Rotten Apples Spoil the Bunch: An Anatomy of Google Play Malware

Michael Cao*
Univ. of British Columbia, Canada
michaelcao@ece.ubc.ca

Khaled Ahmed*
Univ. of British Columbia, Canada
khaledea@ece.ubc.ca

Julia Rubin
Univ. of British Columbia, Canada
mjulia@ece.ubc.ca

ABSTRACT

This paper provides an in-depth analysis of Android malware that bypassed the strictest defenses of the Google Play application store and penetrated the official Android market between January 2016 and July 2021. We systematically identified 1,238 such malicious applications, grouped them into 134 families, and manually analyzed one application from 105 distinct families. During our manual analysis, we identified malicious payloads the applications execute, conditions guarding execution of the payloads, hiding techniques applications employ to evade detection by the user, and other implementation-level properties relevant for automated malware detection. As most applications in our dataset contain multiple payloads, each triggered via its own complex activation logic, we also contribute a graph-based representation showing activation paths for all application payloads in form of a control- and data-flow graph. Furthermore, we discuss the capabilities of existing malware detection tools, put them in context of the properties observed in the analyzed malware, and identify gaps and future research directions. We believe that our detailed analysis of the recent, evasive malware will be of interest to researchers and practitioners and will help further improve malware detection tools.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → **Software reverse engineering**; **Software security engineering**.

KEYWORDS

Android, malware, dataset, malware detection, manual analysis

ACM Reference Format:
Michael Cao, Khaled Ahmed, and Julia Rubin. 2022. Rotten Apples Spoil the Bunch: An Anatomy of Google Play Malware. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510161

1 INTRODUCTION

The popularity of mobile phones has increased rapidly in the past decade. Together with the increased popularity, their wide adoption greatly stimulated the growth of mobile malware. Understanding the characteristics of such malware is an important step towards building effective malware detection tools.

While a number of Android malware datasets were collected in recent years, e.g., [30, 32, 36, 61, 65, 93, 97, 108], only a few authors performed a detailed manual analysis to identify and describe the mechanisms employed by malicious applications (a.k.a. apps). The most prominent of these works are the Malware Genome [108] and AMD [97] projects. In the former, the authors collected and manually analyzed 1,260 Android malware apps from official and alternative markets, focusing on identifying payloads, installation mechanisms, and activation conditions for malicious behavior. Later, the authors of AMD performed a similar manual analysis on 405 apps from 135 distinct malware families. These two annotated datasets, collectively, cover the time period between 2010 and 2016. Yet, to the best of our knowledge, no such detailed analysis was conducted since then for newer malware apps.

To address this gap, we collect and manually analyze a set of difficult to detect malware apps that penetrated the official Android Google Play store between January 2016 and July 2021. The Google Play store arguably has the most advanced defenses for Android apps, considering the low amount of malware that it contains compared with alternative markets [24, 109]. We thus believe that analyzing the mechanisms which allowed this malware to avoid detection by the store is a valuable direction which will help further improve both academic and industrial malware detection tools.

To collect malware apps, we systematically analyzed threat reports of the 19 most prominent mobile security companies according to Gartner [53], identifying reports that describe malware which penetrated the Google Play app store during our target dates. We identified 1,238 apps, grouped them into 134 families, and analyzed one app per family from 105 distinct families; we could not analyze apps from the remaining families due to obfuscation, encryption, etc. We refer to the collected dataset as *Google Play (GP) malware*.

Our malware analysis focused on collecting detailed information about malware implementation strategies and mechanisms it uses to avoid detection. In addition, as malware often has multiple payloads, each triggered under (potentially multiple) complex activation conditions, e.g., that rely on combination of sensor data and random events, we defined a *flow-based malware signature*: a graph-based representation showing activation paths and payloads for an app in a form of a control- and data-flow graph. We then manually generated flow-based signatures for the analyzed apps. We believe such signatures are useful to gain a proper understanding of malicious behaviors, which is required for building effective detection tools, e.g., those based on dynamic analysis [75, 90, 98] or static execution-path exploration [101, 102].

