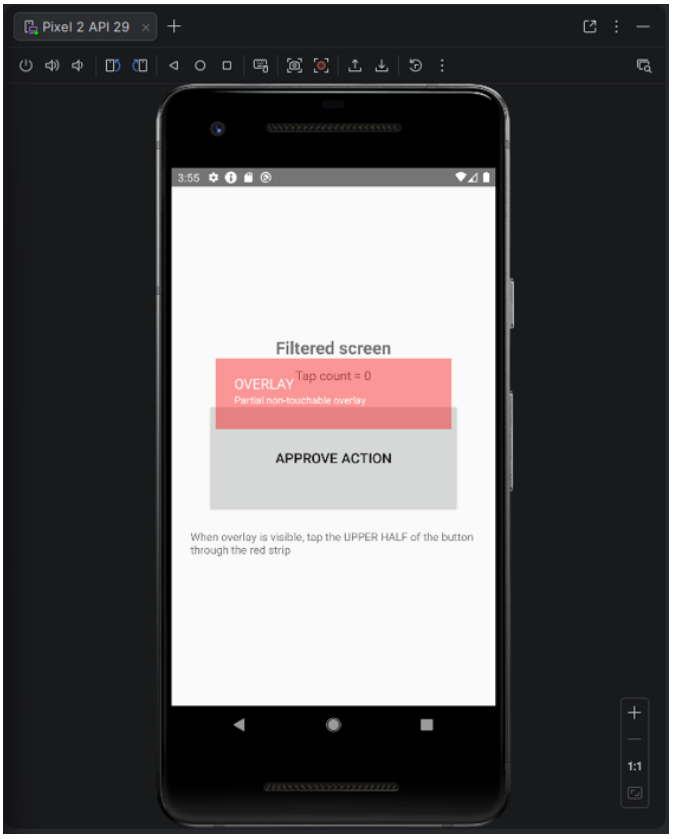

Figure 16. Filtered TargetLab screen rejecting obscured touch events.

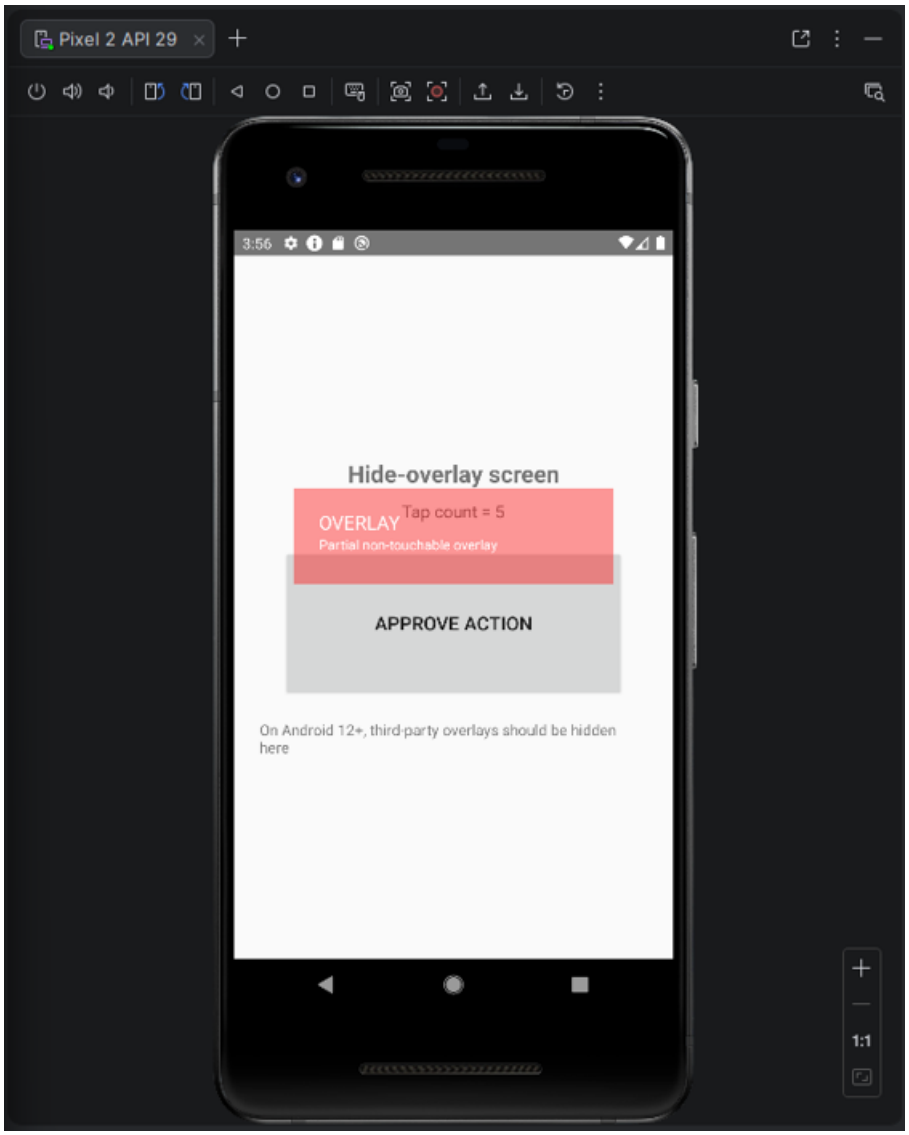

Figure 17. Hide-overlay screen with the overlay visible on Android 10.

