
# DVa: Extracting Victims and Abuse Vectors from Android Accessibility Malware

Haichuan Xu, Mingxuan Yao, and Runze Zhang, *Georgia Institute of Technology;* Mohamed Moustafa Dawoud, *German International University;* Jeman Park, *Kyung Hee University;* Brendan Saltaformaggio, *Georgia Institute of Technology*

https://www.usenix.org/conference/usenixsecurity24/presentation/xu-haichuan

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

# DVa: Extracting Victims and Abuse Vectors from Android Accessibility Malware

Haichuan Xu[1], Mingxuan Yao[1], Runze Zhang[1], Mohamed Moustafa Dawoud[2],
Jeman Park[3]*, Brendan Saltaformaggio[1]*
[1]*Georgia Institute of Technology* [2]*German International University* [3]*Kyung Hee University*

## Abstract

The Android accessibility (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. We 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. We 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 Introduction

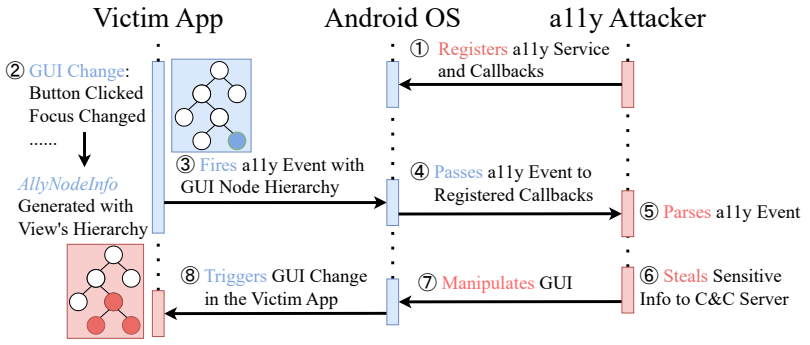
Android's accessibility service [1], called a11y, provides extensive utilities to assist users in better navigating their phones. It stands as an exception to Android's app-isolated sandbox design in the sense that it grants a11y apps the ability to examine and navigate foreground graphical user interface (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 a11y 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.

*Co-corresponding author.

Regarding a11y security, several works have proposed data-flow restraints [5], [6] to counteract the proof-of-concept (PoC) attacks [7]–[10]. However, our research revealed (§6.3) that modern a11y malware can still evade the most recent Android security patch [11] and state-of-the-art (SOTA) data-flow constraint defense [5]. Techniques also have been proposed to identify the misuse of a11y service in benign apps [7], [12]–[14]. However, without considering the targets of these misuses, an investigator will fall short of understanding their in-context abuse vectors because generic a11y service 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 a11y 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 a11y 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.

During our research, we found that a11y malware relied on Android APIs and broadcast a11y 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 a11y malware probes the information of the installed app, it




(a) Steps an a11y attacker uses to illicitly interact with a victim app.

```
1 public class malwareA11yService extends A11yService{
2   @Override
3   public void onA11yEvent (A11yEvent e) {
4     // parse a11y event properties
5     int type = e.getEventType();
6     CharSequence name = e.getPackageName();
7     ...
8     // parse the a11y node structures
9     A11yNodeInfo node = e.getRecord().getSource();
10    // malicious code to record GUI states
11    CharSequence nodeText = node.getText();
12    nodeText.sendToCnC();
13    ...
14    // malicious code to manipulate GUI screen
15    node.performAction(ACTION_CLICK);
16    ...
17  }
18 }
```

(b) Sample malware a11y event handling logic.

Figure 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).

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.

We developed DVa[1], 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 (§3.1). Using dynamic execution traces, DVa further utilizes an abuse-vector-guided symbolic execution strategy to identify and attribute abuse routines to victims (§3.2). Finally, DVa detects a11y-empowered persistence mechanisms (§3.3) to understand how malware obstructs legal queries or removal attempts.

Using DVa, we conducted investigations of five Google Pixel 3 devices infected with 9,850 malware samples collected from VirusTotal [15] 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 Prevention*, exploited by 92% of malware. Lastly, we have made DVa available at: https://github.com/CyFI-Lab-Public/DVa.