To present our findings, we organize malware analysis results around two main research questions:

*Equal contribution

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510161

RQ1 (Malware Characteristics): What are the characteristics of the GP malware and how does it compare with prior findings?

RQ2 (Malware Flow Behaviors): How complex are the control- and data-flow behaviors implemented by the GP malware?

We then discuss our malware analysis results w.r.t. the capabilities of existing malware detection tools and identify possible gaps and future research directions.

Our analysis identifies new payloads not discussed in prior work, such as clipboard hijacking and cryptomining. Interestingly, some of these payloads do not rely on any platform-specific APIs, typically used as an "anchor" by many malware detection approaches. For example, cryptomining can be performed by using simple mathematical operations only. We also identify new malware activation conditions, e.g., checking for device temperature (to decide when it is appropriate to mine cryptocurrency), and sophisticated combinations of multiple events and conditions required for a payload to execute, e.g., sensitive information is retrieved when a phone call is received, stored in a file, and then released on a completely different event, e.g., when another app is installed. Triggering a subset of these events, or even all events but in a different order, will prevent dynamic analysis tools from detecting the malicious behavior. Finally, our analysis shows that some payloads spawn bytecode, JavaScript, and native code, e.g., obtain data in a WebView while leaking it via an Android Java API. Existing tools need to be augmented to track such cross-technology patterns.

Contributions. Our work identifies characteristics and precise flow-based signatures of contemporary malware. It can be used to inform software engineering and security communities who develop efficient malware detection tools and also as a benchmark for evaluating such tools. More specifically, this paper contributes:

1. A systematically collected set of 1,238 Android malware apps from 134 distinct malware families, which bypassed Google Play defenses between January 2016 and July 2021. To the best of our knowledge, this is the first confirmed dataset of such malware.

2. Detailed reports from the manual analysis of one sample from each of the 105 distinct families, as well as aggregated findings and a comparison of malware characteristics with those from earlier years [97, 108], to identify new and obsolete practices (Section 3).

3. Flow-based signatures for each analyzed malware sample, which include per-flow conditions that guard each payload and the location/language of each flow element (Section 4).

4. A categorization and analysis of existing malware detection tool capabilities, and a discussion on gaps and future research directions in context of our collected malware dataset (Section 5).

Data Availability. We responsibly share the collected dataset and our analysis results in an online appendix [35]; the latest citable release can also be found online [34].

2 METHODOLOGY

In this section, we describe our methods for building the dataset and analyzing the malicious samples.

2.1 Building the Dataset

Typically, malware apps are collected by either browsing blogs of security companies [97, 108] or by screening apps found in public repositories [93]. Such screening relies on antivirus scanners, e.g., from the VirusTotal [92] online service, which offers more than 70 different scanners. In our work, we chose the former approach, because blogs of security companies already provide useful (albeit partial) information about malware characteristics, which can assist our manual analysis and help ensure the results are accurate.