Observed result

On Android 10, the unprotected screen accepted obscured touches and logged FLAG_WINDOW_IS_OBSCURED, while the filtered screen rejected the same condition. The hide-overlay screen did not provide effective additional protection in the tested Android 10 configuration, as shown in Figure 18. On Android 12, the unprotected screen remained exposed under partial obscuration and logged FLAG_WINDOW_IS_PARTIALLY_OBSCURED, whereas the filtered screen rejected obscured touches and the hide-overlay screen restored unobscured interaction, as shown in Figure 19 and Figure 20. On Android 17, the same broad pattern remained: the unprotected screen accepted obscured and partially obscured touches, while the filtered and hide-overlay paths prevented effective abuse.

```
05-16 15:52:47.253  6639  6639 D OverlayLab: Overlay added
05-16 15:53:37.573  6760  6760 D TargetLab: Unprotected ACTION_DOWN
flags=FLAG_WINDOW_IS_OBSCURED
05-16 15:53:37.636  6760  6760 D TargetLab: Unprotected click
accepted, tapCount=1
05-16 15:53:37.793  6760  6760 D TargetLab: Unprotected ACTION_DOWN
flags=FLAG_WINDOW_IS_OBSCURED
05-16 15:53:37.866  6760  6760 D TargetLab: Unprotected click
accepted, tapCount=2
05-16 15:53:38.712  6760  6760 D TargetLab: Unprotected ACTION_DOWN
flags=FLAG_WINDOW_IS_OBSCURED
05-16 15:53:38.773  6760  6760 D TargetLab: Unprotected click
accepted, tapCount=3

05-16 15:54:15.959  6760  6760 D TargetLab:
SecureActionButton.onFilterTouchEventForSecurity
flags=FLAG_WINDOW_IS_OBSCURED allowed=false
05-16 15:54:20.268  6760  6760 D TargetLab:
SecureActionButton.onFilterTouchEventForSecurity
flags=FLAG_WINDOW_IS_OBSCURED allowed=false

05-16 15:56:13.786  6760  6760 D TargetLab: setHideOverlayWindows
unavailable on this Android version
05-16 15:56:16.581  6760  6760 D TargetLab: HideOverlay ACTION_DOWN
flags=FLAG_WINDOW_IS_OBSCURED
05-16 15:56:16.797  6760  6760 D TargetLab: HideOverlay click
accepted, tapCount=1
05-16 15:56:16.801  6760  6760 D TargetLab: HideOverlay ACTION_DOWN
flags=FLAG_WINDOW_IS_OBSCURED
05-16 15:56:17.659  6760  6760 D TargetLab: HideOverlay ACTION_DOWN
flags=FLAG_WINDOW_IS_OBSCURED
05-16 15:56:17.844  6760  6760 D TargetLab: HideOverlay click
accepted, tapCount=2
05-16 15:56:18.576  6760  6760 D TargetLab: HideOverlay ACTION_DOWN
flags=FLAG_WINDOW_IS_OBSCURED
05-16 15:56:18.744  6760  6760 D TargetLab: HideOverlay click
accepted, tapCount=3
```

Figure 18. Android 10 overlay logcat evidence for unprotected, filtered, and hide-overlay screens.

