

**EXTRACTING HIDDEN, PROTECTED, AND DELETED USER INTERACTION
CONTEXTS FROM APPLICATIONS AND AI SYSTEMS**

A Dissertation
Presented to
The Academic Faculty

By

Haichuan Xu

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering
Department of Electrical and Computer Engineering

Georgia Institute of Technology

May 2026

© Haichuan Xu 2026

EXTRACTING HIDDEN, PROTECTED, AND DELETED USER INTERACTION CONTEXTS FROM APPLICATIONS AND AI SYSTEMS

Thesis committee:

Dr. Brendan Saltaformaggio, Advisor
School of Cybersecurity and Privacy and
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Saman Zonouz
School of Cybersecurity and Privacy and
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Michael Specter
School of Cybersecurity and Privacy and
School of Computer Science
Georgia Institute of Technology

Dr. Frank Li
School of Cybersecurity and Privacy and
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Ozgur Ozmen
School of Computing and Augmented Intelligence
Arizona State University

Date approved: April 3, 2026

To my parents, Huaiqing Xu and Daihong Liu.

And to the love of my life, Sirui Han.

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Brendan Saltaformaggio. Thank you for giving me the freedom to pursue my own research directions while guiding me with remarkable vision toward great research ideas before they fully emerge. At the same time, you provided tireless, hands-on technical mentorship and insight, demonstrating through your own dedication how to execute research projects. Most importantly, you taught me how to tell a compelling story and transform technical depth into clear contributions. The lessons I learned from you extend far beyond this dissertation.

I am sincerely grateful to my lab mates in the CyFI Lab, past and present, such as Runze Zhang, Mingxuan Yao, David Oygenblik, Ranjita Pai Sridhar, Yiting Sun, and Prof. Jeman Park. More than collaborators, you were companions through the endless nights of debugging, deadlines, and revisions. Those shared struggles and victories made this journey not only productive but also meaningful.

I dedicate this dissertation to my father, Huaqing Xu. Your commitment to my education laid the foundation for everything I have achieved. You taught me logic, reasoning, and what it truly means to be an engineer — disciplined, and persistent. Who I am today is defined by the example you set.

I also dedicate this work to my mother, Daihong Liu. You always believed in me, especially during the most challenging and lost times when I doubted myself. Your constant encouragement and unconditional support carried me through moments of frustration and loss. Most importantly, you taught me how to slow down, take care of myself, and appreciate the quiet, beautiful moments in life amidst all the challenges.

Finally, I dedicate this dissertation to the love of my life, Sirui Han. Your patience, understanding, and unwavering support sustained me throughout this journey. We stood side by side through the most challenging days and shared the happiest moments of my life. This achievement is as much yours as it is mine.

TABLE OF CONTENTS

Acknowledgments	iv
List of Tables	ix
List of Figures	xi
Chapter 1: Introduction	1
1.1 Dissertation Statement	1
1.2 Thesis and Contributions	2
1.2.1 DVa	2
1.2.2 SOMBRA	3
1.2.3 ORISA	4
1.3 Dissertation Organization	4
Chapter 2: DVa: Extracting Victims and Abuse Vectors from Android Accessi- bility Malware	6
2.1 Overview	9
2.1.1 Abusing Android a11y Service	9
2.1.2 Uncovering 0-Day Abuse Evidence from a11y Malware	10
2.1.3 Threat Model	13
2.2 Methodology	13

2.2.1	Victim Detection	14
2.2.2	Abuse Vector Detection	17
2.2.3	Persistence Mechanism Detection	20
2.3	Validating My Techniques	21
2.3.1	Implementation	21
2.3.2	Validation Setup	21
2.3.3	Validation Results	22
2.3.4	Coverage Assessment	24
2.4	Findings	25
2.4.1	Targeted Victims	26
2.4.2	Abuse Vectors	27
2.4.3	Persistence Mechanisms	29
2.5	Case Studies	30
2.5.1	a11y 2FA Stealer	30
2.5.2	a11y Ransomware	31
2.5.3	Defenses Against a11y Malware	32
2.6	Discussion	33
2.7	Prerequisites for Analysis	34
2.7.1	Packed Malware and Dynamic Code Loading (DCL)	35
2.7.2	Anti-Dynamic Techniques	35

Chapter 3: Lock the Door But Keep the Window Open: Extracting App-Protected Accessibility Information from Browser-Rendered Websites 42

3.1	Background	44
-----	----------------------	----

3.2	Exposing Mobile Browser Users' Sensitive Information	45
3.2.1	Leakage Types	46
3.2.2	Attack Workflow	49
3.2.3	Attack Prerequisites	49
3.3	Pinpointing Browser-Side Leakage of a11y Information	50
3.3.1	App-Side a11y Model	50
3.3.2	Browser-Side a11y Model	51
3.3.3	Finding the Mismatch Between App-Side and Browser-Side a11y Protection	52
3.4	Evaluation	60
3.4.1	Dataset & Experiment Setup	60
3.4.2	Browser-Side a11y Leakage in Real Benign Services	61
3.4.3	Security Impact	64
3.4.4	Developers' App-Side and Browser-Side a11y Protection Adoption Comparison	66
3.5	Case Studies	69
3.6	Discussion	71
3.7	Validation	73

**Chapter 4: Recovering and Rehosting Mobile Local LLM Conversations and
Contexts via Memory Forensics 77**

4.1	Overview	79
4.1.1	Key Components of Local LLMs	80
4.1.2	Threat Model	82
4.2	Problem and Opportunity	83

4.3	Methodology	85
4.3.1	Model Binary Extraction From Disk	86
4.3.2	Batch Tokens Recovery and Detokenization	87
4.3.3	Active and Forgotten Context Extraction from KV Cache	89
4.3.4	Active KV Extension	91
4.3.5	Session Rehosting	93
4.4	Evaluation	94
4.4.1	Tokens & KV Cache Recovery	95
4.4.2	Context Retention of Rehosted Sessions	98
4.4.3	Semantic Consistency of Rehosted Sessions	101
4.5	Case Study	104
4.6	Discussion	107
4.6.1	Extension to Other Inference Engines	107
4.6.2	Obfuscation and Anti-Forensics Techniques	107
Chapter 5: Related Work		109
Chapter 6: Conclusion		112
References		113
Vita		131

LIST OF TABLES

2.1	A11y capabilities, generic a11y techniques, and a11y-empowered abuse vectors in the context of victim apps.	36
2.2	Malware’s a11y-empowered persistence mechanisms, their triggering conditions, and their a11y behaviors.	37
2.3	Validation of the victims, abuse vectors, and persistence mechanisms detected by DVa together with AVClass2 labels of the top-10 a11y malware families.	38
2.4	Comparison of victims, abuse vectors, and persistence mechanisms detected by DVa versus FlowDroid.	39
2.5	Categories of victim apps, a11y malware that target them, and abuse vectors statistics.	40
2.6	A11y-empowered persistence mechanisms used by the top-10 a11y malware families.	41
2.7	Effectiveness of the newest security patch and data-flow defense in eliminating a11y malware behaviors.	41
3.1	Element Type Rules SOMBRA Adopt To Match Web Page Elements To Their Android App Element Counterparts And Whether They Contain a11y Text.	58
3.2	SOMBRA’s Discovered a11y Leakage In Benign Services’ Websites Rendered In Chrome, Firefox, Brave, And Edge Browsers.	61
3.3	App-side Protected a11y Information Leakage Category In Top Services’ Websites Rendered With Chrome.	65
3.4	Developers’ App-side And Browser-side a11y Information Protection Adoption Types.	66

3.5	Validation Of SOMBRA’s Detection Of App-side Protected Views And Browser-side Exposed Views.	73
4.1	ORISA’s Recovery of Batch Tokens and KV Cache Within and Beyond the Sessions’ Active Context Window.	96
4.2	Rehosted Sessions’ Retention of Prompted Knowledge Within and Beyond the Active Context Window.	99
4.3	Comparison of Response Behaviors Between Original and Rehosted Sessions Under Forensic Interrogation.	102

LIST OF FIGURES

2.1	Overview of malware workflow to abuse the Android a11y service. Steps used to parse and manipulate a victim app’s GUI by an a11y service registered by an attacker is shown in (a). The a11y attacker’s actions are highlighted in red. A sample implementation logic of the malicious a11y service to achieve the abuse illustrated in (a) is shown in (b).	7
2.2	Dynamically loaded automatic transaction abuse routine targeting the Chime banking app extracted by DVa.	11
3.1	Protections against a11y information leakage for native Android app users. .	46
3.2	An a11y malware’s workflow to steal mobile browser users’ sensitive information.	49
3.3	App-side a11y protection model. Android native views can adopt both static protection labels and dynamic a11y modification handlers. They are directly interpreted by the Android OS.	50
3.4	Browser-side a11y protection model. Web page DOM elements can only contain static ARIA labels. The elements with labels need to be translated by browsers and then interpreted by the Android OS.	52
3.5	Klarna’s a11y implementation of the user bank account page in both the app and the mobile website.	75
3.6	Password leaked in Chrome-rendered Dollar General login page. Declaring alternative ARIA labels is ineffective at protecting browser-side a11y information.	76
4.1	Overview of key components that enable inference in a local LLM app’s chat session.	81

4.2	IRIS’s offline model download screen and chat screen, allowing users to use custom open-source models for completely local inference.	82
4.3	The size of KV cache in memory with respect to conversation turns. Although dereferenced after chat session deletion in IRIS UI, the KV cache still physically remains in memory.	84
4.4	Rehosting a deleted conversation context into a live session allows investigators to interrogate and extract forensics evidence.	84
4.5	Overview of ORISA’s design. ORISA takes a mobile LLM app’s memory and app storage as input, conducts model extraction, batch tokens recovery, KV cache recovery, active KV extension, and session rehosting to output an app-agnostic rehosted session representing the previous app session’s context.	85
4.6	Concrete key-value pair knowledge fed into the LLMs to simulate session context.	95
4.7	Chatter UI is one of the most popular native mobile LLM chat app. It allows users to customize model sampler parameters (top left) to fine-tune response behaviors, load custom local LLMs (bottom left), and instantiate custom chat characters in chat sessions(right).	105
4.8	Recovered deserialized tokens show that the Chatter UI app incorporates user-defined character description prompts into system prompts and feeds them to the inference engine as permanent tokens not to be evicted.	106

CHAPTER 1

INTRODUCTION

1.1 Dissertation Statement

Many forensics failures today share a uniform root cause: critical interaction context – how malware acted on a device, what a software exposed, and what user actions were taken – is transient and difficult to reconstruct afterwards. On mobile platforms, Android’s accessibility (a11y) service intentionally crosses application sandboxes to support assistive technologies [1], but these cross-app visibility and action capabilities have also been widely abused by real-world malware to steal information and automate fraud [2, 3, 4]. At the same time, user interactions are increasingly split across multiple front ends (native apps and browser-rendered websites), creating additional paths through which sensitive UI state can be exposed. Finally, the rise of on-device local LLM applications facilitates high-stakes planning and communication without cloud logging, making traces covert [5, 6, 7].

For a11y abuse, prior work proposed data-flow constraints and sandboxing-style defenses [8, 9] and methods to detect misuse of a11y services in apps [10, 11, 12, 13], yet these approaches do not provide victim-specific evidence of what a malware sample can do in the context of the apps installed on a particular device. For benign services, Android introduced app-side a11y protection mechanisms intended to hide sensitive UI information from untrusted a11y services [14, 15], but prior work did not examine how protections can fail when the same service is accessed via a mobile browser-rendered website. For local LLMs, disk forensics can extract storage artifacts and logs [16, 17, 18, 19], and memory forensics can recover OS and app data structures [20, 21], but neither line of work is designed to recover and rehost deleted conversation context that remains only as model-internal runtime state.

1.2 Thesis and Contributions

Accurately reconstructing these user interaction evidence is challenging because interaction context is dynamic, cross-layer, and often intentionally ephemeral. A11y malware may trigger abuse routines only under victim-specific conditions and dynamically load abuse code, making it hard to attribute capabilities to concrete victims without executing in a realistic, instrumented environment. On the benign-service side, websites and native apps expose a11y information through different mechanisms, and browser translation of ARIA labels can vary across browsers, creating a moving target for developers who want consistent protection. For on-device LLM apps, the relevant context is not stored as human-readable chat logs but as encoded token sequences and attention state (e.g., KV cache) that can be logically evicted, requiring deep model-aware interpretation to validate and rehost.

In this dissertation, I aim to make hidden, protected, and deleted interaction context measurable and actionable by extracting it from real software stacks. In DVa, I uncover victim apps and attribute victim-specific a11y abuse vectors by combining victim modeling with abuse-vector-guided symbolic analysis on real malware. In SOMBRA, I identify where app-side a11y protections fail to transfer to mobile websites by modeling and comparing the a11y information exposed by native apps and by browser-rendered counterparts across four different browsers. In ORISA, I recover and rehost deleted conversation context from local LLM apps by extracting and realigning batch tokens and KV cache state so investigators can interrogate a live session with recovered context. Below I will briefly introduce these techniques, the technical contributions made by each, and the unique challenges that they overcome.

1.2.1 DVa

The Android a11y service is widely abused by malware to conduct on-device monetization fraud. Existing mitigation techniques focus on malware detection but overlook providing

users evidence of abuses that have already occurred and notifying victims to facilitate defenses. I developed DVa, a malware analysis pipeline based on dynamic victim-guided execution and abuse-vector-guided symbolic analysis, to help investigators uncover a11y malware’s targeted victims, victim-specific abuse vectors, and persistence mechanisms. I deployed DVa to investigate Android devices infected with 9,850 a11y malware. From the extractions, DVa uncovered 215 unique victims targeted with an average of 13.9 abuse routines. DVa also extracted six persistence mechanisms empowered by the a11y service.

1.2.2 SOMBRA

The Android a11y service has been widely utilized by malware to abuse benign services. To prevent such abuse, developers need to secure a11y content access in both their apps and mobile websites. However, a misalignment of a11y protection mechanisms exists between them. Prior research has focused on attacking and defending a11y information embedded in native Android apps. However, my research found that a11y malware can retrieve app-protected a11y information in its mobile browser-rendered website counterpart, leaving mobile browser users more vulnerable to a11y attacks than app users. To help benign service developers vet this attack surface, I developed SOMBRA, an automated analysis pipeline to vet browser-side leakage of a11y information that is a11y-protected in apps. Using SOMBRA, I analyzed 294 benign services and found 29 of them deploy app-side a11y protection mechanisms to secure 256 views. SOMBRA discovered that 241, 402, 244, and 251 elements corresponding to their protected app-side views are a11y-exposed in their websites rendered by Chrome, Firefox, Brave, and Edge browsers, respectively. The leaked elements contain sensitive personal identifiable information. Finally, SOMBRA discovered that most developers do not adopt browser-side a11y protections because existing mechanisms either have ineffective protection or hinder the usability of their content.

1.2.3 ORISA

Advances in edge computing devices have enabled local large language model (LLM) apps to execute entirely on-device, powering tasks such as document summarization and virtual assistance while preserving user privacy. However, this also removes centralized logging and enforcement, allowing adversaries to fine-tune or deploy modified models that generate malicious content. When such apps are used in planning crimes, it is hard for forensic investigators to gather evidences because disk logs from app storage are cleared once conversation histories are deleted from the UI by suspects. Existing memory forensics techniques also cannot reconstruct the context embeddings retained within the LLM runtime states. I present ORISA, a memory forensics framework for recovering and rehosting deleted conversation contexts from local LLM apps. ORISA leverages the batch tokens and key-value (KV) caches preserved in process memory to reconstruct token sequences and restore session-level attention states. ORISA recovers previous conversations and outputs a rehosted app-agnostic session that remembers forgotten context from the original session, allowing investigators to reveal suspect’s prior interactions and ask additional information using the live session. I evaluated ORISA across 60 session configurations involving 10 Android local LLM apps and 3 model architectures. ORISA recovered 100% of tokens and KV cache tensors within the active context window, and an average of 32.60% and 38.59% more forgotten context. ORISA’s rehosted sessions revealed an average of 169.6% knowledge compared to the original session.

1.3 Dissertation Organization

This dissertation will present the evolution of this body of work. I will highlight the progression of the shifts in user interaction forensics capabilities proposed by each subsequent technology. The overall organization of this dissertation is as follows:

- Chapter 1 has introduced the forensic benefits and unique challenges of extracting

user interaction evidence from modern software stacks. I have presented an overview of my contributions to this area, with a specific focus on moving investigations away from artifact-only recovery and toward reconstructing *in-context* interaction from hidden, protected, and deleted UI/AI state. For each component within this body of work, I have outlined the research problems it targets and the fundamental principles behind each technique.

- Chapter 2 presents DVa, my malware analysis pipeline for Android accessibility (a11y) malware. I show how victim modeling and abuse-vector-guided analysis uncover targeted victim apps, attribute victim-specific abuse vectors, and extract persistence mechanisms from real malware samples.
- Chapter 3 presents SOMBRA, my system for discovering accessibility leakage caused by misaligned protections between native Android apps and mobile browser-rendered websites. I demonstrate how cross-platform a11y modeling and guided traversal enable developers to identify sensitive fields protected in apps but exposed in websites across different browsers.
- Chapter 4 presents ORISA, my memory forensics framework for recovering and rehosting deleted conversation context from local LLM applications. I describe how batch token and KV cache recovery, realignment, and session rehosting restore actionable context for investigators to interrogate a live session.
- Chapter 5 discusses related research efforts that serve as motivation, background, and technical complements to my work in this dissertation.
- Chapter 6 concludes this dissertation.

CHAPTER 2

DVA: EXTRACTING VICTIMS AND ABUSE VECTORS FROM ANDROID ACCESSIBILITY MALWARE

Android’s accessibility service [1], called `AccessibilityService`, provides extensive utilities to assist users in better navigating their phones. It stands as an exception to Android’s app-isolated sandbox design in the sense that it grants `AccessibilityService` apps the ability to examine and navigate foreground graphical user interface (GUI) screens of other apps or the Android OS. These powerful capabilities are thus widely abused by malware to conduct more intrusive attacks on user-controlled information and services [2, 3]. In fact, these capabilities enable malware to conduct on-device fraud [4] and simplify traditional account takeover practices (e.g., ransomware, RAT). Because of `AccessibilityService` malware’s powerful capabilities, traditional malware mitigation techniques that merely detect and delete malware are inadequate. Users of compromised devices need to know what damage could have occurred during the infection to facilitate restitution. In addition, the developers of victim apps need to know how they are targeted to proactively deploy defenses.

Regarding `AccessibilityService` security, several works have proposed data-flow restraints [8, 9] to counteract the proof-of-concept (PoC) attacks [13, 22, 23, 24]. However, my research revealed (subsection 2.5.3) that modern `AccessibilityService` malware can still evade the most recent Android security patch [25] and state-of-the-art (SOTA) data-flow constraint defense [8]. Techniques also have been proposed to identify the misuse of `AccessibilityService` in benign apps [10, 11, 12, 13]. However, without considering the targets of these misuses, an investigator will fall short of understanding their in-context abuse vectors because generic `AccessibilityService` routines can mean drastically different things in different contexts. A malware analysis technique that provides proof of abuse capabilities to the device owner and alerts targeted apps of the abuse vectors to aid proactive defense remains an unresolved matter.

To achieve these capabilities, the existing malware detection engine (e.g., Google Play Protect) needs to send detected malware to the backend, accurately dissect the malware’s all1y abuse vectors, and send the targeted victim and abuse vector report to the device user and the developers of affected victim apps. Unfortunately, such a technique is very challenging to implement. Malware may not perform abuse vectors without certain conditions (e.g., the absence of victim apps), making it difficult for investigators to examine the abuse intentions. As such, the abuse report that the device user receives must be specific to the victim apps installed on the user’s device. Compounding this issue, modern all1y malware complicates victim identification by dynamically loading the abuse routines and encoding the targeted app names. As a result, it remains largely impossible for investigators to identify all targeted victim apps and alert them to the discovered abuse techniques specific to each app.

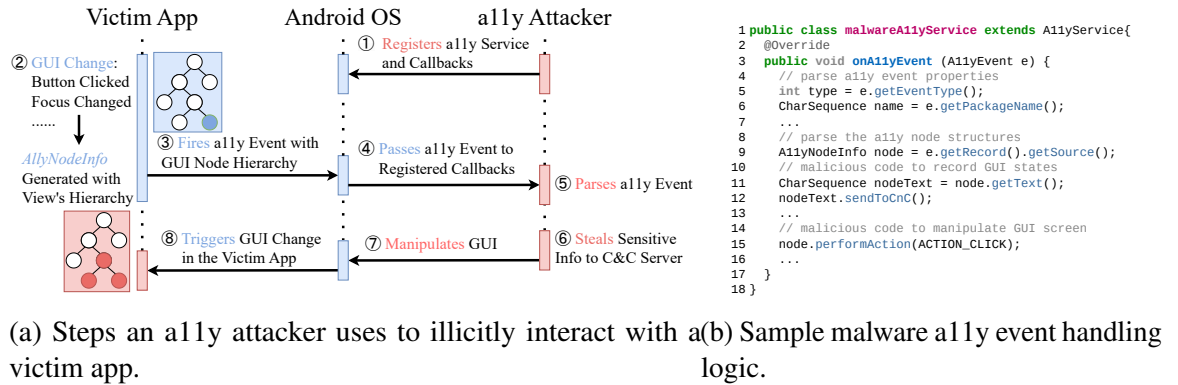


Figure 2.1: Overview of malware workflow to abuse the Android all1y service. Steps used to parse and manipulate a victim app’s GUI by an all1y service registered by an attacker is shown in (a). The all1y attacker’s actions are highlighted in red. A sample implementation logic of the malicious all1y service to achieve the abuse illustrated in (a) is shown in (b).

During my research, I found that all1y malware relied on Android APIs and broadcast all1y events to probe the information of the installed apps on the user’s device. This gives investigators an opportunity to mimic the presence of the target victim apps and drive the execution in an isolated environment without modifying the victim’s device. After all1y malware probes the information of the installed app, it would choose to load the dynamic

a11y abuse routines accordingly. Such trigger-based behaviors allow the investigators to attribute the a11y abuse routines to the trigger — specific victim apps. Attackers have infinite ways to implement each a11y abuse routine. However, these routines still rely on the a11y APIs exposed by the Android system. This enables investigators to uncover the abuse vectors by evaluating its a11y action sequences. These abuse vectors can be reported to all corresponding victim apps to enable blocking or mitigation. Lastly, by comparing the attributed abuse vectors with the installed apps on the victim’s device, the investigators can inform the device users of their targeted apps and the damage they may face.

I developed DVa¹, a malware analysis pipeline to uncover a11y malware’s targeted victims and victim-specific abuse vectors. It can operate as a backend malware analysis service for Google Play Protect, activated when an a11y malware is detected on the user’s device. With the malware APK, DVa adopts a novel lightweight *victim modeling and reconstruction* approach to guide malware to reveal its targeted victims (subsection 2.2.1). Using dynamic execution traces, DVa further utilizes an abuse-vector-guided symbolic execution strategy to identify and attribute abuse routines to victims (subsection 2.2.2). Finally, DVa detects a11y-empowered persistence mechanisms (subsection 2.2.3) to understand how malware obstructs legal queries or removal attempts.

Using DVa, I conducted investigations of five Google Pixel 3 devices infected with 9,850 malware samples collected from VirusTotal [26] that request a11y permissions from August 2022 to December 2022. DVa uncovered 215 unique victim apps across seven categories abused by 4,291 a11y malware samples. DVa found that *Banking* and *Crypto* apps are the most popular targets abused by 3,579 and 1,130 malware samples across 55 and 23 families, respectively. DVa also detected an average adoption of 13.9 unique abuse routines targeting custom UIs of each victim app with an average of 21.1 illicit a11y API calls. To persist on user devices, DVa uncovered that malware adopts six persistence mechanisms empowered by a11y. The most abused mechanism is *Permission Revocation*

¹Detector of Victim-specific a11y abuse

Prevention, exploited by 92% of malware. Lastly, I have made DVa available at: <https://github.com/CyFI-Lab-Public/DVa>.

2.1 Overview

Android apps' GUI contains substantial user-controlled sensitive information [27, 28]. Although the Android *ally service* is designed to help users better interact with their devices, its ability to peek into the on-screen element hierarchy and to simulate user interactions provides a new perspective of malware to abuse victims.

2.1.1 Abusing Android ally Service

Malware uses the eight steps shown as circled numbers in Figure 2.1a to illegally acquire sensitive information or conduct malicious GUI actions. First, the malware registers an *ally service* to the Android system that can retrieve changes in window contents, as shown in ①. After the user confirms the binding of the service, whenever something notable [29] happens in the GUI such as when a window changes, a button is clicked, a textbox is focused, etc. (Step ②), the *View* element in which the change occurred fires an *ally event* to the system, as shown in Step ③. The relationship between the *View* and other elements in the GUI is represented as a GUI tree data structure, as illustrated in the tree in the blue box of Figure 2.1a. The *ally event* contains properties of the changed *View*, together with the node hierarchy of the GUI tree the *View* resides in. These properties of the changed *View* and its relationship with other elements are represented as an *ally node info* data structure, as shown in the blue circle inside the tree.

After the Android OS receives an *ally event*, it dispatches the event to all registered ally services that listen for such an event through a callback, as shown in step ④. The callback is handled by a declared `onAllyEvent` handler in the malware's registered *ally service*. A sample ally event handler is illustrated in Figure 2.1b. This handler can then parse the *ally event* (Lines 4-9 Figure 2.1b, ⑤ Figure 2.1a), steal information fetched from

the event (Lines 10-13 Figure 2.1b, ⑥ Figure 2.1a), and subsequently manipulate the GUI according to the GUI node hierarchy contained in the event (Lines 14-16 Figure 2.1b, ⑦ ⑧ Figure 2.1a). The GUI manipulation is realized by issuing *a11y actions* that can mimic user interactions such as pressing a button, scrolling the screen, inputting text, etc. It can also be realized by sending *a11y global actions* to simulate global controls such as returning to the home screen, locking the screen, etc.

2.1.2 Uncovering 0-Day Abuse Evidence from a11y Malware

DVa's benefit over standard malware analysis techniques is that it dynamically models victim-specific a11y information that malware is probing for. With DVa, an investigator will have access to exclusive live interaction between the malware and this a11y information. In fact, lacking this evidence, traditional techniques are incapable of fully extracting malware's targeted victims and abuse vectors.

Consider the Cerberus malware studied extensively by malware analysts. Based on DVa's analysis of a Google Pixel 3 device infected with the Cerberus malware², DVa discovered a previously unknown *automatic transaction* abuse vector targeting 12 new victims.

Existing Malware Analysis Reports. Cerberus is widely considered to be a RAT targeting multiple banking, utility, and social media apps, capable of stealing users' credentials [30, 31]. Lines 20-27 of Figure 2.2 show the existing credential stealer abuse capabilities targeting financial institutions. When Cerberus receives an a11y event, it compares the source of the event with a hard-coded victim package name. If the source matches with a targeted victim, the malware then receives and loads overlay screen resources tailored to attack identified victim apps (Line 24). The `onA11yEvent` handler will also trigger text logging capabilities after loading the overlay screens to illicitly acquire users' credentials (Line 25). DVa's analysis also revealed all 89 targeted victims with the credential stealer abuse found by industry reports.

²MD5: 9236f4009503b4216e6773741b9d8ec0

```

1 public class cerberusA1lyService extends A1lyService{
2     @Override
3     protected void onServiceConnected() {
4         // hash encoded new targeting victims
5         List<String> newVictimHashes = secretDict.getNewVictimHashes();
6         for (PackageInfo pi : pm.getInstalledPackages()) {
7             // check existence and validity of targeting victims
8             if (newVictimHashes.contains(pi.packageName.hash())) {
9                 if (pm.getLaunchIntentForPackage(pi.packageName).isValid()) {
10                     ...
11                     // load tailored new a1ly attack code from CnC
12                     requestNewPayload(pi.packageName.hash());
13                     classLoader.loadClass(newPayload);
14                 }
15             }
16         }
17     }
18     @Override
19     public void onA1lyEvent (A1lyEvent e) {
20         // existing victims
21         List<String> victims = getVictimPackageNames();
22         if (victims.contains(e.getPackageName())) {
23             // existing overlay attack code to steal credentials
24             loadOverlay();
25             keyLog();
26             ...
27         }
28         // new code targeting chime mobile banking app
29         if (e.getPackageName().hash() == newVictimHashes[0]) {
30             // new a1ly automatic transaction attack routine
31             if (e.checkGUIState() == homePage) {
32                 findAndClickTransferPageButton();
33                 findAndFillTransactionTarget();
34                 ...
35                 findAndClickTransfer();
36             }
37         }
38         // additional routines targeting 11 more victims
39         ...
40     }
41 }

```

New DCL a1ly
Abuse Code

Figure 2.2: Dynamically loaded automatic transaction abuse routine targeting the Chime banking app extracted by DVa.

DVa’s Malware Analysis Discovery. In addition, DVa uncovered 0-day dynamically loaded automatic transaction abuse routines targeting 12 additional victims, as shown in the highlighted red box of Figure 2.2. Lines 3-9 of Figure 2.2 show the victim discovery routines that lead to the loading of new abuse code. When the a1ly service is connected, Cerberus dynamically queries packages installed on the device and compares them with a secret dictionary of hashes of the new victims’ package names. When a package matches, it then executes multiple verification routines such as `getLaunchIntentForPackage()`, `getPackageInfo()`, and `getInstallSourceInfo()` to determine the validity of

the victim app states. Only when all victim states are verified will Cerberus proceed to request and load tailored abuse code targeting the victim, as shown in Lines 11-13 of Figure 2.2. Investigators without DVa may notice these routines statically, but they cannot decode the victim hash in Line 8 to understand the conditions for triggering dynamic code loading. To overcome this, DVa creates a lightweight victim-information model that mimics benign apps' static characteristics and dynamic behaviors of 37K unique popular apps. With access to this model, DVa then remodels all interface functions the malware relies on to acquire victim states such as package query APIs, package installation states APIs, `all` event queries, etc. to mimic the live interactions as if all 37K benign apps are present on the user's device. Through this modeling, DVa guides Cerberus to execute the whole victim parsing routine and trigger dynamic code loading. DVa then extracts the secret victim package names that matched Cerberus's hash dictionary. DVa found the Chime mobile banking app's package name (`com.onedebit.chime`) along with 11 other package names, thus confirming Cerberus's undiscovered victim targets.

With the dynamically loaded abuse routines, DVa also revealed the code containing the newly discovered automatic transaction capability. Lines 28-37 show an example of the capability targeting the Chime mobile banking app. However, even when faced with such routines, a traditional malware analysis technique is unable to attribute the capability to its targeted victim because the trigger conditions of those routines are unknown. Symbolic execution into the entire app at this point will lead to state explosion. DVa uses a novel abuse-vector-guided symbolic analysis to extract routines listed in Lines 32-35 of Figure 2.2. After solving the symbolic constraints, DVa determines that these routines all depend on the value of the `all` event package name and its GUI state, whose symbolic value assignment occurs in Lines 29 and 31. DVa solves the symbolic hash constraint in Line 29 to derive a concrete hash value. DVa then matches this hash value with a hash observed in the dynamic execution traces and reports the plaintext victim package name that generated the hash. As such, DVa attributes Lines 32-35 to the Chime mobile banking app that the malware

targets. Finally, by modeling the `all` API invocation sequences in Lines 32-35 in the context of the victim target’s GUI screen, DVa reveals that each one is capable of navigating to the transactions page, selecting the transaction targets, filling the transaction amount, and sending the transaction requests, etc.

2.1.3 Threat Model

DVa’s goal is to identify the victims targeted by `all` malware and attribute `all` abuse vectors to each victim. I assume the user’s Android device is infected with malware that requests the `all` permission and the user has already granted the requested permissions. This is reasonable because `all` malware has been infiltrating the Google Play Store [32] and can trick users into granting `all` permissions [33]. DVa’s scope of abuse vector detection covers malware actions conducted by `all` APIs and targeting specific victim apps. That said, malware actions that complement `all` abuse but are conducted without `all` APIs (e.g., displaying overlay screens with `SYSTEM_ALERT_WINDOW`) are not considered by DVa. Additionally, `all` actions that do not have a specific targeted victim app (such as recording screenshots with a fixed interval and recording keystrokes whenever there is keyboard input) are outside the scope of DVa.

2.2 Methodology

DVa is a backend service that conducts analysis on `all` malware detected by an on-device AV engine (e.g., Google Play Protect). DVa takes the malware APK in the `user-data` disk of an Android device from adb [34] as input³. DVa pulls the list of installed applications from adb as victim app candidates on the device. DVa requires no prior knowledge of malicious `all` apps. For each `all` malware sample, DVa outputs its targeted victim apps, victim-specific `all` abuse vectors, and persistence mechanisms enabled by `all`.

³In a real-world scenario, Google Play Protect can use its internal extraction method to obtain the malware APK.

2.2.1 Victim Detection

The first goal of DVa is to identify victim apps on users' devices that might have already been abused, as well as potential victim apps that can be abused. However, identifying all possible victims is not as easy as it seems. To evade the security vetting system of the application stores, ally malware dynamically loads payloads locally or fetched from C&C servers during runtime. Specifically, the most advanced ally malware loads attack payloads only after scanning installed apps on victim devices, checking victim app configurations, and receiving acknowledgments from C&C servers [35]. This dynamic loading practice could easily make static analysis ineffective since the payload is not available at the time of the investigation. Worse still, even if the payload is available, the wide adoption of victim information encoding and encryption schemes makes decoding them statically challenging. While dynamic analysis with a SOTA sandbox can fake certain environmental parameters and statuses in system API calls, it cannot generate customized data structures such as ally events that malware is checking for.