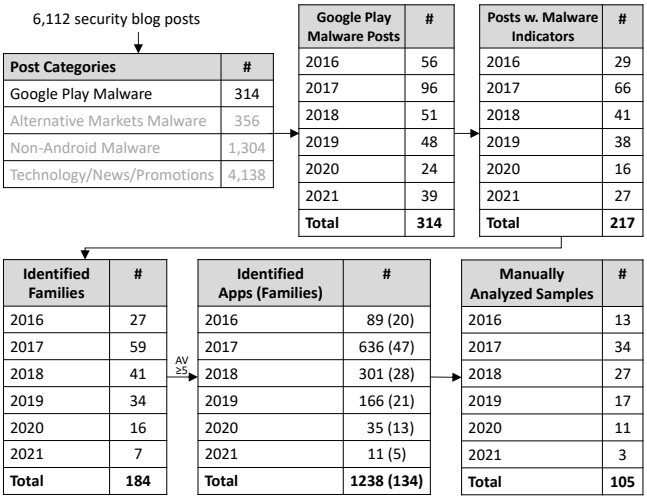

Figure 1: Malware collection process

We started from a list of 21 security companies reported on Gartner's Magic Quadrant for Best Endpoint Security Platforms [53], identifying, for each company, blog sites related to cyber-threat intelligence; we found such sites for 19 companies. For each site, we extracted blog posts describing Android Google Play malware using the following search terms: ("Android" | "Google" | "Playstore" | "Play" | "Store") and ("malware" | "malicious" | "malice"). We limit our search to the time period between January 2016 and July 2021, as we are interested in augmenting the body of knowledge collected in prior studies, which looked at earlier periods. To make our search approach repeatable and reproducible, we implemented an automated web crawler, which is available online, together with the full list of companies and their blogs sites [35].

Running our crawler in July 2021 resulted in 6,377 blog posts, out of which we discarded 265 where the main part of the text was not written in English. Two authors of the paper then independently read the remaining 6,112 posts and classified each post based on its content. We arrived at four categories of posts: Google Play Malware, Alternative Markets Malware, Non-Android Malware, and Technology/News/Promotions. The categorization lists were cross-validated and all disagreements (on 68 posts, 1.07%) were resolved through a discussion with all authors of this paper. The number of blog posts for each category is shown in our schematic representation of the data collection process in Figure 1.

We further focused only on posts from the first category: 314 posts describing malware from the Google Play store. The distribution of Google Play malware blog posts per year is described in Figure 1. For each post, we looked for indicators of apps described in the post: (1) file hash (SHA-1/SHA-256/MD5), which is a unique identifier of an app, (2) app package name, which is the name identifier of an application, and (3) author and app name, which is the information presented to the users. We discarded 97 posts containing none of these indicators.

Analyzing the remaining 217 posts, we observed that a post can describe apps from one or more malware families, e.g., an ESET post [42] describes three families: apps that mine cryptocurrency after a preset period of time, apps that phish for cryptocurrency credentials, and apps that fake cryptocurrency mining functionality to aggressively push ads onto the user. It is also possible that families described by two different posts refer to malware from the same family, i.e., by posts referencing each other directly or describing indicators pointing to the same apps. We identified 235 malware families in total, out of which 51 were duplicates, resulting in 184 unique families. Blog posts provided the names for families in a majority of cases (95%). For the remaining cases, we used AVClass [79] and assigned names based on AVClass labels. The distribution of posts with malware indicators and malware families per year is given in Figure 1.

Next, we collected apps for each of the identified families by scanning the VirusTotal Academic [92], Virus Share [81], and Contagio [70] malware repositories, Android alternative markets, such as APKMonk [28] and APKPure [29], and AndroZoo [24] – a popular repository with over 9.5 million Android apps from various markets. Our assumption was that these repositories may still contain the app even though it was already removed from the official store. We prioritized apps found directly by the file hash. If we could not find such apps, we searched using the app package name, app name, and author names, and prioritized apps having a file hash identical to the one reported by the blog post. For cases when the post did not provide the hash, we uploaded the identified app to VirusTotal and ensured the app was flagged as malicious by the same antivirus company that published the blog post and in the same family as reported in the post. If we could not verify the app, we marked it as not found.

At the end of this process, we collected a total of 1294 identifiable apps from 180 blog posts. We further exclude apps marked by fewer than 5 antivirus tools, to ensure our collected dataset is reliable [32]. This eliminated 56 apps (4%), resulting in a dataset of 1,238 from 134 unique families. The distribution of retrieved apps per year is also given in Figure 1.

2.2 Analyzing Malware Samples

We first selected 20 malware families and one sample app per family at random; two authors of this paper manually analyzed the selected samples to arrive at a common, reliable, and reproducible analysis methodology. The analysis reports were discussed and augmented by all authors of the paper in a number of joint meetings. The remaining families were analyzed by the two authors individually while regularly validating each other's findings. The summary reports were discussed with all the authors and augmented, when needed.