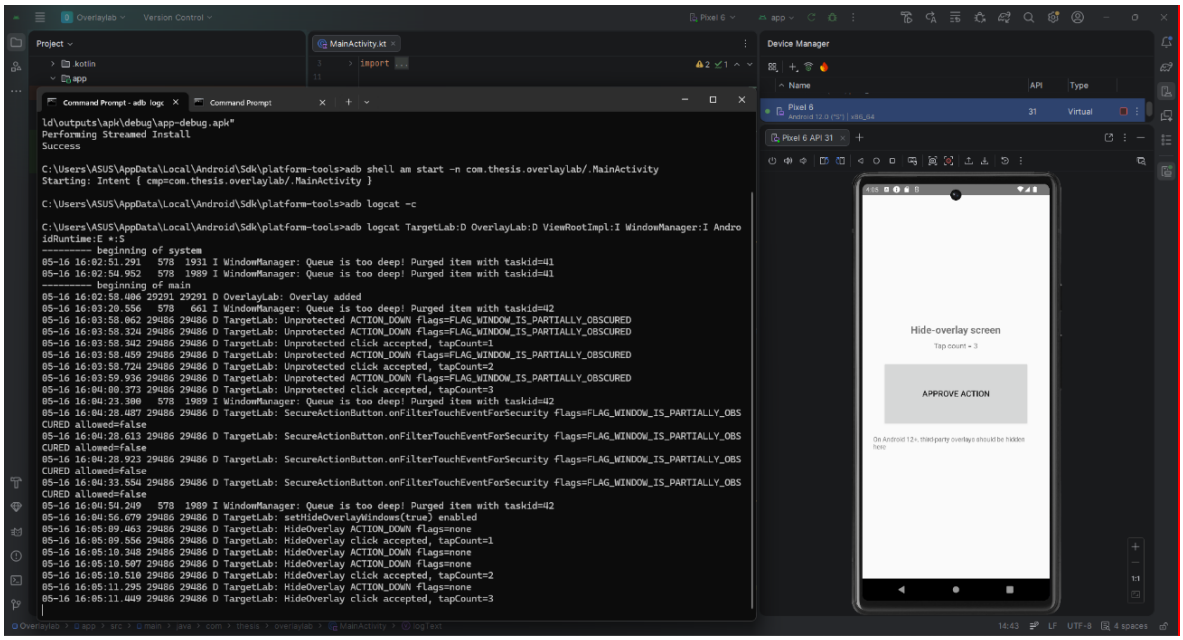

Figure 19. Overlay containment on Android 12 using the hide-overlay window setting.

```
05-16 16:02:58.406 29291 29291 D OverlayLab: Overlay added
05-16 16:03:58.062 29486 29486 D TargetLab: Unprotected ACTION_DOWN
flags=FLAG_WINDOW_IS_PARTIALLY_OBSCURED
05-16 16:03:58.324 29486 29486 D TargetLab: Unprotected ACTION_DOWN
flags=FLAG_WINDOW_IS_PARTIALLY_OBSCURED
05-16 16:03:58.342 29486 29486 D TargetLab: Unprotected click
accepted, tapCount=1
05-16 16:03:58.459 29486 29486 D TargetLab: Unprotected ACTION_DOWN
flags=FLAG_WINDOW_IS_PARTIALLY_OBSCURED
05-16 16:03:58.724 29486 29486 D TargetLab: Unprotected click
accepted, tapCount=2
05-16 16:03:59.936 29486 29486 D TargetLab: Unprotected ACTION_DOWN
flags=FLAG_WINDOW_IS_PARTIALLY_OBSCURED
05-16 16:04:00.373 29486 29486 D TargetLab: Unprotected click
accepted, tapCount=3

05-16 16:04:28.487 29486 29486 D TargetLab:
SecureActionButton.onFilterTouchEventForSecurity
flags=FLAG_WINDOW_IS_PARTIALLY_OBSCURED allowed=false
05-16 16:04:28.613 29486 29486 D TargetLab:
SecureActionButton.onFilterTouchEventForSecurity
flags=FLAG_WINDOW_IS_PARTIALLY_OBSCURED allowed=false

05-16 16:04:56.679 29486 29486 D TargetLab:
setHideOverlayWindows(true) enabled
05-16 16:05:09.463 29486 29486 D TargetLab: HideOverlay ACTION_DOWN
flags=none
05-16 16:05:09.556 29486 29486 D TargetLab: HideOverlay click
accepted, tapCount=1
05-16 16:05:10.348 29486 29486 D TargetLab: HideOverlay ACTION_DOWN
flags=none
05-16 16:05:10.507 29486 29486 D TargetLab: HideOverlay ACTION_DOWN
flags=none
05-16 16:05:10.510 29486 29486 D TargetLab: HideOverlay click
accepted, tapCount=2
05-16 16:05:11.295 29486 29486 D TargetLab: HideOverlay ACTION_DOWN
flags=none
05-16 16:05:11.449 29486 29486 D TargetLab: HideOverlay click
accepted, tapCount=3
```

Figure 20. Android 12 overlay logcat evidence for unprotected, filtered, and hide-overlay screens.

Interpretation

This case is best interpreted as partial containment rather than full prevention. Across all tested versions, an unprotected screen remained exposed to overlay-mediated interaction in the lab setup. At the same time, obscured-touch filtering and overlay hiding consistently reduced or removed exploit value on protected screens. The case therefore shows that Android's practical resilience depends heavily on whether the relevant defence is enabled at the attacked interface boundary (Fratantonio et al., 2017; Zhou et al., 2024).

This case contributes to RQ3 by showing that overlay abuse remains possible on unprotected screens but is reduced where interface-level protections are enabled.

6.4 Cross-case findings

Taken together, the case studies show that Android privilege escalation remains boundary-specific and configuration-sensitive rather than uniformly vulnerable or uniformly fixed. The custom-permission case did not reproduce the expected escalatory effect under the tested update path. The ICC case succeeded only while the privileged deputy remained externally reachable. The hidden-API probe showed reflective reachability without a meaningful privileged outcome. The overlay case showed persistent exposure on unprotected screens but effective reduction of risk where application-side or platform-side protections were enabled. Table 3 summarises the outcome classification across the experiments.

A common pattern across the cases is that exploitability depends less on abstract exploit family alone than on where enforcement is placed. Where the relevant control was applied directly at the attacked boundary, the result shifted towards prevention or containment. Where a weak or exposed boundary remained reachable, the exploit path persisted. The cases therefore support the broader thesis interpretation that Android's security is shaped by the coordination of layers rather than by the existence of any single mechanism.

A second common pattern is the distinction between prevention and containment. Not every technically observable path yielded a meaningful privileged effect. The hidden-API and overlay cases are especially important here because both retained partial technical reachability while reducing practical attacker gain under the tested conditions. These results support evaluating Android security in terms of exploit outcome, not only exploit attempt.