[1] **D**etector of **V**ictim-specific **a**11y abuse

## 2 Overview

Android apps' GUI contains substantial user-controlled sensitive information [16], [17]. Although the Android *a11y 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 Abusing Android a11y Service

Malware uses the eight steps shown as circled numbers in Figure 1a to illegally acquire sensitive information or conduct malicious GUI actions. First, the malware registers an *a11y 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 [18] 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 *a11y 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 1a. The *a11y 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

```
1 public class cerberusA11yService extends A11yService{
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 a11y attack code from CnC
12                    requestNewPayload(pi.packageName.hash());
13                    classLoader.loadClass(newPayload);
14                }
15            }
16        }
17    }
18    @Override
19    public void onA11yEvent (A11yEvent 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 a11y 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 a11y Abuse Code

Figure 2: Dynamically loaded automatic transaction abuse routine targeting the Chime banking app extracted by DVa.

elements are represented as an *a11y node info* data structure, as shown in the blue circle inside the tree.

After the Android OS receives an *a11y event*, it dispatches the event to all registered a11y services that listen for such an event through a callback, as shown in step ④. The callback is handled by a declared `onA11yEvent` handler in the malware's registered *a11y service*. A sample a11y event handler is illustrated in Figure 1b. This handler can then parse the *a11y event* (Lines 4-9 Figure 1b, ⑤ Figure 1a), steal information fetched from the event (Lines 10-13 Figure 1b, ⑥ Figure 1a), and subsequently manipulate the GUI according to the GUI node hierarchy contained in the event (Lines 14-16 Figure 1b, ⑦ ⑧ Figure 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.

## 2.2 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 malware[2], 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 [19], [20]. Lines 20-27 of Figure 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.

**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. Lines 3-9 of Figure 2 show the victim discovery routines that lead to the loading of new abuse code. When the a11y 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. 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, a11y 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

[2]MD5: 9236f4009503b4216e6773741b9d8ec0

(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. After solving the symbolic constraints, DVa determines that these routines all depend on the value of the a11y 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 a11y 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.

## 2.3 Threat Model

DVa's goal is to identify the victims targeted by a11y malware and attribute a11y abuse vectors to each victim. We assume the user's Android device is infected 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 [21] and can trick users into granting a11y permissions [22]. DVa's scope of abuse vector detection covers malware actions conducted by a11y APIs and targeting specific victim apps. That said, malware actions that complement a11y abuse but are conducted without a11y APIs (e.g., displaying overlay screens with SYSTEM_ALERT_WINDOW) are not considered by DVa. Additionally, a11y 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.

# 3 Methodology

DVa is a backend service that conducts analysis on a11y 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 [23] as input[3]. DVa pulls the list of installed applications from adb as victim app candidates on the device. DVa requires no prior knowledge of malicious a11y apps. For each a11y malware sample, DVa outputs its targeted victim apps, victim-specific a11y abuse vectors, and persistence mechanisms enabled by a11y.

## 3.1 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, a11y malware dynamically loads payloads locally or fetched from C&C servers during runtime. Specifically, the most advanced a11y malware loads attack payloads only after scanning installed apps on victim devices, checking victim app configurations, and receiving acknowledgments from C&C servers [24]. 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 a11y events that malware is checking for.

To overcome these challenges, DVa uses dynamic hooks to mimic the existence of both targeted victims apps (§3.1.1) and their generated a11y events (§3.1.2).

### 3.1.1 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 [25].

Algorithm 1 presents DVa's strategy to model victim query APIs. DVa keeps a predetermined victim database ($\Pi$) 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 [26] and SensorTower [27]. Currently, the database contains 37K unique Android packages. As shown in Lines 2-28 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)

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

**Algorithm 1:** DVa's victim query modeling.

```
   // Victim model, Π_x is model of a single victim
1  Π = {Π_1, Π_2, Π_3, .....};
   // 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              Π_x = Π.get(pm.param.packageName);
10             R = new pm.returnType;
               // Pierce back custom victim fields
11             for field ∈ pm.returnType do
12                 a = Π_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 = new pm.returnType;
               // Pierce back custom victim field from
                  many victim apps
20             for package ∈ pm.param.packageNames do
21                 Π_x = Π.get(package);
22                 field = Π_x.get(pm.returnType);
23                 R.set(package, field);
24             end
25             R.pad();
26             return R;
27         end
28     end
29 end
```

package (e.g., App $X$), DVa looks up the associated victim model $\Pi_x$. DVa then pierces all fields required in the API return value from $\Pi_x$ 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 $A$, $B$ ... $N$), DVa obtains the required fields from each victim model $\Pi_A, \Pi_B ... \Pi_N$, 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 Appendix A).

#### 3.1.2 Mimicking Victim a11y Events

For advanced malware that eavesdrops on device window states and launches victim-specific attacks only after the user

**Algorithm 2:** DVa's large-scale triggers of victim a11y events.