Similar to Wei et al. [97], we used both focused and exploratory analysis of sample apps. In focused analysis, we read the blog post to obtain a high-level description of the malware in plain human language, noticing all malicious behaviors described in the post. We then decompiled the app using JADX [83] to collect decoded resources, Java source code, and native binaries, and attempted to map the described behavior to Android implementation mechanisms. For example, if the post stated that the malware "subscribes to premium numbers", we looked for Android API calls to *android.telephony.SmsManager::sendTextMessage()*. Once we located the API in the app, we performed manual backwards reachability analysis to identify paths leading to the payload; we further analyzed all components found along the execution path.

We complemented focused analysis with an exploratory analysis to understand the overall workflow of the application and to find additional unexpected behaviors. We observed that malicious payloads are often triggered in response to system events, e.g., boot completed or phone unlocked, likely because malware developers attempt to avoid detection by moving the payload execution away from the app main execution path while still ensuring the payload is triggered. We thus thoroughly explored code triggered by system and user events, in addition to the app main launch activity.

Some malware apps utilize native code, web assets, and/or bytecode executables found in application resources. We treated these files as an extension of the application code, analyzing them together with the main code of the app. We used IDA disassembler [76] to reverse engineer native binaries and convert machine code to human-readable assembly code. We used IntelliJ IDE [54], which supports multiple programming languages, to perform additional analysis tasks, such as class hierarchy analysis.

Dealing with anti-analysis techniques. Surprisingly, only less than half of the malware in our dataset used renaming as an obfuscation technique (i.e., producing package, method, and variable names like *a.a* and *a.b*). While APIs of the Android framework methods, such as registering for an event or opening an HTTP connection, cannot be obfuscated, obfuscation of application-specific code complicates analysis. We used JADX's built-in deobfuscation functionality to map obfuscated names to more readable unique identifiers, which simplified the analysis of the code.

Some of the malware apps used off-the-shelf and custom mechanisms to encode and later decode Strings comprising reflective calls and network addresses. We followed the logic implemented in the source code of the apps and re-implemented the routine to decrypt the data outside of the application. In a few cases where we could not fully understand the particular routine, we instrumented the application to print out the decrypted Strings.

In addition, several apps used Android packers, which encrypt DEX files using ELF binaries [64] and decrypt them at runtime, to increase the difficulty of reverse engineering the application [107]. In cases where malware apps used commercial packers, e.g., Jiagu [55] and Bangcle [99], we were able to decrypt DEX files of these apps using Android Unpacker [85] and DrizzleDumper [38].

Furthermore, some apps used encryption/decryption mechanisms to obfuscate Strings, asset files, and code files loaded from local storage. In the majority of the cases, we could identify the decryption key, which was simply stored in the APK file.

Dependence on external code/data. Several apps hide malicious functionality in code downloaded from external sources, e.g., the developer's backend servers. Once the malware is discovered, Internet Service Providers shut down its backend server permanently. As we could not access these servers and analyze the malicious behavior of the downloaded code, we borrowed the description of its payloads from the corresponding blog post. We explicitly marked (with ☆) such payloads, to differentiate them from those we

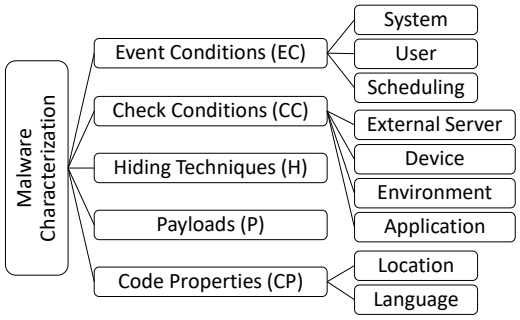

Figure 2: Malware characterization schema

have observed independently. For 16 apps, the description of all app payloads relied on the blog posts; in 24 additional cases, part of the payloads was observed independently and part was inferred from posts. For another 16 cases, no adequate description of a payload was given in the post and we marked it as *unknown*.