Table 3. Experimental outcomes across the four case studies.

Case	**Main observed result**	**Interpretation**
Custom-permission mutation	Provider access persisted, but CALL_PHONE stayed denied	Failure to reproduce escalation
ICC / delegated privilege misuse	Exported deputy leaked contact data; fixed configuration blocked it	Success, then prevention
Hidden API probe	Reflection reached selected APIs, but no privileged action followed	Containment
Overlay-mediated UI abuse	Unprotected screens accepted obscured input; filtering and hide-overlay reduced impact	Partial containment

6.5 Interim answer to RQ3

RQ3 asked how selected privilege-escalation exploits behave across Android versions and configurations. The experimental evidence shows that the tested exploit families behaved unevenly. Some paths remained effective when a locally weak boundary persisted, some failed when the relevant exposure was removed, and some remained technically reachable without yielding a meaningful privileged outcome.

The results therefore support three conclusions. First, exploit behaviour is version-aware, but not in a simple linear progression from vulnerable to fixed. Second, configuration remains decisive, especially where exported components, obscured-touch filtering, or overlay hiding alter the result directly. Third, Android’s practical security is better understood through the distinction between success, failure, and containment than through a binary vulnerable-versus-patched model.

In that sense, the experiments support the broader thesis argument that Android's main weakness lies less in missing controls than in incomplete coordination between them. Newer versions improve security, but the practical result still depends heavily on whether the correct defence is activated at the boundary being traversed.

6.6 Experimental limitations

These findings should be interpreted within the limits of an emulator-first validation strategy. The use of Android Studio, the Android Emulator, adb, and filtered logcat supported repeatability and clear instrumentation, but it also limits direct generalisation to the full Android ecosystem. The emulator-only design especially limits claims about OEM-specific Android behaviour. Stock emulator images do not reproduce vendor kernels, proprietary drivers, customised SEAndroid policies, preinstalled privileged services, or vendor-specific storage implementations. Therefore, this thesis experimentally validates application-layer and framework-layer boundary behaviour, while vendor-specific conclusions remain grounded primarily in the literature review and documented-case analysis.

The case set was selected purposively rather than statistically. This was appropriate for representative validation across exploit families, but it means the chapter does not estimate prevalence across all Android vulnerabilities or vendor environments. Some historically important exploit classes also remain difficult to reproduce in stock emulator images because they depend on firmware state, vendor services, patch level, or hardware-specific behaviour.

A further limitation is that some proof-of-concepts were deliberately scoped for safe academic validation rather than maximal offensive effect. The hidden-API probe did not attempt dangerous privileged actions, and the overlay case was implemented as a benign obscuration lab rather than a credential-theft chain. These choices strengthen methodological defensibility, but they also mean that the chapter demonstrates exploit conditions and defence responses more directly than maximum attacker impact.

7 Discussion

This chapter interprets the combined findings of the thesis and answers the research questions at an integrated level. The earlier chapters established the relevant exploit families, compared representative documented cases, and validated selected exploit paths under controlled conditions. The purpose of the present chapter is therefore not to restate those results in detail, but to explain what they collectively show about Android privilege boundaries, defensive coherence, and practical security behaviour.

The combined evidence supports a consistent central argument. Android privilege escalation is best understood as a cross-layer security problem in which exploitability depends on the interaction between application configuration, framework behaviour, policy state, vendor customisation, and platform version. Android's defences are meaningful, but their effect is uneven when checks are separated across layers or applied inconsistently at the attacked boundary. This interpretation supports the thesis claim that Android's main weakness lies less in the absence of controls than in incomplete coordination between them.

7.1 Integrated answer to RQ1

RQ1 asked how attackers successfully escalate privileges on Android and what techniques they use. The combined thesis evidence shows that Android privilege escalation is not one stable exploit type, but a family of boundary-crossing mechanisms. The most important recurring routes identified in this thesis are kernel and driver exploitation, permission and custom-permission abuse, delegated privilege misuse through ICC, hidden or non-SDK service API exploitation, policy inconsistency driven by vendor customisation, and UI-mediated abuse through overlay or accessibility control.

What unifies these otherwise different exploit families is not their surface, but their logic. In each case, attackers begin from a constrained foothold and exploit a boundary where trust is delegated, incompletely enforced, or inconsistently coordinated with neighbouring controls. Some routes begin at the kernel or driver layer, whereas others begin in

application logic, service exposure, or permission-state handling. The combined result is that Android privilege escalation is best interpreted as a path through interacting enforcement domains rather than as a single bug class. (He et al., 2023; Li et al., 2021; Meng et al., 2018)

The thesis therefore refines a purely kernel-centric interpretation of Android escalation. Low-level compromise remains important, but it does not explain the full landscape. The evidence instead shows that sandbox bypass may begin above the kernel boundary and only later extend into stronger privilege domains. This supports a broader architectural interpretation in which Android's privilege boundaries are repeatedly tested wherever one layer assumes that another has already enforced the relevant check.