```
   // Trigger startup events for each victim model
1  for Π_x ∈ Π do
2      ae = a11yEvent.obtain();
       // Mimics an app startup event
3      ae.setEventType(WINDOW_STATE_CHANGED);
       // Pierce together a11y event with victim model
4      for field ∈ a11yEvent do
5          Π_xField = Π_x.get(field);
6          ae.set(Π_xField);
7      end
       // Broadcast victim a11y event to the malware
8      am = getSystemService(Context.a11yService);
9      am.sendA11yEvent(ae);
10 end
```

instantiates the victim app, DVa needs to automatically fire victim app a11y events on a large scale to trigger the malware's victim-specific attack routines.

Algorithm 2 presents DVa's strategy for triggering victim a11y events on a large scale. For each victim model $\Pi_x$ in the victim database $\Pi$, DVa obtains a default a11y 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 a11y event, DVa queries the victim model $\Pi_x$ and populates the a11y 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 a11y manager and broadcasts the customized a11y 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 a11y 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.

### 3.2 Abuse Vector Detection

After identifying the targets of a11y malware, DVa next finds unique abuse vectors empowered by the a11y service. However, detecting abuse vectors from the dynamic victim-guided execution is challenging. Specifically, the a11y event structures, embedded with trees of GUI elements that malware looks for, are enormous considering all victim apps.

#### 3.2.1 a11y Capabilities

To accurately model the abuse vectors, we first systematically categorize the capabilities of a11y APIs according to the official a11y API doc [1], [18], [28]. Since an a11y abuse relies on the Android a11y service, the malware has to use

Table 1: A11y capabilities, generic a11y techniques, and a11y-empowered abuse vectors in the context of victim apps.

| **a11y Capabilities**[1] | **Sample APIs or Identifiers** |
|---|---|
| **a11yService** | |
| GlobalAction | `performGlobalAction()`, `dispatchGesture()`, `findFocus()`, `takeScreenshot()`, `takeScreenshotOfWindow()` |
| **a11yEvent** | |
| 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` |
| **a11yNodeInfo** | |
| QueryNodeInfo | `findA11yNodeInfoByText()`, `getBoundsInScreen()`, `getText()`, `getPackageName()`, `isPassword()` |
| NodeAction | `performAction()` |
| **a11y 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.

capabilities provided by the service [1], the `a11yEvent` [18] dispatched to the service, or the `a11yNodeInfo` [28] embedded within the `a11yEvent`. The a11y capabilities section in Table 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 1, for an app to query the information of an on-screen element, it has to use APIs provided by the `a11yEvent` 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 `a11yNodeInfo` class, as shown in Row 12 of Table 1. DVa uses these a11y capabilities as basic blocks for modeling a11y behaviors.

### 3.2.2 a11y Techniques

Next, DVa models the generic techniques an app can achieve using the above capabilities. Since an a11y service receives an event whenever notable changes occurred in the GUI [18], 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. A11y Techniques section in Table 1 shows the techniques that can either read screen changes or inject input and their action models.

To detect these techniques, given malware's a11y 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 a11y event handler or reaches the top parent. DVa then selects the call sequences that originate from the a11y 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 [29].

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 *a11y 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

Table 2: Malware's a11y-empowered persistence mechanisms, their triggering conditions, and their a11y 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 a11y Permission Revocation | a11y Settings Screen | Back / Return Home |
| Escalating Privileges | a11y Permission Granted | Permission Screen with Screen Navigation |
| Uninstall Other Apps | Uninstall Package Setting(w/ Permission) | Screen Navigation |
| Disable Power Options | Power Options Dialog | Back / Return Home |

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.

#### 3.2.3 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. We 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) [2]–[4], [19], [20], [30]–[39]. We also queried in VirusTotal [15] and manually investigated all 43 unique a11y malware appended in the above reports and found the seven abuse vectors, as listed in Table 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 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.** We 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.

### 3.3 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 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 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

Table 3: Validation of the victims, abuse vectors, and persistence mechanisms detected by DVa together with AVClass2 labels of the top-10 a11y 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.

activity stabilizes after granting the a11y permission.