To overcome these challenges, DVa uses dynamic hooks to mimic the existence of both targeted victims apps (subsubsection 2.2.1) and their generated ally events (subsubsection 29).

Dynamic Victim-Guided Execution

DVa first must be able to intervene with malware's victim-probing process and guide it to believe that the victims are present and valid. On Android, the access points to scan the status of installed apps are the package manager APIs [36].

algorithm 1 presents DVa's strategy to model victim query APIs. DVa keeps a pre-determined victim database (II) containing each victim app's traits and properties. The victim candidates are collected by querying the top-25 Android apps in 34 categories across 92 countries from market intelligence platforms AppBrain [37] and SensorTower [38]. Currently, the database contains 37K unique Android packages. As shown in Lines 2-28

Algorithm 1: DVa’s victim query modeling.

```
// Victim model,  $\Pi_x$  is model of a single victim
1  $\Pi = \{\Pi_1, \Pi_2, \Pi_3, \dots\};$ 
// Model all Android packageManager APIs, override before-method
  handler
2 @Override
3 Function Object beforePmMethod():
4   switch pm do
// API queries Generic info
5     case G do
// Use default system handler
6       return;
7     end
// API queries Individual package info
8     case I do
// Model of a single victim app
9        $\Pi_x = \Pi.get(pm.param.packageName);$ 
10       $R = \text{new } pm.returnType;$ 
// Pierce back custom victim fields
11      for  $field \in pm.returnType$  do
12         $a = \Pi_x.get(field);$ 
13         $R.set(a);$ 
14      end
15       $R.pad();$ 
16      return  $R;$ 
17    end
// API queries Collective info of packages
18    case C do
19       $R = \text{new } pm.returnType;$ 
// Pierce back custom victim field from many victim apps
20      for  $package \in pm.param.packageNames$  do
21         $\Pi_x = \Pi.get(package);$ 
22         $field = \Pi_x.get(pm.returnType);$ 
23         $R.set(package, field);$ 
24      end
25       $R.pad();$ 
26      return  $R;$ 
27    end
28  end
29 end
```

of algorithm 1, for each package manager API, DVa applies a dynamic hook and models it to return customized information containing legitimate victim information. If the query is generic, DVa returns the handle to the system’s default handler, as shown in Lines 5-7. If the hooked API queries the information of a single (specific) package (e.g., App X), DVa looks up the associated victim model Π_x . DVa then pierces all fields required in the API return

Algorithm 2: DVa’s large-scale triggers of victim ally events.

```
// Trigger startup events for each victim model
1 for  $\Pi_x \in \Pi$  do
2    $ae = allyEvent.obtain();$ 
   // Mimics an app startup event
3    $ae.setEventType(WINDOW\_STATE\_CHANGED);$ 
   // Pierce together ally event with victim model
4   for  $field \in allyEvent$  do
5      $\Pi_{xField} = \Pi_x.get(field);$ 
6      $ae.set(\Pi_{xField});$ 
7   end
   // Broadcast victim ally event to the malware
8    $am = getSystemService(Context.allyService);$ 
9    $am.sendAllyEvent(ae);$ 
10 end
```

value from Π_x and returns the custom value (Lines 8-17) such as package name, app icon, package install time, package launch intent, etc. Similarly, when the hooked API queries collective information of several packages (e.g., Apps $A, B \dots N$), DVa obtains the required fields from each victim model $\Pi_A, \Pi_B \dots \Pi_N$, and pierces them together (Lines 18-27). By returning the device status that malware is looking for, DVa tricks it into believing that its target victims, together with their traits, exist and are valid. DVa is also able to handle general anti-dynamic-analysis techniques equipped by the malware (detailed in section 2.7).

Mimicking Victim ally Events

For advanced malware that eavesdrops on device window states and launches victim-specific attacks only after the user instantiates the victim app, DVa needs to automatically fire victim ally events on a large scale to trigger the malware’s victim-specific attack routines.

algorithm 2 presents DVa’s strategy for triggering victim ally events on a large scale. For each victim model Π_x in the victim database Π , DVa obtains a default ally event. DVa then lets the event mimic the change in the foreground GUI when a user first opens up the victim app by setting the event’s type to `WINDOW_STATE_CHANGED`. For each field in the ally event, DVa queries the victim model Π_x and populates the ally event with the acquired custom victim trait, as shown in Lines 4-7 of algorithm 2. The fields populated in

the event contain key information that represents the GUI screen during the initiation of a victim app such as event time, content change type, view locations, view content, etc. DVa then acquires the system ally manager and broadcasts the customized ally event, as shown in Lines 8-9. This tricks malware to believe that a victim app is open and loads targeted abuse routines.

With DVa’s capability of mimicking victim existence and triggering victim ally events on a large scale, investigators can now notify users of victim apps targeted on their device as well as notify developers of additional malware targets.

2.2.2 Abuse Vector Detection

After identifying the targets of ally malware, DVa next finds unique abuse vectors empowered by the ally service. However, detecting abuse vectors from the dynamic victim-guided execution is challenging. Specifically, the ally event structures, embedded with trees of GUI elements that malware looks for, are enormous considering all victim apps.

ally Capabilities

To accurately model the abuse vectors, I first systematically categorize the capabilities of ally APIs according to the official ally API doc [1, 29, 39]. Since an ally abuse relies on the Android ally service, the malware has to use capabilities provided by the service [1], the allyEvent [29] dispatched to the service, or the allyNodeInfo [39] embedded within the allyEvent. The ally capabilities section in Table 2.1 shows the categorization of official APIs based on their purpose supplemented by sample APIs or identifiers within the categories. For example in Row 5 of Table 2.1, for an app to query the information of an on-screen element, it has to use APIs provided by the allyEvent class that can query the source nodes, such as `getSource()` or `getPackageName()`. Similarly, for an app to click on an on-screen element on behalf of the user, it has to rely on node action API `performAction()` supplemented by the `click` action provided by the

`allyNodeInfo` class, as shown in Row 12 of Table 2.1. DVa uses these ally capabilities as basic blocks for modeling ally behaviors.

ally Techniques

Next, DVa models the generic techniques an app can achieve using the above capabilities. Since an ally service receives an event whenever notable changes occurred in the GUI [29], it can read the event to understand what changes occurred. Based on this knowledge, the service can then inject inputs on behalf of the user to conduct actions based on the conditions of the GUI change. Ally Techniques section in Table 2.1 shows the techniques that can either read screen changes or inject input and their action models.

To detect these techniques, given malware’s ally event handler, DVa uses it as the entry point to build a static Call-Graph (CG). Within the CG build, DVa sets the sink functions as the last action function in any of the techniques. For each sink function marked, DVa then queries its caller methods in the CG until it finds the ally event handler or reaches the top parent. DVa then selects the call sequences that originate from the ally event handler as candidates and constructs Control-Flow-Graphs (CFG) for each function within the call chain using the built-in CFG constructor in Soot [40].

To confirm the complete action model and to resolve parameters of action APIs to concretize actions, DVa starts symbolic execution from the entry point along each CFG. DVa first marks the input *ally event* parameter as symbolic. Whenever DVa encounters an unseen variable, it introduces a new symbolic label. If any part of the variable is reassigned, DVa propagates the label and marks the new variable as symbolic. When encountering branches, DVa duplicates the symbolic label, continues execution, and records along both directions. DVa ends the traversal along a path when it successfully reaches the targeted sink APIs that it is searching for, or when it encounters new functions not in the marked call chains, indicating an irrelevant path. After completing traversal along all call paths, DVa checks the validity of each call path by confirming the symbolic labels’ existence in

each intermediate routine. It also concretizes sink APIs to exact actions by matching the value of the symbolic parameters in the APIs back to their original value. For example, DVa confirms a path to contain the *InjClick* technique when the parameter of the node action resolves to the identifier of `ACTION_CLICK` instead of other actions.

Victim-Specific Abuse Vectors

Preliminary Study. Next, DVa resolves the action models in the context of the victim app to detect abuse vectors of a11y malware. I started by searching for reports of a11y malware published between 2017 and 2022 (reviewing and cataloging results from the Google search query: Android accessibility malware) [41, 42, 43, 31, 4, 30, 2, 44, 45, 3, 46, 47, 48, 49, 50]. I also queried in VirusTotal [26] and manually investigated all 43 unique a11y malware appended in the above reports and found the seven abuse vectors, as listed in Table 2.1.

Dynamic Victim Stack Trace. DVa first searches for any routines that examine the state of an a11y event in the previously detected a11y techniques action models that match with any dynamic victim detection stack traces. A match implies that the following technique is specifically targeting the victim detected in the dynamic victim-guided execution. For example, as shown in Row 24 of Table 2.1, when a `QueryWindowChange` call of a detected `ScreenLog` and consecutive `InjClick` or `FillText` technique has a matching stack trace from a financial victim app up until the call, DVa marks the trace to be an *Auto Transaction* abuse vector and attributes it to the specific financial victim app.

To match additional abuse vectors to the victim app, DVa compares the concretized parameter solved in symbolic constraints of the a11y event query APIs against other such APIs from all detected abuse vector call sequences. If the same value is resolved in another call, DVa confirms that it also constitutes the abuse routine against the same victim.

Other Victim Hints. DVa also looks for other call sequences before or after a11y techniques to infer victim abuse vectors. DVa scans the marked technique call sequences for intent crafting and firing functions and uses the resolved symbolic value in the parameters to

infer victims. For example, when DVa detects that before invocations to `EvdpText` or `ScreenLog` in a call sequence, an activity launch intent is concretized to the *Google Authenticator*'s package name, it is marked with the *Steal Authentication Code* abuse vector.

With DVa's ability to model a11y-specific abuse vectors and match dynamic victim traces, an investigator can attribute the abuse vectors to each previously detected victim and notify users and victim app developers of victim-specific assets and behaviors targeted by a11y malware.

Extendability of DVa. I designed DVa to be modular and extendable to detect new abuse vectors. When a new malware emerges with novel techniques or abuse vectors, the investigator can easily extend DVa by (1) adding the abuse vector's sink API into DVa's sink list and (2) adding the new action sequence in DVa's symbolic model.

2.2.3 Persistence Mechanism Detection

Aside from abusing victims, a11y malware modifies the user's interaction with system settings to persist for as long as possible on user devices. DVa next detects how malware utilizes a11y service to do so and notify users about possible changes to system settings. To achieve this, DVa invokes the triggers of persistence mechanisms that malware looks for and deploys dynamic hooks to capture their a11y responses.

Triggers to a11y Behaviors. Table 2.2 Column 1 lists the mechanisms empowered by a11y service malware use to persist on user devices. Since malware tries to adopt them as early as possible after installation and acquiring a11y permissions, DVa tries to match the triggering conditions listed in Table 2.2 Column 2 for each of the mechanisms, satisfy them, and trigger them to observe malware's reactions to them. For example, to examine if malware prevents a user from looking up its information and uninstalling it, DVa crafts an intent to open the app detail menu of the malware in *Settings* and sends it to navigate to the screen; to examine if malware prevents users from powering off or restarting the device, DVa sends an action to open the device power options dialog. A similar strategy is adopted

to trigger other mechanisms listed, except for when testing for escalation of privileges, in which case DVa dumps the malware’s runtime permissions and device admin apps when the foreground activity stabilizes after granting the `all` permission.

Intercepting `all` Behaviors. To interpret `all` responses of the mechanisms, DVa applies dynamic hooks to capture their `all` actions (listed in Table 2.2 Column 3). For example, after DVa triggers the power options dialog, if back or return home actions to send users away from the screens are captured, DVa confirms malware’s abuse of `all` service to persist on a user device through *disable power options*. With DVa’s persistence mechanisms detection, investigators can notify users of illicit changes to their device configurations.

2.3 Validating My Techniques

2.3.1 Implementation

Abuse-vector-guided symbolic analysis in DVa is implemented in Java (7.3K lines) leveraging Soot [40], used in top-tier research [51]. Dynamic hooks leverage EdXposed [52] with 1.6K lines of Java code. Dynamic analysis management is implemented in Python (1.3K lines). AVClass2 [53], a SOTA malware labeling tool is used to classify malware families. For deriving malware labels, I used the latest PyPI `avclass-malicialab` 2.8.7 with `-t` flag to extract all CLASS and BEH (behavior) tags from VirusTotal reports. The victim application dataset is queried from AppBrain [37] and SensorTower [38], SOTA Android market intelligence services. Benign application icons and UI screens are collected from Google Image Search. I used an Ubuntu 20.04 LTS system to host DVa’s static analysis module. DVa’s dynamic analysis and `all` malware are hosted on 5 Google Pixel 3 (64GB, 4GB RAM) phones running Android 9.0 (Pie).

2.3.2 Validation Setup

Due to the physical disk size limitation, I installed the top-300 banking apps across all regions together with 100 other apps evenly distributed among the other five categories

of apps such as crypto, authentication, social media, communication, and shopping apps on each device. Then I used AVClass2 [53] to label my malware dataset and randomly picked samples from the top-10 malware families until I found two samples in each family with industry reports and live C&C response, resulting in 20 samples. The C&C response is determined by matching with at least one of the contacted IP addresses in VirusTotal’s malware relation reports. I installed the two samples from each family onto the five devices respectively in two batches and dumped the disk image, resulting in 10 device investigations, as listed in Column 1 of Table 2.3. I manually reverse-engineered these malware samples using Jadx [54] and used industry reports to derive the ground truth⁴.

2.3.3 Validation Results

Table 2.3 presents DVa’s validation results. Column 2 shows the top-10 all-y malware families that are installed on each device investigation. Columns 3, 7, and 12 show the ground truth (GT) number of abused victim apps, abuse vectors, and categories of all-y-empowered persistence mechanisms. The following columns present DVa’s detection on the respective tasks. DVa’s evaluation metrics, including True Positives (TP), False Positives (FP), and False Negatives (FN), are presented in the rest of the columns.

Victim Detection. As shown in Column 5 of Table 2.3, DVa correctly identified (TP) 206 instances of targeted victims. My manual investigation together with industry reports revealed 220 instances, indicating DVa’s 94% accuracy in detecting victims. In Row 6, DVa produced 11 FNs while detecting victims for the Anubis family. With investigation, I found that while matching abuse code with dynamic execution trace, DVa’s symbolic engine failed to reconstruct the victim package names. This is due to Anubis’s adoption of a complex customized data structure for encoding victim information. I confirmed that this is a rare behavior. Additionally, while investigating devices 1 and 8, DVa introduced 2 FNs in a Spynote sample and 1 FN in an Androlua sample. I found that the 3 FN victims – *Cajasur*,

⁴Note that DVa does not need nor have access to the ground truth data.

Kutxabank, and *Banca Móvil Laboral Kutxa* – were reported in industry reports but were not installed on the devices. After installing them manually, DVa observed the abuse vectors and confirmed them as victims.

Abuse Vectors Detection. As shown in Columns 7 and 9 in Table 2.3, DVa successfully detected (TP) 53 categories of abuse vectors among 55 categories in the GT, yielding a 96% detection accuracy. As shown in Row 5, DVa introduced 1 FP while analyzing Spymax malware on device 4. I manually investigated and found that it contained a screen navigation error handler to immediately return to the malware’s screen after failed attempts to parse the on-screen element tree, making DVa incorrectly attribute its abuse vector. I confirmed that this is a rare occurrence. DVa also introduced 1 FN while investigating device 5. Due to the unreachability of Anubis malware’s C&C server for remote code loading, DVa was unable to detect the *ScreenLog* behavior in industry reports, which is solely contained in the dropped payload.

Persistence Mechanisms Detection. Shown in Columns 12-16 of Table 2.3, DVa achieved 100% accuracy, detecting 55 instances of persistence routines.

Comparison with AVClass2. To understand the advantage of DVa in reporting fine-grained abuse vectors specific to a11y, I compared DVa’s result with labels reported by the SOTA malware intelligence engine AVClass2. The last column of Table 2.3 shows all class and behavior tag labels reported by AVClass2’s extraction from VirusTotal reports. As observed in all rows, the labels reported are high-level and coarse-grained without containing evidence of detailed abuse behaviors. For example in Row 4, the only labels reported for the Spymax malware are *spyware* and *grayware*, which do not contain evidence of targeted victim or specific techniques abused as reported by DVa. Similarly in Row 2, although the labels contain *bankbot*, *execdownload*, and *infosteal*, no specific bank victims or abuse vectors that constitute infosteal or executable download are evident.

2.3.4 Coverage Assessment

Next, I evaluated how much of an advantage DVa’s abuse vector modeling provides over a generic data-flow analysis. I hypothesized that a generic data-flow analysis would over-taint paths and lead to false positives in discovering ally abuse and be insufficient in detecting behaviors reliant on dynamic execution traces. I compared victims, abuse vectors, and persistence mechanisms extracted from paths explored by DVa and those explored by the SOTA taint data-flow analysis tool, FlowDroid [51].

Experiment Setup. I used FlowDroid’s latest stable-build version (2.11.1) as the starting point to derive data-flow path coverage. The input APKs are the ones repackaged with dynamically loaded DEX files extracted by DVa. I set the source function to be the `onAllEvent` handler and the sink functions to be all functions declared in `ally service`, `ally event`, and `ally node info` classes (in total 246 functions). I found that initially, FlowDroid was unable to track data-flow from the `onAllEvent` callback function because it doesn’t have a concrete invocation. Because of this issue, I modified FlowDroid’s source tracking logic of the `onAllEvent` handler to taint the first assignment statement of the function parameter instead. I used the standard CHA algorithm for FlowDroid to generate the CG.

Assessment result. Table 2.4 shows the victims detected, number of paths flagged with ally abuse, and persistence mechanisms detected by DVa and FlowDroid. The FP and FN columns for DVa are omitted because they were discussed in Table 2.3. FP and FN columns for FlowDroid are derived by manually examining the data-flow paths. As discussed in subsection 2.3.3, I counted seven FP paths that DVa flagged in the Spymax samples as TP for FlowDroid in Row 4. For the Anubis samples in Row 5, I omitted FN paths for both DVa and FlowDroid because the payload containing the reported behavior was statically unavailable. As shown in the *Total* row of Columns 4 and 7, DVa flagged 2,327 paths with ally abuse, while FlowDroid flagged a total of 13,680. However, in Column 8 I see that FlowDroid incurred FP in 11,629 (85%) paths that do not constitute ally abuse. This is

because FlowDroid does not have access to the abuse vector modeling and flags irrelevant paths with generic `all` APIs. Column 9 shows that 269 paths were marked by DVa as `all` abuse but not traversed by FlowDroid. I found that these paths contain global action system APIs such as `dispatchGesture()` and `findFocus()` for which FlowDroid does not have pre-extracted method summaries, thus breaking the taint propagation. Additionally, Columns 3 and 6 show the number of victims detected along the paths with `all` abuse. While DVa extracted 206 victim apps, I manually verified the paths flagged by FlowDroid and found that 0 victim apps could be resolved with the data-flow paths alone. This is because the constraints along all paths can only be resolved to one-way encoded hashes and cannot be further matched. With the help of execution traces that DVa extracted in victim-guided dynamic analysis, DVa has a unique advantage in resolving statically unsolvable victim constraints. Finally, Columns 5 and 10 show the number of persistence mechanisms detected. While DVa reported a total of 55 instances of persistence mechanisms, the data-flow paths alone generated by FlowDroid cannot derive any of them because their trigger conditions and sink actions are unknown and unmodeled by FlowDroid.

2.4 Findings

To help gather large-scale malware behavioral insights from `all` malware, I deployed DVa on 5 Google Pixel 3 phones. These phones were repetitively infected with each of the 9,850 malware collected from VirusTotal [26] from August 2022 to December 2022. The selection criteria for the samples are that they (1) are labeled by at least five antivirus engines, ensuring that they are indeed malware [55] and (2) contain the `BIND_ALL_SERVICE` permission declaration string in the app manifest file, which indicates that they abuse the Android `all` service. Out of the 9,850 samples collected, 7,700 samples are labeled by AVClass2 [53] spanning 197 families. The rest of the 2,150 samples contain no family labels and are treated as singleton malware.

DVa’s dynamic analysis has an average runtime of 110 seconds per sample. Since DVa

is deployed on a backend server, this overhead is acceptable. The execution overhead on the frontend user’s device is negligible (no more than the current Google Play Protect).

2.4.1 Targeted Victims

Table 2.5 presents the identified victims grouped by category in Column 1, their most popular geolocation country code in Column 2, victim count and victim median download number in Columns 5 and 6, and their abused malware count and family count in Columns 7 and 8. As shown in Columns 5, 7, and 8 in the *Total* row, DVa detected a total of 4,291 malware samples spanning 65 families abusing 215 victim apps. DVa reported no targeted victims for the other 5,559 samples. Upon further investigation, I found that 4,614 of the samples did not have any live C&C responses, so I excluded them from the victim-detection evaluation. For the other 945 samples, I manually investigated 10 random samples and found that six of them are generic utility apps that misuse the all y service such as file manager, overlay widget, notification modifier apps, etc. In fact, prior work [56, 13] confirmed that benign apps also misuse the all y service to achieve functionality purposes. These utility apps do not target specific victim apps, so I exclude them from the victim-detection evaluation.

Of the 4,291 malware samples with detected victims, 2,575 require C&C response to drop victim-specific information and still have live C&C connections. The rest of the 1,716 malware samples contain local abuse routines that are present statically or loaded at runtime. Column 6 in the *Total* row shows that the median download number is 10M+ across all 215 abused victim apps. Columns 1 and 5 show the abused victim apps’ categories along with their ranked victim app count in each category. The 215 victim apps span a total of seven categories – Banking, Crypto, Shopping, Social Media, Transportation, Authentication, and Communication. Banking apps are the most abused victim app category with 159 (74%) apps targeted by 3,579 (83%) malware samples across 55 (85%) families, corroborating the banking apps’ vulnerability to all y-based attacks found by prior work [57]. Following Banking apps are Crypto and Shopping apps, with, respectively, 16 (7%) and 13 (6%) victim

apps abused by 1,130 (26%) and 257 (6%) malware samples across 23 (35%) and five (8%) families. As shown in the *Auth.* category row, although only five authentication apps (e.g., Microsoft Authenticator, Google Authenticator) are abused, they are abused by a majority of 3,022 (70%) malware samples across 52 (80%) families. The median victim app download numbers in Column 5 reveal that Social Media apps are the most popular apps that are targeted, with a median of 1B+ downloads. Communication apps such as Gmail, WhatsApp, etc. are the next most popular apps, with a median of 100M+ downloads. All victim app categories have at least a median of 5M+ downloads, indicating a large batch of renowned victim apps targeted by a11y malware.

Takeaway. DVa identified 215 victim apps spanning seven categories abused by 4,291 a11y malware samples across 65 families. *Banking* apps are the most widely targeted category, with 74% of all victim apps falling in the category and targeted by 83% a11y malware samples across 85% families. Seventy percent of a11y malware samples also target a set of five *Authentication* apps, extending their capabilities into infiltrating other benign apps. A11y malware targets renowned victim apps with a median download of over 10M across all categories. With the victim targets extracted, investigators can compare the targets with existing services on the user's device to identify abuse that has already occurred, as well as notify developers of malware targets that have not yet been abused.

2.4.2 Abuse Vectors

After extracting the victims of a11y malware, DVa helps investigators extract abuse vectors and attribute them to the victims. Columns 3, 4, 9, 10, and 11 of Table 2.5 present the a11y techniques and abuse vectors tailored for each category of victim apps. Columns 3 and 4 show the different abuse vectors and their average numbers targeting each victim.

As shown in the *Total* row of Column 4, a malware sample adopts an average of 13.9 vectors to target each victim. Looking at the *Subtotal* rows of Column 4, malware targeting a *Banking* or *Crypto* victim implements an average of 21.1 and 24.1 abuse vectors toward

each victim, 52% and 73% higher than the average. This indicates that malware targeting these two categories adopts more event checks to handle a diverse set of window change events. Comparing the subrows of each victim category, the most adopted abuse vector across banking, crypto, shopping, and transportation apps is *Steal Credentials*, followed by *Auto Transactions*. The *Authentication* apps are also targeted with 13.7 *Steal Credentials* abuse vectors, accounting for 79% of all abuse vectors. This corroborates the feasibility of a11y abuse proposed by prior PoC attacks [23, 58, 59].

Column 9 of Table 2.5 shows the a11y techniques used by each abuse vector. I further categorize APIs adopted by the techniques into *query APIs* (those that query window states) and *action APIs* (those that conduct GUI actions) in Columns 10 and 11. As shown in the *Total* row of Columns 10 and 11, an a11y technique routine on average adopts 19.0 APIs to query the state of on-screen elements and 2.1 APIs to perform a11y actions. In the *Subtotal* rows of Column 10, I observe that an average of 29.0 and 25.4 query APIs are used in each a11y technique that targets *Banking* and *Crypto* apps, 53% and 34% more than that of the category average. This indicates that malware employs more a11y node checks to properly handle more on-screen elements that are present on each banking and crypto app screen.

The *Subtotal* rows of Column 11 show that the average number of a11y actions performed is similar across each category, having an average of 2.1 a11y action APIs to perform each a11y technique. This shows that the differences that malware adopts in handling victim UIs lies in the queries of on-screen elements instead of the actions to them. Instead of focusing on the actions that malware performs in an abuse technique, it is more important to measure the query strategy of the UI that malware deploys against each victim.

Takeaway. DVa extracted an average of 19.0 a11y query APIs and 2.1 actions APIs in each a11y technique to deploy 13.9 abuse vectors towards each victim app. A11y malware implements more handler routines to parse UI screens (52% and 73% higher than average) as well as more on-screen element query routines (53% and 34% higher than average) to abuse *Banking* and *Crypto* apps. The most frequently adopted abuse vectors are *Steal Credential*

and *Auto Transaction* across most categories of victims. Although all action APIs are used to perform concrete all actions on elements, the query API usage is the pattern that differentiates abuse routines across different victim categories. With victim-specific abuse vectors extracted, investigators can enrich the notification to both users and victim developers with concrete abuse behaviors to guide loss remediation and develop proactive defenses.

2.4.3 Persistence Mechanisms

After extracting victim-specific abuse vectors adopted by all malware, DVa then helps investigators understand what all-empowered mechanisms malware adopts to persist on users' devices. Table 2.6 presents the persistence mechanisms adopted by the top-10 malware families. Columns 1-2 list the malware families and their respective sample counts. Columns 3, 4, 5, 8, and 9 show the number in each family that adopts mechanisms to disable device protection, prevent app info lookup/uninstallation, prevent all permission revocation, uninstall other apps, and disable device power options. Columns 6 and 7 show the number of samples that can escalate to device admin privilege and other privileges. As shown in the *Total* row, preventing all permission revocation, preventing info lookup, and disabling device protection are the three most widely adopted persistence mechanisms, observed in 9,102 (92%), 9,024 (92%), and 8,741 (89%) malware samples. Disabling power options is the least adopted, used by only 1,852 (19%) malware samples. Although 4,861 (49%) samples escalate privileges, only 157 (2%) of them (all from the *Fakeapp* family) escalate to device admin. Although the powerful device admin privilege can be illegally escalated by abusing all permission, very few malware samples choose to do that to avoid static and runtime antivirus scans since it is a strong indicator of malware. Similarly, few malware samples employ the intrusive mechanism of disabling power options to avoid alerting users. This adoption trend indicates that most malware deploys all-based measures to turn off system malware scanning and prevent users from querying, disabling,

or uninstalling them.

Takeaway. DVa extracted six categories of a11y-empowered persistence mechanisms. Most malware abuses a11y to prevent users from revoking a11y permission (92%), looking up malware info / uninstalling malware (92%), and disabling Google Play Protect (89%). More intrusive behaviors such as preventing users from turning off or restarting the device (19%) and escalating to device admin privilege (2%), are less common to avoid antivirus detection and alerting users. Investigators now can understand how a11y malware persists on users' devices and notify users of malware's illicit modifications to system settings.

2.5 Case Studies

2.5.1 a11y 2FA Stealer

Authenticator apps use 2FA codes to provide users additional protection against compromised static passwords. In my experiments, DVa found that 471/545 malware samples from the Bianlian family can steal 2FA codes generated from the Google Authenticator app. Of the initial 545 Bianlian samples, DVa found that 224 implemented dynamic loading of their a11y service class or a11y event handlers. After DVa's victim-guided dynamic analysis, DVa captured and repackaged new DEX payloads from 144 samples using dynamic class loaders. With DVa's abuse-vector-guided symbolic analysis from the `onA11yEvent` handler, DVa found that 471 malware samples contained valid execution paths to *EvdpText* and *Screen-Log* techniques with the victim package name resolved to the Google Authenticator App, confirming their 2FA code stealing capability. The routines start with malware creating and sending an `Intent` that launches the main activity of the app. Then, the malware checks for package names of the `WINDOW_STATE_CHANGED` event to confirm the start of the app. After that, the malware creates an iterator from the source node of the event (a `ViewGroup` containing all child elements of the main activity) to capture all user's 2FA codes. Within each child view representing a 2FA code, the malware uses customized parsing logic tailored for the app to extract the text element of the code and the account name associated with the

code.

a1ly Blocking by 2FA Apps. Although a1ly malware can steal 2FA codes, authenticator apps can choose to block untrusted a1ly services from accessing the code. I picked the top-11 authenticator apps listed on the Google Play Store, installed them on one Google Pixel 4 device running Android 14, and manually registered 2FA code for *Twitch* in all apps. I then installed a custom app with an a1ly service that listens for and parses a1ly events from the views in each app that contain the 2FA codes. Only two of the 11 apps protect their 2FA codes from untrusted a1ly services. I observed that the text properties of 2FA code fields in the *2FAS* (com.twofasapp) and *Dashlane* (com.dashlane.authenticator) apps are set to null, confirming their protection against untrusted a1ly services. I manually reverse engineered the apps and found that both protect their views containing 2FA codes with the `AllyDataSensitive` property to block interaction with untrusted a1ly services.

2.5.2 a1ly Ransomware

DVa detected 131 instances of the Doublelocker malware that abuses a1ly service for ransom, indicating new strategies for mobile ransomware. DVa also reported the *privilege escalation* persistence mechanism. DVa found that the malware automatically brings up device administration and default home app user dialog and utilizes hard-coded a1ly node parsing routines to locate and click the confirm buttons without user acknowledgment. By escalating to the default home app, the malware ensures that whenever a user tries to press the *Home* button or the *Back* button, the malware is relaunched instead. This renders the device unusable to the user, achieving an intrusive abuse and persistence measure. The main activity captured by DVa indicates a classic ransom screen, asking users to send an equivalent of 50 US dollars worth of bitcoin to an attacker's account.

Traditional ransomware requires multiple user interactions to achieve its purpose. As DVa observed in the Doublelocker family, all user interactions required for the malware to achieve its ransom purpose is granting the a1ly permission, thus increasing the attack's

success rate.

2.5.3 Defenses Against a11y Malware

To understand how SOTA defenses are (in)effective at preventing a11y malware abuse, I evaluated how a11y malware behaves under the newest Android security patch (Android 13.0.0_r31) [25] and the most recent data-flow defense framework proposed by Huang et. al. [8]. I re-implemented the defense framework as stated in the study [8] as a patch to the AOSP that constrains data-flow from user inputs to non-GUI actions and from the GUI event change to non-user-perception APIs. I deployed DVa on one Google Pixel 4 device with the latest Android security patch and another one with my re-implemented defense framework. I evaluated using 14 malware samples that target Android SDK Level 33 (Android 13) from the dataset.

Table 2.7 shows the effectiveness of the previous two defenses. Column 2 shows the baseline behaviors extracted by DVa from the 14 malware samples. In total, 42 instances of malware behaviors are extracted under Android 9. Column 3 shows the behaviors extracted by DVa on the newest Android security patch (Android 13). Four malware failed to install with manifest malformed errors due to the incompatibility of code on Android 13. Of the remaining 10 malware samples, the system initially blocked the installation of eight of them with Google Play Protect scanning. However, all eight malware samples successfully bypassed such scanning after I added a trivial asset (one JSON file in APK assets) and reinfected the re-signed APK. In particular, aside from the signature scanning, 100% of malware behaviors are observed under Android 13. Column 4 shows the effectiveness of the data-flow defense framework on Android 13. It successfully eliminated *ScreenLog* behaviors from all malware due to the constraints set to prevent information leakage from a11y events. However, 6/13 (46%) *screen navigation* behaviors still persist. Upon investigation, I found that malware relies on hardcoded screen coordinates not associated with a11y events to navigate GUI, thus circumventing the constraints. A similar technique is also utilized to

prevent restriction to 4/7 (57%) illegal *info lookup prevention* behaviors, etc.

Shown in the Total row of Table 2.7, 67% of malicious a11y behaviors are still observed in the latest Android security patch and 40% can still bypass the SOTA data-flow defense framework on top of the latest Android security patch.

2.6 Discussion

Malware Detection. Since DVa acts as an add-on service to existing malware detection engines, DVa relies on identified a11y malware. Google Play Protect already has signature-based and behavior-based malware detection [60] incorporated. Researchers have also proposed methods to distinguish benign and malicious a11y apps based on explicit user intention used as sources for a11y actions [8].