Analysis Report. Overall, we were able to successfully analyze at least one app from 105 distinct malware families. We excluded four families where we could not identify the malicious behavior described in the blog post, nine families whose apps used a commercial packer we could not unpack, nine families whose apps were heavily obfuscated and we could not reliably analyze, five families whose apps relied on reflective calls we could not track, and two families whose apps contained strings that were encrypted with a custom encryption key that was dynamically downloaded from the app backend server and was no longer functional at the time of our analysis. The detailed breakdown of the apps we analyzed per year is given in Figure 1.

At the end of the analysis, we produced a detailed report for each app, describing (a) app meta-information, including the file hash, blog URL, and a brief description of the malware's behavior, (b) the relevant source code related to the execution path(s) required to activate the payload(s), (c) a tabular-form characterization of the malware app, as described in Section 3, and (d) a flow-based signature of the app, as described in Section 4.

3 MALWARE CHARACTERIZATION

We augment and refine malware categorization schemas used in previous work [97, 108], proposing the categorization in Figure 2. In what follows, we first describe our schema using the example in Figure 3 for illustration purposes. We then use the categorization to discuss the malware in our dataset.

3.1 Malware Characterization Schema

Figure 3 shows a simplified version of a malware app from the SpyBankerHU family [41]. In a nutshell, this malware intercepts SMS messages and steals the user's banking credentials. It is activated when the user unlocks the phone or receives an SMS message. Both events are handled by the *onReceive(...)* method of the *UnlockSMSRecver* component (lines 3-32 in Figure 3). Using the *Intent* parameter, the method checks whether an SMS was received (lines 4-22). If so, the method checks if the field *stealSMS* (the name that we gave to this variable for illustration purposes) is set to *true*, to determine whether the SMS stealing functionality is activated (line

```
1 class UnlockSMSRecver extends BroadcastReceiver {
2 static boolean stealSms = false;
3 void onReceive(Context ctx, Intent in) {
4  if (in.getAction().contains("SMS_RECEIVED")){
5   if (stealSms){
6    ContentResolver cr = ctx.getContentResolver();
7    SmsMessage sms = SmsMessage.createFromPdu(
8            in.getExtra().get("pdus")[0]);
9    Cursor c = cr.query(Uri.parse("content://sms"),
10           new String[]{"_id", "body"});
11   while (c.moveToNext()) {
12    int id = c.getInt(0);
13    String body = c.getString(1);
14    if (body.equals(sms.getMessageBody())) {
15     cr.delete(Uri.parse("content://sms/"+id));
16    }
17   }
18   String device = context.getDeviceId();
19   Connection con = new URL("leaksms.com/?u="+
        device).open();
20   con.write(sms);
21  }
22 }
23 String device = context.getDeviceId();
24 Connection con = new URL("commands.com/?u="+
      device).open();
25 String commands = con.read();
26 if (commands.contains("steal_sms")){
27  stealSms = true;
28 }
29 String running = getRunningProcesses();
30 if (running.contains("com.garanti.cepbank")) {
31  WebView.loadUrl("fakebank.com");
32 }
33 }}
```

Figure 3: SpyBankerHU malware

5). The method then retrieves all SMS messages on the device (lines 6-10) and iterates over them to identify and delete the received SMS (lines 11-17). It further leaks the SMS content to the developer's server, together with the device id (lines 18-20). This is most likely done to steal the one-time access passcode the bank might send to the user and to hide from the user messages which could indicate that someone else (i.e., the malicious party) is manipulating their bank account.

In any event – whether an SMS is received or the user unlocks their phone, the app also attempts to communicate with the command and control server and to steal banking credentials (lines 23-33). Specifically, the app retrieves the device id (line 23) and contacts its server to retrieve a set of commands (lines 24-25). If it receives the "steal_sms" command, the app sets the field *stealSMS* to *true* (lines 26-28), indicating that the user is going to be attacked and enabling the SMS leaking behavior described above. The app further checks if a process associated with the cepbank banking application is running (lines 29-30) and, if so, loads an HTML page that impersonates the login screen of the banking app (line 31). An unsuspecting user will enter their banking login credentials on the fake overlay, which sends the credentials to the malicious server.