7.2 Integrated answer to RQ2

RQ2 asked how effective Android's existing security mechanisms are in mitigating privilege-escalation attacks. The answer emerging from the thesis is that Android's defences are effective in important ways, but uneven in their practical operation. Permissions, SEAndroid, patching, storage controls, component restrictions, and platform hardening all reduce exploitability. However, no single mechanism consistently explains defensive success across all exploit families.

The strongest defensive situations were those in which enforcement was applied directly at the attacked boundary. The experiments showed this clearly in the ICC and overlay cases, where removing component reachability or enabling obscured-touch protection changed the result immediately. The literature points to the same pattern in service-side validation, SEAndroid policy quality, and filesystem control. Android's defences work best when they are direct, contextually relevant, and consistently aligned with the privilege transition being attempted. (Chen et al., 2017; He et al., 2023; Y.-T. Lee et al., 2023)

The weakest defensive situations were those involving cross-layer inconsistency. Permission abuse remained possible where state transitions or custom definitions became unsafe. Hidden-API risk emerged where helper-side and service-side checks diverged. Vendor customisation weakened otherwise sound protections where OEM policies, drivers, or storage rules no longer preserved AOSP assumptions. The evidence therefore

shows that Android defences fail most often not because one control is entirely absent, but because several controls no longer enforce the same trust boundary coherently.

A further conclusion is that defence effectiveness should be interpreted through the distinction between prevention and containment. Some exploit paths were blocked before privilege gain, while others remained technically reachable but no longer yielded meaningful attacker control. This is especially important for interpreting modern Android hardening, because later versions often narrow or constrain exploit outcomes without making the underlying family disappear entirely. In that sense, Android security should be judged not only by whether escalation begins, but also by whether the platform limits what follows. (Ho et al., 2014; Park et al., 2012)

7.3 Integrated answer to RQ3

RQ3 asked how selected privilege-escalation exploits behave across Android versions and configurations. The controlled validation showed that exploit behaviour is strongly version-aware and configuration-sensitive, but not in a simple linear progression from vulnerable to fixed. Newer Android versions did improve security, yet the improvement often appeared as narrowing, repositioning, or containment rather than as the full disappearance of an exploit family.

More specifically, the custom-permission case failed to reproduce escalation, the ICC case succeeded only under vulnerable component exposure, the hidden-API case showed containment, and the overlay case showed partial containment.

The experiments showed that configuration was often as decisive as version. Exported component exposure, obscured-touch filtering, overlay hiding, and the exact form of a permission-state transition materially changed exploit outcome even when the surrounding platform remained similar. This means Android privilege-escalation behaviour cannot be inferred reliably from architecture or version label alone. The practical outcome depends on whether the relevant control is enabled at the local boundary being traversed.

The broader implication is that Android hardening should be interpreted as a problem of defensive coherence rather than as a sequence of isolated fixes. If one layer improves while a neighbouring boundary remains exposed or inconsistently enforced, exploitability

may persist in altered form. The thesis therefore concludes that version change matters, but its real security value depends on whether platform controls, application logic, and policy state continue to align across the exploit path.

7.4 Comparison with prior literature

The thesis broadly confirms prior literature, but also refines its interpretation. Earlier exploit surveys argued that Android privilege escalation was shifting from generic low-level exploitability towards more device-specific, service-specific, and policy-specific paths (Meng et al., 2018). The findings here support that pattern. Kernel-centric cases remain important, but they no longer explain the whole landscape. The comparative analysis and controlled validation both show that application logic, delegated privilege, hidden APIs, and UI conditions are equally important to understanding how sandbox bypass occurs in practice.

The thesis also aligns with permission-focused research but extends it by embedding permission abuse within a broader system argument. Earlier work established overprivilege, permission re-delegation, and custom-permission weakness as important Android security problems (Felt et al., 2011; Li et al., 2021). The present findings confirm that permission boundaries remain central but show that their practical significance depends on how they interact with service interfaces, update state, delegated execution, and application configuration. In this sense, permission weakness is not isolated. It is one component of a wider cross-layer escalation pattern.

The findings further support prior work on service inconsistency and policy divergence. He et al. (2023) showed that helper-side and service-side enforcement can diverge, while policy research showed that OEM modification and multi-policy interaction can reopen exploit conditions even when individual controls appear sound (Chen et al., 2017; Y.-T. Lee et al., 2023; Yu et al., 2021). The thesis confirms both claims and strengthens them by showing that their practical significance becomes clearer when literature synthesis, documented case comparison, and experimental validation are considered together rather than separately.

Where this thesis extends earlier work is in the integration of evidence types. Much prior research examines one exploit family, one subsystem, or one defensive mechanism at a

time. This thesis instead combines thematic synthesis, comparative documented cases, and controlled validation under a shared analytical framework. Its contribution is therefore not a new exploit or a new defence tool, but a more coherent explanation of how Android privilege boundaries fail, how they are partially restored, and why prevention and containment must be distinguished when evaluating platform security.

7.5 Implications

The findings have three main implications. First, Android's sandbox should not be treated as a self-sufficient security property. Its practical strength depends on a wider set of mediating controls, including permissions, Binder-facing services, SEAndroid policy, storage controls, and UI trust conditions. A security model that treats sandboxing as one isolated boundary will therefore overestimate platform resilience.