Limitations. Although DVa can be extended to adapt to new abuse vectors, it still relies on the modeling of explicit functionalities defined by a11y APIs. That said, DVa will fail to detect abuse that relies on side-channel exploitation of a11y APIs. Additionally, although DVa optimizes its symbolic exploration strategy based on abuse vectors, path explosion and unsolvable constraints are still possible when malware adopts overly complex a11y event handlers, as seen in the Anubis samples in subsection 2.3.3. Since DVa relies on dynamic analysis, malware can deploy new evasion techniques and time-sensitive behaviors to hinder the analysis. Although not observed in subsection 2.3.3, malware could implement stringent runtime checks to find a very rare a11y event. Since DVa can only reconstruct a finite set of a11y events in dynamic analysis, DVa will miss detecting malware capabilities if they do so. If this happens, DVa’s victim model can be extended to incorporate these a11y events.

Implementation Alternatives. While designing DVa, I considered alternative methods to implement victim detection. Fuzzing could be used to enumerate victim apps’ properties. However, fuzzing would generate many a11y events that do not make sense in the context of any real victim app and risk alerting the malware of my analysis. Honeypots could be deployed to mimic the dynamic traits of victim apps and record abuse behaviors. However,

honeypots passively execute malware, which would not drive the malware’s execution through victim-generated ally events. DVa’s victim-guided dynamic analysis resolves both of these challenges by actively generating ally events in the context of real victim apps’ behaviors.

Future of Android’s ally Malware. Although Android 14 allows developers to restrict ally information delivered to unvetted ally services by checking the `ALLY_TOOL` flag [61], I expect future malware to subvert its vetting process. Since the process is manual and conducted at Google Play Store’s submission stage, malware can still infiltrate the store by loading malicious code dynamically or through app updates just as it did before [32, 62].

Developer’s Defense. App developers can choose not to broadcast ally events if untrusted ally services are found on a user’s device [63]. However, this sacrifices the usability of the app because legitimate ally services would also be blocked. The Coinbase app [64] adopted an out-of-band ally verification that malware could not intercept: It displays a view that ally cannot interact with, instructing users to shake the device to approve the use of an untrusted ally service. Another approach is to protect an app’s GUI in a more fine-grained manner. Specifically, wrap only the minimal views containing sensitive information with ally delegates [15] to customize their exposed ally events and declare `allyDataSensitive` on views so that untrusted ally services cannot interact with them.

2.7 Prerequisites for Analysis

DVa initiates the investigation by pulling a list of registered ally services from the user’s device using the *ally manager* API and matching it with the identified malware’s ally service name. For each identified ally service, DVa then finds the package it belongs to and extracts its base APK file from the user’s internal `app-data` storage directory. During my extraction, I encountered multiple families of malware with anti-static and anti-dynamic techniques to thwart analysis. Here, I briefly describe strategies DVa adopts to bypass them.

2.7.1 Packed Malware and Dynamic Code Loading (DCL)

Malware heavily relies on packers to hide static malicious payloads by decrypting and loading them only after the application is loaded on a device. Some malicious payloads are loaded only after dynamic environment checks. To accurately collect all malicious payloads for static analysis, DVa deploys dynamic hooks to class loaders and utilize code reflection to locate and dump them. For each direct and indirect subclasses of `ClassLoader`, such as `BaseDexClassLoader`, `PathClassLoader`, etc., DVa deploys dynamic hooks on the `loadClass` API to capture every DCL attempt. Before the control logic is handed back to the routine, DVa intercepts the `ClassLoader` parameter and uses reflection to gather the path lists, dex elements, dex files being loaded, and the paths of the loaded files. DVa then copies the loaded files from the path in malware's internal storage, unzips them if necessary, and collects the final dex payload. After bypassing dynamic victim checks, DVa gathers all dumped payloads, eliminates duplicated ones, zips, and signs them into an APK for static analysis.

2.7.2 Anti-Dynamic Techniques

To avoid detection, malware also halts malicious code execution when it detects dynamic environment traces that suggest they are being analyzed. I observed multiple techniques such as emulator detection and side-channel inference on dynamic analysis framework artifacts. I reverse-engineered all malware from major families and deployed the following counteractions. DVa circumvents them first by running dynamic analysis on real Android devices. To counter side-channel inferences when malware infers the existence of a dynamic analysis framework by differentiating exception types when querying their class loaders, DVa applies dynamic hooks and directly throws `ClassNotFoundException`. Similarly, to counter malware from querying the existence of artifacts used by dynamic analysis frameworks in the file system, DVa applies dynamic hooks to file IO APIs and throws `FileNotFoundException` when detecting such queries.

Table 2.1: Allly capabilities, generic allly techniques, and allly-empowered abuse vectors in the context of victim apps.

allly Capabilities ¹	Sample APIs or Identifiers
alllyService	
GlobalAction	performGlobalAction(), dispatchGesture(), findFocus(), takeScreenshot(), takeScreenshotOfWindow()
alllyEvent	
QuerySourceNode	getSource(), getPackageName(), getRecord()
QueryViewChange	TYPE_VIEW_CLICKED, TYPE_VIEW_SELECTED, TYPE_VIEW_TEXT_CHANGED, TYPE_VIEW_TEXT_SELECTION_CHANGED
QueryWindowChange	TYPE_WINDOW_STATE_CHANGED, TYPE_WINDOW_CONTENT_CHANGED, TYPE_WINDOWS_CHANGED
QueryNotiChange	TYPE_NOTIFICATION_STATE_CHANGED
OtherQueries	TYPE_VIEW_HOVER_ENTER, TYPE_TOUCH_INTERACTION_START, TYPE_ANNOUNCEMENT, TYPE_SPEECH_STATE_CHANGE
alllyNodeInfo	
QueryNodeInfo	findAlllyNodeInfoByText(), getBoundsInScreen(), getText(), getPackageName(), isPassword()
NodeAction	performAction()
allly Techniques	Action Models
Read	
EvdpText ²	QueryWindowChange → QuerySourceNode → QueryNodeInfo(Text)
Evdpclick	QueryViewChange → QuerySourceNode → QueryNodeInfo(ViewProperties)
Evdpgesture	QueryGestureDetection
ScreenLog	QueryWindowChange → QuerySourceNode → QueryNodeInfo(Text) — GlobalAction(Screenshot) → FileWrite — DBWrite — InternetSend
Inject	
FillText	NodeAction(FillText)
InjClick ³	NodeAction(Click) — GlobalAction(Back/Home)
InjGesture	GlobalAction(Gesture)
Abuse Vectors	Technique Models
Auto Transaction	QueryWindowChange(FinancialStackTrace) ↔ ScreenLog & InjClick — ScreenLog & FillText
Steal Credentials	QueryWindowChange(LoginScreen) → ScreenLog — EvdpText
Steal Auth. Code	GetLaunchIntent(Auth./SMS) → SendIntent → EvdpText — ScreenLog
Hide/Delete Noti.	QueryNotiChange → InjClick & InjGesture
USSD Code	GetLaunchIntent(Phone) → SendIntent → ScreenLog → InjClick
Fake Calls	QueryWindowChange(Phone) → EvdpText & InjClick → LaunchActivity
Ransom Screen	QueryWindowChange — Evdpclick ↔ InjClick — LaunchActivity

1: Excluding APIs that are default callbacks, used to construct objects, acquire handles, and misc APIs.

2: Eavesdrop Text. 3: Inject Click.

Table 2.2: Malware’s ally-empowered persistence mechanisms, their triggering conditions, and their ally behaviors.

Persistence Mechanisms	Triggers	Behaviors
Disable Device Protection	Security Settings Screen	Back / Return Home
Prevent Info Lookup / Uninstall	App Detail Setting Screen	Back / Return Home
Prevent ally Permission Revocation	ally Settings Screen	Back / Return Home
Escalating Privileges	ally Permission Granted	Screen Navigation
Uninstall Other Apps	Uninstall Package Setting(w/ Permission)	Screen Navigation
Disable Power Options	Power Options Dialog	Back / Return Home

Table 2.3: Validation of the victims, abuse vectors, and persistence mechanisms detected by DVa together with AVClass2 labels of the top-10 all malware families.

D#	Family	# Victims				# Vectors				# P Mechs. ¹				AVClass2 CLASS & BEH Labels		
		GT	D ²	TP	FN	GT	D	TP	FP	FN	GT	D	TP		FP	FN
1	Spynote	13	11	11	2	6	6	6	0	0	6	6	6	0	0	spyware
2	Hqwar	5	5	5	0	7	7	7	0	0	4	4	4	0	0	execdownload, infosteal, bankbot, grayware
3	Bianlian	20	20	20	0	5	5	5	0	0	6	6	6	0	0	execdownload, infosteal, bankbot
4	Spymax	17	17	17	0	7	7	6	1	0	7	7	7	0	0	spyware, grayware
5	Anubis	89	78	78	11	4	3	3	0	1	8	8	8	0	0	infosteal, bankbot
6	Fakecall	4	4	4	0	3	3	3	0	0	3	3	3	0	0	execdownload, infosteal, grayware
7	Cerberus	29	29	29	0	6	6	6	0	0	7	7	7	0	0	execdownload, infosteal, bankbot
8	Androlua	15	14	14	1	6	6	6	0	0	6	6	6	0	0	grayware, clicker
9	Mobtes	16	16	16	0	5	5	5	0	0	4	4	4	0	0	execdownload, grayware
10	Mobtool	12	12	12	0	6	6	6	0	0	4	4	4	0	0	infosteal, grayware
Total ³		220	206	206	14	55	54	53	1	1	55	55	55	0	0	—

1: Persistence Mechanisms. 2: DVa's detection result. 3: Indicates total instances of detection, including duplicates.

Table 2.4: Comparison of victims, abuse vectors, and persistence mechanisms detected by DVa versus FlowDroid.

D#	Family	DVa			FlowDroid				
		V ¹	PAV ²	PM ³	V	PAV	FP	FN	PM
1	Spynote	11	207	6	0	1,348	1,165	24	0
2	Hqwar	5	63	4	0	418	360	5	0
3	Bianlian	20	217	6	0	1,719	1,540	38	0
4	Spymax	17	236	7	0	1,968	1,771	32	0
5	Anubis	78	746	8	0	3,458	2,767	55	0
6	Fakecall	4	51	3	0	342	300	9	0
7	Cerberus	29	269	7	0	1,317	1,097	49	0
8	Androlua	14	199	6	0	813	639	25	0
9	Mobtes	16	182	4	0	1,006	847	23	0
10	Mobtool	12	157	4	0	1,291	1,143	9	0
Total		206	2,327	55	0	13,680	11,629	269	0

1: Detected victims. 2: Paths flagged with ally abuse.

3: Detected persistence mechanisms.

Table 2.5: Categories of victim apps, ally malware that target them, and abuse vectors statistics.

Vic. Catg.	Vic. Geo. (Top-5)	Abu.Vec.	Avg. Vec. ¹	# Vic.	Med. DL ²	# Mal	# Fam	ally Tech.	Avg. Q ³	Avg. A ⁴
Banking	RU BR GB US MX	Auto Tran.	11.7	35	10M+	755	12	Scr. Nav. AutoFill	46.5 15.5	3.4 2.8
		Steal Cre.	19.3	147	10M+	3,296	52	Scr. Log.	9.6	1.7
		Steal Auth.	3.4	122	10M+	2,444	47	Scr. Nav. Scr. Log.	25.9 18.3	3.1 3.0
		Noti.	1.2	98	10M+	1,930	20	Scr. Nav.	7.6	1.2
		Ussd	5.2	103	10M+	2,067	16	Scr. Nav. AutoFill	59.7 22.1	2.8 1.1
		Calls	1.0	5	10M+	340	3	Scr. Nav.	55.4	2.7
Subtotal	-	-	21.1	159	10M+	3,579	55	-	29.0	2.4
Crypto	US IR AU BE PO	Auto Tran.	9.8	3	10M+	787	12	Scr. Nav. AutoFill	63.0 24.7	3.8 2.2
		Steal Cre.	21.5	16	5M+	1,051	23	Scr. Log.	7.9	1.6
		Steal Auth.	4.3	12	5M+	775	16	Scr. Nav. Scr. Log.	32.8 17.1	2.8 2.3
		Noti.	1.2	12	5M+	713	10	Scr. Nav.	7.1	1.3
Subtotal	-	-	24.1	16	5M+	1,130	23	-	25.4	2.3
Shopping	US ES JP IT DE	Auto Tran.	7.0	5	50M+	110	4	Scr. Nav. AutoFill	34.4 12.7	2.5 1.8
		Steal Cre.	15.8	13	50M+	219	5	Scr. Log.	6.6	1.6
		Noti.	1.3	10	50M+	143	5	Scr. Nav.	5.8	1.3
Subtotal	-	-	15.2	13	50M+	257	5	-	14.9	1.8
Social	IN US CA MX ID	Auto Tran.	9.3	7	1B+	171	11	Scr. Nav. AutoFill	38.0 12.4	3.1 2.4
		Steal Cre.	4.3	10	1B+	450	14	Scr. Log.	7.8	1.5
		Steal Auth.	2.7	7	1B+	268	7	Scr. Nav. Scr. Log.	28.9 15.8	2.5 2.3
		Noti.	1.2	2	5B+	22	1	Scr. Nav.	6.4	1.4
Subtotal	-	-	6.0	11	1B+	539	17	-	18.2	2.2
Transp.	US CA BR AU FR	Auto Tran.	3.5	6	50M+	45	2	Scr. Nav. AutoFill	33.6 15.3	2.5 1.9
		Steal Cre.	14.5	6	50M+	45	2	Scr. Log.	8.8	1.7
		Noti.	2.8	6	50M+	45	2	Scr. Nav.	5.6	1.4
Subtotal	-	-	20.8	6	50M+	45	2	-	15.8	1.9
Auth.	US CA BE DE NO	Steal Cre.	13.7	5	10M+	3,022	52	Scr. Log.	23.9	2.4
		Noti.	3.6	5	10M+	3,022	52	Scr. Nav.	4.5	1.2
Subtotal	-	-	17.3	5	10M+	3,022	52	-	14.2	1.8
Comm.	UK JP US CA AU	Auto Tran.	12.0	3	100M+	45	7	Scr. Nav. AutoFill	39.5 11.8	3.4 2.7
		Steal Cre.	2.8	5	100M+	1,374	34	Scr. Log.	5.4	1.1
		Noti.	1.4	3	500M+	45	7	Scr. Nav.	5.2	1.4
Subtotal	-	-	3.1	5	100M+	1,374	34	-	15.5	2.2
Others	-	Ransom Scr.	-	-	-	12	3	-	1.5	-
Subtotal	-	-	-	-	-	12	3	-	1.5	-
Total	-	-	13.9*	215	10M+	4,291*	65 ⁵	-	19.0†	2.1†

1: Average Abuse Vectors per victim app per malware. 2: Medium download number of abused victim apps.

3: Average of System ally Query APIs used per ally Technique. 4: Average of System ally Action APIs used per ally Technique.

5: 1,235 samples contain no AVClass family name. *: Not a sum of subtotals, multiple targeted victims per malware.

†: Excluding Others Column.

Table 2.6: A11y-empowered persistence mechanisms used by the top-10 a11y malware families.

Family	#	D.P. ¹	P.L. ²	P.R. ³	E.P. ⁴		U.A. ⁵	P.O. ⁶
					Adm.	Others		
Spynote	1,421	1,278	1,397	1,378	0	1,281	1,071	0
Hqwar	1,400	1,291	1,270	1,306	0	1,183	989	1,277
Bianlian	545	523	539	521	0	0	0	0
Spymax	461	429	446	451	0	384	351	0
Anubis	449	413	443	443	0	0	278	0
Fakecalls	351	319	290	303	0	323	0	0
Cerberus	349	244	305	305	0	0	102	0
Androlua	298	278	242	238	0	256	108	0
Mobtes	245	214	227	219	0	0	210	220
Mobtool	212	168	188	194	0	0	0	0
Others	4,119	3,584	3,677	3,744	157	1,277	979	355
Total	9,850	8,741	9,024	9,102	157	4,704	4,088	1,852

1: Disable device protection. 2: Prevent info lookup/uninstall.

3: Prevent a11y permission revocation. 4: Escalate privileges.

5: Uninstall other apps. 6: Disable power options.

Table 2.7: Effectiveness of the newest security patch and data-flow defense in eliminating a11y malware behaviors.

Behaviors	OS 9	OS 13	OS 13+DFD*
ScreenNav	13	9	6
Autofill	2	2	1
ScreenLog	7	5	0
Prevent Info Lookup	7	4	4
Prevent a11y Revocation	9	5	3
Escalating Privileges	3	2	2
Disable Power Options	1	1	1
Total	42(100%)	28(67%)	17(40%)

*: Data-Flow Defense.

CHAPTER 3

LOCK THE DOOR BUT KEEP THE WINDOW OPEN: EXTRACTING APP-PROTECTED ACCESSIBILITY INFORMATION FROM BROWSER-RENDERED WEBSITES

Android’s accessibility service [65], called `AccessibilityService`, although designed to help users better interact with their devices, has been widely utilized by malware to abuse benign services [2, 3, 4]. For developers to secure their content, they need to deploy defenses to all resources an `AccessibilityService` attacker can access, including both native Android apps and websites. However, an intrinsic misalignment exists between how `AccessibilityService` protection mechanisms are supported by native apps and browser-rendered websites. In particular, native apps can adopt stronger `AccessibilityService` protection mechanisms provided by Android, while websites still rely on browser-translated ARIA labels declared by developers alone.

Researchers have proposed a line of proof-of-concept (PoC) attacks [13, 23, 24, 22] abusing `AccessibilityService` to retrieve sensitive information from native Android apps. Malware analysis work [66] also has studied how real `AccessibilityService` malware can conduct in-app GUI attacks on benign services. Motivated by these attacks, Android has introduced several app-side `AccessibilityService` protection mechanisms to enable developers to hide their sensitive information from untrusted `AccessibilityService` services [14, 15]. Existing work also has proposed to use data-flow constraints [8, 9] to counteract several attacks that compromise the victim app’s GUI by abusing the `AccessibilityService` service. However, no existing work has explored how `AccessibilityService` attackers can still steal sensitive information when benign service developers actively deploy these `AccessibilityService` protection mechanisms in their Android apps.

In my research, I discovered that an `AccessibilityService` attacker can retrieve app-protected `AccessibilityService` information in its mobile browser-rendered website counterpart due to the misalignment between native app’s and website’s `AccessibilityService` support. Specifically, Android allows native app

developers to adopt both fine-grained static labels and dynamic a11y handlers to customize the a11y information exposed to on-device a11y services. On the other hand, mobile website developers can only rely on ARIA labels assigned to the elements to indicate their intended access level and announcement behavior. Since the labels are then translated by browsers before being interpreted by the Android OS, the same declaration by developers may result in different a11y output if rendered with different browsers. This leaves mobile browser users more vulnerable than app users to exposing their sensitive information to a11y attackers.

To help benign service developers vet this attack surface, I developed SOMBRA¹, an automated system to discover browser-side leakage of a11y information that is app-side a11y protected. SOMBRA first derives an app-side a11y model (subsection 3.3.1) and a browser-side a11y model (subsection 3.3.2) as a guide to its analysis. Given a service’s Android app, SOMBRA conducts an app-side a11y-model-guided traversal (subsubsection 3.3.3) to extract native fields protected by developers. SOMBRA then attributes each protected field to its app-side a11y protection mechanism (subsubsection 24) by conducting static analysis. Using the traversal logic of the app, SOMBRA then guides the browser-side automation engine to discover the fields in the service’s mobile website corresponding to the app-side protected components (subsubsection 46). SOMBRA then extracts the embedded browser-side a11y information and checks whether any leakage has occurred. Finally, SOMBRA compares the ARIA labels declared by the website developers with app-side a11y protections declared by app developers (subsubsection 46).

Using SOMBRA, I conducted a study of browser-side a11y leakage in real benign services’ websites rendered by four different mobile browsers – Chrome, Firefox, Brave, and Edge. From the 294 benign services collected, SOMBRA discovered 29 services that utilized at least one a11y protection mechanism to secure 256 views in their Android apps. While matching the views to the ones rendered in their websites, SOMBRA found that 241, 402, 244, and 251 elements corresponding to their a11y-protected app-side counterparts are

¹Scanner Of Mobile Browser Rendered Accessibility leakage

a11y-exposed in the four browsers, respectively (subsection 3.4.2). The Firefox browser exposes more elements than the others because of its difference in a11y translation logic. For the 256 views protected on the app-side, SOMBRA found that their website developers deploy fewer browser-side a11y protection mechanisms. SOMBRA found that only 12 (4.7%) elements are hidden from all a11y services and only 48 (18.8%) elements have alternative a11y announcements. Most developers choose to declare no browser-side a11y protection because existing mechanisms either are ineffective at protecting their content or hinder the usability of their content (subsection 3.4.4). The leaked a11y information on the browser side contains common sensitive personal identifiable information (PII). Among the 241 leaked elements in Chrome-rendered websites, 34.4% contain user account or credit card information and 7.5% contain user passwords (subsection 3.4.3).

3.1 Background

a11y Implementation in Android Apps. Android’s `a11yService` [65] allows developers to make their apps more accessible and usable. For each view [67] declared by developers in an app’s GUI, the Android OS populates the view with `a11yNodeInfo` [68] representing its a11y properties. Whenever the view changes in the GUI, it will broadcast an `a11yEvent` [69] containing the changes of the view and its properties to the Android OS. The Android OS then redirects the `a11yEvent` to all `a11yServices` on the device that are registered and allowed to receive such a type of event. After parsing the `a11yEvents`, `a11yServices` can translate that information to other types of output to users such as audio, making the content more accessible. Similarly, `a11yServices` can also translate users’ various input actions to text input or gestures on the app’s GUI screen, realizing functionalities such as voice control and gesture recognition.

a11y Implementation in Mobile Browsers. A mobile browser’s render engine is responsible for interpreting the HTML page and resolving dynamic content to show the web page to end users. Similarly, it is also responsible for translating and constructing the a11y informa-

tion embedded in the HTML page to a11y constructs that are understandable and parsable by the Android OS. Specifically, a mobile browser render engine parses the Document Object Model (DOM) [70], translates ARIA labels [71], constructs view elements in the Accessibility Object Model (AOM) [72], and populates a11y node information within each view. With the constructed views interpretable by the Android OS, registered `a11yServices` can then parse `a11yEvents` generated by those views and read a11y information embedded in those views when the screen content changes.

Since each browser’s render engine can have its own interpretation of the a11y labels and hierarchy, it can construct different AOMs. This is different from the native Android app’s a11y support where the app’s declared view hierarchy is directly interpreted by the Android OS, making the a11y events generated and their embedded a11y node information consistent across different devices. Additionally, the customizability of a11y information in mobile browsers is more limited than that in native Android apps. This is because mobile browsers rely entirely on ARIA labels declared in HTML pages to customize and render elements, while Android apps have access to multiple customizable `a11yEvent` [69], `a11yNode` [68], and `a11yDelegate` [15] methods to do so. As shown in subsection 3.2.1, the inconsistency in the mobile browser’s interpretation of a11y labels and its lack of customizability lead to failed a11y protection of sensitive information otherwise inaccessible to a11y attackers.

3.2 Exposing Mobile Browser Users’ Sensitive Information

Due to the differences in a11y support between native Android apps and mobile browsers, the apps provide stronger a11y protections. This leaves users who access their accounts and services through mobile browsers more vulnerable to a11y attackers than users who do so in apps. Although there are fewer users of mobile browsers than users of native apps [73], they deserve the same attention and protection.

```

1 TextView accountNum = findViewById(R.id.accountNum);
2 // Android 14 (API Level 34)
3 if (Build.VERSION.SDK_INT >= 34) {
4     // Only accessible to ally tools
5     accountNum.setAccessibilityDataSensitive(View.ACCESSIBILITY_DATA_SENSITIVE_YES);
6 }

```

(a) `allyDataSensitive` label protects ally text.

```

1 TextView accountID = findViewById(R.id.accountID);
2 // Bind an AccessibilityDelegate
3 accountID.setAccessibilityDelegate(new View.AccessibilityDelegate() {
4     @Override
5     public void onInitializeAccessibilityNodeInfo(View host, AccessibilityNodeInfo info) {
6         super.onInitializeAccessibilityNodeInfo(host, info);
7         // Override and hide text
8         info.setText("");
9     }
10 });

```

(b) `allyDelegate` overrides and protects ally text.

```

1 TextView userBalance = findViewById(R.id.userBalance);
2 // Set a custom listener for accessibility events
3 userBalance.setOnPopulateAccessibilityEventListener(event -> {
4     // Override and hide text
5     event.getText().clear();
6     event.getText().add("User balance");
7 });

```

(c) Customized `allyEvent` handler protects ally text.

```

1 EditText password = findViewById(R.id.password);
2 // Set as password type
3 password.setInputType(android.text.InputType.TYPE_TEXT_VARIATION_PASSWORD);
4 // Mask input characters
5 password.setTransformationMethod(PasswordTransformationMethod.getInstance());

```

(d) Password field with transformation protects ally text.

Figure 3.1: Protections against ally information leakage for native Android app users.

3.2.1 Leakage Types

Next, I illustrate the four types of ally leakage against mobile browser users. App users are protected from these leakages because of app-side protection mechanisms, as illustrated in Figure 3.1. I provide real leakage examples SOMBRA discovered in two apps from the Google Play Store – Klarna (`com.myklarnamobile`) and Varo (`com.varomoney.bank`).

App User Protection 1: `allyDataSensitive`. In the Klarna app, whenever a user binds a bank account to the app, the app displays a `ViewGroup` [74] for users to access the account. Within the `ViewGroup`, a standalone `TextView` [75] stores the bank account number. The `TextView`'s initialization routine [76] is set with the `allyDataSensitive` [14]

flag, as shown in Line 5 of Figure 3.1a. This ensures that whenever an `allyService` not approved by the Google Play Store as an `allyTool` [61] tries to access the view's `ally` content, it will be displayed as null to protect its information.

Browser User Leak 1: Absence of Fine-grained ally Access Control. The view that displays the bank account number on the Klarna website is directly focusable and visible while traversing the user profile page. Upon investigation, SOMBRA found that the HTML text field is declared with no labels that suggest its being hidden from `ally` services not approved by Google. In fact, no fine-grained `ally` access control label exists for website developers that only blocks untrusted `allyServices`. As a result, SOMBRA found the view on the browser side by locating its `allyNodeID` [77] in the top-level `ViewGroup` in the `window change` event [78]. After acquiring the view's `allyNodeID`, SOMBRA inspected the `ally` text field [79] of the view and found that the full account number is present and visible to `ally` attackers.

App User Protection 2: allyDelegate Override. In the Klarna app, a user can also check orders placed under the privacy and security tab on the user profile page. For each of the accounts bound, a `TextView` displays its account ID. SOMBRA found that when the view initializes, a custom `allyDelegate` class is bound to the view, as shown in Line 3 of Figure 3.1b. The `allyDelegate` class then sets the text field of the view to an empty string when it is called, as shown in Line 8 of Figure 3.1b. This ensures that the `ally` text field of the view is never interpretable by an `ally` attacker eavesdropping on the device.

Browser User Leak 2: Absence of Element Delegate. After locating the view showing the user account ID in the Klarna app, SOMBRA found that the view is traversable and contains the full account ID in the `ally` text field of `ally` events generated while focusing on the element. No labels or delegates exist for developers to achieve the same protection similar to the app-side delegates.

App User Protection 3: Customized allyEvent. In the Varo app, a user can check the balance of the account in a top-level `TextView` on the user profile page. While the text

field inside the view shows the aggregated numeral balance, the `allY` text announcement of the field only contains the string “user balance,” regardless of the actual balance. Upon investigation, SOMBRA found that the view customizes its `allY` event broadcast by modifying the initialization of its `allYNodeInfo` [76], as shown in Line 3 of Figure 3.1c. When the `allYNodeInfo` initializes, the view overrides its existing text with a constant string, as shown in Lines 5 and 6 of Figure 3.1c, thus preventing `allY` attackers from reading the sensitive information.

Browser User Leak 3: Uncustomizable `allYEvent`. The view that displays the balance field in the Varo website is traversable by SOMBRA and contains the full numeral balance in the `allY` text field. No customization is applied to the view’s content.

App User Protection 4: Password Field. In the account login activity of the Varo app, the password box `EditText` [80] view is declared with a `textPassword` [81] flag, as shown in Line 3 of Figure 3.1d. The view is then applied with a customized transformation method to mask every user input character with a dot, including the most recently typed character, as shown in Line 5 of Figure 3.1d. This ensures that when a user types in a password, every `allY` event generated from the view change contains only the masked dots. An `allY` attacker eavesdropping on the field thus cannot piece together the typed-in password by concatenating the last visible characters from a sequence of `allY` events.

Browser User Leak 4: Inconsistent Password Input. The password input box in the Varo website is focusable while traversing the main page by SOMBRA. While inputting characters in the `EditText` box, SOMBRA found that the most recently typed-in character is visible in the `allY` text field of the view in window change events. Although the previously type-in characters are masked out, SOMBRA can piece together the user password by concatenating the last visible character in the `allY` text fields. Upon investigation, SOMBRA found that although the input box is declared as a password type in the HTML page, the input box is applied with a JavaScript function that reveals the last typed-in character in an input event listener.

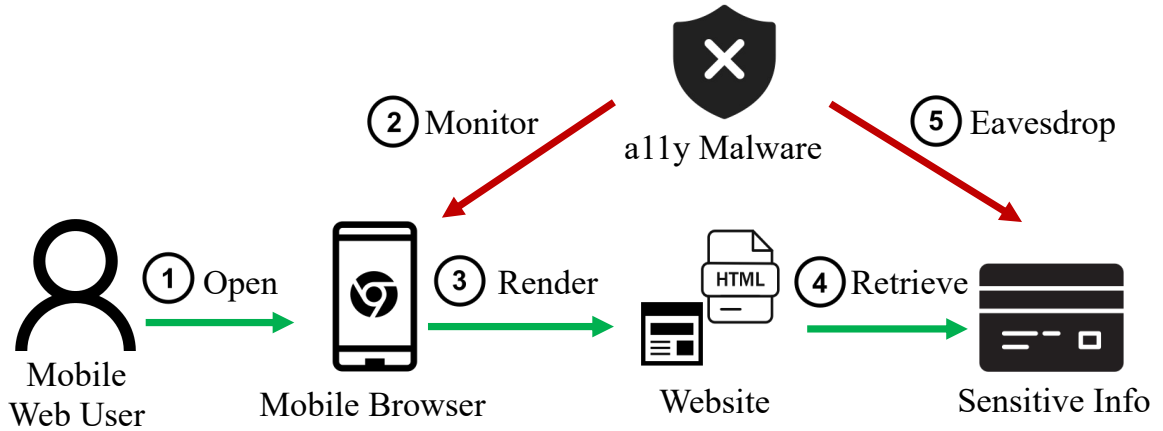


Figure 3.2: An a11y malware’s workflow to steal mobile browser users’ sensitive information.

3.2.2 Attack Workflow

Figure 3.2 shows an a11y malware’s workflow to steal sensitive information from a mobile browser user. When the user opens a browser, as shown in ①, the a11y malware can monitor the user’s actions by parsing `WINDOW_STATE_CHANGE` events, as shown in ②. When the browser renders the website in ③ and the user is viewing sensitive information in ④, the malware can eavesdrop on the sensitive information by parsing `a11y events` generated by the views rendered by the unprotected browser-rendered elements.

3.2.3 Attack Prerequisites

To expose mobile browser users’ sensitive information, as shown in subsection 3.2.2, I assume an attacker has infected users’ devices with malware that requests the a11y permission and the user has already granted the requested permissions. This is reasonable because a11y malware has been infiltrating the Google Play Store [32, 62] and can trick users into granting a11y permissions [33]. Furthermore, a11y malware remains the most popular mobile remote access trojan, targeting a wide range of benign services [66].

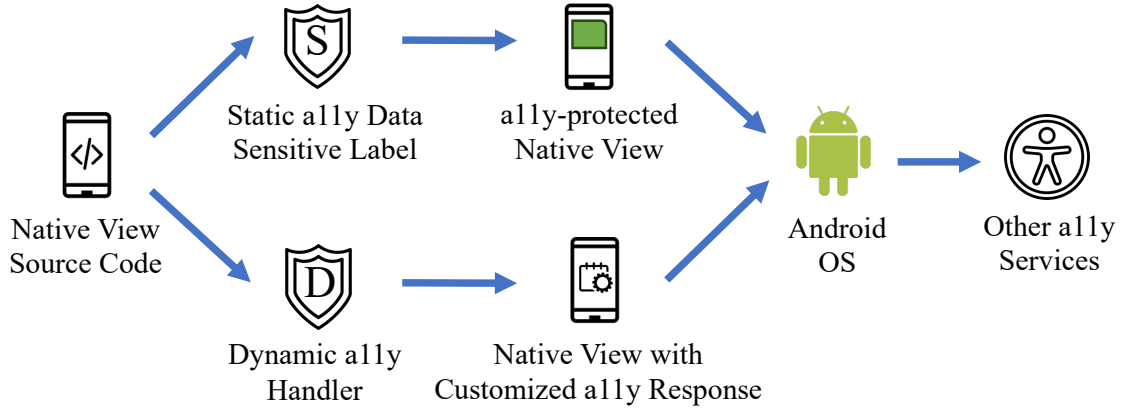


Figure 3.3: App-side a11y protection model. Android native views can adopt both static protection labels and dynamic a11y modification handlers. They are directly interpreted by the Android OS.

3.3 Pinpointing Browser-Side Leakage of a11y Information

To help developers vet a11y-exposed information for browser users that is protected for app users, I developed SOMBRA, an automated hybrid analysis pipeline that finds information exposed on mobile-browser-rendered websites but is protected in the app. My study focused on services with both apps and websites because SOMBRA uses app-side protected a11y information as ground truth. That said, browser-side a11y leakage extends to websites that don’t also have companion apps. I leave the analysis of these websites as future work since SOMBRA doesn’t have ground truth for them.