**Intercepting a11y Behaviors**. To interpret a11y responses of the mechanisms, DVa applies dynamic hooks to capture their a11y actions (listed in Table 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 a11y service to persist on a user device through *disable power options*. With DVa's persistence mechanisms detction, investigators can notify users of illicit changes to their device configurations.

# 4 Validating Our Techniques

## 4.1 Implementation

Abuse-vector-guided symbolic analysis in DVa is implemented in Java (7.3K lines) leveraging Soot [29], used in top-tier research [40]. Dynamic hooks leverage EdXposed [41] with 1.6K lines of Java code. Dynamic analysis management is implemented in Python (1.3K lines). AVClass2 [42], a SOTA malware labeling tool is used to classify malware families. For deriving malware labels, we 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 [26] and SensorTower [27], SOTA Android market intelligence services. Benign application icons and UI screens are collected from Google Image Search. We used an Ubuntu 20.04 LTS system to host DVa's static analysis module. DVa's dynamic analysis and a11y malware are hosted on 5 Google Pixel 3 (64GB, 4GB RAM) phones running Android 9.0 (Pie).

## 4.2 Validation Setup

Due to the physical disk size limitation, we 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 we used AVClass2 [42] to label our malware dataset and randomly picked samples from the top-10 malware families until we 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. We 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 3. We manually reverse-engineered these malware samples using Jadx [43] and used industry reports to derive the ground truth[4].

## 4.3 Validation Results

Table 3 presents DVa's validation results. Column 2 shows the top-10 a11y 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 a11y-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 3, DVa correctly identified (TP) 206 instances of targeted victims. Our 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, we 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

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

information. We 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. We found that the 3 FN victims – *Cajasur*, *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 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. We 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. We 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 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, we compared DVa's result with labels reported by the SOTA malware intelligence engine AVClass2. The last column of Table 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.

## 4.4 Coverage Assessment

Next, we evaluated how much of an advantage DVa's abuse vector modeling provides over a generic data-flow analysis. We hypothesized that a generic data-flow analysis would over-taint paths and lead to false positives in discovering a11y abuse and be insufficient in detecting behaviors reliant on dynamic execution traces. We 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 [40].

**Experiment Setup.** We 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. We set the source function to be the `onA11yEvent` handler and the sink functions to be all functions declared in `a11y service`, `a11y event`, and `a11y node info` classes (in total 246 functions). We found that initially, FlowDroid was unable to track data-flow from the `onA11yEvent` callback function because it doesn't have a concrete invocation. Because of this issue, we modified FlowDroid's source tracking logic of the `onA11yEvent` handler to taint the first assignment statement of the function parameter instead. We used the standard CHA algorithm for FlowDroid to generate the CG.

Table 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 a11y abuse.
3: Detected persistence mechanisms.

**Assessment result.** Table 4 shows the victims detected, number of paths flagged with a11y abuse, and persistence mechanisms detected by DVa and FlowDroid. The FP and FN columns for DVa are omitted because they were discussed in Table 3. FP and FN columns for FlowDroid are derived by manually examining the data-flow paths. As discussed in §4.3, we 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, we 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 a11y abuse, while FlowDroid flagged a total of 13,680. However, in Column 8 we see that FlowDroid incurred FP in 11,629 (85%) paths that do not constitute a11y abuse. This is because FlowDroid does not have access to the abuse vector modeling and flags irrelevant paths with generic a11y APIs. Column 9 shows that 269 paths were marked by DVa as a11y abuse but not traversed by FlowDroid. We 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 a11y abuse. While DVa extracted 206 victim apps, we 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.

## 5 Findings

To help gather large-scale malware behavioral insights from a11y malware, we deployed DVa on 5 Google Pixel 3 phones. These phones were repetitively infected with each of the 9,850 malware collected from VirusTotal [15] from August 2022 to December 2022. The selection criteria for the samples that they be (1) are labeled by at least five antivirus engines, ensuring that they are indeed malware [44] and (2) contain the `BIND_A11Y_SERVICE` permission declaration string in the app manifest file, which indicates that they abuse the Android a11y service. Out of the 9,850 samples collected, 7,700 samples are labeled by AVClass2 [42] 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).

### 5.1 Targeted Victims

Table 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, we found that 4,614 of the samples did not have any live C&C responses, so we excluded them from the victim-detection evaluation. For the other 945 samples, we manually investigated 10 random samples and found that six of them are generic utility apps that misuse the a11y service such as file manager, overlay widget, notification modifier apps, etc. In fact, prior work [7], [45] confirmed that benign apps also misuse the a11y service to achieve functionality purposes. These utility apps do not target specific victim apps, so we 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 a11y-based attacks found by prior work [46]. 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 tha has already occured, as well as notify developers of malware targets that have not yet been abused.