In summary, the app contains three different payloads: it leaks the device id when an SMS is received or the phone is unlocked (lines 18-20 and 23-24); it leaks the content and deletes SMS messages when instructed by the server (lines 14-20); and it steals banking credential when the cepbank application is running (lines 29-32). Each of the payloads is triggered under different conditions

and activation events, part of which we omitted for the simplicity of the presentation. We also omitted yet another payload: ad abuse, as well as additional events that the app intercepts to trigger the malicious behaviors, e.g., power change.

To accurately describe the behavior of such malware, we use five categories specified in Figure 2:

1. **Event Conditions (EC)** are events that an app intercepts and that trigger the activation of a malicious path. We further divide them into *System* events, which are triggered by the Android system when the device is booted, an SMS is received, etc.; *User* events, which are triggered by the app user, when logging in into the app, pressing a particular button, copying text, etc.; and *Scheduling* events, which are triggered at specific time intervals. Our SpyBankerHU sample in Figure 3 intercepts SMS- and phone unlocking system events.

2. **Check Conditions (CC)** are conditions that need to be satisfied for the malicious behavior to fire. These conditions can depend on an *External server*, e.g., when the malicious payload is executed only when a particular command is received from the app backend, like in the SpyBankerHU sample in Figure 3. Further, check conditions can be *Device*-dependent, i.e., rely on a certain hardware or software specification; *Environment*-dependent, e.g., be executed at a certain time or location; and *Application*-dependent, e.g., rely on a certain permission granted to the application or on a certain data format.

3. **Hiding Techniques (H)** are used by the app to hide its malicious actions from the user. These include removing the app icon, so the user cannot identify and uninstall the app, and blocking information, such as deleting SMS messages in the SpyBankerHU sample.

4. **Payloads (P)** describe the main malicious functionality of the app itself. We identified a variety of payloads, including stealing of personal, device, and banking information, like in the SpyBankerHU sample in Figure 3.

5. **Code Properties (CP)** are the code-level details describing how these behaviors are implemented. We consider the *Location* of the code, e.g., being in the application directly, hidden in resources, or downloaded from a remote server and the *Language* in which the malicious code is implemented, e.g., bytecode, native code, or a web-based language (HTML/JavaScript). For the SpyBankerHU app in Figure 3, the *UnlockSMSRecver* class is implemented directly in bytecode, while the fake login overlay is implemented in JavaScript and is loaded from a remote server.

Next, we outline, for each of the categories, the implementation strategies that malware in our dataset employs. Section 4 further describes our malware signature, which accurately relates the events, conditions, hiding techniques, and payloads to each other, in a form of a control- and data-flow graph.

3.2 Malware Characteristics

Figure 4 shows the specific behaviors we observed for each of the categories in our schema. We mark with an arrow (→) categories that were introduced based on information observed in our dataset, compared with prior work [97, 108]. We also provide the number of analyzed sample apps exhibiting each of the behaviors. As one app can contain multiple (or no) event and check conditions, payloads, etc., it can be counted in multiple categories; thus, the number of apps in a category does not sum up to 105. A detailed description of each analyzed sample app is available in our online appendix [35].