The input to SOMBRA is the Android APK and the website URL of a service. After SOMBRA’s automated a11y information scanning, SOMBRA outputs all elements in the mobile-browser-rendered websites that are accessible to a11y services but inaccessible in their Android app counterparts.

3.3.1 App-Side a11y Model

The first step for SOMBRA to vet the attack surface is to understand how a11y protection to native Android apps is implemented and formulate an app-side a11y protection model. I

scoured the Android a11y service implementation [65] to find all mechanisms that allow app developers to modify a11y output or restrict a11y access to a11y services. Figure 3.3 shows the app-side a11y protection model available to app developers. Since developers can directly control the declaration and implementation of native views, they can adopt both static and dynamic a11y protection mechanisms. For static a11y protection, developers can declare `a11yDataSensitive` label to views. During app runtime, Android interprets this protection label and only broadcasts a11y information within the protected view to Google-approved a11y services. For dynamic a11y protection, developers can execute a11y output modification logic during runtime while views are constructed. Specifically, developers can modify the `a11yEvent` a view broadcasts, modify the static `a11yNodeInfo` property of a view, and assign an `a11yDelegate` to take over the a11y information broadcast. Given the app-side protection model, SOMBRA can further discover and attribute app-side protected information during the app traversal (see subsection 3.3.3).

3.3.2 Browser-Side a11y Model

Next, to discover browser-side a11y leakage, SOMBRA needs to understand how a11y protection to browser-rendered websites is implemented in the web a11y standard. I studied the web a11y standard guideline [82] to extract all ARIA mechanisms that allow developers to change the a11y output or restrict a11y access to their website elements. Figure 3.4 shows the browser-side a11y protection model available to developers. With a given rendered web page, the only a11y-related fields in the DOM are static ARIA labels whether they are declared statically or assigned with JavaScript dynamically. In specific, the `aria-hidden` label allows developers to declare an element should be hidden from all a11y services. Alternative ARIA labels that allow developers to change the default a11y announcements of their fields include `aria-label`, `aria-labelledby`, `aria-describedby`, etc. However, since developers have no control over how browsers translate their ARIA labels, the actual exposed a11y information accessible to the Android OS in the AOM depends

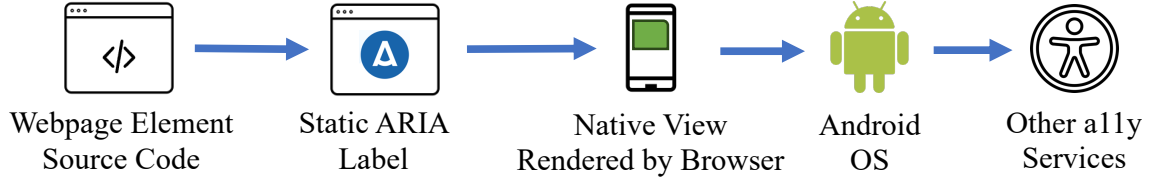


Figure 3.4: Browser-side a11y protection model. Web page DOM elements can only contain static ARIA labels. The elements with labels need to be translated by browsers and then interpreted by the Android OS.

on each browser’s implementation of a11y parsing logic. Given this extra layer of a11y translation, the same element declared by developers, if rendered with different browsers, may result in different a11y properties in views interpreted by Android and broadcast to a11y services. With the browser-side a11y protection model, SOMBRA is then able to extract exposed a11y information (see subsection 3.3.3).

3.3.3 Finding the Mismatch Between App-Side and Browser-Side a11y Protection

Given the misalignment of a11y protection between native Android app and browser-rendered web page components, SOMBRA next analyzes the Android app and mobile website of a given benign service and finds a11y leakage on the browser side.

a11y-Model-Guided App-Side Analysis

Pinpointing protected a11y information in the app is challenging because app developers use complex layered structures to organize the views shown to users in the frontend GUI. To reveal an app’s underlying a11y information, a system should exhaustively traverse a given user GUI screen together with all redirections and child screens, as well as explore layered view structures within groups of views. To achieve this, SOMBRA adopts a customized depth-first strategy to guide the discovery and hierarchy breakdown of a11y information within the GUI.

After the manual page setup (details in subsection 3.4.1), SOMBRA initializes the retrieval of a11y information from the user profile page. algorithm 3 shows SOMBRA’s

traversal strategy for exhaustively recording the ally information of GUI elements in the user account page. When the page is first initialized, SOMBRA captures the `TYPE_WINDOW_STATE_CHANGED` ally event broadcast by the app that indicates the start of a new GUI screen, as shown in Line 3. SOMBRA then retrieves the source node (represented as an *allyNodeInfo*) of the ally event and marks it as the root node of the user account page. Given the root node, SOMBRA adopts a depth-first pre-order traversal to visit all descendants of the node. For each node visited, SOMBRA records the ally text embedded and broadcast by the view, together with its properties such as `viewType` and `allyDescription`, as shown in Lines 8 and 9. When encountering clickable elements that can redirect to a new GUI screen, such as a `Button` or `ViewGroup`, SOMBRA prioritizes the traversal of the new window state triggered by clicking the element, as shown in Lines 10 and 11. SOMBRA then captures the new ally window state change event and continues the traversal. To keep a record of the layout hierarchy of the current page, given each GUI element, SOMBRA records all children of the node and pushes them into a stack for future processing, as shown in Lines 14-16. When all nodes within the current screen are traversed, SOMBRA issues `global_action_back` ally command to navigate back to the previous screen and continues the traversal, as shown in Lines 18-20. Finally, the traversal ends when no elements remain in the node stack.

After the app traversal of ally information concludes, SOMBRA records all ally nodes with null text properties or empty strings. I designed SOMBRA to ignore constant strings in ally nodes that are different than the nodes' visible text. That said, SOMBRA will miss developers using constant strings to protect sensitive fields (e.g., hiding a user balance amount with the string "user balance"). SOMBRA users could enable this but will introduce higher false positives caused by developers' deliberate definition of different ally node text (e.g. describing a "search" button as "search this app"). SOMBRA excludes an ally node when it is a view type that originally shouldn't have a text property such as `imageView`, `layouts`, `switches`, `toggles`, `scrollView`, etc. Now, SOMBRA has a collection

Algorithm 3: SOMBRA’s depth-first pre-order traversal of app-side ally node information.

```
1 nodeStack =  $\emptyset$ ;
2 Function onAllyEvent(event):
    // Capture the initial ally event when navigating to a page
3     if event.type == TYPE_WINDOW_STATE_CHANGED then
        // Extract the top-level source node of the event
4         node = event.getSource();
5         nodeStack.push(node);
        // Depth-first traversal of the children of the ally nodes
6         while !nodeStack.isEmpty() do
7             node = nodeStack.pop();
            // Record the ally node’s embedded text and type
8             record(node.text, node.viewType,
9                 node.allyDescription);
10            if node.isClickable() then
                // Click buttons, expandable views to navigate to new
                // screen change
11                node.actionClick();
12            end
13            else
14                for childNode  $\in$  node.getChildNodes() do
15                    nodeStack.push(childNode);
16                end
17            end
            // Traverse back to the previous screen when no new nodes
            // exist on the current GUI screen
18            if CurScreenNodeCount == 0 then
19                global_action_back();
20            end
21        end
        // End the traversal
22        return;
23    end
24 end
```

of elements whose ally information is intended to be hidden from unwanted ally services by app developers. This collection indicates the app-side ally protected GUI elements by app developers and will be used to compare with the same elements on the browser-rendered website elements.

Attributing App-Side Protection Mechanisms

Since each type of browser-side leakage is caused by the inefficacy of providing a strong app-side ally protection mechanism, SOMBRA first examines the app-side protected view

and finds its implemented app-side protection mechanism. However, attributing the app-side protection mechanism given a dynamically captured `ally` event is challenging. This is because multiple instances of the same view type can be instantiated by developers in the app, and multiple mechanisms, including both static XML declaration and dynamic view handler routine, can be used by developers to define and customize the `ally` behaviors of a view type. To accurately pinpoint all `ally` protection mechanisms declared by developers, a system must explore and capture all initialization characteristics involving a view. To achieve this, SOMBRA combines static `ally` label scanning with `ally` label data-flow analysis to resolve all `ally` protection mechanisms used by a view.

Given an `allyNodeInfo` property of a dynamically captured view that hides its `ally` information, as discovered in subsection 3.3.3, SOMBRA first captures its `ally` node resource ID, as this ID indicates the static view type declared by developers. SOMBRA then searches all static view declarations in the APK (all XML files in `res/layout`) to pinpoint the view properties that match the same resource ID. SOMBRA then extracts static `ally` labels declared by developers, in particular the `allyDataSensitive` label that makes it inaccessible to untrusted `ally` services. Then, to determine all dynamic `ally` customization made to the view, SOMBRA searches for all view initialization routines in Activities. Under all functions that allow customization to a view component such as `setContentView`, `inflate`, etc. SOMBRA first taints the view data structure with the same Android ID that matches the resource ID found in dynamic `ally` node capture. SOMBRA then propagates the taint and marks when it determines the tainted tag appears in any `ally` label modification function such as `setAllyDelegate`, `setAllyNodeText`, etc. Given all `ally` label customization routines specific to the view, SOMBRA finally maps the customization method to the four `ally` protection routines discussed in subsection 3.2.1.

Algorithm 4: SOMBRA ’s browser-side traversal for identifying elements corresponding to app-side ally-protected Views. SOMBRA matches element types and aria labels in addition to app-side traversal sequence when browser layout is different than the app layout.

```

1  stepIdx = 0;
2  Function startTraversal(appAllySequence):
    // Start from the root ally node in AOM
3  currentRoot = AOM.getRootAllyNode();
4  while stepIdx < appAllySequence.length do
5      targetStep = appAllySequence[stepIdx];
6      matchingNode = null;
    // Match app-side traversal sequence
7      nodeStack = {};
8      nodeStack.push(currentRoot);
9      while !nodeStack.isEmpty() do
10         node = nodeStack.pop();
    // Match app-side element with text and type
11         if node.matches(targetStep.text, targetStep.type) then
12             matchingNode = node;
13             break
14         end
15         for child ∈ node.getChildNodes() do
16             nodeStack.push(child);
17         end
18     end
    // When browser layout differs from app layout
19     if matchingNode == null then
    // Global page search with type and label
20         nodeStack = {};
21         candidates = [];
22         nodeStack.push(currentRoot);
23         while !nodeStack.isEmpty() do
24             node = nodeStack.pop();
25             if typeMatch(targetStep.type, node.role) then
26                 candidates.push(node);
27             end
28             for child ∈ node.getChildNodes() do
29                 nodeStack.push(child);
30             end
31         end
    // Match aria-label
32         for node ∈ candidates do
33             if node.ariaLabel == targetStep.contentDescription then
34                 matchingNode = node;
35                 break
36             end
37         end
38     end
    // Click buttons, expandable views to navigate to new screen
39     if targetStep.isClickable then
40         matchingNode.actionClick();
41         currentRoot = AOM.getRootAllyNode();
42     end
43     stepIdx ++;
44 end
45 return matchingNode;
46 end

```

a11y-Model-Guided Browser-Side Analysis

After obtaining the GUI elements in the app that are a11y-protected, SOMBRA next finds the same elements on the browser-rendered website and examines whether they are unprotected and contain a11y text information. However, locating the same GUI elements on the mobile browser-rendered website is challenging. A service’s website and app, although intended to convey the same information to users, usually are developed by two teams of developers with different content layouts. Even for hybrid services whose apps are translated versions of the website, their GUI element hierarchy still differs between the two versions. Additionally, since each mobile browser implements its own interpretation of the website layout and adopts its own translation of a11y ARIA labels, the sequence for an a11y service to traverse the same website differs depending on the browser that renders it.

However, I found during my research that in addition to the DOM, the AOM browsers created while rendering a web page contain crucial a11y node structural information and that neighboring element properties can help adjust and calibrate the traversal to adhere to the app-side traversal sequence. To accurately pinpoint the same elements on a service website rendered by different browsers, SOMBRA adopts a dynamic app-side-guided AOM traversal strategy. After the manual setup (see subsection 3.4.1), SOMBRA starts the traversal on the mobile-browser-rendered website in search of the corresponding app-side a11y-protected elements.

As shown in algorithm 4, SOMBRA first acquires the AOM created by the browser at the initial state of the current page. For each of the traversal sequences on the app side that led to the discovery of an app-side a11y protected element, SOMBRA matches the button and expandable view clicking sequences and searches for the element within the final landing page. If no clickable elements are present in the app-side traversal, SOMBRA matches the child hierarchy of the a11y-protected view to the DOM hierarchy on the web page, as shown in Lines 7-18.

When the browser page layout differs from the app layout and no element is found

Table 3.1: Element Type Rules SOMBRA Adopt To Match Web Page Elements To Their Android App Element Counterparts And Whether They Contain a11y Text.

HTML Element	Android Element	a11y Text
<div>	FrameLayout	✗
	TextView	✓
<p>	TextView	✓
<a>	TextView	✓
<h1>, <h2>, etc.	TextView	✓
<button>	Button	✓
	ImageView	✗
<input> _{text}	EditText	✓
<input> _{button}	Button	✓
<input> _{checkbox}	CheckBox	✗
,	ListView	✓
<select>	Spinner	✗

according to the app-side traversal sequence, SOMBRA conducts a global page search to locate the element, as shown in Line 19. SOMBRA first narrows down the element candidates by finding all website elements that correspond to the app-side views according to Table 3.1, as shown in Lines 20-30 of algorithm 4. For example, when an app-side a11y-protected `TextView` element's traversal sequence matches a `<span>`, `<p>`, `<a>`, or `<h1>`, etc., SOMBRA confirms that it is a valid candidate. Given these elements, SOMBRA further attributes the ARIA label to the app-side content description to infer the matching role of the element in Lines 32-37.

When a clickable element is matched, SOMBRA sends an *a11y action* to click it and continues the traversal on the updated page and its updated AOM, as shown in Lines 39-42. When a match of the app-side a11y-protected element is found in the DOM, SOMBRA then queries and records the *a11y role*, *a11y states*, and *a11y properties* fields within the element's AOM node and its *ARIA labels* declared in the DOM. SOMBRA finally compares the a11y text information accessible to any a11y services in the *a11y properties* fields with the a11y text field in the app-side element. If the browser-side information is not null or an empty string, SOMBRA confirms that the browser-side element leaks a11y information

otherwise protected on the app side.

Since the discovered browser-side a11y information visibility can be caused by four different types of browser leakage as discussed in subsection 3.2.1, SOMBRA next attributes the reason for each found browser-side a11y leakage.

Attributing Browser-Side Leakage

Since the development team of a service's app and website can be different, SOMBRA further needs to attribute the reason for the found browser-side leakage. If the app-side a11y-protected information is not declared with ARIA protection labels by website developers, the a11y leakage is caused by the deliberate inconsistency between a service's app and website development team. If the same app-side a11y-protected information is also declared with ARIA protection labels, the a11y leakage on the browser-side is inherent to the browser's translation of a11y information according to web a11y standards and cannot be avoided by website developers alone.

SOMBRA examines the *ARIA labels* declared in the DOM that correspond to the AOM element with the *a11y properties* leakage. If the element in the DOM does not contain either the `aria-hidden` label or alternative ARIA labels that can change an element's a11y announcement, as discussed in the browser-side a11y model subsection 3.3.2, it is the inconsistency between the website development team and the app development team that caused the a11y leakage. Otherwise, if such a label is present, the browser's render engine reveals the a11y information to a11y services according to the web a11y standard and causes the a11y information leakage.

SOMBRA now has finished vetting benign services' a11y information leaked in mobile browser-rendered websites but protected in their native apps. I discuss the developer's defense as well as mitigation to the attack surface in section 4.6.

3.4 Evaluation

I deployed SOMBRA to vet all-y-exposed information for browser users that are protected for app users in real services. App-side all-y information traversal is implemented in Java (1.1K lines) leveraging the Android all-y service [65]. Browser-side AOM retrieval and traversal is implemented in Python (0.5K lines) leveraging Appium [83], the SOTA mobile UI automation tool. Extracted all-y field categorization into common PII types is queried through Google Cloud natural language APIs [84]. Dynamic analysis of the applications and mobile browser-rendered websites is hosted on a Google Pixel 5 device running Android 14.

3.4.1 Dataset & Experiment Setup

Dataset. To collect a benign services dataset, I queried AppBrain [37], a state-of-the-art (SOTA) Android market intelligence service, for the top-150 free finance, shopping, and transportation Android applications in the U.S. This is selected according to the most abused categories of apps from the most recent Android all-y malware study [66]. For each application, I collected both the Android package name and its website URL (if it exists). To acquire their Android applications, I downloaded their most recent versions from AndroZoo [85], the SOTA Android application dataset used in top-tier research, resulting in a collection of 294 APKs, excluding duplicates, and have their service websites. I selected Chrome, Firefox, Brave, and Edge to render the services’ websites.

Experiment Setup. I created test credentials for 226 services that support signing up with email/phone numbers. I used pre-existing personal accounts for 23 services. I asked for and received test credentials from four services that require real accounts that I did not have personal accounts for. I failed to acquire valid test accounts for 41 services. For each service, I logged into the app and browser web pages, filled in personal information fields, and bound one Chase credit card and one Chase banking account when possible to mimic normal users’ sensitive information. I manually left the app and the website at the user account page for

SOMBRA to start the traversal because this page contains the most sensitive information. SOMBRA users can pick any page to start the traversal. For services for which I failed to obtain valid login credentials, I manually left them at the login page for SOMBRA to start the traversal. This conforms to the experiment setup procedures from prior work [86].

Table 3.2: SOMBRA’s Discovered a11y Leakage In Benign Services’ Websites Rendered In Chrome, Firefox, Brave, And Edge Browsers.

Category	Package Name	App-P. ¹	# P. ²	Chrome		Firefox		Brave		Edge	
				Ex. ³	w/o Dup. ⁴	Ex.	w/o Dup.	Ex.	w/o Dup.	Ex.	w/o Dup.
Finance	com.varomoney.bank	4	2	7	3	11	6	7	3	7	3
Finance	com.DailyPay.DailyPay	5	1	12	7	21	12	12	7	12	7
Finance	com.syf.mysynchrony	6	1	9	6	17	8	12	8	15	8
Finance	com.usaa.mobile.android.usaa	5	1	5	3	9	5	5	3	5	3
Finance	com.propel.ebenefits	3	1	3	2	6	5	3	2	3	2
Finance	com.meetcleo.cleo	2	1	10	7	25	19	10	7	16	11
Finance	com.intuit.turbotax.mobile	6	1	17	12	30	22	17	12	17	12
Finance	com.squareup.cash	4	1	6	3	15	10	9	6	6	3
Finance	io.metamask	7	2	9	7	16	13	9	7	9	7
Transportation	com.yandex.yango	2	1	8	3	13	8	8	3	8	3
Transportation	com.coulombtech	7	1	22	13	53	39	22	13	22	13
Transportation	com.trailbehind.android.gaiagsps.pro	6	1	8	8	19	9	8	8	10	8
Transportation	org.rajman.neshan.traffic.tehran.navigat	10	1	15	8	21	12	15	8	15	8
Transportation	net.sharewire.parkmobilev2	5	2	7	4	17	9	7	4	11	4
Transportation	com.xatori.Plugshare	3	2	7	5	12	8	7	5	7	5
Transportation	com.ventrarchicago.riderapp	11	1	14	9	18	10	14	9	14	9
Shopping	com.affirm.central	3	1	3	2	3	2	3	2	3	2
Shopping	com.puma.ecom.app	8	3	26	17	42	28	26	17	28	17
Shopping	com.route.app	42	2	93	16	179	28	87	16	78	16
Shopping	com.belk.android.belk	12	2	11	7	16	9	11	7	11	7
Shopping	com.dollargeneral.android	8	1	18	12	27	21	16	8	18	12
Shopping	com.myklarnamobile	4	2	8	7	10	7	8	7	8	7
Shopping	com.acehardware.rewards	1	1	5	2	6	2	5	2	7	3
Shopping	com.sneakerhotsapm.app	30	2	43	27	77	40	43	27	43	27
Shopping	com.cvs.launchers.cvs	2	2	6	4	8	6	6	4	6	4
Shopping	com.biglotsltds.biglotsam	5	1	9	5	9	5	9	5	9	5
Shopping	com.adidas.confirmed.app	9	2	21	16	44	20	21	16	25	19
Shopping	com.shopmium	40	1	79	15	132	23	86	17	81	15
Shopping	com.einnovation.temu	6	1	18	11	37	16	18	11	16	11
Total	29	256	41	499	241	893	402	504	244	510	251

1: Number of app-side a11y-protected views. 2: Number of types of Android native a11y protection mechanisms.

3: Number of elements exposed with a11y information on the browser-rendered websites.

4: Number of unique elements exposed with a11y information, eliminating duplicates caused by ViewGroup handlers.

3.4.2 Browser-Side a11y Leakage in Real Benign Services

Table 3.2 shows SOMBRA’s findings of browser-side a11y information leakage in the Chrome, Firefox, Brave, and Edge browsers in real-world benign services. I manually verified and confirmed these results. I conducted additional validation of SOMBRA that

shows SOMBRA can detect app-side all y protection and match browser-side elements with low false positives and false negatives (shown in section 3.7 due to space constraints).

As shown in the Total Row of Table 3.2, of 294 benign hybrid services I collected, SOMBRA discovered a total of 29 (9.9%) services that deploy at least one type of app-side all y protection mechanisms in their Android apps. Upon further investigation, I extracted their Android app manifest information and found that all 29 services have updated their apps to target Android 14, which provides enhanced all y protection mechanisms such as `all yDataSensitive` declarations to help protect apps from non-Google-approved all y services. I expect that more developers will gradually update and adopt the new app-side all y protection mechanisms.

Columns 1 and 2 of Table 3.2 show the benign services' category and package name. Out of the 29 apps that adopt app-side all y protection, nine (31.0%) are finance apps, seven (24.1%) are transportation apps, while the remaining 13 (44.8%) apps are shopping apps.

Columns 3 and 4 of Table 3.2 show the number of view elements that are all y protected discovered by SOMBRA in the app-side dynamic analysis and the number of all y protection types they adopted, respectively. As shown in the Total Row of Columns 3 and 4, a total of 256 views are all y-protected, with an average of 8.8 views protected in each app. The number of views protected within each app varies significantly across different apps. For the largest numbers of app-side all y-protected views, `com.route.app` contains 42 with two types of protection, namely `all yDataSensitive` and customized `all yEvent` handler. `com.shopmium` also contained 40 views protected by `all yDataSensitive` fields. For the least number of all y-protected views, `com.acehardware.rewards` only protects one view, which is the account login password `EditText` view, using the all y password protection. As shown in Column 4, the majority of apps (18) adopt only one type of all y protection, 10 apps adopt two different types of all y protection, and only one app (`com.puma.ecom.app`) adopts three types of protection (`all yDataSensitive` label, customized `all yEvent` handler, and all y password protection).

Columns 5 - 12 of Table 3.2 show the number of a11y-exposed elements discovered by SOMBRA in the benign services' websites rendered by four different mobile browsers (Chrome, Firefox, Brave, and Edge). The detailed exposed content type is discussed in subsection 3.4.3). As shown in the Total Row of Table 3.2, the 29 benign services' websites rendered in Chrome, Firefox, Brave, and Edge mobile browsers exposed a total of 499, 893, 504, and 510 elements that correspond to the app-side a11y-protected views; that is 1.9x, 3.5x, 2.0x, and 2.0x more than their app-side counterparts. Upon further investigation, I found that the reason more elements are exposed on the browser side is that all elements within a `ViewGroup` element, including expandable views rendered in the browsers, inherit the a11y text information in the AOM. This means that for a single view in an Android app, its parent or child element in the browser-rendered element should there be any, all contain the exposed a11y information. To mitigate this duplication and avoid over-counting, I eliminated the duplicates and showed the unique elements exposed on the browser side in Columns 6, 8, 10, and 12. As shown in these columns, a total of 241, 402, 244, and 251 unique elements corresponding to the app-side protected views are exposed in the Chrome, Firefox, Brave, and Edge browsers, respectively. Averaging across all benign services, they exposed an average of 8.3, 13.9, 8.4, and 8.7 elements per service.

While rendering the same website, I found that the Firefox browser in particular contains more a11y-exposed views than the other three browsers. I studied the rendering logic and found that the Chrome, Brave, and Edge browsers have similar rendering logic because they all adopt the same Blink render engine, which is responsible for interpreting the website DOM structure and translating the ARIA labels into the AOM. The Firefox browser, on the other hand, adopts the Gecko render engine, which differs in AOM construction logic. Specifically, while both render engines implement the same `aria-hidden` label translation logic by excluding it from the AOM tree, they treat elements marked with alternative labels, such as `aria-describedby` and `aria-labelledby`, differently. For each DOM element that contains an alternative ARIA label, the Chrome, Brave, and

Edge browsers utilizing the Blink render engine only expose the a11y text information in the alternative element, while the Firefox browser that utilizes the Gecko render engine exposes the a11y information in both the original and the alternative element.

Takeaway. SOMBRA identified a total of 256 app-side a11y-protected views across 29 benign services' Android apps. They adopted at least one and at most three types of Android a11y-protection mechanisms. While examining the benign services' websites, SOMBRA discovered that 241, 402, 244, and 251 elements matching their a11y-protected app-side counterparts are a11y-exposed in the Chrome, Firefox, Brave, and Edge browsers with 8.3, 13.9, 8.4, and 8.7 exposed elements per service. The Firefox browser in particular exposes more elements than the other three browsers because of the difference in adopted render engines. Specifically, the Gecko render engine adopted by Firefox has different interpretation logic for elements declared with alternative ARIA labels such as `aria-describedby` and `aria-labelledby`.

3.4.3 Security Impact

I categorized the types of information exposed to a11y services on benign services' websites that are a11y-protected in their apps. The exposed information causes compromised credit card/account/password and PII stalking to browser users. For each a11y-exposed field, SOMBRA extracts its a11y label and/or hint text fields and uses Google Cloud natural language APIs [84] to categorize them into common PII types.

Table 3.3 shows the extracted a11y information category exposed in Chrome-rendered websites while protected in their Android app-side counterparts. Column 1 shows the top 10 benign services with the most a11y-protected views in their apps. Column 2 shows unique elements protected in their apps but exposed in Chrome-rendered websites. As shown in the Total row of Table 3.3, a total of 241 elements are exposed out of the 29 apps that adopt app-side a11y protection mechanisms, an average of 8.3 elements per app.

Columns 3 - 7 show the a11y field types of the exposed elements such as passwords,

Table 3.3: App-side Protected a11y Information Leakage Category In Top Services' Websites Rendered With Chrome.

Name	Chrome Ex. Elements ¹	Psw.	Act. / Card	Key / Ident.	Addr. / Contact	Others
route	16	0	4	2	5	5
shopmium	15	1	5	0	0	9
sneakerapm	27	1	11	0	0	15
belk	7	0	4	1	0	2
ventra	9	0	0	3	0	6
neshan	8	1	4	1	0	2
confirmed	16	2	3	2	7	2
puma	17	1	4	0	4	8
dollar general	12	2	6	2	0	2
metamask	7	1	3	0	0	3
Others	107	9	39	18	24	17
Total	241	18	83	29	40	71

1: Number of unique elements protected in app but exposed in Chrome-rendered websites.

account or credit card, key or identifiers, address or contact, etc. As shown in Column 3, a total of 18 password fields are a11y-exposed in the Chrome-rendered websites. The *puma* app, which adopts app-side a11y password protection (see subsection 3.2.1), does not contain any browser-side protections, making the passwords accessible to a11y attackers eavesdropping on its mobile website. A total of 83 (34.4%) elements expose user account or credit card information fields in the Chrome-rendered websites. As shown in Columns 5 and 6, a total of 29 elements contain keys or identifiers such as tokens, wallet IDs, etc., while 40 elements expose address or personal contact information. In-depth examples of browser-side a11y leakage that causes compromised financial accounts and passwords to browser users are further illustrated in section 3.5.

Takeaway. SOMBRA uncovered a total of 241 elements with exposed a11y information in Chrome-rendered services' websites. While their corresponding app-side views are a11y protected, the browser-side elements leak common sensitive PII information such as account or credit card, keys or identifiers, address or contact, and passwords. This exposure causes

Table 3.4: Developers’ App-side And Browser-side a11y Information Protection Adoption Types.

Name	App-Side		Browser-Side		
	# Views	P. Types ¹	ARIA Hidden	ARIA Change	No Protection
route	42	①, ③	0	4	38
shopmium	40	①	0	0	40
sneakerapm	30	①, ②	2	11	17
belk	12	②, ③	0	3	9
ventra	11	②	3	0	8
neshan	10	①	0	2	8
confirmed	9	①, ②	0	0	9
puma	8	①, ②, ④	0	0	8
dollar general	8	①	0	5	3
metamask	7	①, ②	0	0	7
Others	79	–	7	23	49
Total	256	–	12	48	196

1: Android app-side a11y protection types. ①: a11yDataSensitive label, ②: custom a11yEvent, ③: custom a11yDelegate, ④: password protection.

compromised financial accounts/passwords and PII stalking to browser users. Among the 241 leaked elements, 34.4% contain user account or credit card information, while 7.5% contain user account passwords.

3.4.4 Developers’ App-Side and Browser-Side a11y Protection Adoption Comparison

With the extracted a11y information leakage in the browser-rendered websites, SOMBRA next compares the a11y protection adopted by the app and website developers. Table 3.4 shows the ARIA protection labels declared in benign services’ websites corresponding to each app-side a11y-protected view.

Column 1 of Table 3.4 shows the 10 benign services that have the most views protected against a11y attackers in their Android apps. Columns 2 and 3 show the number of a11y-protected views in these apps, as well as the types of Android a11y protection mechanisms adopted in these views. As shown in the top row of Columns 2 and 3, the *route* app

protected 42 of its views, which is the most among all 29 apps that adopted app-side a11y protection. The *route* app used two types of Android a11y-protection mechanism, declaring `a11yDataSensitive` labels and assigning customized `a11yDelegate` to views to alter their a11y exposure. The next apps that protected the most app-side views are *shopmium* and *sneakerapm*, containing 40 and 30, respectively. However, *shopmium* only utilized the `a11yDataSensitive` label to wrap all 40 views, while *sneakerapm* utilized both the label and customized `a11yEvent` handlers. Looking at Column 3, eight of the top 10 apps (80%) adopted the `a11yDataSensitive` label in Android 14 to protect them from being accessed by non-Google-approved a11y services. Four of the top 10 apps generate customized `a11yEvent` for the protected views, while two assign customized `a11yDelegate`. The *puma* app also protects its login password field with a11y-password protection subsection 3.2.1.

While the benign services' app developers have the aforementioned methods to protect their app-side a11y information, the misalignment between app-side and browser-side a11y protection limits the ways to protect their website a11y information. Columns 4 - 6 show the ARIA labels declared by benign services' website developers. Only one browser-side element is counted for each matching app-side a11y-protect view to avoid duplication.

SOMBRA found from the DOM structures that only 12 out of 256 (4.7%) of app-side a11y-protected views are completely hidden on the browser side (Chrome, Firefox, Brave, and Edge all exclude `aria-hidden` elements from the AOM, thus making them inaccessible to any a11y services). Among the top-10 services that protected the most app-side views, only the *sneakerapm* app and the *ventra* app protected a total of five elements with `aria-hidden` label. Although the label provides the strongest protection on the browser side, it completely disallows any a11y services to access them, thus hindering the usability of the website for users of benign a11y utility apps. The inability to fine-grain the a11y access level to different a11y services makes this mechanism impractical for developers to adopt.

Column 5 shows the number of elements that are declared with `aria-label`, `aria-labelledby`, and `aria-describedby` labels to change the `ally` field exposed and announced to `ally` services. Looking at the Total row, of the 256 app-side protected views, 48 (18.8%) contain the label to alter the browser-side elements' `ally` response. However, as discussed in subsection 3.3.2, although the website developers can alter the default ARIA announcement, it is up to the browser's translation and interpretation to determine the actual exposed `ally` information to the AOM and subsequently to the Android `ally` services. As shown in subsection 3.4.2, the Chrome, Firefox, Brave, and Edge browsers all still include the original and altered `ally` information in the AOM, making them visible to any `ally` services. Although some (18.8% elements) developers adopt the alternative labels, browsers' interpretation according to the website `ally` model renders them ineffective at limiting the `ally` exposure to `ally` services.

Column 6 of Table 3.4 shows the number of elements that are completely free of any ARIA labels. As shown in the Total row, 196 (76.6%) of 256 elements protected in their app-side counterparts have no ARIA protections declared on their websites. The *shopmium* app, although having 40 app-side protected views, has no `ally` protection for any of them on the browser-side. Similar behavior also exists in other apps such as *confirmed*, *puma*, and *metamask*. Not declaring ARIA protections ensures the usability of the website elements to all `ally` services but at the same time makes them vulnerable to `ally` malware.

Takeaway. While SOMBRA discovered 256 `ally`-protected views declared by Android app developers, few website developers declare ARIA labels to protect their browser-side `ally` information. Specifically, 12 (4.7%) of the 256 elements declare `aria-hidden` labels to hide them from all `ally` services. Since this protection makes the `ally` content inaccessible to all `ally` services, it hinders the usability of benign `ally` utility apps. While 48 (18.8%) of the elements are declared with alternative ARIA announcement labels, browsers still expose the `ally` information in the browser-side AOM and subsequently make them visible to `ally` services due to the existing web `ally` standard, making the protection ineffective.

Most developers (196 / 256 elements) adopt no browser-side a11y protections, ensuring the functional usability of their content.