### 5.2 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 5 present the a11y techniques and abuse vectors tailored for each category of victim apps. Columns 3 and 4 show the different abuse

Table 5: Categories of victim apps, a11y 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 | a11y Tech. | Avg. Q[3] | Avg. A[4] |
|---|---|---|---|---|---|---|---|---|---|---|
| Banking | RU BR GB US MX | Auto Tran. | 11.7 | 35 | 10M+ | 755 | 12 | Scr. Nav.<br>AutoFill | 46.5<br>15.5 | 3.4<br>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.<br>Scr. Log. | 25.9<br>18.3 | 3.1<br>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.<br>AutoFill | 59.7<br>22.1 | 2.8<br>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.<br>AutoFill | 63.0<br>24.7 | 3.8<br>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.<br>Scr. Log. | 32.8<br>17.1 | 2.8<br>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.<br>AutoFill | 34.4<br>12.7 | 2.5<br>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.<br>AutoFill | 38.0<br>12.4 | 3.1<br>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.<br>Scr. Log | 28.9<br>15.8 | 2.5<br>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.<br>AutoFill | 33.6<br>15.3 | 2.5<br>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.<br>AutoFill | 39.5<br>11.8 | 3.4<br>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 a11y Query APIs used per a11y Technique. 4: Average of System a11y Action APIs used per a11y Technique.
5: 1,235 samples contain no AVClass family name. *: Not a sum of subtotals, multiple targeted victims per malware.
†: Excluding Others Column.

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 [9], [47], [48].

Column 9 of Table 5 shows the a11y techniques used by each abuse vector. We 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, we 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 a11y action APIs are used to perform concrete a11y 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.

## 5.3 Persistence Mechanisms

After extracting victim-specific abuse vectors adopted by a11y malware, DVa then helps investigators understand what a11y-empowered mechanisms malware adopts to persist on users' devices. Table 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 a11y 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 a11y 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 a11y 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 a11y-based measures to turn off system malware scanning and prevent users from querying, disabling, or uninstalling them.

Table 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.

**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.

## 6 Case Studies

### 6.1 a11y 2FA Stealer

Authenticator apps use 2FA codes to provide users additional protection against compromised static passwords. In our 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 *ScreenLog* 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.

**a11y Blocking by 2FA Apps.** Although a11y malware can steal 2FA codes, authenticator apps can choose to block untrusted a11y services from accessing the code. We 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. We then installed a custom app with an a11y service that listens for and parses a11y events from the views in each app that contain the 2FA codes. Only two of the 11 apps protect their 2FA codes from untrusted a11y services. We 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 a11y services. We manually reverse engineered the apps and found that both protect their views containing 2FA codes with the `A11yDataSensitive` property to block interaction with untrusted a11y services.

### 6.2 a11y Ransomware

DVa detected 131 instances of the Doublelocker malware that abuses a11y 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 a11y 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 a11y permission, thus increasing the attack's success rate.

Table 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.

### 6.3 Defenses Against a11y Malware

To understand how SOTA defenses are (in)effective at preventing a11y malware abuse, we evaluated how a11y malware behaves under the newest Android security patch (Android 13.0.0_r31) [11] and the most recent data-flow defense framework proposed by Huang et. al. [5]. We re-implemented the defense framework as stated in the study [5] 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. We deployed DVa on one Google Pixel 4 device with the latest Android security patch and another one with our re-implemented defense framework. We evaluated using 14 malware samples that target Android SDK Level 33 (Android 13) from the dataset.

Table 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 we 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, we 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 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.