Second, layered defence remains a valid principle only when the layers are coherent. The thesis repeatedly showed that protections work best when neighbouring controls enforce the same trust assumptions. Permissions are weaker when service logic bypasses them. Policy is weaker when OEM customisation changes its meaning. UI protections are weaker when sensitive surfaces remain unprotected despite platform support. The findings therefore suggest that defensive layering without cross-layer consistency can produce a false sense of security.

Third, policy coherence should be treated as a core security property rather than as a maintenance detail. Many of the most important failures examined in this thesis emerged not from one missing control, but from disagreement between adjacent controls about who should be trusted, where a check should occur, or which capability should apply. This implies that Android hardening should prioritise consistency across permission handling, service validation, policy enforcement, and vendor modification rather than adding isolated mitigations that remain locally correct but globally misaligned.

7.6 Limitations of the combined findings

The combined findings remain bounded by the quality and visibility of the available evidence. Public exploit information is uneven, some exploit chains are only partially disclosed, and others may never enter the public record in sufficient technical detail for

academic analysis. The thesis therefore privileges well-documented cases that can be classified, compared, and validated rigorously. This supports strong analytical conclusions, but not exhaustive coverage of all Android privilege-escalation routes.

A second limitation lies in controlled validation. The experiments were designed as representative validation rather than exhaustive replication. This made the study safer, more reproducible, and better aligned with its research purpose, but it also means that some exploit families were observed as structured proof-of-concept conditions rather than as maximum attacker-impact scenarios. Hardware-dependent, OEM-specific, and patch-state-sensitive exploit classes remain especially difficult to validate comprehensively in emulator-based environments.

A third limitation concerns ecosystem generalisation. The literature and comparative analysis both show that Android security varies across vendors, firmware, drivers, and policy configurations, yet the thesis does not attempt complete vendor-comparative coverage. The combined findings are therefore best interpreted as robust evidence of recurring patterns in Android privilege-boundary failure rather than as a ranking of all devices or OEMs.

A final limitation concerns future platform change. The thesis includes recent Android versions in controlled validation, but Android security continues to evolve. Hidden-API policy, permission semantics, service structure, overlay behaviour, and OEM obligations may change again in later releases. The conclusions should therefore be read as a strong account of Android privilege escalation up to the versions examined here, not as a timeless statement about every future platform state.

8 Conclusion

This thesis examined whether Android's layered security architecture mitigates privilege escalation in practice. The central problem was that Android combines sandboxing, permissions, SEAndroid, kernel hardening, and service-level controls, yet documented exploit paths still cross intended privilege boundaries. The thesis therefore treated Android privilege escalation not as an architectural abstraction, but as an empirical question about how those controls behave under real exploit conditions.

The study addressed that problem through three connected phases. It first synthesised the literature on Android privilege escalation and sandbox bypass, then compared representative documented cases across major exploit families, and finally validated selected exploit paths in controlled environments across versions and configurations. This structure allowed the thesis to move from prior knowledge to comparative case evidence, to direct observation of Android defence behaviour.

8.1 Summary of findings

The thesis found that Android privilege escalation is best understood as a cross-layer problem rather than as a single exploit class. Across the literature, comparative case analysis, and controlled validation, the most important routes were kernel and driver exploitation, permission and custom-permission abuse, inter-component communication misuse, hidden or non-SDK service API exploitation, vendor and policy inconsistency, and UI-mediated abuse. These routes differ in surface, but they share a common logic: each exploits a weak or misaligned boundary between less-trusted and more-trusted execution contexts.

The thesis also found that Android's security mechanisms are meaningful but uneven in their practical effect. Permissions, SEAndroid, patching, storage controls, and application- or service-level protections do reduce exploitability. However, their effectiveness declines when enforcement becomes context-insensitive, split across layers, or weakened by application design and vendor customisation. The central conclusion is

therefore that Android's main weakness lies less in the absence of defences than in incomplete coordination between them.

The controlled validation further showed that exploit behaviour is strongly version-aware and configuration-sensitive. Newer Android versions do improve security, but improvement often appears as narrowing or containment rather than as the complete disappearance of an exploit family. In several cases, the decisive factor was not the platform version alone, but whether the relevant protection was enabled at the attacked boundary.

8.2 Answers to the research questions

For RQ1, the thesis shows that attackers escalate privileges on Android through several recurring exploit families rather than through one dominant route. Successful escalation typically depends on boundary traversal across permissions, components, services, policy layers, or lower-level subsystems rather than on one isolated flaw.

For RQ2, the thesis shows that Android's existing security mechanisms are effective in important ways, but not uniformly. They work best when enforcement is direct, local to the attacked boundary, and consistent with neighbouring controls. They fail most often where trust is delegated indirectly or where different layers no longer enforce the same assumptions.

For RQ3, the thesis shows that exploit behaviour changes across Android versions and configurations, but not in a simple vulnerable-versus-fixed pattern. Newer versions often constrain exploitability, yet configuration choices such as exported components, overlay protection, or permission-state handling remain decisive to the practical outcome. The experimental contribution is therefore not a claim that all Android versions are uniformly secure or insecure, but that exploitability depends on the tested boundary, platform version, and configuration.