3.5 Case Studies

Browser-Side a11y-Leaked Bank Account Number. Klarna is a hybrid shopping service provider with access in both its Android app and their website. Its app is one of the most popular shopping apps in the Google Play Store with 10M+ downloads. The user setting page in both the app and website allows users to bind and view bank accounts for purchases. Figure 3.5 shows SOMBRA's a11y information extraction from both the Android app and mobile website rendered in Chrome while traversing the user payment method page. SOMBRA is able to extract users' sensitive bank account numbers through a11y access in the Chrome-rendered mobile website while unable to extract the same field in the Android app counterpart.

Figure 3.5a shows the screenshot of the payment method page of its Android app. During the a11y page traversal, SOMBRA discovered that the `ViewGroup` labeled as *Account Number* has a child element that has a null value in its a11y text field. After matching the child view's a11y resource ID with Klarna app's static view declarations extracted from the APK, SOMBRA found that the `TextView` representing the *Account Number* field is declared with the Android `A11Y_DATA_SENSITIVE_YES` property. This effectively forces the Android OS to only broadcast its a11y information to Google-approved a11y services with the *a11yTool* verification. Any non-Google-approved a11y services such as the one used by SOMBRA and the ones used by a11y malware cannot access this a11y-protected information.

Figure 3.5b shows the screenshot of the same payment method page of Klarna's mobile website rendered by Chrome. As shown in the figure, the structural hierarchy of views resembles that of its Android app counterpart. Utilizing the same view hierarchy traversal sequence as SOMBRA did in the Android app, SOMBRA found an element with the same

Account Number label. However, the account number is visible in the node's `ally` text field and SOMBRA is able to retrieve the information. With access to the DOM tree during the traversal, SOMBRA found that no ARIA labels are declared for the *Account Number* field. As discussed in subsection 3.4.4, most mobile website developers refrain from declaring `aria-hidden` protections to their sensitive information because it renders the field inaccessible to all `ally` services, including benign utility ones, and hinders the usability of their websites. With no ARIA protection in the DOM, the Chrome browser then renders the `ally` node fully visible in the AOM. Subsequently, Android extracts the field's `ally` text from the AOM and broadcasts it to all registered `ally` services, making them accessible to all `ally` services registered on users' devices, including the ones controlled by `ally` malware.

Browser-Side `ally`-Leaked Password. As discussed in subsection 3.4.4, since the `aria-hidden` protection renders the browser-side elements inaccessible to all `ally` services, some developers seek other ways to protect their content by declaring ARIA alternative labels that customize the announcement of a field when accessed by an `ally` service while making the elements still accessible. These alternative labels include `aria-label`, `aria-labelledby`, `aria-describedby`, etc. Although they are only intended by the web `ally` standard to improve website usability, their ability to customize the `ally` announcement of elements motivates developers to protect their browser-side elements.

Figure 3.6 shows the screenshot of Dollar General's Chrome-rendered website's login page. While SOMBRA's website traversal engine enters the registered password, SOMBRA's `ally` service component is able to see and retrieve the newly entered digits by listening and parsing the screen change events. SOMBRA found that the password input box is declared with an `aria-label` field and set to a constant string "password." However, the browser still renders every input digit in the AOM. This is because the existing web `ally` standard only recommends end-level `ally` services to announce the alternative label while still advising browsers to make the field visible in the AOM. As a result, declaring

alternative ARIA labels is ineffective at protecting against browser-side a11y leakage. Any a11y services on the device, including the ones used by a11y malware, can still access the fields declared with ARIA alternative labels.

3.6 Discussion

Limitations. Because Google implements fine-grained a11y protections on the app side, my study uses those protections as a baseline for what should be protected on the browser side. That said, if app developers accidentally misconfigure their app-side a11y protection for a view, SOMBRA will miss the corresponding browser-side element. Detecting misconfigurations in the app is out of scope, as I aim to align the existing protections. Additionally, SOMBRA requires valid credentials to test each service. However, SOMBRA users (service developers) should not face this challenge.

WebViews and Custom Tabs. During my study, SOMBRA found that 138 apps contain WebView [87] and 47 contain Custom Tab [88] elements. However, no a11y information is protected in those WebView and Custom Tab elements. This is because both use the Chromium-based render engine and Android’s native a11y protection mechanisms are inapplicable to them.

Developer’s Defense. As shown in subsection 3.4.4, the existing browser-side `aria-hidden` mechanism sacrifices the usability of website content to benign a11y services by removing a11y content entirely from the AOM. Alternative a11y labels are also ineffective at hiding sensitive a11y information due to the existing web a11y standard. To allow developers to protect their browser-side a11y content while ensuring usability, I recommend they remove the sensitive content from their website and redirect the user to their Android app counterpart for access. For example, banking apps such as Chase advise users to install or redirect to the banking apps to conduct transactions.

Mitigating the Attack Surface. The existing Android native a11y support intends to both improve the usability and security of app content, while the existing web a11y standard

only focuses on usability. To fundamentally mitigate this attack surface, the misalignment of app-side and browser-side a11y protection mechanisms needs to be eliminated. This would ideally consist of a three-party collaboration that involves redesigning the web a11y standard, enforcing browsers' interpretation of the web a11y model, and adapting Android's translation of browser-rendered AOM. At the base level, the web a11y standard should provide developers more freedom to fine-grain the level of a11y access to different a11y services. For example, mobile website developers can be allowed to delegate the screening of legitimate a11y services to Android and declare their content to be only accessible to Google-approved a11y services. At the browser level, a third-party should be introduced to enforce each browser to implement a consistent and correct interpretation of the same a11y model declared by website developers. This ensures that the a11y content access level does not differ among different browsers that the end users choose to use. Finally, the Android OS needs to adapt to the newly introduced web a11y standard with fine-grained access to ensure each protection type is fully translated and broadcast to on-device a11y services.

Disclosure and Open-Source. I disclosed my findings to the Chrome, Firefox, Brave, and Edge teams. At the time of writing this paper, I received confirmation from the Firefox and Edge team, acknowledging the exposure of a11y information in browser-rendered elements. I also disclosed my findings to all 29 service developers who implement native a11y protections in their Android apps but leave their counterparts exposed on their websites. I recommended that they remove the sensitive content from their websites and redirect users to their Android app counterparts for access. Unlike addressing a traditional vulnerability, mitigating browser-side a11y leakage requires a long-term redesign of the web a11y standard, as the current standard lacks the fine-grained a11y access control available in Android. I hope my findings can advocate for a three-party collaboration to mitigate this attack surface, as discussed earlier in section 3.6. Finally, SOMBRA is open-sourced and available at <https://github.com/CyFI-Lab-Public/SOMBRA>.

Table 3.5: Validation Of SOMBRA’s Detection Of App-side Protected Views And Browser-side Exposed Views.

Name	App-Side			Browser-Side ¹		
	SOMBRA ²	FP	FN	SOMBRA ³	FP	FN
MySynchrony	6	0	0	9	0	0
MetaMask	7	0	0	9	0	0
Varo Bank	4	0	0	7	0	0
ParkMobile	5	0	2	7	0	2
ChargePoint	7	0	0	22	0	0
Ventra	11	0	0	14	0	3
Klarna	4	0	0	8	0	0
CVS	2	0	0	6	0	0
Belk	12	0	0	11	0	4
Dollar General	8	0	0	18	0	0
Total	66	0	2	111	0	9

1: Rendered by the mobile Chrome browser.

2: SOMBRA’s detected app-side protected views.

3: SOMBRA’s detected browser-side exposed views.

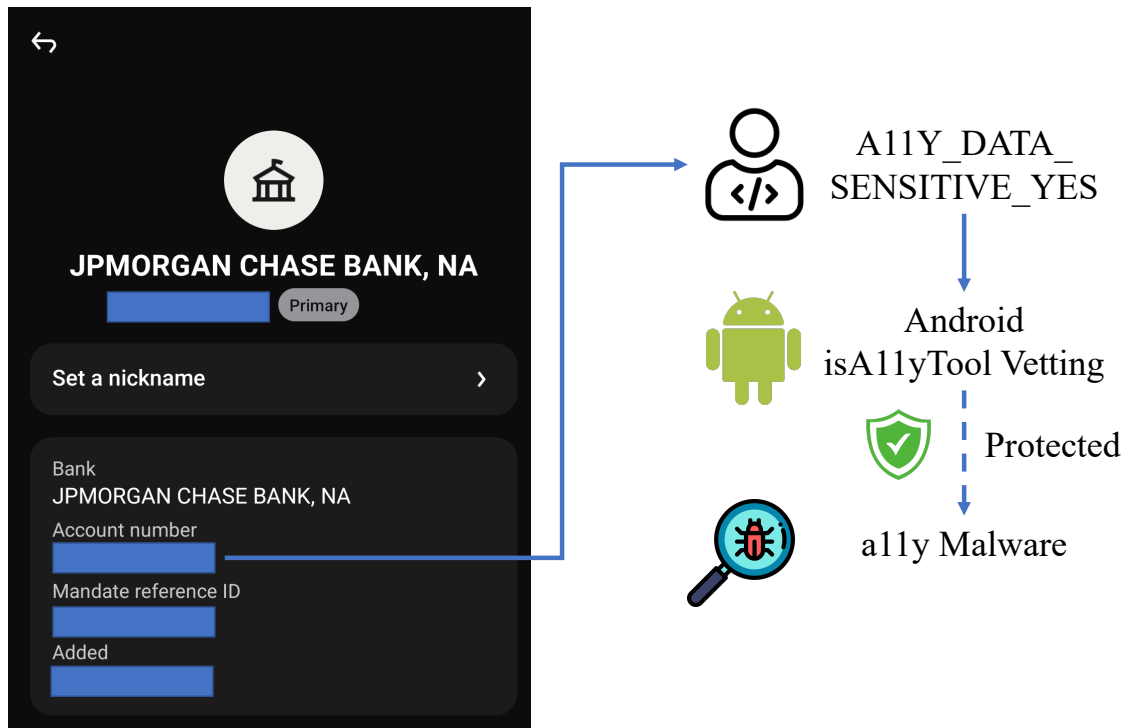
3.7 Validation

I validated SOMBRA’s accuracy in detecting app-side protected views and matching them with browser-rendered elements. From my dataset, I randomly selected an app and included it in the validation set if it had at least one app-side protected native element identified by SOMBRA, continuing this process until I found 10 such apps. This approach ensures the existence of app-side ground truth for all y-protected views, enabling comparison with the corresponding browser-side elements. I then installed them on the Google Pixel 5 device running Android 14 and navigated to the Chrome-rendered websites of the 10 apps on the same device. I obtained the ground truth of app-side and browser-side protection utilizing Android’s built-in TalkBack [89] service, manually traversing the app’s and website’s user profile screen, retrieving the text of each on-screen element and comparing them with the embedded all y text broadcast in Viewall yFocused [90] events.

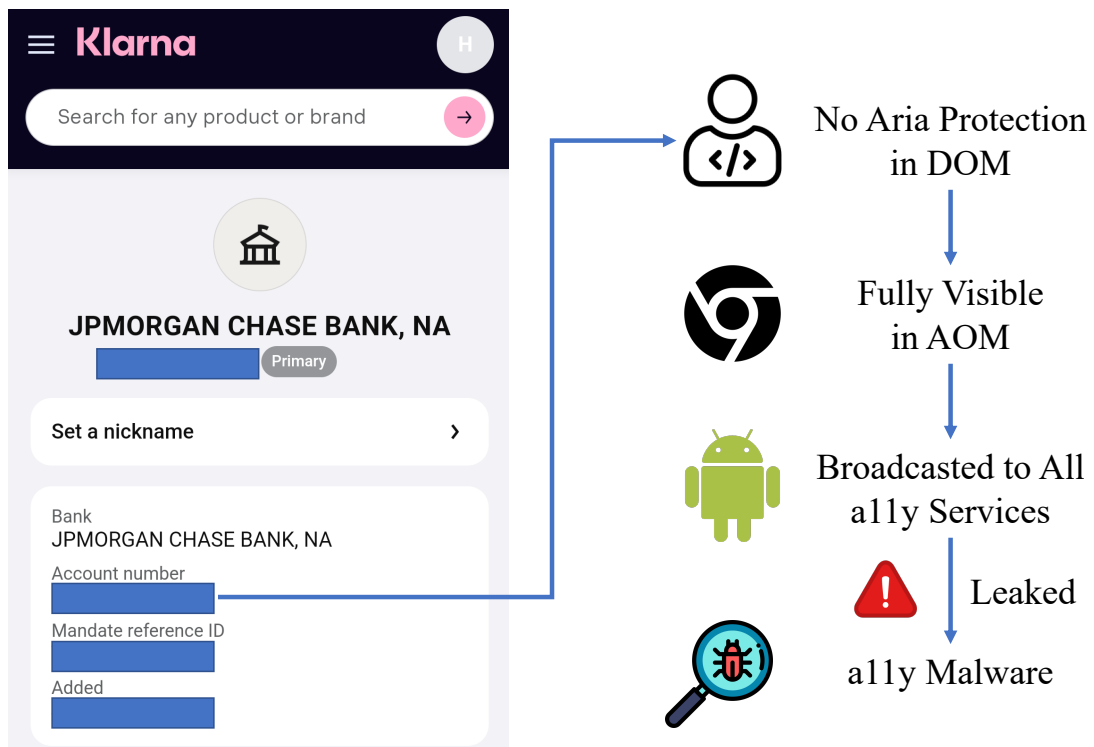
Validation Results. For app-side validation, as shown in Columns 2-4 of Table 3.5, SOMBRA has 97% accuracy in detecting app-side protected views. SOMBRA missed two

(FN) while analyzing the Park Mobile app. Upon further investigation, I found that the Park Mobile app broadcasts fixed placeholder addresses for two views that represent users' addresses, acting as an effective ally protection. I confirmed that this is a rare occurrence.

For browser-side validation, as shown in Columns 5-7 of Table 3.5, SOMBRA achieved 93% accuracy in matching and detecting browser-side exposed elements. Because of the two FN app-side views missed by SOMBRA in the Park Mobile app, SOMBRA also missed the two corresponding browser-side elements. For the Ventra app, SOMBRA missed three elements (FN) that match the app-side protected user account number and balance fields because of missing element labels by website developers. Similarly in the Belk app, SOMBRA missed four elements (FN) that match the app-side protected reward member number and contact detail because of empty element labels as well.



(a) The account number field is a11y-protected in the Klarna app and inaccessible to a11y malware.



(b) The account number field is not a11y-protected in the Klarna mobile website rendered by Chrome and accessible to a11y malware.

Figure 3.5: Klarna's a11y implementation of the user bank account page in both the app and the mobile website.

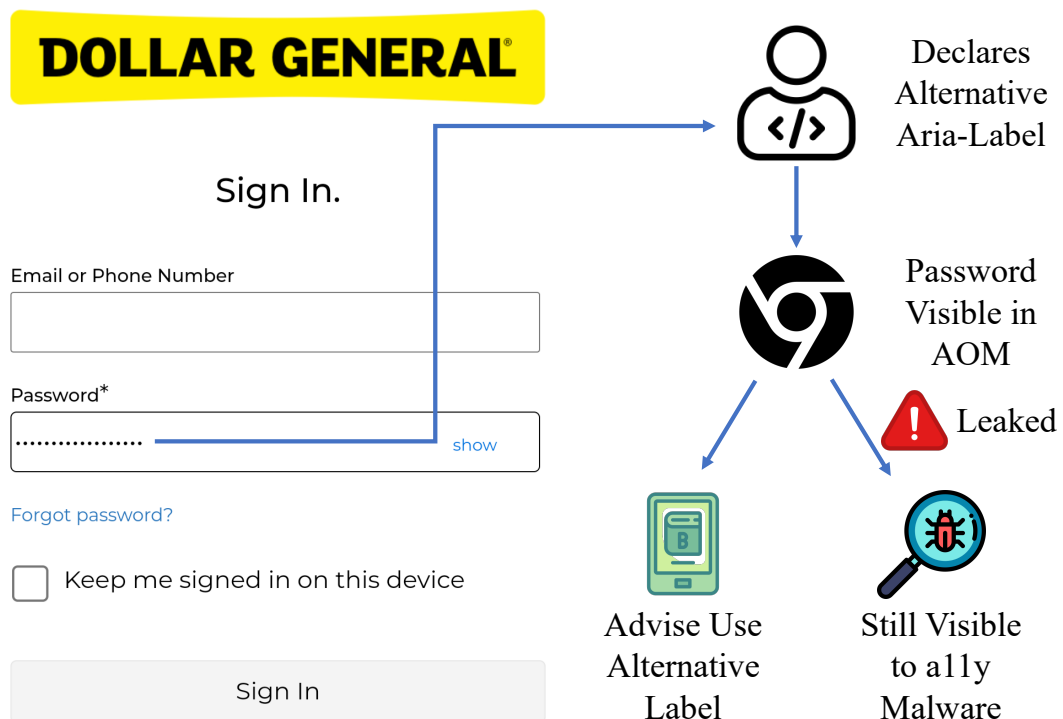


Figure 3.6: Password leaked in Chrome-rendered Dollar General login page. Declaring alternative ARIA labels is ineffective at protecting browser-side all y information.

CHAPTER 4

RECOVERING AND REHOSTING MOBILE LOCAL LLM CONVERSATIONS AND CONTEXTS VIA MEMORY FORENSICS

Local LLM apps are increasingly more popular as edge computing devices grow more powerful [5, 6, 7]. They power a variety of applications — from search-engines to virtual assistants [91], document summarization [92] to personalized tutoring [93] — all while running directly on the user’s own device. By operating locally, such models give users greater control over their data and privacy compared to cloud-hosted alternatives, because no proprietary provider processes or logs every query and response. Because of their strong privacy protections, a suspect who uses a local LLM app to plan or coordinate a crime makes their traces covert. When this happens, it is crucial for an investigator to reveal the previous conversations the suspect had with the LLM app, and even better, to interact with the LLM to understand and reveal the suspect’s additional actions, goals, and motives.

Unfortunately, no research exists to provide investigators with such evidence. Investigators have two options for where to look for evidence: in app storage (internal and external storage for an Android app [94]) on disk and in the runtime memory of the LLM process. Existing disk forensics work focuses on extracting OS and app storage artifacts such as sensitive data [16, 17] and log messages [18, 19]. However, when the suspect deletes the conversation from the app UI, corresponding disk artifacts will also be removed, leaving no evidence for recovery. Traditional memory forensic techniques can recover kernel objects and app/OS specific data structures [20, 21]. Data structure recovery from LLM apps would enable evidence recovery, but it falls short of rehosting the LLM so the investigator can interact with it. Few prior memory forensic work recovered and rehosted deep learning (DL) model internals [95], and Android app GUIs [28, 27]. However, no prior work has considered rehosting a complex layered software stack required for LLM rehosting, e.g.

model context manager, token sequences, KV cache, etc. Rehosting LLMs creates unique challenges due to the complexity of LLM internals. Because of how LLM apps manage the context, part of the context is dereferenced (logically evicted and forgotten in the current session) but not overwritten in memory. Without a deep understanding of LLM internals, no existing research has been able to take advantage of the forgotten context beyond the model’s context limit, realign, extend, and rehost them into a live session.

LLM apps commonly use KV cache to preserve the attention of the model and batch tokens to represent the vocabulary of the context [96]. To continue generation for a conversation, both are required to represent the existing conversation context. During my preliminary study (section 4.2), I found that local LLM apps preserve both the KV cache and the batch tokens in memory until overwritten by new conversations. This gives me a unique vantage point to reconstruct previous conversation context by recovering both from memory. However, unlike application-level chat records, tokens and KV cache are encoded, unstructured, and lack boundary markers, making their interpretation challenging. Furthermore, the active regions are dereferenced once a session is deleted in the UI, making the recovery and validation difficult.

Additionally, since LLMs have intrinsic context limits, local LLM apps keep track of the active KV cache region according to the session’s context limit. I discovered that since the available allocated memory for the app is usually larger than that required to represent the model’s context, logically evicted and forgotten context still resides in memory before being physically cleared and recycled by the OS. In fact, ORISA found an average of over 30% of tokens and KV tensors beyond the context limit are still present (subsection 4.4.1). This gives me an opportunity to recover extra forgotten context from an earlier conversation and rehost it into a live session for investigators to interact with. However, recovering these logically evicted tensors are challenging because the LLM sessions only book-keeps positions of tensors within the currently active context window. Even when recovered, the session will never incorporate them to continue inference without properly extending and

realigning the active region.

Based on these insights, I developed ORISA¹, an automated system for recovering conversation histories and rehosting the context of a local LLM app into a live session for investigators to interact with. ORISA takes an LLM app process’s memory image and its app storage as input. ORISA first performs model binary extraction from disk (subsection 4.3.1). ORISA then does batch token recovery (subsection 4.3.2), giving investigators evidences and a starting point for interrogation. Next, ORISA conducts KV cache recovery and active KV extension (subsection 4.3.3, subsection 4.3.4) to recover and realign both active and forgotten context from the previous session. ORISA then reconstructs and rehosts the recovered context into an app-agnostic session (subsection 4.3.5), allowing investigators to ask additional questions to the live session with previous conversation context. Finally, I will open-source ORISA upon the paper’s acceptance.

I evaluated ORISA with 60 configurations of LLM app sessions consisting of 10 Android Local LLM apps, with 3 different models under 2 different context windows. For tokens and KV cache recovery, ORISA recovered 100% of batch tokens and KV cache tensors from within the sessions’ active context window and 32.60% and 38.59% beyond it. ORISA’s rehosted session retained an average of 169.6% of knowledge compared to the original context, enabling investigators to interrogate the rehosted session with the forgotten context of the session. Finally, ORISA’s rehosted sessions with the same context window preserved the response behaviors of the original session, allowing investigators to treat the rehosted sessions’ interrogation outputs as credible evidence.

4.1 Overview

Cloud-hosted LLMs are traditionally deployed on centralized GPU/TPU clusters, where all user prompts and model outputs are processed on remote servers. These providers routinely log requests, responses, telemetry, and security metadata for service quality and

¹REcovering and rehosting LLM Interactions and Contexts

abuse detection. Major technology companies can also disclose such server-side data to law enforcement under legal processes such as subpoenas, warrants, or national security orders [97, 98, 99]. As a result, cloud LLM interactions leave extensive traces, including network traffic, server logs, and access metadata that investigators can obtain during criminal inquiries.

In contrast, local LLMs run entirely on the user’s device, typically optimized via runtimes such as llama.cpp [100], GGML [101], or other on-device inference backends. For suspects seeking to avoid cloud visibility, local models are especially attractive. Prompts and responses never leave the device, eliminating server-side audit trails or logs that investigators can obtain via legal processes. In these cases, the device itself becomes the only meaningful evidence source, motivating the need for forensic techniques that can recover and rehost the local LLM context.

4.1.1 Key Components of Local LLMs

Figure 4.1 shows the overview of a mobile LLM app’s workflow of generating chat session responses from user input. Local LLM inference necessitates several artifacts that have to reside on disk and in memory, giving investigators a unique vantage point to recover evidence.

Model Binary. To load a local LLM model, the model binary itself has to reside on disk, as shown in Figure 4.1. This binary contains the serialized model with the learned parameters specific to the model. It also contains key metadata about the model architecture, including vocabulary size, embedding dimension, number of layers, RoPE parameters, etc. For a local LLM app to instantiate a live session, it has to first load the model binary from disk to the inference engine.

Batch Tokens. When a user interacts with a local LLM app, their textual input is first passed to the tokenizer, which encodes each piece of text into a unique integer ID from the model’s vocabulary, named as batch tokens, as shown in Figure 4.1. These integers are then stored

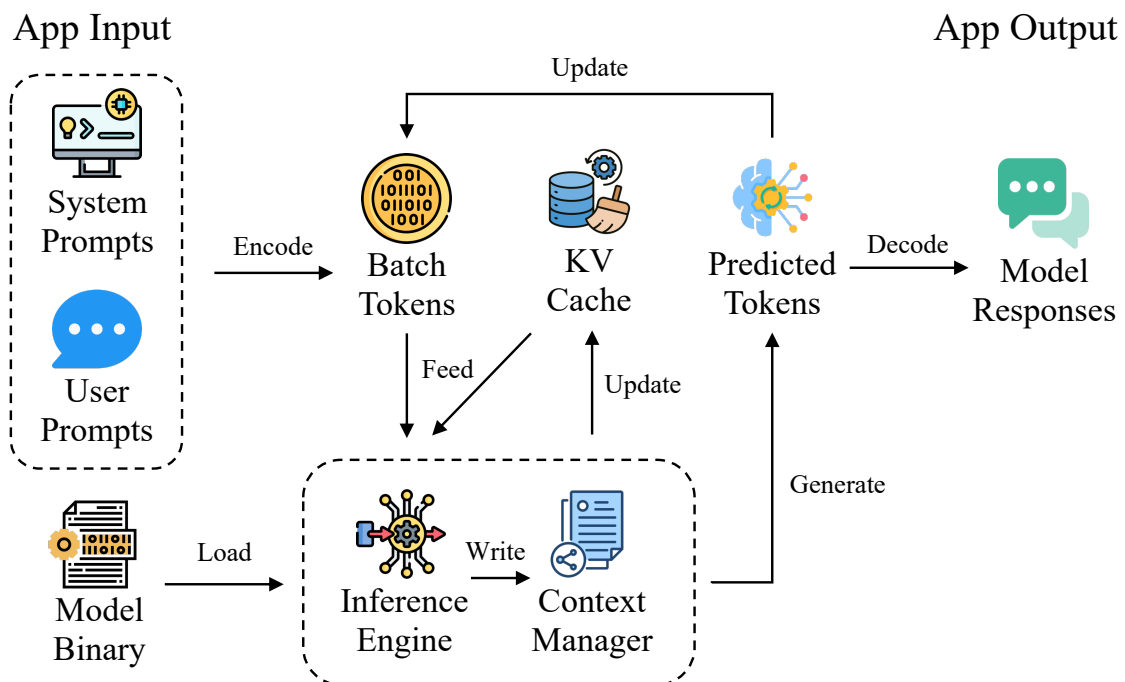


Figure 4.1: Overview of key components that enable inference in a local LLM app's chat session.

in token buffers in process memory. The token buffer acts as the linearized memory of the textual conversation so far. It also provides positional references used to index the KV cache (discussed below). Without this buffer, the model has no way to align its internal cache to the original text. Recovering and deserializing them allows investigators to reconstruct previous prompts and responses of a session as evidence.

KV Cache. The KV cache is a collection of tensors that store the LLM's internal attention state for every token already processed, managed by the inference engine's context manager, as shown in Figure 4.1. It represents the current conversation context and is updated with each new token. The K layer stores the projection of previous tokens into the key space used by self-attention layers, while the V layer stores the corresponding value projections. Even when a suspect deletes the original conversation text from the device's UI, the KV cache preserves the semantic embedding of those tokens. Recovering and rehosting the KV cache together with batch tokens allows investigators to resume generation of the session from where it was left off to further gather evidence.

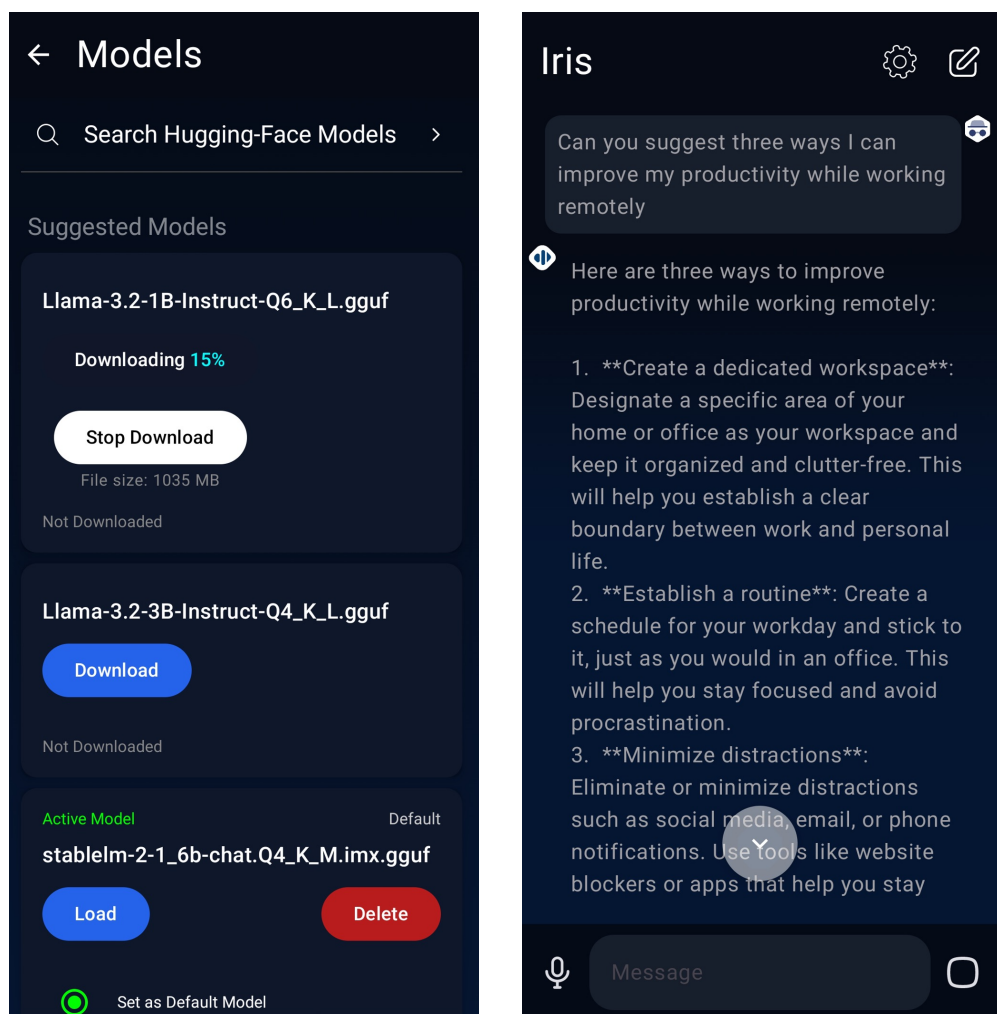


Figure 4.2: IRIS’s offline model download screen and chat screen, allowing users to use custom open-source models for completely local inference.

4.1.2 Threat Model

I assume that the suspect of interest is an average, normal user of a mobile device and utilizes a local LLM app on an Android device to coordinate and plan a crime. The suspect opens the app, engages in conversation with the local LLM, then deliberately deletes the chat history via the app UI, leaving no logs or transcripts on disk in app storage. They rely purely on local execution; thus, no server-side log is available. At the time of seizure, the device is still running the LLM app’s process. More advanced and technical suspects who can rewrite the app or use anti-forensics techniques such as those that obfuscate or

encrypt the app’s memory are discussed in section 4.6. Although they can still be handled by ORISA with additional perpendicular research, they are outside the scope of this paper.

4.2 Problem and Opportunity

During my preliminary study, I found a popular Android local LLM app on the Google Play Store [102] named IRIS [103] with over 10K+ downloads. It is advertised as ”Privacy-Focused” [104] and works completely offline without an internet connection. Figure 4.2 shows the model download and chat UI of the app. It allows users to download custom open-source models from Hugging Face [105] and use them for local inference. While interacting with the app, I dumped both the external (under `\sdcard\Android\data`) and internal (under `\data\data`) storage directory of the app and found that no cache or log of chat sessions is stored on disk. Clicking the new chat button also deletes the existing session. No way exists in the app UI to resume a previous chat session once deleted. This makes me wonder if there are still traces in memory that could reconstruct a previous chat session, if it has already been deleted in the app.

I then instrumented the app and dumped the memory image of the app process as I interacted and prompted the model in the app. In the memory image, I referenced `llama_batch` pointer and found token IDs that, when detokenized, matched with my previous prompts. In addition, when I examine the region pointed by `llama_kv_cache` pointer in `llama_context`, I found KV cache tensors that grow with conversation turns. Figure 4.3 shows the KV cache size in memory with respect to the turns of the current conversation, growing from 7.71 MB to 19.48 MB. I then deleted the chat session after turn 5 by pressing the new chat button in the IRIS UI and dumped the memory image again. I found that although the `llama_kv_cache` pointer is dereferenced in the context block, when examining the same region before the deletion, the KV cache tensor values remain intact and are not physically cleared. This gives me a unique opportunity to rehost a previous session from batch tokens and KV cache recovered from memory, even after the session

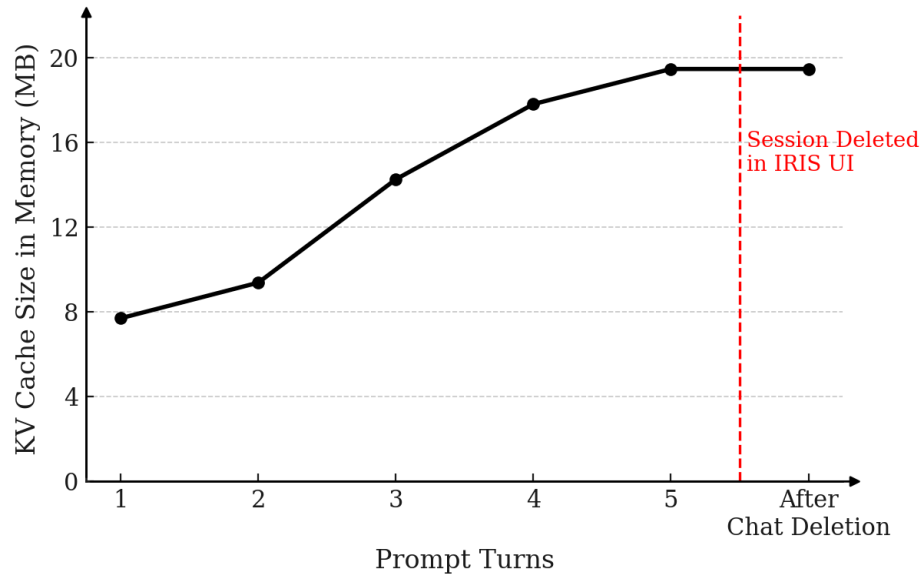


Figure 4.3: The size of KV cache in memory with respect to conversation turns. Although dereferenced after chat session deletion in IRIS UI, the KV cache still physically remains in memory.