## 7 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 [49] incorporated. Researchers have also proposed methods to distinguish benign and malicious a11y apps based on explicit user intention used as sources for a11y actions [5].

**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 §4.3. Since DVa relies on dynamic analysis, malware can deploy new evasion techniques and time-sensitive behaviors to hinder the analysis. Although not observed in §4.3, 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, we 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 our 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 a11y events. DVa's victim-guided dynamic analysis resolves both of these challenges by actively generating a11y events in the context of real victim apps' behaviors.

**Future of Android's a11y Malware.** Although Android 14 allows developers to restrict a11y information delivered to unvetted a11y services by checking the `A11Y_TOOL` flag [50], we 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 [21], [51].

**Developer's Defense.** App developers can choose not to broadcast a11y events if untrusted a11y services are found on a user's device [52]. However, this sacrifices the usability of the app because legitimate a11y services would also be blocked. The Coinbase app [53] adopted an out-of-band a11y verification that malware could not intercept: It displays a view that a11y cannot interact with, instructing users to shake the device to approve the use of an untrusted a11y 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 a11y delegates [54] to customize their exposed a11y events and declare `a11yDataSensitive` on views so that untrusted a11y services cannot interact with them.

## 8 Related Work

**Benign Misuse of a11y Service.** Benign services such as antivirus engines [45] and utility apps [7] can abuse the a11y service to automate legitimate tasks. Multiple works have focused on evaluating the misuse of the a11y service. Chen et al. [12] proposed dynamic analysis to automatically extract a11y issues while traversing the Android apps. Salehnamadi et al. [13] proposed Latte to automatically assess the functional correctness of an app's a11y features. Naseri et al. [14] conducted a study on how a11y functionalities are misused commonly by Android apps. Unlike detecting and analyzing benign misuse of the a11y service, focuses on dissecting malicious use of a11y to target victim apps.

**Attacks on a11y Service.** The a11y service provides malware with a unique surface [47] to launch phishing attacks [55] and make them more evasive [56]. Multiple works also focus on proposing PoC attacks that exploit the a11y service [9], [48]. Lei et al. [57] exposed a side channel of using consecutive content queries to guess passwords through a11y service. Mehralian et al. [58] exposed information leakage through overly accessible elements in Android's a11y service. Jang et al. [8] evaluated the a11y support for four operating systems and identified 12 attacks on them. Fratantonio et al. [10] uncovered an attack that can

control the UI feedback of an Android device should the malware be granted both the SYSTEM_ALERT_WINDOW and a11y permissions. Motivated by these attacks, DVa focuses on understanding a11y abuse conducted by real malware and the victims they target.

**Mobile Banking Security.** E-banking fraud [59]–[61] and attacks [62]–[65] have led to huge financial losses worldwide. Mobile banking apps are vulnerable to malware attacks [66], [67]. Multiple works focus on evaluating the security measures imposed by these apps [68], [69]. Chen et al. [70] 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. [46] evaluated the security flaws of Brazilian mobile banking apps, uncovering their susceptibility to UI and a11y-based attacks. Corroborating the security flaws in mobile banking apps, DVa contributes to this field of research by detecting a11y abuse vectors exploiting real victim banking apps targeted by mobile a11y malware.

**Defenses Against a11y Abuse.** Defenses have been proposed to restrict malicious usage of the a11y service [71], [72]. Fernandes et al. [6] proposed data-flow restriction on Android apps that only allows declared data-flow patterns by users while blocking all other undeclared flows. Huang et al. [5] introduced a more fine-grained sandbox design across the a11y service lifecycle that uses least-privileged data-flow constraints to secure the Android a11y service. §6.3 shows how existing defenses are ineffective in eliminating all a11y malware abuse – motivating the need for malware analysis techniques like DVa.

**Malware Analysis.** Some works use taint analysis [73], [74], API trace analysis [75]–[79], and network traffic analysis [80]–[82] to reveal malware behaviors. However, to attribute a11y attack vectors, DVa uses symbolic analysis [83]–[86] to match the constraints of a11y behaviors to their targets. Inspired by forced execution [87]–[89], DVa uses victim modeling to guide the execution of malware and loading of victim-specific abuse payloads.

## 9 Conclusion

We introduced DVa, a malware analysis pipeline to notify users and victims of a11y abuse vectors. Using DVa, we conducted analysis for 9,850 malware samples extracted from Google Pixel devices to uncover 215 victim apps that were abused with an average of 13.9 abuse vectors and six categories of persistence mechanisms empowered by a11y.

## Acknowledgments

We thank the anonymous reviewers for their constructive comments and feedback. In particular, we thank our shepherd for guidance throughout the revision process. We also thank our collaborators at Netskope for their support, insights, and suggestions throughout this research. This material was supported in part by the Office of Naval Research (ONR) under grants N00014-19-1-2179 and N00014-23-1-2073; the National Science Foundation (NSF) under grant 2143689; and Cisco Systems under an unrestricted gift. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of our sponsors and collaborators.