8.3 Contributions

This thesis makes three main contributions. First, it provides a cross-layer explanation of Android privilege escalation that integrates exploit construction, defence behaviour,

version change, and configuration sensitivity within one analytical framework. This helps explain why Android privilege escalation persists even when individual controls appear locally sound.

Second, the thesis contributes a structured way of comparing Android privilege-escalation families across layers, preconditions, and defensive encounters. By treating documented cases and controlled validation under one shared analytical lens, the study clarifies how exploit routes differ technically while still reflecting the same deeper problem of privilege-boundary inconsistency.

Third, the thesis contributes an evaluation perspective that distinguishes prevention from containment. This distinction is important because Android hardening does not always remove an exploit path entirely. In many cases, it changes the exploit outcome from direct success to limited or non-useful attacker gain. That distinction is necessary for realistic evaluation of Android security.

8.4 Recommendations

The findings support three main recommendations. First, Android hardening should place stronger emphasis on enforcement at the actual execution boundary. Helper-only validation is weaker than service-side validation, and declarative protection is weaker when state can drift across updates, mappings, or indirect call paths.

Second, OEM customisation should be treated as security-sensitive engineering rather than as a deployment detail. Changes to drivers, SEAndroid policy, storage rules, or preinstalled services can materially alter exploitability, even where the AOSP baseline appears better protected.

Third, Android security evaluation should distinguish clearly between prevention and containment. A platform may reduce the practical severity of an exploit without removing its technical reachability altogether. Future evaluation practices should therefore report not only whether an exploit path exists, but also what effective attacker capability remains after platform and application-level defences respond.

8.5 Future work

Future work should expand comparative evaluation across OEM firmware and policy variants because vendor divergence remains one of the least uniformly documented but most security-relevant aspects of Android privilege escalation. Stronger vendor-comparative work would help clarify whether many escalation paths are primarily architectural, vendor-driven, or the product of interaction between both.

Future research should also examine newer Android releases and newer service or policy surfaces as the platform evolves. The present thesis includes recent versions in controlled validation, but Android security remains a moving target, particularly in hidden-API policy, UI protection, service architecture, and firmware-level customisation.

A further priority is the development of stronger cross-layer verification methods. The literature and experiments both indicate that Android's most important weaknesses often arise when permissions, services, policy rules, and application logic no longer align. Future tools should therefore verify the coherence of these controls together rather than analyse them only in isolation.

8.6 Final remarks

Android privilege escalation should not be framed only as a collection of isolated exploits. It should be understood as a recurring test of whether Android's layered defences remain coherent when confronted with real adversarial paths. The evidence presented in this thesis shows that those defences are meaningful, but not uniformly aligned across application, service, policy, vendor, and version boundaries.

The conclusion of this thesis is therefore that Android security has improved, but its practical resilience still depends on whether the right protections are applied consistently at the boundaries attackers actually traverse. Incomplete coordination between layers remains the central condition that allows privilege escalation to persist.

References

Armando, A., Merlo, A., Migliardi, M., & Verderame, L. (2013). Breaking and fixing the Android Launching Flow. *Computers & Security*, *39*, 104–115. https://doi.org/10.1016/j.cose.2013.03.009

Au, K. W. Y., Zhou, Y. F., Huang, Z., & Lie, D. (2012). PScout: Analyzing the Android permission specification. *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 217–228. https://doi.org/10.1145/2382196.2382222

Bhandari, S., Jaballah, W. B., Jain, V., Laxmi, V., Zemmari, A., Gaur, M. S., Mosbah, M., & Conti, M. (2017). Android inter-app communication threats and detection techniques. *Computers & Security*, *70*, 392–421. https://doi.org/10.1016/j.cose.2017.07.002

Chen, H., Li, N., Enck, W., Aafer, Y., & Zhang, X. (2017). Analysis of SEAndroid Policies: Combining MAC and DAC in Android. *Proceedings of the 33rd Annual Computer Security Applications Conference*, 553–565. https://doi.org/10.1145/3134600.3134638

Chin, E., Felt, A. P., Greenwood, K., & Wagner, D. (2011). Analyzing inter-application communication in Android. *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, 239–252. https://doi.org/10.1145/1999995.2000018

Demissie, B. F., Ghio, D., Ceccato, M., & Avancini, A. (2016). Identifying Android inter app communication vulnerabilities using static and dynamic analysis.