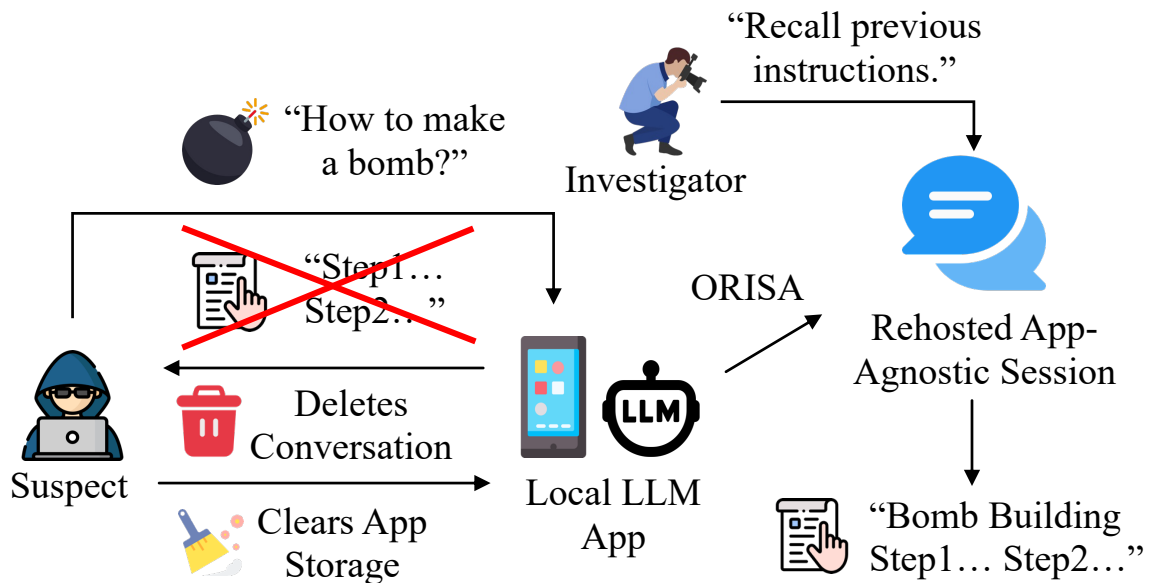


Figure 4.4: Rehosting a deleted conversation context into a live session allows investigators to interrogate and extract forensics evidence.

was cleared through the app UI.

Opportunity in Forensics Investigation. Figure 4.4 shows an example of applying this opportunity in a forensics investigation. Suppose a suspect uses a local LLM app with a

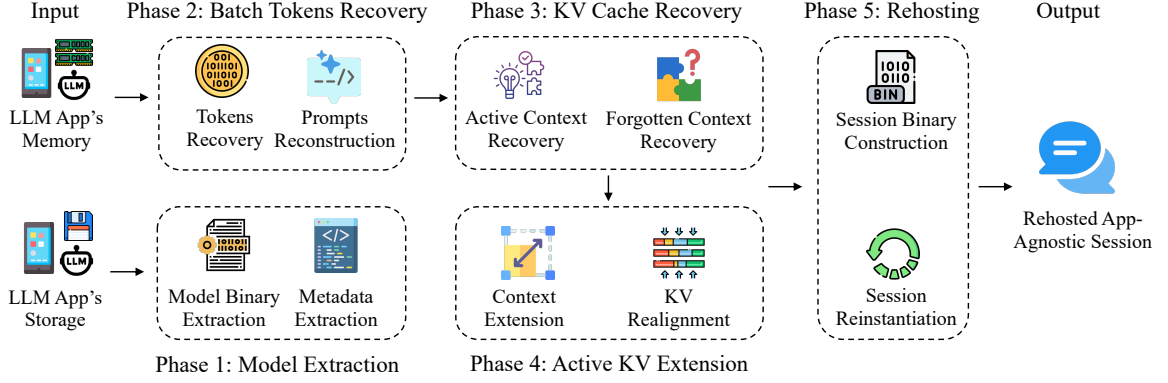


Figure 4.5: Overview of ORISA’s design. ORISA takes a mobile LLM app’s memory and app storage as input, conducts model extraction, batch tokens recovery, KV cache recovery, active KV extension, and session rehosting to output an app-agnostic rehosted session representing the previous app session’s context.

jailbroken model to plan a crime and asks it, ”How to build a bomb?” and then deletes the responses and clears the app storage. When an investigator wants to determine if the LLM assisted in the suspect’s crime planning and confiscates the device, they turn to ORISA for help. ORISA recovers deleted conversation context in memory and rehosts it into a live app-agnostic session. The investigator can then interrogate the rehosted session and ask it to recall previous bomb making instructions and ask additional questions regarding the suspect’s intentions. The responses then reveal previous illicit instructions and serve as evidence to convict the suspect.

4.3 Methodology

ORISA’s goals are to reconstruct previous prompts and responses from a local LLM app session, recover active and forgotten context of the LLM session, and rehost them into a live session to enable live interrogation by investigators. Figure 4.5 shows the overview of ORISA’s design consisting of 5 phases. ORISA takes a mobile LLM app’s process memory and app storage on disk as input. ORISA first locates and extracts the model binary and metadata from disk (subsection 4.3.1). ORISA then recovers the previous session’s batch tokens from memory and reconstructs them into the prompt history (subsection 4.3.2).

Next, ORISA recovers the KV cache representing both the active and logically evicted forgotten context from memory (subsection 4.3.3). ORISA then extends the KV cache’s active region to incorporate the forgotten context into a larger context window and realigns the KV tensors (subsection 4.3.4). Finally, ORISA reconstructs and rehosts the recovered context into an app-agnostic session (subsection 4.3.5), enabling investigators to conduct a live interrogation of the previous app session.

4.3.1 Model Binary Extraction From Disk

To enable session rehosting, ORISA first needs to extract the exact model binary used in the session as batch tokens, and KV cache only makes sense given a specific model. Pinpointing the exact model file used in the original run is challenging, as multiple model files may reside on disk, which could be used in different sessions. Even when a model with the same type is recovered, a slight variation in the model’s hyperparameters, such as embedding dimension or head counts, will make the rehosting invalid.

However, I discovered in my preliminary study that during runtime, LLM apps’ context embeds key descriptors of the model file into memory, including the model architecture identifier, and hyperparameters such as the number of layers, trained context length. Additional constants, such as the quantization layout, are also embedded in the KV cache configuration, which can be used to identify different versions of the same model type. These parameters together form a unique fingerprint to recover the exact model file used by the session. To accurately extract the exact same model used in the original run, ORISA uses these model metadata recovered from memory to locate and validate the model file.

Since the apps use local LLMs, the model file has to reside on the disk. ORISA scans the LLM app’s both external (under `\sdcard\Android\data`) and internal (under `\data\data`) app storage directories to locate and extract the model file. ORISA first extracts all files within the two directories with a matching file header with the model file format, e.g. `0x47475546` for GGUF models. ORISA then compares the model architec-

Algorithm 5: ORISA’s recovery of batch tokens from memory.

```
// Metadata in llama_context manager structure
1 n_past = processedTokenCount;
2 Function recoverAndDetokenizeBatchTokens():
    // Use batch_ptr if not cleared
3     if llama_context.batch_ptr.isValid() then
4         A_token = *batch_ptr;
5         for Ai < A_token do
6             if Ai[token] ∈ [0, vocab_size] & Ai_pos.notEmpty() & Ai_seq_id.notEmpty() then
7                 continue;
8             end
9             A_token = Ai;
10            break;
11        end
12    end
    // Else match token vocab size and check valid pos and seq_id
    // pointers
13    else
14        Set(A_bases) = scanBlocksLargerThan(llama_batch.size+n_past*token_type.size);
15        for A ∈ A_bases do
16            if ∀ An ∈ [0, vocab_size] & A_pos.notEmpty() & A_seq_id.notEmpty() then
17                A_token = A;
18                break;
19            end
20        end
21    end
    // Detokenization
22    D = Detokenize(T, A_token, A_token.length);
23    D.partition(T.delimiters);
24 end
```

ture identifier, quantization layout, number of layers, embedding dimension, head counts, and trained context length in the candidate model file’s metadata key-value section with the recovered metadata in the model’s hparams and context’s cparams from memory. ORISA finds the exact model file when all metadata matches.

4.3.2 Batch Tokens Recovery and Detokenization

The first artifact to recover in memory is the batch token sequences used in the previous session. Together with KV cache recovery (subsection 4.3.3), it represents the context state of the session exactly where it was left off. When detokenized, the batch tokens also reveal the previous prompts and responses strings, providing investigators a starting point to

conduct live interrogation of the session (subsection 4.3.5).

However, unlike application-level chat records, the token buffer is encoded and unstructured, containing non-human-readable and interleaved sequences. These tokens are also unlabeled, lacking explicit message boundary markers. ORISA recovers the batch tokens by checking the vocabulary sizes of elements as well as valid positions and unique sequence IDs bound to each token. algorithm 5 shows ORISA’s strategy to recover batch tokens from memory. ORISA first checks whether the *batch_ptr* in the session’s context manager (`llama_context` for *llama.cpp*) is still present and points to a valid page. If the region is valid, ORISA then scans backward from the currently active *batch_ptr* to locate additional tokens with valid buffer within the model’s vocabulary range and that the position and sequence id buffers are not empty, as shown in Lines 3-12 of algorithm 5. If the pointer has been cleared, ORISA then scans for contiguous blocks of aligned regions of at least size $llama_batch.size + n_past * token_type.size$, representing previously processed tokens. For each candidate, ORISA validates the token buffer’s vocabulary range and its pointers to position and sequence ids are valid, as shown in Lines 13-21 of algorithm 5. Finally, ORISA detokenizes the verified token sequence back into human-readable prompt and response sequence strings with delimiters by applying the decode function of the exact tokenizer in the recovered model binary file from disk (model binary recovery detailed in subsection 4.3.1). Each token ID is mapped back deterministically to its string representation, preserving role delimiters and formatting tokens injected by the app layer. To separate the prompts into system prompts, user prompts, and model responses, ORISA partitions the detokenized scripts with delimiters of the session, as shown in Lines 22-24 of algorithm 5.

Now ORISA has recovered the batch tokens of the previous session and detokenized conversation history, which serve as human-readable evidence to investigators for interrogation. To further allow live interrogation of the session, ORISA next recovers active and forgotten contexts from memory (subsection 4.3.3, subsection 4.3.4) and rehosts them into a live session (subsection 4.3.5).

Algorithm 6: ORISA’s recovery of the active and forgotten KV cache.

```
// Previously recovered token length and metadata
1  $T_{old} = tokenBufferLength, H = numberOfHeads, D_h = headDim, E =$ 
    $elementSize, ctx = originalContextWindow;$ 
2 Function recoverAndAlignKVCache():
   // Per-token kv cache slot size
3    $B_{token} = H * D_h * E * 2;$ 
   // Locate kv cache region
4    $Set(A_{bases}) = scanBlocksLargerThan(B_{token} * ctx);$ 
5    $A_{base} = null;$ 
6   for  $A \in A_{bases}$  do
7      $h = cacheHeadOffset(A_{base});$ 
8      $u = cacheUsedOffset(A_{base});$ 
9     if  $header(A_{base}).matches(header(KVCache)) \ \& \ h < ctx \ \& \ u \leq T_{old}$  then
10       $A_{base} = A$ 
11    end
12  end
   // Determine previous active region
13   $A_{active\_start} = A_{base} + ctx * B_{token};$ 
14  if  $h + u \leq ctx$  then
15     $A_{ctx} = [A_{active\_start}, A_{active\_start} + u * B_{token});$ 
16  end
   // Handle wrap around
17  else
18     $A_{ctx} = [A_{active\_start}, A_{end}) \cup [A_{base}, A_{base} + ((h + u) \bmod ctx) * B_{token});$ 
19  end
   // Tensor validation with token position
20  for  $cell[i] \in A_{ctx}$  do
21     $check(cell[i].pos = token[i].pos);$ 
22     $check(cell[i].seq\_id = token[i].seq\_id);$ 
23  end
   // Scan for valid forgotten context sensors beyond the context
   window
24  for  $cell[j] < A_{active\_start}$  do
25    if  $j > T_{old}$  then
26      break;
27    end
28    if  $cell[j].pos = token[j].pos \ \& \ cell[j].seq\_id = token[j].seq\_id$  then
29       $insert(cell[j], A_{ctx});$ 
30      continue;
31    end
32    break;
33  end
34 end
```

4.3.3 Active and Forgotten Context Extraction from KV Cache

Next, ORISA recovers the LLM session’s internal attention state after token processing at the time of extraction by retrieving the KV cache. Combined with the batch tokens generated

and the model file recovered previously, it allows investigators to resume generation exactly from the point when the scene was seized with rehosting.

However, fully recovering the KV cache with forgotten context is challenging. Once a session is deleted in the apps' UI, the active KV cache region is dereferenced as I found in my preliminary study (section 4.2), making the recovery from known pointers in `llama-context` infeasible. Additionally, the LLM session only book-keeps positions of KV tensors that are within the currently active context window, making recovery and validation of tensors representing logically evicted tokens challenging. Finally, every tensor recovered needs to be validated and match exactly with previous tokens in the correct sequence in order to represent the correct attention state. ORISA approaches by scanning the memory region with unique KV cache headers and uses previously recovered session metadata and batch token identifiers to guide the recovery and validation of KV cache tensors. algorithm 6 shows the steps ORISA adopts to recover both the active and forgotten KV cache from memory.

ORISA first uses previously recovered model's number of attention heads (H), head dimension (D_h), element size (E) and the original context window (ctx) as a reference to calculate the per-token KV cache slot size to recover $B_{token} = H * D_h * E * 2$, as shown in Line 3 of algorithm 6. Then to locate the candidate KV cache ring buffer regions, ORISA scans the heap for a contiguous block of size larger than the minimum size to represent the previous context $B_{token} * ctx$ that are page-aligned and resident. For each candidate base address A , ORISA matches the header with the `llama-kv-cache`, and verifies that the value h at offset to the `cache.head` offset is within the valid region between $[0, ctx)$, and that the value u at offset to the `cache.used` offset is within the recovered token buffer length T_{old} . ORISA accepts the candidate KV cache region when all the above are matched, as shown in Lines 4-12 of algorithm 6.

Next, ORISA recovers the previously active KV tensors within the context window. ORISA first calculates the base address for the previous active region as $A_{active.start} =$

$A_{base} + ctx * B_{token}$. If the previously used tokens are within the context window, then the previously active tensors $A_{ctx} = [A_{active_start}, A_{active_start} + u * B_{token})$. If they are larger, then there is a wrap-around in the ring buffer so that the part with the most recent tokens' tensors is wrapped to the start of the region. The total active region is thus a union of $[A_{active_start}, A_{end})$ and $[A_{base}, A_{base} + ((h + u) \bmod (ctx)) * B_{token})$, where A_{end} is the end of the buffer that have no valid tensors in succeeding slots.

Finally, ORISA validates the KV cache slots recovered by validating their mapping to the batch token positions recovered earlier. From the slot representing the tensors for the most recent tokens, ORISA checks backward in A_{ctx} until it reaches A_{active_start} . For each tensor slot, ORISA validates the $cell[i].pos$ matches with the $token[i].pos$ previously recovered, and that the token sequence identifier stored in the KV cache slot $cell[i].seq_id$ matches the $token[i].seq_id$, as shown in Lines 20-23 in algorithm 6. When it reaches the active region's head A_{active_start} , ORISA continues to scan beyond it for valid tensors logically evicted but still present in the buffer. If any cells' position and sequence ids $cell[j].pos$ and $cell[j].seq_id$ matches with $token[j].pos$ and $token[j].seq_id$ of recovered tokens beyond the context window, ORISA marks them as valid tensors representing the forgotten context and inserts them into the recovered KV cache region A_{ctx} . ORISA stops the forgotten context recovery when it reaches the end of the recovered token buffer length or when the new cell position doesn't contain a valid tensor anymore, as shown in Lines 24-33 of algorithm 6.

4.3.4 Active KV Extension

After the recovery of the KV cache with logically evicted tensors representing the forgotten context of the session beyond the context window, next ORISA needs to realign the KV cache and enlarge the active region of it so that when rehoted, the new session treats these forgotten contexts as active contexts.

This is challenging because although the recovered forgotten context tensors are phys-

Algorithm 7: ORISA’s extension of recovered KV cache tensors with forgotten context and realignment of metadata for rehosting into a larger context window.

```

// Previously recovered KV cache with forgotten context
1  $A_{ctx} = recoveredKVRegion, h = originalCacheHeadOffset(A_{base}), ctx =$ 
    $originalContextWindow, ctx_{new} = newContextWindow;$ 
2 Function extendAndRealignKVCache():
   // Reset KV cache metadata
3    $cache.head = 0;$ 
4    $cache.used = u_{valid} = A_{ctx}.length;$ 
5    $cache.size = ctx_{new};$ 
   // Remap new kv cache slot index
6    $i_{new} = (i_{old} - h) \bmod (ctx)$  for each  $i_{old} \in A_{ctx}.length;$ 
   // Move tensor values and metadata fields
7   for  $\forall i_{new}$  do
8      $cells[i_{new}].pos = cells[i_{old}].pos;$ 
9      $cells[i_{new}].seq\_id = cells[i_{old}].seq\_id;$ 
10     $cells[i_{new}].buf = cells[i_{old}].buf;$ 
11  end
   // Clear unused cells
12  for  $i_{new} \in [u_{valid}, ctx_{new}]$  do
13     $cells[i_{new}].pos = -1;$ 
14     $cells[i_{new}].pos = 0;$ 
15  end
   // Rest session past token count
16   $runtime\_context\_set(n\_past, u_{valid});$ 
17 end

```

ically present, the session still keeps them as invalid and will never incorporate them to continue generation. ORISA extends the session’s perception of the active KV by enlarging the cache size, resetting the cache head and active used section pointer, and realigning the cells with the new positions to the start of the new KV cache region.

algorithm 7 shows the steps to extend and realign the KV cache after its recovery with forgotten context. ORISA first records the previously recovered KV region A_{ctx} , the original cache head offset with respect to the base h , the previous context window ctx , and the new enlarged context window ctx_{new} . Then, ORISA resets KV cache regions’ metadata to represent the new valid region with forgotten context by setting the $cache.head$ pointer to 0, the $cache.used$ count to u_{valid} which is the length of the newly recovered region A_{ctx} , and the $cache.size$ to ctx_{new} , as shown in Lines 3-5 of algorithm 7. ORISA then calculates the new kv cache slot index $i_{new} = (i_{old} - h) \bmod (ctx)$ for each $i_{old} \in A_{ctx}.length$ slot

for the realignment, as shown in Line 6 of algorithm 7. For each of the new slots, ORISA then copies the tensor values and metadata fields that match with token positions and token sequence ids from the previous position to the new position. For the regions outside the new active region, ORISA resets the cell position counters and cell sequence ids, as shown in Lines 7-15 of algorithm 7. Finally, in the rehosted sessions' context manager (`llama_context` for *llama.cpp*), ORISA sets the processed token counters `n_past` to the currently valid context recovered u_{valid} so that the rehosted session can incorporate both the previously active and forgotten context recovered into subsequent generations.

Now ORISA has recovered the batch token sequences and the realigned and extended KV cache with forgotten context from memory for rehosting.

4.3.5 Session Rehosting

Batch tokens and KV cache need to be loaded into a live LLM session with the original model to continue conversation generation with previous context. After recovering the KV cache with forgotten context and the batch tokens of a conversation, ORISA rehosts them into a new session using the same model configurations with an enlarged context window to enable live interrogation of the previous session for investigators.

Session Binary Construction. To rehost an LLM conversation, I need to represent the previous context as a session state binary that the new session understands and can continue the generation from. This session binary should contain the exact configuration and attention of the model when the previous session stopped. For *llama.cpp* applications, this binary should represent the `llama_context` state that `llama_state_set_data()` takes as input. ORISA constructs the binary by first aligning the recovered context-level pseudo-random generator state, followed by the most recent output logits kept in the `llama_context` recovered from memory to preserve determinism in subsequent sampling. Then ORISA aligns the KV cache with the extended context to represent the exact attention state of the previous conversation. Finally, ORISA sets the new session binary's context window

to the enlarged window, and the bookkeeping parameter of processed tokens `n_past` to the recovered token length. Now the reconstructed session file encodes not only the full attention history of all processed tokens but also the precise sampling state needed to ensure continuity.

Session Reinstantiation. ORISA then loads a fresh session with the exact model binary file recovered in subsection 4.3.1, the session binary constructed (by calling `llama_state_set_data()` for *llama.cpp*), and the batch tokens recovered with the new processed token counter `n_past`. ORISA now continues the inference from the previous session (by calling `decode()`) and allows investigators to interrogate the new session with the previous context.

4.4 Evaluation

I deployed ORISA to evaluate its capability to recover batch tokens and KV cache from memory, as well as rehosting them into new sessions for investigators to interrogate using real Android local LLM apps.

Dataset & Implementation. I collected the 10 most popular (according to number of downloads or stars) Android local LLM apps from the Google Play Store [102] and public GitHub repositories that allow users to choose and use their own GGUF LLMs. Notably, they are all powered by *llama.cpp* [100], which is the most widely adopted mobile LLM inference engine. I installed them and conducted the evaluation on a rooted Google Pixel 6 device running Android 15 [106]. For the local LLM models, I collected 3 different lightweight models from the leading local model repository Hugging Face [105] suitable for local execution on a Google Pixel 6 device, namely Llama 3.2 [107], Qwen 2.5 [108], and Gemma 2 [109]. LLM app processes' memory images are acquired with Frida [110] using Objection [111]. Superuser privilege for Frida is acquired and granted with Magisk [112]. The rehosted sessions are run on an Ubuntu 24.04 LTS system running *llama.cpp* [100].

```
We're keeping some internal project notes.  
- Project codename = FERN-9e2f1b5d  
- API key hint = live_68cd-PLUM  
- Contact = Dana R. <dana@example.org> id=R-77b1  
- Ops window: Mondays 09:00–11:00 UTC  
- Staging bucket: s3://violet-bridge-14  
- On-call PIN = 5421-ALTO  
.....
```

Figure 4.6: Concrete key-value pair knowledge fed into the LLMs to simulate session context.

4.4.1 Tokens & KV Cache Recovery

I first evaluated ORISA’s recovery of batch tokens and KV cache tensors from within the model’s active context window as well as from beyond the context window.

Experiment Setup. For each of the 10 apps, I ran 6 sessions using the 3 models mentioned before under 2 different active context windows (256 and 512 tokens). To concretely determine if a response from the LLM is derived from previous chat knowledge rather than from the training knowledge, I randomly created concrete key-value pair information representing a project internal note that the LLMs do not know in advance and fed them into each LLM session to simulate the session context, template as shown in Figure 4.6. For each session, I continuously fed in the key-value pairs in a batch of 10 until the aggregated token count of the current session from both prompts and LLM responses reached at least 1024 tokens to overfill the 256 and 512-token active context window. This ensures that at least 512 tokens in both sessions are over the model’s active context window and are forgotten, enabling ORISA to recover batch tokens and KV cache representing those forgotten context. I then deleted the current conversation from each app’s UI, dumped the app process’s memory, and ran ORISA to recover the batch tokens and KV cache.

Findings. Table 4.1 shows the results for batch token and KV cache recovery from each app’s memory image running the 3 models under 2 different active context windows. Columns 1-2 of Table 4.1 show the 10 apps and 3 models loaded in each session. Column 3 shows the initial active context window set in each session for each model. Each model is

Table 4.1: ORISA’s Recovery of Batch Tokens and KV Cache Within and Beyond the Sessions’ Active Context Window.

App	Model	Initial Ctx	GT Tokens	Recovery Within Ctx					Recovery Beyond Ctx				
				Tokens #	%	KV #(K)	Tensors %	KV Size (MB)	Tokens #	%	KV #(K)	Tensors %	KV Size (MB)
Iris	Llama 3.2	256 512	1,073 1,062	256 512	100.00 100.00	4,194 8,389	100.00 100.00	8.66 16.33	138 327	16.89 59.45	2,848 3,756	21.28 41.68	5.56 7.34
	Qwen 2.5	256 512	1,048 1,074	256 512	100.00 100.00	3,670 7,340	100.00 100.00	7.56 14.55	173 265	21.84 47.15	2,429 3,820	21.39 47.41	4.74 7.46
Avg.	Gemma 2	256 512	1,044 1,068	256 512	100.00 100.00	13,631 27,263	100.00 100.00	26.85 52.88	179 308	22.72 55.40	9,312 14,624	22.19 49.40	18.19 28.56
	-	-	1,062	384	100.00	10,748	100.00	21.14	232	37.24	6,132	33.89	11.98
ChatterUI	Llama 3.2	256 512	1,058 1,067	256 512	100.00 100.00	4,194 8,389	100.00 100.00	8.66 16.33	173 234	21.57 42.16	2,343 3,719	17.83 40.90	4.58 7.26
	Qwen 2.5	256 512	1,059 1,043	256 512	100.00 100.00	3,670 7,340	100.00 100.00	7.56 14.55	149 202	18.56 38.04	2,007 2,652	17.43 34.84	3.92 5.18
Avg.	Gemma 2	256 512	1,055 1,039	256 512	100.00 100.00	13,631 27,263	100.00 100.00	26.85 52.88	184 258	23.03 48.96	9,158 13,418	21.53 47.82	17.89 26.21
	-	-	1,054	384	100.00	10,748	100.00	21.14	200	32.05	5,550	30.06	10.84
Layla	Llama 3.2	256 512	1,071 1,059	256 512	100.00 100.00	4,194 8,389	100.00 100.00	8.66 16.33	127 392	15.58 71.66	2,703 3,034	20.24 33.85	5.28 5.93
	Qwen 2.5	256 512	1,077 1,033	256 512	100.00 100.00	3,670 7,340	100.00 100.00	7.56 14.55	189 254	23.02 48.75	2,235 3,676	18.99 49.22	4.37 7.18
Avg.	Gemma 2	256 512	1,052 1,066	256 512	100.00 100.00	13,631 27,263	100.00 100.00	26.85 52.88	140 349	17.59 63.00	9,815 8,997	23.16 30.50	19.17 17.57
	-	-	1,060	384	100.00	10,748	100.00	21.14	242	39.93	5,077	29.33	9.92
Solia	Llama 3.2	256 512	1,064 1,056	256 512	100.00 100.00	4,194 8,389	100.00 100.00	8.66 16.33	176 341	21.78 62.68	2,889 3,560	21.82 39.94	5.64 6.95
	Qwen 2.5	256 512	1,070 1,069	256 512	100.00 100.00	3,670 7,340	100.00 100.00	7.56 14.55	155 386	19.04 69.30	2,597 2,404	22.25 30.11	5.07 4.70
Avg.	Gemma 2	256 512	1,049 1,067	256 512	100.00 100.00	13,631 27,263	100.00 100.00	26.85 52.88	188 224	23.71 40.36	9,067 12,585	21.47 42.58	17.71 24.58
	-	-	1,063	384	100.00	10,748	100.00	21.14	245	39.48	5,517	29.70	10.78
Maid	Llama 3.2	256 512	1,046 1,076	256 512	100.00 100.00	4,194 8,389	100.00 100.00	8.66 16.33	147 373	18.61 66.13	2,534 4,414	19.58 47.77	4.95 8.62
	Qwen 2.5	256 512	1,039 1,060	256 512	100.00 100.00	3,670 7,340	100.00 100.00	7.56 14.55	180 297	22.99 54.20	2,129 3,053	18.97 38.86	4.16 5.96
Avg.	Gemma 2	256 512	1,045 1,038	256 512	100.00 100.00	13,631 27,263	100.00 100.00	26.85 52.88	163 333	20.66 63.31	10,045 15,221	23.91 54.34	19.62 29.73
	-	-	1,051	384	100.00	10,748	100.00	21.14	249	40.98	6,233	33.90	12.17
LLMHub	Llama 3.2	256 512	1,041 512	256 512	100.00 100.00	4,194 8,389	100.00 100.00	8.66 16.33	135 205	17.20 37.21	3,122 3,156	24.28 34.96	6.10 6.16
	Qwen 2.5	256 512	1,072 1,075	256 512	100.00 100.00	3,670 7,340	100.00 100.00	7.56 14.55	190 362	23.28 64.30	2,524 4,267	21.58 52.87	4.93 8.33
Avg.	Gemma 2	256 512	1,040 1,057	256 512	100.00 100.00	13,631 27,263	100.00 100.00	26.85 52.88	122 256	15.56 46.97	8,928 10,428	21.39 35.93	17.44 20.37
	-	-	1,058	384	100.00	10,748	100.00	21.14	212	34.09	5,404	31.83	10.56
PocketPal	Llama 3.2	256 512	1,050 1,043	256 512	100.00 100.00	4,194 8,389	100.00 100.00	8.66 16.33	171 391	21.54 73.63	3,070 4,797	23.60 55.14	6.00 9.37
	Qwen 2.5	256 512	1,036 1,042	256 512	100.00 100.00	3,670 7,340	100.00 100.00	7.56 14.55	143 217	18.33 40.94	2,418 4,042	21.62 53.20	4.72 7.89
Avg.	Gemma 2	256 512	1,078 1,053	256 512	100.00 100.00	13,631 27,263	100.00 100.00	26.85 52.88	175 344	21.29 63.59	9,813 16,011	22.42 55.58	19.17 31.27
	-	-	1,050	384	100.00	10,748	100.00	21.14	240	39.89	6,692	38.59	13.07
Local AI	Llama 3.2	256 512	1,037 1,055	256 512	100.00 100.00	4,194 8,389	100.00 100.00	8.66 16.33	186 323	23.82 59.48	2,745 3,083	21.45 34.65	5.36 6.02
	Qwen 2.5	256 512	1,035 1,051	256 512	100.00 100.00	3,670 7,340	100.00 100.00	7.56 14.55	177 398	22.72 73.84	2,148 3,928	19.23 50.83	4.20 7.67
Avg.	Gemma 2	256 512	1,065 1,034	256 512	100.00 100.00	13,631 27,263	100.00 100.00	26.85 52.88	169 278	20.89 53.26	10,290 14,293	23.89 51.42	20.10 27.92
	-	-	1,046	384	100.00	10,748	100.00	21.14	255	42.33	6,081	33.58	11.88
AnythingLLM	Llama 3.2	256 512	1,077 1,061	256 512	100.00 100.00	4,194 8,389	100.00 100.00	8.66 16.33	192 209	23.39 38.07	2,936 3,686	21.83 40.98	5.73 7.20
	Qwen 2.5	256 512	1,054 1,058	256 512	100.00 100.00	3,670 7,340	100.00 100.00	7.56 14.55	133 357	16.67 65.38	2,344 3,155	20.49 40.31	4.58 6.16
Avg.	Gemma 2	256 512	1,073 1,047	256 512	100.00 100.00	13,631 27,263	100.00 100.00	26.85 52.88	162 232	19.83 43.36	9,252 15,872	21.27 55.72	18.07 31.00
	-	-	1,062	384	100.00	10,748	100.00	21.14	214	34.45	6,208	33.43	12.12
LM Playground	Llama 3.2	256 512	1,032 1,031	256 512	100.00 100.00	4,194 8,389	100.00 100.00	8.66 16.33	151 319	19.46 61.46	2,811 4,009	22.11 47.14	5.49 7.83
	Qwen 2.5	256 512	1,068 1,070	256 512	100.00 100.00	3,670 7,340	100.00 100.00	7.56 14.55	181 370	22.29 66.31	2,718 2,699	23.35 33.74	5.31 5.27
Avg.	Gemma 2	256 512	1,072 1,071	256 512	100.00 100.00	13,631 27,263	100.00 100.00	26.85 52.88	157 288	19.24 51.52	9,787 12,045	22.53 40.47	19.12 23.53
	-	-	1,057	384	100.00	10,748	100.00	21.14	244	40.05	5,678	31.56	11.09
Total Avg.	-	-	1,056	384	100.00	10,748	100.00	21.14	233	38.02	5,860	32.60	11.44

run with a 256 and 512-token context window in 2 sessions. Column 4 shows the ground truth token count, including both prompt and model responses, that the session contains before the deletion of the chat content from UI and gathering the memory image. Columns 5-9 show the batch tokens, KV cache tensors, and KV cache size within the respective context window recovered by ORISA. Additionally, Columns 10-14 show the logically evicted batch tokens, KV cache tensors, and KV cache size recovered beyond the active context window, representing forgotten context.

Looking at the Total Avg. Row, Column 4 of Table 4.1, each session had processed an average of 1,056 tokens, overfilling the 256 and 512 active context windows by at least 4 and 2 times. Columns 5-8 show that ORISA recovered 100% of batch tokens and KV tensors from all sessions within the context window, indicating that active tokens and KV cache still fully reside in memory even after the deletion of the chat session from the apps' UI. For KV cache, ORISA recovered 4,194K, 3,670K, and 13,631K tensors from the Llama 3.2, Qwen 2.5, and Gemma 2 models, respectively, for the 256 tokens context window. The recovered tensors from the Gemma 2 models are more than 3 times larger than those recovered from the other 2 models because of the significantly larger number of KV heads and KV head dimensions of the Gemma 2 model. As a result, Column 9 of the Total Row shows that the KV cache size recovered from the Gemma 2 models (26.85 MB) is also 3 times larger than that from the other 2 models (8.66 MB and 7.56 MB, respectively).