## 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-android-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] J. Huang, M. Backes, and S. Bugiel, "A11y 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] *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.

[12] 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.

[13] 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.

[14] 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.

[15] *Virustotal*. [Online]. Available: http://www.virustotal.com/.

[16] 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.

[17] 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.

[18] *Accessibilityevent (event types)*. [Online]. Available: https://developer.android.com/reference/android/view/accessibility/AccessibilityEvent#developer-guides.

[19] ThreatFabric, *2020 - year of the rat*. [Online]. Available: https://www.threatfabric.com/blogs/2020_year_of_the_rat.html.

[20] 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/.

[21] *Goldoson android malware infects over 100 million google play store downloads*. [Online]. Available: https://thehackernews.com/2023/04/goldoson-android-malware-infects-over.html.

[22] *Cybercrime service bypasses android security to install malware*. [Online]. Available: https://www.bleepingcomputer.com/news/security/cybercrime-service-bypasses-android-security-to-install-malware/.

[23] *Android debug bridge (adb)*. [Online]. Available: https://developer.android.com/tools/adb.

[24] 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.

[25] *Package manager*. [Online]. Available: https://developer.android.com/reference/android/content/pm/PackageManager.

[26] *Appbrain - everything you need for a successful android app*. [Online]. Available: https://www.appbrain.com/.

[27] *Sensortower, actionable intelligence for the digital world*. [Online]. Available: https://app.sensortower.com/.

[28] *Accessibilitynodeinfo*. [Online]. Available: https://developer.android.com/reference/android/view/accessibility/AccessibilityNodeInfo.

[29] 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.

[30] *Pixstealer malware detected in banking sector*. [Online]. Available: https://www.cybertalk.org/2021/09/30/pixstealer-malware-detected-in-banking-sector/.

[31] 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.

[32] 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/.

[33] 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.

[34] 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/.

[35] 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/.

[36] G. Shinde, *Malware on google play abusing accessibility service*. [Online]. Available: https://www.zscaler.com/blogs/security-research/malware-google-play-abusing-accessibility-service.

[37] R. Suau, *Analysis of a malware exploiting android accessibility services*. [Online]. Available: https://blog.pradeo.com/accessibility-services-mobile-analysis-malware.

[38] Y. Amit, *Inside the android accessibility clickjacking malware*. [Online]. Available: https://techbeacon.com/app-dev-testing/inside-android-accessibility-clickjacking-malware.

[39] T. Meskauskas, *How to remove the cerberus banking trojan*. [Online]. Available: https://www.pcrisk.com/removal-guides/17387-cerberus-banking-trojan-android.

[40] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, 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.

[41] *Edxposed: Elder driver xposed framework*. [Online]. Available: https://github.com/ElderDrivers/EdXposed.

[42] 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.

[43] Skylot, *Skylot/jadx: Dex to java decompiler*. [Online]. Available: https://github.com/skylot/jadx.

[44] O. Alrawi, C. Lever, K. Valakuzhy, R. Court, K. Snow, F. Monrose, 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] *Google play protect on-device protections*. [Online]. Available: https://developers.google.com/android/play-protect/client-protections.

[50] *Accessibilityserviceinfo: Isaccessibilitytool()*. [Online]. Available: https://developer.android.com/reference/android/accessibilityservice/AccessibilityServiceInfo#isAccessibilityTool().

[51] *Malware on google play abusing accessibility service*. [Online]. Available: https://www.zscaler.com/blogs/security-research/malware-google-play-abusing-accessibility-service.

[52] *Accessibilitymanager: Getenabledaccessibilityservicelist(int)*. [Online]. Available: https://developer.android.com/reference/android/view/accessibility/AccessibilityManager#getEnabledAccessibilityServiceList(int).

[53] *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.

[54] *View.accessibilitydelegate*. [Online]. Available: https://developer.android.com/reference/android/view/View.AccessibilityDelegate.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

[89] 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.

# A Prerequisites for Analysis

DVa initiates the investigation by pulling a list of registered a11y services from the user's device using the *a11y manager* API and matching it with the identified malware's a11y service name. For each identified a11y 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 our extraction, we encountered multiple families of malware with anti-static and anti-dynamic techniques to thwart analysis. Here, we briefly describe strategies DVa adopts to bypass them.

## A.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.

## A.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. We observed multiple techniques such as emulator detection and side-channel inference on dynamic analysis framework artifacts used to hinder analysis. We reverse-engineered a11y 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.