Proceedings of the International Conference on Mobile Software Engineering and Systems, 255–266. https://doi.org/10.1145/2897073.2897082

Fang, Z., Han, W., & Li, Y. (2014). Permission based Android security: Issues and countermeasures. *Computers & Security*, *43*, 205–218. https://doi.org/10.1016/j.cose.2014.02.007

Felt, A. P., Chin, E., Hanna, S., Song, D., & Wagner, D. (2011). Android permissions demystified. *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 627–638. https://doi.org/10.1145/2046707.2046779

Feng, H., & Shin, K. G. (2016). Understanding and defending the binder attack surface in Android. *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 398–409. https://doi.org/10.1145/2991079.2991120

Fratantonio, Y., Qian, C., Chung, S. P., & Lee, W. (2017). Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. *2017 IEEE Symposium on Security and Privacy (SP)*, 1041–1057. https://doi.org/10.1109/SP.2017.39

He, Y., Gu, Y., Su, P., Sun, K., Zhou, Y., Wang, Z., & Li, Q. (2023). A Systematic Study of Android Non-SDK (Hidden) Service API Security. *IEEE Transactions on Dependable and Secure Computing*, *20*(2), 1609–1623. https://doi.org/10.1109/TDSC.2022.3160872

Ho, T.-H., Dean, D., Gu, X., & Enck, W. (2014). PREC: Practical root exploit containment for android devices. *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, 187–198. https://doi.org/10.1145/2557547.2557563

Lee, Y. K., Bang, J. Y., Safi, G., Shahbazian, A., Zhao, Y., & Medvidovic, N. (2017). A SEALANT for Inter-App Security Holes in Android. *2017 IEEE/ACM 39th*

International Conference on Software Engineering (ICSE), 312–323. https://doi.org/10.1109/ICSE.2017.36

Lee, Y.-T., Chen, H., Enck, W., Vijayakumar, H., Li, N., Qian, Z., Petracca, G., & Jaeger, T. (2023). PolyScope: Multi-Policy Access Control Analysis to Triage Android Scoped Storage. *IEEE Transactions on Dependable and Secure Computing*, 1–14. https://doi.org/10.1109/TDSC.2023.3310402

Li, R., Diao, W., Li, Z., Du, J., & Guo, S. (2021). Android Custom Permissions Demystified: From Privilege Escalation to Design Shortcomings. *2021 IEEE Symposium on Security and Privacy (SP)*, 70–86. https://doi.org/10.1109/SP40001.2021.00070

Mathew, R. (2012). STUDY OF PRIVILEGE ESCALATION ATTACK ON ANDROID AND ITS COUNTERMEASURES. *International Journal of Engineering Science and Technology*.

Meng, H., Thing, V. L. L., Cheng, Y., Dai, Z., & Zhang, L. (2018). A survey of Android exploits in the wild. *Computers & Security*, *76*, 71–91. https://doi.org/10.1016/j.cose.2018.02.019

Nauman, M., Khan, S., & Zhang, X. (2010). Apex: Extending Android permission model and enforcement with user-defined runtime constraints. *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 328–332. https://doi.org/10.1145/1755688.1755732

Park, Y., Lee, C., Lee, C., Lim, J., Han, S., Park, M., & Cho, S.-J. (2012). *RGBDroid: A Novel Response-Based Approach to Android Privilege Escalation Attacks*.

Reshetova, E., Bonazzi, F., & Asokan, N. (2017). SELint: An SEAndroid policy analysis tool. *Proceedings of the 3rd International Conference on Information Systems Security and Privacy*, 47–58. https://doi.org/10.5220/0006126600470058

Zhang, H., She, D., & Qian, Z. (2015). Android Root and its Providers: A Double-Edged Sword. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 1093–1104. https://doi.org/10.1145/2810103.2813714

Zhang, H., She, D., & Qian, Z. (2016). Android ION Hazard: The Curse of Customizable Memory Management System. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 1663–1674. https://doi.org/10.1145/2976749.2978320

Zhou, H., Wu, S., Qian, C., Luo, X., Cai, H., & Zhang, C. (2024). Beyond the Surface: Uncovering the Unprotected Components of Android Against Overlay Attack. *Proceedings 2024 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. https://doi.org/10.14722/ndss.2024.24035

Wijesekera, P., Egelman, S., Wagner, D., & Beznosov, K. (2015). *Android Permissions Remystified: A Field Study on Contextual Integrity*.

Wu, J., Cui, T., Ban, T., Guo, S., & Cui, L. (2015). PaddyFrog: Systematically detecting confused deputy vulnerability in Android applications: PaddyFrog: systematically detecting confused deputy vulnerability in Android applications. *Security and Communication Networks*, *8*(13), 2338–2349. https://doi.org/10.1002/sec.1179

Wu, L., Grace, M., Zhou, Y., Wu, C., & Jiang, X. (2013). The impact of vendor customizations on android security. *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security - CCS '13*, 623–634. https://doi.org/10.1145/2508859.2516728

Xing, L., Pan, X., Wang, R., Yuan, K., & Wang, X. (2014). Upgrading Your Android, Elevating My Malware: Privilege Escalation through Mobile OS Updating. *2014*

IEEE Symposium on Security and Privacy, 393–408. https://doi.org/10.1109/SP.2014.32

Yu, D., Yang, G., Meng, G., Gong, X., Zhang, X., Xiang, X., Wang, X., Jiang, Y., Chen, K., Zou, W., Lee, W., & Shi, W. (2021). SEPAL: Towards a Large-scale Analysis of SEAndroid Policy Customization. *Proceedings of the Web Conference 2021*, 2733–2744. https://doi.org/10.1145/3442381.3450007

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Kristjan Tamm

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Bypassing Android’s Sandbox Security - A Deep Dive into Privilege Escalation Exploits” supervised by Matthew James Sorell.
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

17.05.2026

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.