Moving on to the Total Avg. Row of Columns 10-11 in Table 4.1, ORISA recovered an average of 233 extra batch tokens beyond the active context window, representing an extra 38.02% of prior tokens. ORISA recovered the most tokens beyond the context window from the *Local AI* app, an average of 255 tokens from all 6 sessions. I noted that when the sessions' ground truth processed tokens are around the same, ORISA recovers more tokens from the session with a larger initial context window. For example, in the *Iris* app running the Llama 3.2 model, ORISA recovered 138 extra tokens when the initial context window is 256, while it recovered 327 extra tokens when that is 512. For KV cache recovery

beyond the context window, Columns 12-14 show that ORISA recovered an average of 5,860K extra tensors (32.60%) across all apps with an average size of 11.44 MB. ORISA recovered the most extra KV tensors from the *PocketPal* app, with an average of 6,692K tensors representing 38.59% of extra context. I observed that ORISA can recover more KV tensors from sessions with a larger initial context window because the allocated KV cache buffer is significantly larger, e.g., 16,011K from the *PocketPal* app running Gemma 2 with a 512 context window compared with 9,813K from the same app and model with a 256 context window.

Takeaway. ORISA recovered 100% batch tokens and KV cache tensors within sessions' active context window, confirming that they still persist in memory even after chat content deletion from apps' UI. ORISA additionally recovered an average of 32.60% and 38.59% of batch tokens and KV tensors beyond the models' active context limit, revealing significant forgotten context retention in the memory image. Recovered tensor count varied with model size and configuration. ORISA recovered over 3 times larger KV cache from Gemma 2 sessions due to its higher head count and dimension, and sessions initialized with larger context windows allowed more tensor recovery.

4.4.2 Context Retention of Rehosted Sessions

Recall from the previous section subsection 4.4.1, for each chat session, I keep prompting the LLM key-value pair information representing sample internal project notes (Figure 4.6) for the model to remember. I next evaluate the retention of this key-value pair knowledge in the rehosted sessions constructed with the token and KV cache recovery.

Experiment Setup. After the recovery of tokens and KV cache from memory, ORISA reconstructs the new session binary and rehosts it into a new *llama.cpp* session with a larger active context window (subsection 4.3.5). Immediately after rehosting, I interrogate the model with this exact sentence to extract its prior knowledge – "List out all the internal project notes I told you earlier." In the response, I record the number of key-value pairs of

Table 4.2: Rehosted Sessions’ Retention of Prompted Knowledge Within and Beyond the Active Context Window.

App	Model	Initial Ctx	Fed Pairs	Pairs In Ctx	Rehost Ctx	Extr ¹ Pairs	Recovery Rate ²
Iris	Llama 3.2	256	100	21	1,024	34	161.90%
		512	100	45	1,024	68	151.11%
	Qwen 2.5	256	110	25	1,024	43	172.00%
		512	110	48	1,024	76	158.33%
	Gemma 2	256	80	15	1,024	28	186.67%
		512	80	34	1,024	55	161.76%
Avg.	-	-	97	31	1,024	51	165.30%
Chatter UI	Llama 3.2	256	100	14	1,024	26	185.71%
		512	100	36	1,024	58	161.11%
	Qwen 2.5	256	110	15	1,024	29	193.33%
		512	100	39	1,024	59	151.28%
	Gemma 2	256	70	10	1,024	23	230.00%
		512	70	30	1,024	49	163.33%
Avg.	-	-	92	24	1,024	41	180.80%
Layla	Llama 3.2	256	100	18	1,024	29	161.11%
		512	100	41	1,024	59	143.90%
	Qwen 2.5	256	110	20	1,024	36	180.00%
		512	110	47	1,024	73	155.32%
	Gemma 2	256	80	13	1,024	23	176.92%
		512	80	31	1,024	43	138.71%
Avg.	-	-	95	28	1,024	45	168.93%
Others	-	-	94	27	1,024	46	168.70%
Total Avg.	-	-	94	27	1,024	46	169.59%

1: Extracted KV pairs from LLMs’ responses.

2: Recovery rate is the extracted KV pairs in the rehosted session during interrogation over the number of KV pairs within the initial context window.

knowledge it correctly recalled.

Findings. Table 4.2 shows the extracted key-value pair knowledge in the rehosted sessions. Columns 1 and 2 show the apps and models used. Column 3 shows the initial context window before the rehosting. Each model is run with both a 256 and a 512 active context window for each app with 2 separate sessions. Column 4 shows the number of key-value pairs (Figure 4.6) I prompted each session in a batch of 10 until the processed tokens of each session reached at least 1,024 to overfill the context window. Column 5 shows the number of key-value pairs that are still within each session’s context window before the rehosting. Columns 6-8 show the rehosted session’s context window, extracted KV pairs from the LLMs’ responses during the interrogation, and the recovery rate with respect to the key-value pairs inside the context window before the rehosting.

Looking at the Total Avg. Row, Columns 4 and 5 of Table 4.2, I prompted an average of 94 key-value pair knowledge into each session before overfilling the sessions with at least 1,024 tokens. I found that an average of 27 key-value pairs still reside in the active context window. I noticed that this number is smaller than the tokens needed to represent those key-value pairs on average because a portion of the context window is filled with system prompt tokens set up by each app, in addition to the prompted key-value pairs. For example, as shown in the Llama 3.2 Rows of the Iris and the Chatter UI app, although the number of fed key-value pairs is both 100, fewer pairs (14, 36) still reside in the context window of the Chatter UI app than the Iris app (21, 45). This is because the Chatter UI app had an average of 120 tokens to represent system prompts compared to around 42 for the Iris app. In addition to system prompt tokens, I also observed that different models’ tokenizers also impacted the number of key-value pairs inside the context window.

Moving to the Total Row, Columns 6 and 7 of Table 4.2, ORISA rehosted each session with a larger 1,024 tokens context window after the recovery and active KV manipulation subsection 4.3.5. In the interrogated responses, ORISA extracted an average of 46 key-value pairs contained in the originally fed pairs shown in Column 4, significantly larger

than those originally retained before the rehosting. I noted that for each app, the number of extracted pairs for the Llama and Qwen models is larger than that of the Gemma model. For example, as shown in the Layla Row Column 7, sessions with Llama 3.2 (29, 59) and Qwen 2.5 (36, 73) had a similar number of pairs extracted in the rehosted sessions, while Gemma 2 model retained fewer pairs (23, 43). This is because the prior 2 models had a similar tokenization rate of around 9.5 tokens per pair, while the Gemma 2 model had around 12.7. As shown in Column 8 of Table 4.2, ORISA extracted an average of 169.59% of knowledge in the rehosted session compared with the original session, enabling investigators to interrogate prior sessions with more logically evicted context.

Takeaway. On average, although only 27 key-value pairs of knowledge remained within the original 256–512 token context windows for each app session, ORISA’s recovery and rehosting with a 1,024 context window successfully extracted 46 pairs—an average recovery rate of 169.6% compared to the original context. Recovery rate varied by models’ tokenization rate and apps’ system prompt token length. This shows that ORISA not only preserves but substantially amplifies the retrievable knowledge from deleted sessions, enabling investigators to interrogate prior sessions with more context.

4.4.3 Semantic Consistency of Rehosted Sessions

When ORISA reconstructs and rehosts a session, the forensic validity of any responses generated during interrogation depends on whether those responses are consistent with what the model would have produced if the original session had continued uninterrupted. To measure this validity, I then evaluate the semantic consistency of responses during interrogation between the original and the rehosted sessions.

Experiment Setup. For each of the 10 apps, I ran 3 sessions with one of each model, similar to subsection 4.4.1. For each session, I prompt the model with 20 key-value pairs of internal project notes to represent new knowledge the model didn’t know beforehand (example shown in Figure 4.6). I set the context window of each session to 2048 tokens, greatly

Table 4.3: Comparison of Response Behaviors Between Original and Rehosted Sessions Under Forensic Interrogation.

App	Model	Original Session				Rehosted Session			
		TP ¹	FP ²	TN ³	FN ⁴	TP	FP	TN	FN
Iris	Llama 3.2	10	1	4	0	10	1	4	0
	Qwen 2.5	9	0	5	1	9	0	5	1
	Gemma 2	10	0	5	0	10	0	5	0
Chatter UI	Llama 3.2	10	2	3	0	10	2	3	0
	Qwen 2.5	9	0	5	1	9	0	5	1
	Gemma 2	10	0	5	0	10	0	5	0
Layla	Llama 3.2	10	2	3	0	10	2	3	0
	Qwen 2.5	10	1	4	0	10	1	4	0
	Gemma 2	10	1	4	0	10	1	4	0
Solia	Llama 3.2	9	2	3	1	9	2	3	1
	Qwen 2.5	9	1	4	1	9	1	4	1
	Gemma 2	10	1	4	0	10	1	4	0
Maid	Llama 3.2	10	0	5	0	10	0	5	0
	Qwen 2.5	10	1	4	0	10	1	4	0
	Gemma 2	10	0	5	0	10	0	5	0
Others	-	145	14	61	5	145	14	61	5
Total	-	291	26	124	9	291	26	124	9

1: The model correctly recalls the factual value of a key that was previously fed into the session.

2: The model outputs a value for a key that was never fed but asserts it as if known.

3: The model answers "Not in prior messages" for a key that was never fed.

4: The model fails to recall or incorrectly answers the value of a key that was fed earlier in the session.

exceeding the number of tokens needed to represent that knowledge and any follow-up interrogations to avoid the original sessions from losing context. I then dumped the process memory for each app session and let ORISA recover and rehost it into a new *llama.cpp* session with the same 2048 context window. To compare the response behaviors of the rehosted and the original session, I then prompted both sessions with the exact instructions listed as "Rule: You must answer ONLY using information that appears visibly in my prior chat messages. If the requested information does not appear in my prior messages, respond exactly: Not in prior messages." I then prompted each session to recall 10 values of keys that was prompted before in prior messages (e.g. I prompted the session "What is the project codename?" when I previously fed the model with "Project codename = FERN-9e2f1b5d") and 5 values for keys that did not (e.g. I prompted the session "What is the client passphrase?" when this information was never fed into the model before). I used a 2:1 ratio for the recall and non-recall questions to avoid over-inflating the false positive rate due to small denominator effects. Finally, I record the TP, FP, TN, and FN of the models' responses to all 15 questions in both the original and the rehosted sessions.

Findings. Table 4.3 shows the comparison of models' responses in the original and the rehosted sessions, given the same prior knowledge, context window, and interrogation prompts. Columns 1 and 2 show the app and model used in each session. Columns 3-6 and 7-10 show the TP (The model correctly recalls the factual value of a key that was previously fed into the session), FP (The model outputs a value for a key that was never fed but asserts it as if known), TN (The model answers "Not in prior messages" for a key that was never fed), and FN (The model fails to recall or incorrectly answers the value of a key that was fed earlier in the session) of the models' responses to the 15 interrogation questions in the original session and the rehosted sessions respectively.

As shown in the Total Row, Column 4 of Table 4.3, 26 FP responses are recorded during interrogation of the original sessions. For example, in the Iris, Llama 3.2 Row, I recorded a FP response from the model where it answers "DEBUG-ON-2025" to the question "What is

the client passphrase?” when the client passphrase was never fed into the session. As shown in the Total Row, Column 6, a total of 9 FN responses were also recorded. For example, in the Chatter UI, Qwen 2.5 Row, I recorded an FN response from the model where it answers “Not in chat” to the question “What tracking label did I use?” when I previously fed the session with the knowledge “Tracking label = TKT-8224-amber”.

As shown in the Total Row, Columns 8 and 10 of Table 4.3, the rehosted sessions recorded the exact same number of FP (26) and FN (9) across all app sessions, indicating that the rehosted sessions preserved the exact context of the original session. This is because ORISA’s rehosting not only preserves the original sessions’ batch tokens and KV cache, it also preserves the pseudo-random generator state in the original session for subsequent generation. Since the FP and FN responses in the rehosted session are also present in the interrogation of the original session, they stem from the intrinsic behavior of the LLMs used themselves (e.g., FP from hallucination and FN from incomplete retrieval) – not from rehosting inaccuracy. This shows that ORISA enables investigators to treat the rehosted sessions’ responses as credible evidence, since they faithfully represent what the original model would have produced had the session continued uninterrupted.

Takeaway. ORISA’s rehosted sessions produced identical true and false responses (26 FPs and 9 FNs from 450 questions) to those of the original sessions, confirming that ORISA preserves the model’s semantic state and internal reasoning behavior. The matching FP and FN patterns indicate that any inaccuracies arise from the model’s inherent hallucination or recall limitations rather than rehosting artifacts. This demonstrates that ORISA allows investigators to treat the rehosted session’s interrogation outputs as credible evidence of what the model would have generated if the session had continued.

4.5 Case Study

Chatter UI [113] is one of the most popular native mobile frontend apps for hosting on-device LLMs. It is hosted on GitHub and has received over 1.9K stars. Aside from

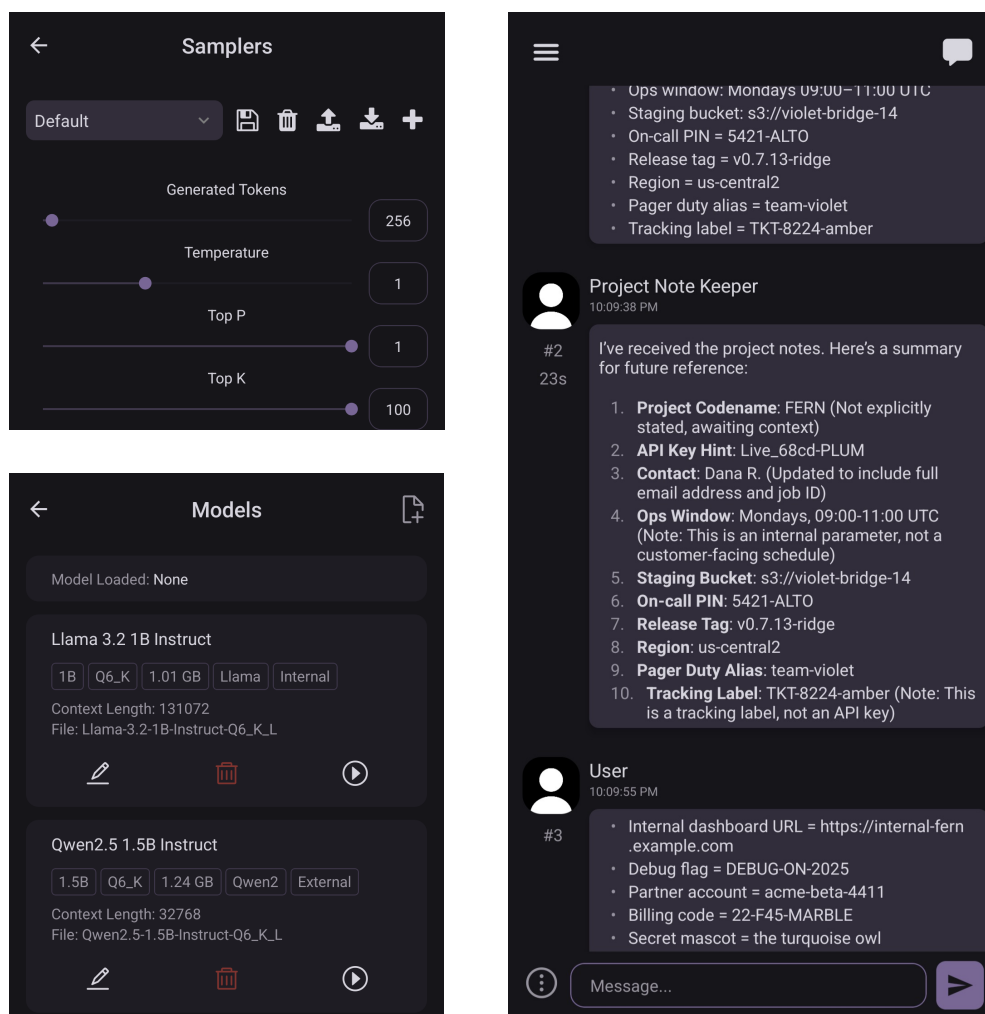


Figure 4.7: Chatter UI is one of the most popular native mobile LLM chat app. It allows users to customize model sampler parameters (top left) to fine-tune response behaviors, load custom local LLMs (bottom left), and instantiate custom chat characters in chat sessions(right).

allowing users to load custom models from disk, it also provides users freedom to fine tune the model sampler’s parameters, such as temperature, Top P, Top K, etc., for different response behaviors. Users can also build different chat templates for different characters by customizing the system prompt sequence, prefix, and suffix, as shown in Figure 4.7.

To test ORISA’s recovery of tokens and KV tensors from a Chatter UI chat session, I downloaded and loaded a Llama 3.2 1B model and configured it with a 256-token context window. I then created a custom ”Project Note Keeper” character with description tokens indicating – ”You are a helpful project note-keeping bot. Only remember factual statements

```

Write Project Note Keeper's next reply in a chat
between Project Note Keeper and User. You are a
helpful project note-keeping bot. Only remember
factual statements that I tell you about the
project.<|eot_id|><|start_header_id|>assistant<|end_
header_id|>

```

Figure 4.8: Recovered deserialized tokens show that the Chatter UI app incorporates user-defined character description prompts into system prompts and feeds them to the inference engine as permanent tokens not to be evicted.

that I tell you about the project.” in the Chatter UI app and fed it with internal project notes key-value pairs to simulate knowledge not known by the model previously. I instrumented the app and kept prompting the session with key-value pair knowledge until the token count reached past 1024, where 100 pairs had been fed into the session. Notably, after each of the model’s responses to my prompts, even after navigating away from the session and back, Chatter UI allows users to continue generating the response without reprompting it. This implies that Chatter UI keeps the previous conversation turns in memory and does not require a new prompt to restart the conversation.

I then deleted the conversation from the app UI and dumped the app process’s memory. After running ORISA, ORISA recovered a total of 429 tokens and 6537K tensors, representing 22% and 18% more from beyond the 256 tokens context window. While deserializing the tokens recovered, I found that the model retains the description tokens I fed into the app during the initiation of the chat session, representing the custom character of the current session, as shown in Figure 4.8. This indicates that the Chatter UI app stores those character description tokens as system prompts and feeds them to the inference engine as permanent tokens not to be evicted outside the context window (`n_keep` tokens for *llama.cpp*).

After ORISA rehosted the recovered context in a fresh *llama.cpp* session, I resumed the session with the interrogation prompt for it to “List out all the internal project notes I told you earlier”. The rehosted session’s response correctly outputted 26 values for the keys previously fed in, exceeding the 14 pairs still residing within the context window by 186%,

indicating that ORISA’s rehosted session successfully retained the forgotten context outside the context window recovered in memory.

4.6 Discussion

4.6.1 Extension to Other Inference Engines

I have designed ORISA to recover and rehost mobile local LLM app sessions’ context using the *llama.cpp* inference engine. However, I acknowledge that although *llama.cpp* remains the most widely adopted inference engine for mobile apps [114, 115] due to its ARM CPU optimizations and extensive model support, there are other frameworks for mobile local LLM deployment (mostly supported through each engine’s official demo app) such as *mllm* [116], *MediaPipe* [117], etc. It is possible to extend ORISA to these engines. Since all transformer inference engines require the same core structures – a token buffer that hosts token IDs and positions, a KV cache that stores attention head state, and a runtime state manager that keeps past token count and other metadata – extending ORISA to these engines only requires implementing new frontends to interpret their respective in-memory signatures for each core context-management data structure. The same methodology, namely token recovery, detokenization, active and forgotten KV cache extraction, active KV extension, and session rehosting, still applies.

4.6.2 Obfuscation and Anti-Forensics Techniques

Adversarial inference frameworks that deviate from the standard, efficient representation of token and KV cache tensors or those that obscure the generic structure of context-management objects can make ORISA’s session recovery and rehosting more difficult. Nevertheless, prior work shows that even heavily obfuscated platforms can be analyzed using established memory forensics techniques [118, 119, 120]. With additional effort, investigators can apply this perpendicular line of research to deobfuscate and reconstruct the data structures for context management objects [121, 122]. Once these structures

are recovered, the investigator can build a corresponding frontend for the adversarial or obfuscated inference engine, similar to a regular and benign engine. Importantly, these frameworks must still retain the minimal information required to represent a session's context to continue the inference, enabling ORISA's recovery and rehosting.

CHAPTER 5

RELATED WORK

Benign Misuse of a11y Service. A11y services are often misused by benign applications such as anti-virus engines [56] and file system management apps [13] to achieve automated functionalities. Multiple works focus on dissecting the misuse of utility apps. Salehnamadi et al. [11] proposed a framework to assess mobile applications' a11y functionality correctness. Naseri et al. [12] introduced a study on how Android apps misuse the a11y service to achieve utility shortcuts. Chen et al. [10] proposed a dynamic traversal technique to extract a11y feature malfunctions in Android a11y apps. Instead of analyzing the benign misuse of the a11y service, I discovered an attack surface that can be abused by malicious a11y attackers.

Attacks on a11y Service. The a11y service is widely abused by malware to conduct automated phishing attacks [123, 58]. The powerful functionality of a11y service allows malware to launch attacks in an evasive manner [124]. Multiple works also proposed PoC attacks to exploit the a11y service [23, 59]. Fratantonio et al. [24] proposed an attack that enables malware to control the GUI of an Android device with the `SYSTEM_ALERT_WINDOW` and a11y permissions. Mehralian et al. [125] uncovered sensitive information leakage through overly accessible a11y elements in Android. Jang et al. [22] identified 12 a11y attacks on four different operating systems. Lei et al. [126] exposed an a11y side-channel attack that allows password leakage through guessing consecutive content queries. Unlike the attacks that target native Android apps, I uncovered an attack surface that allows a11y attackers to extract app-side inaccessible sensitive information in mobile-browser rendered websites. I found this attack surface impactful because mobile browser users of the same service are less protected from a11y attacks than app users.

Defenses against a11y Attacks. Malware and PoC a11y attacks have led to the development of multiple works to counteract malicious abuse of the a11y service [127, 128]. Fernandes et

al. [9] introduced a technique to block all undeclared data-flows in Android apps by enforcing runtime restrictions. Huang et al. [8] proposed a more fine-grained Android app service design to enforce least-privileged data-flow constraints in runtime. Android also introduced new features to app developers to block app-side app access to untrusted app services [61]. With all the above defenses considered, I found the new attack surface introduced to still be feasible because it allows app malware to circumvent app-side protections. DVa also helps benign service developers vet this attack surface and guides its mitigation.

Mobile Banking Security. E-banking fraud [129, 130, 131] and attacks [132, 133, 134, 135] have led to huge financial losses worldwide. Mobile banking apps are vulnerable to malware attacks [136, 137]. Multiple works focus on evaluating the security measures imposed by these apps [138, 139]. Chen et al. [140] exposed weaknesses in mobile banking apps' sensitive data storage and transmission, confirming their proneness to being targeted by malware, as illustrated by DVa. Botacin et al. [57] evaluated the security flaws of Brazilian mobile banking apps, uncovering their susceptibility to UI and app-based attacks. Corroborating the security flaws in mobile banking apps, DVa contributes to this field of research by detecting app abuse vectors exploiting real victim banking apps targeted by mobile app malware.

Program Analysis. Prior work use API trace analysis [141, 142, 143, 144], network traffic analysis [145, 146, 147], symbolic analysis [129, 148] and forensic analysis [95, 27, 28] to reveal program behaviors. However, to discover app-side app-protected elements, DVa uses a combination of dynamic app traversal and static attribution.

Malware Analysis. Some works use taint analysis [149, 150], API trace analysis [141, 142, 151, 143, 144], and network traffic analysis [152, 147, 153] to reveal malware behaviors. However, to attribute app attack vectors, DVa uses symbolic analysis [154, 148, 155, 156] to match the constraints of app behaviors to their targets. Inspired by forced execution [157, 158, 159], DVa uses victim modeling to guide the execution of malware and loading of victim-specific abuse payloads.

Browser Instrumentation. Prior work have instrumented the browser engine to collect activities of web pages from DOM [160, 161, 162, 163, 164]. DVa is inspired by these techniques but focuses the traversal on the `all` tree to reveal unprotected elements.

LLM Forensics. Prior research has explored forensic techniques to trace the sources of data poisoning in LLMs and neural networks [165, 166, 167, 168]. Similarly, a growing body of research [169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180] surveyed and found LLM security vulnerabilities. For instance, Kim et al. [181] introduced a framework aimed at reducing the leakage of personally identifiable information from LLMs. Greshake et al. [182] explored indirect prompt injection attacks against LM frameworks. Furthermore, several work [183, 173, 184, 185, 186, 187, 188, 189] evaluated the robustness of current LLM frameworks against jailbreak attacks. However, no prior work explored the forensic investigation and rehosting of deleted local LLM sessions in mobile applications.

Memory Forensics. Memory forensics techniques have been proposed to recover evidence and aid program analysis [190, 21, 20]. Oygenblik et al. [95] proposed a technique to accurately recover parameters that define DL models and rehost them for a live investigation. Saltaformaggio et al. [27] developed a spatial-temporal memory forensics technique to recover previous screens of Android apps. While memory forensics techniques [191] have been applied to areas such as GUI reconstruction [192, 28], value-set analysis [193], malware analysis [194], and sequencing user activity [195], no prior work aimed to recover and rehost LLM’s context artifacts. This is due to challenges discussed earlier in recovering active and forgotten KV cache regions in LLM apps’ memory and realigning them in a rehosted session. DVa is inspired by these prior works to scan and recover LLM context data structures and use key pointers and context management objects to guide the validation and realignment.

CHAPTER 6

CONCLUSION

Modern forensic investigations often fail when critical user interaction context is hidden, protected, or deliberately deleted. Across chapter 2–chapter 4, I presented three systems that make such transient context measurable and actionable by extracting it from real software stacks and AI systems. First, in chapter 2, I introduced DVa to attribute Android app-specific malware behaviors to concrete, victim-specific abuse vectors. Rather than stopping at generic capability detection, DVa connects app-driven actions and observations to the victim apps present on a device, enabling investigators to answer *which* apps were targeted and *how* the abuse could manifest in those apps. Second, in chapter 3, I showed that securing sensitive app-specific information requires reasoning across multiple front ends. SOMBRA demonstrates that protections deployed in native Android apps may fail to transfer to browser-rendered counterparts, and it provides a practical workflow for developers to identify where protected app-side fields become exposed through mobile browsers. Third, in chapter 4, I introduced ORISA, a memory forensics framework for recovering and rehosting deleted conversation context from on-device local LLM applications. ORISA shows that even when chat histories are removed from the UI and from disk artifacts, model-internal runtime state can retain recoverable evidence of prior interactions. By reconstructing and rehosting this recovered context into a live session, ORISA enables investigators to interrogate recovered interaction state and extract actionable evidence.

REFERENCES

- [1] *Accessibilityservice*. [Online]. Available: <https://developer.android.com/reference/android/accessibilityservice/AccessibilityService>.
- [2] R. Lakshmanan, *Teabot android banking malware spreads again through google play store apps*. [Online]. Available: <https://thehackernews.com/2022/03/teabot-and-roid-banking-malware-spreads.html>.
- [3] A. Groenewald, *New google play malware poses major threat to mobile banking*. [Online]. Available: https://www.cyberghostvpn.com/en_US/privacyhub/google-play-malware/.
- [4] ThreatFabric, *2022 mobile threat landscape update*. [Online]. Available: <https://www.threatfabric.com/blogs/h1-2022-mobile-threat-landscape.html>.
- [5] C. Franzen, *Google unveils ultra-small and efficient open source ai model gemma 3 270m that can run on smartphones*. [Online]. Available: <https://blog.n8n.io/local-llm/>.
- [6] A. G., *The 6 best llm tools to run models locally*. [Online]. Available: <https://getstream.io/blog/best-local-llm-tools/>.
- [7] M. Farcas, *How to run a local llm: Complete guide to setup & best models (2025)*. [Online]. Available: <https://blog.n8n.io/local-llm/>.
- [8] J. Huang, M. Backes, and S. Bugiel, “AIly and privacy don’t have to be mutually exclusive: Constraining accessibility service misuse on android,” in *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual Conference, Aug. 2021.
- [9] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash, “FlowFence: Practical data protection for emerging IoT application frameworks,” in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.
- [10] S. Chen, C. Chen, L. Fan, M. Fan, X. Zhan, and Y. Liu, “Accessible or not? an empirical investigation of android app accessibility,” *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3954–3968, Oct. 2022.
- [11] N. Salehnamadi, A. Alshayban, J.-W. Lin, I. Ahmed, S. Branham, and S. Malek, “Latte: Use-case and assistive-service driven automated accessibility testing framework for android,” in *Proceedings of the 2021 Conference on Human Factors in Computing Systems (CHI)*, Yokohama, Japan, May 2021.

- [12] M. Naseri, N. Borges Jr, A. Zeller, and R. Rouvoy, “Accessileaks: Investigating privacy leaks exposed by the android accessibility service,” in *Proceedings of Privacy Enhancing Technologies Symposium (PETS)*, Stockholm, Sweden, Apr. 2019.
- [13] W. Diao, Y. Zhang, L. Zhang, Z. Li, F. Xu, X. Pan, X. Liu, J. Weng, K. Zhang, and X. Wang, “Kindness is a risky business: On the usage of the accessibility APIs in android,” in *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Beijing, China, Sep. 2019.
- [14] *Accessibilityevent: Isaccessibilitydatasensitive()*. [Online]. Available: [https://developer.android.com/reference/android/view/accessibility/AccessibilityEvent#isAccessibilityDataSensitive\(\)](https://developer.android.com/reference/android/view/accessibility/AccessibilityEvent#isAccessibilityDataSensitive()).
- [15] *View.accessibilitydelegate*. [Online]. Available: <https://developer.android.com/reference/android/view/View.AccessibilityDelegate>.
- [16] C. C.-C. Cheng, C. Shi, N. Z. Gong, and Y. Guan, “Evihunter: Identifying digital evidence in the permanent storage of android devices via static analysis,” in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.
- [17] J. Lee, A. Chen, and D. S. Wallach, “Total recall: Persistence of passwords in android,” in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.
- [18] A. Lyons, J. Gamba, A. Shawaga, J. Reardon, J. E. Tapiador, S. Egelman, and N. Vallina-Rodríguez, “Log: It’s big, it’s heavy, it’s filled with personal data! measuring the logging of sensitive information in the android ecosystem,” in *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.
- [19] X. Yuan, O. Setayeshfar, H. Yan, P. Panage, X. Wei, and K. H. Lee, “Droidforensics: Accurate reconstruction of android attacks via multi-layer forensic logging,” in *Proceedings of the 12th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Abu Dhabi, UAE, Apr. 2017.
- [20] M. Andrea, P. Antonio, O. Andrea, A. Simone, B. Davide, and L. Andrea, “Nveiling byovd threats: Malware’s use and abuse of kernel drivers,” in *Proceedings of the 2026 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2026.
- [21] A. Oliveri, M. Dell’Amico, and D. Balzarotti, “An os-agnostic approach to memory forensics,” in *Proceedings of the 2023 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2023.

- [22] Y. Jang, C. Song, S. P. Chung, T. Wang, and W. Lee, "A11y attacks: Exploiting accessibility in operating systems," in *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Scottsdale, AZ, Nov. 2014.
- [23] J. Kraunelis, Y. Chen, Z. Ling, X. Fu, and W. Zhao, "On malware leveraging the android accessibility framework," in *Proceedings of Mobile and Ubiquitous Systems: Computing, Networking, and Services (MobiQuitous)*, Tokyo, Japan, Dec. 2013.
- [24] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee, "Cloak and dagger: From two permissions to complete control of the UI feedback loop," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*, San Jose, CA, May 2017.
- [25] *The android open source project, android-13.0.0_r31*. [Online]. Available: https://android.googlesource.com/platform/hardware/knowles/athletico/sound_trigger_hal/+/refs/tags/android-13.0.0_r31.
- [26] *Virustotal*. [Online]. Available: <http://www.virustotal.com/>.
- [27] B. Saltaformaggio, R. Bhatia, X. Zhang, D. Xu, and G. G. Richard, "Screen after previous screens: Spatial-temporal recreation of android app displays from memory images," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.
- [28] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, "Guitar: Piecing together android app guis from memory images," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, CO, Oct. 2015.
- [29] *Accessibilityevent (event types)*. [Online]. Available: <https://developer.android.com/reference/android/view/accessibility/AccessibilityEvent#developer-guides>.
- [30] *ThreatFabric, 2020 - year of the rat*. [Online]. Available: https://www.threatfabric.com/blogs/2020_year_of_the_rat.html.
- [31] S. Gatlan, *Cerberus android malware can bypass 2fa, unlock devices remotely*. [Online]. Available: <https://www.bleepingcomputer.com/news/security/cerberus-android-malware-can-bypass-2fa-unlock-devices-remotely/>.
- [32] *Goldoson android malware infects over 100 million google play store downloads*. [Online]. Available: <https://thehackernews.com/2023/04/goldoson-android-malware-infects-over.html>.
- [33] *Cybercrime service bypasses android security to install malware*. [Online]. Available: <https://www.bleepingcomputer.com/news/security/cybercrime-service-bypasses-android-security-to-install-malware/>.

- [34] *Android debug bridge (adb)*. [Online]. Available: <https://developer.android.com/tools/adb>.
- [35] Z. Qu, S. Alam, Y. Chen, X. Zhou, W. Hong, and R. Riley, "DyDroid: Measuring dynamic code loading and its security implications in android applications," in *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Denver, CO, Jun. 2017.
- [36] *Package manager*. [Online]. Available: <https://developer.android.com/reference/android/content/pm/PackageManager>.
- [37] *Appbrain - everything you need for a successful android app*. [Online]. Available: <https://www.appbrain.com/>.
- [38] *Sensortower, actionable intelligence for the digital world*. [Online]. Available: <https://app.sensortower.com/>.
- [39] *Accessibilitynodeinfo*. [Online]. Available: <https://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo>.
- [40] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The Soot framework for java program analysis: A retrospective," in *Proceedings of Cetus Users and Compiler Infrastructure Workshop (CETUS)*, Oct. 2011.
- [41] *Pixstealer malware detected in banking sector*. [Online]. Available: <https://www.cybertalk.org/2021/09/30/pixstealer-malware-detected-in-banking-sector/>.
- [42] ThreatAdvice, *New android malware bypasses two factor authentication to steal passwords*. [Online]. Available: <https://www.threatadvice.com/blog/new-android-malware-bypasses-two-factor-authentication-to-steal-passwords>.
- [43] D. Palmer, *This new android malware bypasses multi-factor authentication to steal your passwords*. [Online]. Available: <https://www.zdnet.com/article/this-new-android-malware-bypasses-multi-factor-authentication-to-steal-your-passwords/>.
- [44] T. Micro, *Tgtoxic malware's automated framework targets southeast asia android users*. [Online]. Available: https://www.trendmicro.com/en_us/research/23/b/tgtoxic-malware-targets-southeast-asia-android-users.html.
- [45] D. Goodin, *Found: New android malware with never-before-seen spying capabilities*. [Online]. Available: <https://arstechnica.com/information-technology/2018/01/found-new-android-malware-with-never-before-seen-spying-capabilities/>.

- [46] D. Palmer, *This android banking malware steals data by exploiting smartphone accessibility services*. [Online]. Available: <https://www.zdnet.com/article/five-easy-steps-to-keep-your-smartphone-safe-from-hackers/>.
- [47] G. Shinde, *Malware on google play abusing accessibility service*. [Online]. Available: <https://www.zscaler.com/blogs/security-research/malware-google-play-abusing-accessibility-service>.
- [48] R. Suau, *Analysis of a malware exploiting android accessibility services*. [Online]. Available: <https://blog.pradeo.com/accessibility-services-mobile-analysis-malware>.
- [49] Y. Amit, *Inside the android accessibility clickjacking malware*. [Online]. Available: <https://techbeacon.com/app-dev-testing/inside-android-accessibility-clickjacking-malware>.
- [50] T. Meskauskas, *How to remove the cerberus banking trojan*. [Online]. Available: <https://www.pcrisk.com/removal-guides/17387-cerberus-banking-trojan-android>.
- [51] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Oteau, and P. McDaniel, “FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Edinburgh, UK, Jun. 2014.
- [52] *Edxposed: Elder driver xposed framework*. [Online]. Available: <https://github.com/ElderDrivers/EdXposed>.
- [53] S. Sebastián and J. Caballero, “AVClass2: Massive malware tag extraction from AV labels,” in *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC)*, Virtual Conference, Dec. 2020.
- [54] Skylot, *Skylot/jadx: Dex to java decompiler*. [Online]. Available: <https://github.com/skylot/jadx>.
- [55] O. Alrawi, C. Lever, K. Valakuzhy, R. Court, K. Snow, F. Monroe, and M. Antonakakis, “The circle of life: A Large-Scale study of the IoT malware lifecycle,” in *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual Conference, Aug. 2021.
- [56] M. Botacin, F. D. Domingues, F. Ceschin, R. Machnicki, M. A. Zanata Alves, P. L. de Geus, and A. Grégio, “AntiViruses under the microscope: A hands-on perspective,” *Computers and Security*, vol. 112, p. 102 500, Jan. 2022.
- [57] M. Botacin, A. Kalysch, and A. R. A. Grégio, “The internet banking [in]security spiral: Past, present, and future of online banking protection mechanisms based

on a brazilian case study,” in *Proceedings of the 14th International Conference on Availability, Reliability and Security (ARES)*, Canterbury CA, United Kingdom, Aug. 2019.

- [58] D. Bove and A. Kalysch, “In pursuit of a secure UI: The cycle of breaking and fixing android’s UI,” *it - Information Technology*, vol. 61, no. 2-3, pp. 147–156, Mar. 2019.
- [59] A. Kalysch, D. Bove, and T. Müller, “How android’s UI security is undermined by accessibility,” in *Proceedings of the 2nd Reversing and Offensive-Oriented Trends Symposium (ROOTS)*, Vienna, Austria, Nov. 2018.
- [60] *Google play protect on-device protections*. [Online]. Available: <https://developers.google.com/android/play-protect/client-protections>.
- [61] *Accessibilityserviceinfo: Isaccessibilitytool()*. [Online]. Available: [https://developer.android.com/reference/android/accessibilityservice/AccessibilityServiceInfo#isAccessibilityTool\(\)](https://developer.android.com/reference/android/accessibilityservice/AccessibilityServiceInfo#isAccessibilityTool()).
- [62] *Malware on google play abusing accessibility service*. [Online]. Available: <https://www.zscaler.com/blogs/security-research/malware-google-play-abusing-accessibility-service>.
- [63] *Accessibilitymanager: Getenabledaccessibilityservicelist(int)*. [Online]. Available: [https://developer.android.com/reference/android/view/accessibility/AccessibilityManager#getEnabledAccessibilityServiceList\(int\)](https://developer.android.com/reference/android/view/accessibility/AccessibilityManager#getEnabledAccessibilityServiceList(int)).
- [64] *When your phone gets sick: Flubot abuses accessibility features to steal data*. [Online]. Available: <https://www.srlabs.de/blog-post/flubot-abuses-accessibility-features-to-steal-data>.
- [65] Android, *Accessibilityservice*, 2025. [Online]. Available: <https://developer.android.com/reference/android/accessibilityservice/AccessibilityService>.
- [66] H. Xu, M. Yao, R. Zhang, M. M. Dawoud, J. Park, and B. Saltaformaggio, “Dva: Extracting victims and abuse vectors from android accessibility malware,” in *Proceedings of the 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, Aug. 2024.
- [67] Android, *View*, 2025. [Online]. Available: <https://developer.android.com/reference/android/view/View>.
- [68] Android, *Accessibilitynodeinfo*, 2025. [Online]. Available: <https://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo>.

- [69] Android, *Accessibiltiyevent*, 2025. [Online]. Available: <https://developer.android.com/reference/android/view/accessibility/AccessibilityEvent>.
- [70] Mozilla, *Document object model*, 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Document%5C_Object%5C_Model.
- [71] Mozilla, *Aria*, 2025. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA>.
- [72] A. B. et al., *Accessibility object model*, 2025. [Online]. Available: <https://wicg.github.io/aom/explainer.html>.
- [73] A. Buck, *Mobile apps vs mobile websites: Why people spend 90% of their time in apps*, 2025. [Online]. Available: <https://www.mobiloud.com/blog/mobile-apps-vs-mobile-websites>.
- [74] Android, *Viewgroup*, 2025. [Online]. Available: <https://developer.android.com/reference/android/view/ViewGroup>.
- [75] Android, *Textview*, 2025. [Online]. Available: <https://developer.android.com/reference/android/widget/TextView>.
- [76] Android, *SetText(java.lang.charsequence)*, 2025. [Online]. Available: [https://developer.android.com/reference/android/widget/TextView%5C#setText\(java.lang.CharSequence\)](https://developer.android.com/reference/android/widget/TextView%5C#setText(java.lang.CharSequence)).
- [77] Android, *Getuniqueid()*, 2025. [Online]. Available: [https://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo%5C#getUniqueId\(\)](https://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo%5C#getUniqueId()).
- [78] Android, *Type_window_state_changed*, 2025. [Online]. Available: https://developer.android.com/reference/android/view/accessibility/AccessibilityEvent%5C#TYPE%5C_WINDOW%5C_STATE%5C_CHANGED.
- [79] Android, *Gettext()*, 2025. [Online]. Available: [https://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo%5C#getText\(\)](https://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo%5C#getText()).
- [80] Android, *Edittext*, 2025. [Online]. Available: <https://developer.android.com/reference/android/widget/EditText>.
- [81] Android, *Specify the input method type*, 2025. [Online]. Available: <https://developer.android.com/develop/ui/views/touch-and-input/keyboard-input/style>.
- [82] Mozilla, *Accessibility*, 2025. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/Accessibility>.

- [83] Appium, *Appium*, 2025. [Online]. Available: <https://appium.io/docs/en/latest/>.
- [84] Google, *Google cloud natural language api basics*, 2025. [Online]. Available: <https://cloud.google.com/natural-language/docs/basics>.
- [85] Androzoo, *Androzoo*, 2025. [Online]. Available: <https://androzoo.uni.lu/>.
- [86] J. Ren, M. Lindorfer, D. Dubois, A. Rao, D. Choffnes, and N. Vallina-Rodriguez, "Bug fixes, improvements, ... and privacy leaks - a longitudinal study of pii leaks across android app versions," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.
- [87] Android, *Webview*, 2025. [Online]. Available: <https://developer.android.com/reference/android/webkit/WebView>.
- [88] Android, *Overview of android custom tabs*, 2025. [Online]. Available: <https://developer.chrome.com/docs/android/custom-tabs>.
- [89] Google, *Navigate your device with talkback*, 2025. [Online]. Available: <https://support.google.com/accessibility/android/answer/6006598?sjid>.
- [90] Android, *Type_view_accessibility_focused*, 2025. [Online]. Available: https://developer.android.com/reference/android/view/accessibility/AccessibilityEvent%5C#TYPE%5C_VIEW%5C_ACCESSIBILITY%5C_FOCUSED.
- [91] K. Anderson, *Create your own local ai assistant for enhanced privacy*. [Online]. Available: <https://www.cognativ.com/blogs/post/create-your-own-local-ai-assistant-for-enhanced-privacy/271>.
- [92] A. Maiya, *Document intelligence with onprem.llm*. [Online]. Available: <https://medium.com/data-science-collective/document-intelligence-with-onprem-llm-5ea69e5d8204>.
- [93] S. Sinha, *Personal ai that runs locally: How small llms are powering privacy-first experiences*. [Online]. Available: <https://brimlabs.ai/blog/personal-ai-that-runs-locally-how-small-llms-are-powering-privacy-first-experiences/>.
- [94] *Android developers. data and file storage overview*. [Online]. Available: <https://developer.android.com/training/data-storage>.
- [95] D. Oygenblik, C. Yagemann, J. Zhang, A. Mastali, J. Park, and B. Saltaformaggio, "Ai psychiatry: Forensic investigation of deep learning networks in memory images," in *Proceedings of the 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, Aug. 2024.

- [96] H. Hichri. [Online]. Available: <https://huggingface.co/blog/not-lain/kv-caching>.
- [97] *Revealed: The contentious tool us immigration uses to get your data from tech firms.* [Online]. Available: <https://www.theguardian.com/us-news/2023/may/25/us-immigration-surveillance-google-twitter-meta-personal-data>.
- [98] *Apple, google, meta are sharing more data with the us government than ever, proton finds.* [Online]. Available: <https://thenextweb.com/news/big-tech-sharing-more-data-with-us-government>.
- [99] *Government requests for customer data report.* [Online]. Available: <https://www.microsoft.com/en-us/corporate-responsibility/reports/government-requests/customer-data>.
- [100] *Llama.cpp. llm inference in c/c++.* [Online]. Available: <https://github.com/ggml-org/llama.cpp>.
- [101] *Ggml. tensor library for machine learning.* [Online]. Available: <https://github.com/ggml-org/ggml>.
- [102] *Google play store.* [Online]. Available: https://play.google.com/store/games?hl=en_US.
- [103] *Iris (offline gpt).* [Online]. Available: https://play.google.com/store/apps/details?id=com.nervesparks.irisGPT%5C&hl=en_IN.
- [104] *Iris – features.* [Online]. Available: https://github.com/nerve-sparks/iris_android?tab=readme-ov-file#features.
- [105] *Hugging face. the platform where the machine learning community collaborates on models, datasets, and applications.* [Online]. Available: <https://huggingface.co/>.
- [106] *Android 15.* [Online]. Available: <https://developer.android.com/about/versions/15>.
- [107] *Llama-3.2-1b-instruct-q6_k_l.* [Online]. Available: <https://huggingface.co/bartowski/Llama-3.2-1B-Instruct-GGUF>.
- [108] *Qwen2.5-1.5b-instruct-q6_k_l.* [Online]. Available: <https://huggingface.co/bartowski/Qwen2.5-1.5B-Instruct-GGUF>.
- [109] *Gemma-2-2b-it-q6_k.* [Online]. Available: https://huggingface.co/roshanai/gemma-2-2b-it-Q6%5C_K-GGUF.
- [110] *Frida. dynamic instrumentation toolkit for developers, reverse-engineers and security researchers.* [Online]. Available: <https://frida.re/>.

- [111] *Objection - runtime mobile exploration*. [Online]. Available: <https://github.com/sensepost/objection?tab=readme-ov-file>.
- [112] *Magisk. the magic mask for android*. [Online]. Available: <https://github.com/topjohnwu/Magisk>.
- [113] *Chatterui - a simple app for llms*. [Online]. Available: <https://github.com/Vali-98/ChatterUI>.
- [114] *Running on the edge: What, why and how. alternatives for running on-device ai on android*. [Online]. Available: <https://iurysouza.dev/android-on-device-ai/>.
- [115] *Llama.cpp: The lightweight engine behind local llms*. [Online]. Available: <https://www.sandgarden.com/learn/llama-cpp>.
- [116] *Mllm. fast and lightweight multimodal llm inference engine for mobile and edge devices*. [Online]. Available: <https://github.com/UbiquitousLearning/mllm>.
- [117] *Mediapipe. on-device machine learning for everyone*. [Online]. Available: <https://github.com/google-ai-edge/mediapipe?tab=readme-ov-file>.
- [118] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang, "Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2011.
- [119] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, Illinois, Nov. 2009.
- [120] B. Saltaformaggio, Z. Gu, X. Zhang, and D. Xu, "Dscrite: Automatic rendering of forensic information from memory images via application logic reuse," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.
- [121] Z. Lin, J. Rhee, C. Wu, X. Zhang, and D. Xu, "Dimsum: Discovering semantic data of interest from un-mappable memory with confidence," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2012.
- [122] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled Reverse Engineering of Types in Binary Programs," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2011.

- [123] S. Aonzo, A. Merlo, G. Tavella, and Y. Fratantonio, “Phishing attacks on modern android,” in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.
- [124] Y. Leguesse, M. Vella, C. Colombo, and J. C. H. Castro, “Reducing the forensic footprint with android accessibility attacks,” in *Proceedings of International Workshop on Security and Trust Management*, Guildford, UK, Sep. 2020.
- [125] F. Mehralian, N. Salehnamadi, S. F. Huq, and S. Malek, “Too much accessibility is harmful! automated detection and analysis of overly accessible elements in mobile apps,” in *Proceedings of the 37th International Conference on Automated Software Engineering (ASE)*, Rochester, MI, Oct. 2022.
- [126] C. Lei, Z. Ling, Y. Zhang, K. Dong, K. Liu, J. Luo, and X. Fu, “Do not give a dog bread every time he wags his tail: Stealing passwords through content queries (CONQUER) attacks,” in *Proceedings of the 2023 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2023.
- [127] Y. Yan, Z. Li, Q. A. Chen, C. Wilson, T. Xu, E. Zhai, Y. Li, and Y. Liu, “Understanding and detecting overlay-based android malware at market scales,” in *Proceedings of the 17th ACM International Conference on Mobile Computing Systems (MobiSys)*, Seoul, South Korea, Jun. 2019.
- [128] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi, “ASM: A programmable interface for extending android security,” in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.
- [129] M. Yao, R. Zhang, H. Xu, S.-H. Chou, V. C. Paturi, A. K. Sikder, and B. Saltaformaggio, “Pulling off the mask: Forensic analysis of the deceptive creator wallets behind smart contract fraud,” in *Proceedings of the 45th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, May 2024.
- [130] X. Li, A. Yepuri, and N. Nikiforakis, “Double and nothing: Understanding and detecting cryptocurrency giveaway scams,” in *Proceedings of the 2023 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2023.
- [131] E. Liu, S. Rao, S. Havron, G. Ho, S. Savage, G. Voelker, and D. McCoy, “No privacy among spies: Assessing the functionality and insecurity of consumer android spyware apps,” in *Proceedings of Privacy Enhancing Technologies Symposium (PETS)*, Lausanne, Switzerland, Jul. 2023.
- [132] S. Li, S. A. A. Shah, M. U. Khan, S. A. Khayam, A.-R. Sadeghi, and R. Schmitz, “Breaking e-banking CAPTCHAs,” in *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, Austin, TX, Dec. 2009.

- [133] A. R. A. Grégio, D. S. F. Filho, V. M. Afonso, P. L. de Geus, V. F. Martins, and M. Jino, “An empirical analysis of malicious internet banking software behavior,” in *Proceedings of ACM Symposium on Applied Computing (SAC)*, Coimbra, Portugal, Mar. 2013.
- [134] M. Botacin, H. Aghakhani, S. Ortolani, C. Kruegel, G. Vigna, D. Oliveira, P. L. de Geus, and A. R. A. Grégio, “One size does not fit all: A longitudinal analysis of brazilian financial malware,” *ACM Transactions on Privacy and Security*, vol. 24, no. 2, 11:1–11:31, May 2021.
- [135] T. Holz, M. Engelberth, and F. C. Freiling, “Learning more about the underground economy: A case-study of keyloggers and dropzones,” in *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS)*, Saint Malo, France, Sep. 2009.
- [136] C. Li, D. He, S. Li, S. Zhu, S. Chan, and Y. Cheng, “Android-based cryptocurrency wallets: Attacks and countermeasures,” in *Proceedings of 2020 IEEE International Conference on Blockchain (Blockchain)*, Rhodes Island, Greece, Nov. 2020.
- [137] S. Houy, P. Schmid, and A. Bartel, “Security aspects of cryptocurrency wallets—a systematic literature review,” *ACM Computing Surveys*, vol. 56, no. 1, 4:1–4:31, Jan. 2024.
- [138] S. Chen, T. Su, L. Fan, G. Meng, M. Xue, Y. Liu, and L. Xu, “Are mobile banking apps secure? what can be improved?” In *Proceedings of the 26th joint meeting of European Software Engineering Conference (ESEC) and ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Lake Buena Vista, FL, Oct. 2018.
- [139] A. Kellner, M. Horlboge, K. Rieck, and C. Wressnegger, “False sense of security: A study on the effectivity of jailbreak detection in banking apps,” in *Proceedings of the 4th European Symposium on Security and Privacy (EuroS&P)*, Stockholm, Sweden, Jun. 2019.
- [140] S. Chen, L. Fan, G. Meng, T. Su, M. Xue, Y. Xue, Y. Liu, and L. Xu, “An empirical assessment of security risks of global android banking apps,” in *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, Seoul, South Korea, Jun. 2020.
- [141] J. Fuller, R. P. Kasturi, A. Sikder, H. Xu, B. Arik, V. Verma, E. Asdar, and B. Saltaformaggio, “C3po: Large-scale study of covert monitoring of c&c servers via over-permissioned protocol infiltration,” in *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*, Seoul, South Korea, Nov. 2021.

- [142] R. Pai Kasturi, J. Fuller, Y. Sun, O. Chabklo, A. Rodriguez, J. Park, and B. Saltaformaggio, “Mistrust plugins you must: A large-scale study of malicious plugins in wordpress marketplaces,” in *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, Aug. 2022.
- [143] R. Pai Kasturi, Y. Sun, R. Duan, O. Alrawi, E. Asdar, V. Zhu, Y. Kwon, and B. Saltaformaggio, “Tardis: Rolling back the clock on cms-targeting cyber attacks,” in *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*, Virtual Conference, May 2020.
- [144] O. Alrawi, M. Ike, M. Pruett, R. P. Kasturi, S. Barua, T. Hirani, B. Hill, and B. Saltaformaggio, “Forecasting malware capabilities from cyber attack memory images,” in *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual Conference, Aug. 2021.
- [145] R. Zhang, R. P. Sridhar, M. Yao, Z. Yang, D. Oygenblik, H. Xu, V. Dave, C. Herley, P. England, and B. Saltaformaggio, “Identifying incoherent search sessions: Search click fraud remediation under real-world constraints,” in *Proceedings of the 46th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, May 2025.
- [146] R. Zhang, M. Yao, H. Xu, O. Alrawi, J. Park, and B. Saltaformaggio, “Hitchhiking vaccine: Enhancing botnet remediation with remote code deployment reuse,” in *Proceedings of the 2025 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2025.
- [147] O. Alrawi, C. Zuo, R. Duan, R. P. Kasturi, Z. Lin, and B. Saltaformaggio, “The betrayal at cloud city: An empirical analysis of cloud-based mobile backends,” in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.
- [148] M. Yao, J. Fuller, R. P. Sridhar, S. Agarwal, A. K. Sikder, and B. Saltaformaggio, “Hiding in plain sight: An empirical study of web application abuse in malware,” in *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.
- [149] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell, “A layered architecture for detecting malicious behaviors,” in *Proceedings of the 11th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Cambridge, Massachusetts, Sep. 2008.
- [150] E. Stinson and J. C. Mitchell, “Characterizing bots’ remote control behavior,” in *Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, Lucerne, CH, Jul. 2007.

- [151] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, “CopperDroid: Automatic reconstruction of android malware behaviors,” in *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2015.
- [152] X. Deng and J. Mirkovic, “Malware analysis through high-level behavior,” in *Proceedings of the 11th USENIX Workshop on Cyber Security Experimentation and Test (CSET)*, Baltimore, MD, Aug. 2018.
- [153] B. Saltaformaggio, H. Choi, K. Johnson, Y. Kwon, Q. Zhang, X. Zhang, D. Xu, and J. Qian, “Eavesdropping on fine-grained user activities within smartphone apps over encrypted network traffic,” in *Proceedings of the 10th USENIX Workshop on Offensive Technologies (WOOT)*, Austin, TX, Aug. 2016.
- [154] C. Zuo and Z. Lin, “SMARTGEN: Exposing server urls of mobile apps with selective symbolic execution,” in *Proceedings of the 26th International World Wide Web Conference (WWW)*, Perth, Australia, Apr. 2017.
- [155] Y. Fratantonio, A. Bianchi, W. K. Robertson, E. Kirda, C. Krügel, and G. Vigna, “TriggerScope: Towards detecting logic bombs in android applications,” in *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, San Jose, CA, May 2016.
- [156] M. Y. Wong and D. Lie, “IntelliDroid: A targeted input generator for the dynamic analysis of android malware,” in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.
- [157] X. Wang, Y. Yang, and S. Zhu, “Automated hybrid analysis of android malware through augmenting fuzzing with forced execution,” *IEEE Transactions on Mobile Computing*, vol. 18, no. 12, pp. 2768–2782, Dec. 2019.
- [158] Z. Lin, R. Wang, X. Jia, J. Yang, D. Zhang, and C. Wu, “ForceDROID: Extracting hidden information in android apps by forced execution technique,” in *Proceedings of the 15th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, Tianjin, China, Aug. 2016.
- [159] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, “X-Force: Force-executing binary programs for security applications,” in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.
- [160] Z. Yang, J. Allen, M. Landen, R. Perdisci, and W. Lee, “Trident: Towards detecting and mitigating web-based social engineering attacks,” in *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

- [161] J. Allen, Z. Yang, M. Landen, R. Bhat, H. Grover, A. Chang, Y. Ji, R. Perdisci, and W. Lee, “Mnemosyne: An effective and efficient postmortem watering hole attack investigation system,” in *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Virtual Conference, Nov. 2020.
- [162] F. Xiao, Z. Yang, J. Allen, G. Yang, G. Williams, and W. Lee, “Understanding and mitigating remote code execution vulnerabilities in cross-platform ecosystem,” in *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*, Los Angeles, CA, Nov. 2022.
- [163] J. Allen, Z. Yang, F. Xiao, M. Landen, R. Perdisci, and W. Lee, “Webrr: A forensic system for replaying and investigating web-based attacks in the modern web,” in *Proceedings of the 33rd USENIX Security Symposium (Security)*, Philadelphia, PA, Aug. 2024.
- [164] Z. Yang, S. Chung, J. Chen, R. Zhang, B. Saltaformaggio, and W. Lee, “Coindef: A comprehensive code injection defense for the electron framework,” in *Proceedings of the 46th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, May 2025.
- [165] W. Zou, R. Geng, B. Wang, and J. Jia, “PoisonedRAG: Knowledge Corruption Attacks to Retrieval-Augmented Generation of Large Language Models,” in *Proceedings of the 34th USENIX Security Symposium (Security)*, Seattle, WA, Aug. 2025.
- [166] K.-H. Hung, C.-Y. Ko, A. Rawat, I. Chung, W. H. Hsu, P.-Y. Chen, et al., “Attention Tracker: Detecting Prompt Injection Attacks in LLMs,” in *Proceedings of the 2025 Annual Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics*, Albuquerque, New Mexico, USA, May 2025.
- [167] S. Shan, A. N. Bhagoji, H. Zheng, and B. Y. Zhao, “Poison Forensics: Traceback of Data Poisoning Attacks in Neural Networks,” in *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, Aug. 2022.
- [168] Z. Chen, Z. Xiang, C. Xiao, D. Song, and B. Li, “AgentPoison: Red-teaming LLM Agents via Poisoning Memory or Knowledge Bases,” in *Proceedings of the 38th Conference on Neural Information Processing Systems (NeurIPS)*, Vancouver, Canada, Dec. 2024.
- [169] G. Wu, Z. Zhang, Y. Zhang, W. Wang, J. Niu, Y. Wu, and Y. Zhang, “I know what you asked: Prompt leakage via kv-cache sharing in multi-tenant llm serving,” in *Proceedings of the 2025 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2025.

- [170] E. Shayegani, M. A. A. Mamun, Y. Fu, P. Zaree, Y. Dong, and N. Abu-Ghazaleh, “Survey of Vulnerabilities in Large Language Models Revealed by Adversarial Attacks,” *arXiv preprint arXiv:2310.10844*, 2023. [Online]. Available: <http://arxiv.org/abs/2310.10844>.
- [171] X. Liu, J. Wang, J. Sun, X. Yuan, G. Dong, P. Di, W. Wang, and D. Wang, “Prompting Frameworks for Large Language Models: A Survey,” *arXiv preprint arXiv:2311.12785*, 2023. [Online]. Available: <http://arxiv.org/abs/2311.12785>.
- [172] F. Perez and I. Ribeiro, “Ignore Previous Prompt: Attack Techniques For Language Models,” *arXiv preprint arXiv:2211.09527*, 2022. [Online]. Available: <http://arxiv.org/abs/2211.09527>.
- [173] A. Wei, N. Haghtalab, and J. Steinhardt, “Jailbroken: How Does LLM Safety Training Fail?” In *Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS)*, New Orleans, Louisiana, Dec. 2023.
- [174] M. Ibrahim, G. S. Tuncay, Z. B. Celik, A. Machiry, and A. Bianchi, “LM-Scout: Analyzing the Security of Language Model Integration in Android Apps,” *arXiv preprint arXiv:2505.08204*, 2025. [Online]. Available: <http://arxiv.org/abs/2505.08204>.
- [175] Y. Liu, G. Deng, Y. Li, K. Wang, T. Zhang, Y. Liu, H. Wang, Y. Zheng, and Y. Liu, “Prompt Injection attack against LLM-integrated Applications,” *arXiv preprint arXiv:2306.05499*, 2023. [Online]. Available: <http://arxiv.org/abs/2306.05499>.
- [176] Alessandro Brucato, *LLMjacking: Stolen Cloud Credentials Used in New AI Attack*, <https://sysdig.com/blog/llmjacking-stolen-cloud-credentials-used-in-new-ai-attack/>, [Online; accessed: 28-October-2025], 2024.
- [177] T. Liu, Z. Deng, G. Meng, Y. Li, and K. Chen, “Demystifying RCE Vulnerabilities in LLM-Integrated Apps,” in *Proceedings of the 31st ACM Conference on Computer and Communications Security (CCS)*, Salt Lake City, UT, Oct. 2024.
- [178] W. M. Si, M. Backes, and Y. Zhang, “Mondrian: Prompt Abstraction Attack Against Large Language Models for Cheaper API Pricing,” *arXiv preprint arXiv:2308.03558*, 2023. [Online]. Available: <http://arxiv.org/abs/2308.03558>.
- [179] T. Cui, Y. Wang, C. Fu, Y. Xiao, S. Li, X. Deng, Y. Liu, Q. Zhang, Z. Qiu, P. Li, Z. Tan, J. Xiong, X. Kong, Z. Wen, K. Xu, and Q. Li, “Risk Taxonomy, Mitigation, and Assessment Benchmarks of Large Language Model Systems,” *arXiv preprint arXiv:2401.05778*, 2024. [Online]. Available: <http://arxiv.org/abs/2401.05778>.
- [180] D. Ersoy, B. Lee, A. Shreeksumar, A. Arunasalam, M. Ibrahim, A. Bianchi, and Z. B. Celik, “Investigating the Impact of Dark Patterns on LLM-Based Web Agents,”

in *Proceedings of the 47th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, May 2026.

- [181] S. Kim, S. Yun, H. Lee, M. Gubri, S. Yoon, and S. J. Oh, “ProPILE: Probing Privacy Leakage in Large Language Models,” in *Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS)*, New Orleans, Louisiana, Dec. 2023.
- [182] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz, “Not What You’ve Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection,” in *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, Copenhagen, Denmark, Nov. 2023.
- [183] G. Deng, Y. Liu, Y. Li, K. Wang, Y. Zhang, Z. Li, H. Wang, T. Zhang, and Y. Liu, “MasterKey: Automated Jailbreak Across Multiple Large Language Model Chatbots,” in *Proceedings of the 2024 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2024.
- [184] E. Shayegani, Y. Dong, and N. Abu-Ghazaleh, “Jailbreak in Pieces: Compositional Adversarial Attacks on Multi-Modal Language Models,” *arXiv preprint arXiv:2307.14539*, 2023. [Online]. Available: <http://arxiv.org/abs/2307.14539>.
- [185] X. Shen, Z. Chen, M. Backes, Y. Shen, and Y. Zhang, “Do Anything Now: Characterizing and Evaluating In-The-Wild Jailbreak Prompts on Large Language Models,” in *Proceedings of the 31st ACM Conference on Computer and Communications Security (CCS)*, Salt Lake City, UT, Oct. 2024.
- [186] J. Yu, X. Lin, Z. Yu, and X. Xing, “GPTFUZZER: Red Teaming Large Language Models with Auto-Generated Jailbreak Prompts,” *arXiv preprint arXiv:2309.10253*, 2023. [Online]. Available: <http://arxiv.org/abs/2309.10253>.
- [187] F. Jiang, Z. Xu, L. Niu, Z. Xiang, B. Ramasubramanian, B. Li, and R. Poovendran, “ArtPrompt: ASCII Art-based Jailbreak Attacks against Aligned LLMs,” in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, Bangkok, Thailand, Jun. 2024.
- [188] Y. Liu, G. Deng, Z. Xu, Y. Li, Y. Zheng, Y. Zhang, L. Zhao, T. Zhang, and K. Wang, “A Hitchhiker’s Guide to Jailbreaking ChatGPT via Prompt Engineering,” in *Proceedings of the 4th International Workshop on Software Engineering and AI for Data Quality in Cyber-Physical Systems/Internet of Things*, Porto de Galinhas, Brazil, Jul. 2024.
- [189] W. M. Si, M. Backes, J. Blackburn, E. De Cristofaro, G. Stringhini, S. Zannettou, and Y. Zhang, “Why So Toxic? Measuring and Triggering Toxic Behavior in Open-

Domain Chatbots,” in *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*, Los Angeles, CA, Nov. 2022.

- [190] R. Petrik, B. Arik, and J. M. Smith, “Towards architecture and os-independent malware detection via memory forensics,” in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.
- [191] F. Pagani and D. Balzarotti, “Back to the whiteboard: A principled approach for the assessment and design of memory forensic techniques,” in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.
- [192] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, “Vcr: App-agnostic recovery of photographic evidence from android device memory images,” in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, CO, Oct. 2015.
- [193] W. Guo, D. Mu, X. Xing, M. Du, and D. Song, “DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis,” in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.
- [194] M. Yao, J. Fuller, R. P. Sridhar, S. Agarwal, A. K. Sikder, and B. Saltaformaggio, “Hiding in plain sight: An empirical study of web application abuse in malware,” in *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.
- [195] R. Bhatia, B. Saltaformaggio, S. J. Yang, A. I. Ali-Gombe, X. Zhang, D. Xu, and G. G. Richard III, “Tipped off by your memory allocator: Device-wide user activity sequencing from android memory images,” in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

VITA

Haichuan Xu earned a Master of Science in Computer Engineering from Georgia Institute of Technology in 2021 and a Bachelor of Science with Honors in Computer Engineering from University of Illinois at Urbana-Champaign in 2019. His research interests lie in fraud and abuse detection, with a focus on malware analysis, cyber forensics, and privacy leakage discovery. He will become a Research Scientist at Meta in June 2026.