Event conditions	#	%
System	80	76.2
Boot status	44	41.9
Device status	30	28.6
Network status	26	24.8
→ Developer-defined	26	24.8
Package changes	22	21.0
→ Service bind	11	10.5
SMS delivery	11	10.5
Battery status	9	8.6
Call status	8	7.6
USB status	1	1.0
User	95	90.5
Application launch	92	87.6
→ Button click	46	43.8
→ Sensitive input	28	26.7
→ Permissions	25	23.8
→ App install	15	14.3
→ Clipboard text	1	1.0
Scheduling	58	55.2
Scheduling	58	55.2

(a)

Check conditions	#	%
External server	79	75.2
Internet	78	74.3
SMS	2	1.9
Device	82	78.1
Software specs	62	59.0
Network	43	41.0
→ Hardware specs	28	26.7
→ Sensors	4	3.8
Environment	41	39.0
Time	38	36.2
Location	6	5.7
Application	42	40.0
Permissions	31	29.5
Data format	10	9.5
→ Probability	8	7.6
Version	1	1.0

(b)

Hiding techniques	#	%
Rich functionality	65	61.9
Icon manipulation	34	32.4
Device admin	15	14.3
Information blocking	12	11.4
→ Self-uninstallation	6	5.7
→ Automated gesture input	5	4.8
Screen locking	3	2.9

(c)

Payloads	#	%
Information stealing	69	65.7
Ad abuse	54	51.4
Premium charges	10	9.5
→ Cryptomining	5	4.8
Root exploit	4	3.8
→ Clipboard hijacking	3	2.9
→ Port forwarding	3	2.9
Ransom	1	1.0
Unknown	16	15.2

(d)

Code properties	#	%
Location		
Direct	97	92.4
→ Downloaded (Remote)	60	57.1
→ Hidden (Resources)	25	23.8
Language		
Bytecode	102	97.1
→ Web	29	27.6
Native	11	10.5

(e)

→ indicates categories identified in our work compared with prior studies [97, 108]

Figure 4: Characteristics of Google Play malware

1. **Event Conditions** (Figure 4a) have three sub-categories:

System Events. The majority of the analyzed samples are triggered by at least one *System* event, most commonly when the phone is booted or when the device status changes, e.g., from locked to unlocked. Similar system events include network, battery, call status, and SMS delivery events. We believe malware uses system events to avoid executing the malicious payload on the (easier to trigger) main application launch while still ensuring the malicious functionality is eventually triggered. Interestingly, most system events we observed are less likely to happen on the emulator, e.g., lock status change, which further complicates the detection of these apps at testing time.

We observed two new types of system events not discussed in earlier reports: *Developer-defined* are custom events not defined by the Android platform. For example, Solid (ID: 94) is activated by the reception of a Firebase message [50]. Considering such custom event types could challenge dynamic-analysis-based tools. *Service bind* events, which allow the app to communicate with bound services (e.g., accessibility), were described in the literature [47], but we believe we are the first to observe them in practice. For example, SpyBankerAJZ (ID: 98) abuses the accessibility service, which has access to user interactions with other apps, to activate itself when the user interacts with a banking app and overlays a phishing browser to steal banking credentials.

User Events. While almost 90% of the samples activate a payload once the user launches the app, we identified additional user events

triggering malicious behavior, such as inputting specific data or copying text (see Figure 4a for details). For example, ClipperC (ID: 28) triggers when the user copies text and checks if the text is formatted as a wallet address. If so, the app executes its payload.

Scheduling Events. Around half of the samples periodically repeat the malicious behavior. For example, HidenAd (ID: 67) schedules an alarm that displays full-screen ads every 15 minutes, even while the user is interacting with other apps.

Comparison with earlier reports. Our analysis identified two types of system events, developer-defined and service bind, not discussed in earlier reports [97, 108]. We also provided a more detailed characterization of user events according to their types.

2. **Check Conditions** (Figure 4b) have four sub-categories:

External Server. Three quarters of the analyzed sample apps execute malicious payloads on receiving particular data from an *external server* – either through the Internet or by an SMS (a.k.a. command and control or C&C malware). Most frequently, the commands are retrieved over the Internet. In two cases, the apps receive an SMS command to enable intercepting and leaking SMS messages and calling premium numbers, e.g., AceCard (ID: 1).

Device. Three quarters of the samples also condition payloads on *software* specs of the underlying Android system, such as system version and which apps are installed. We identified several checks not described in earlier reports: *hardware specifications*, i.e., CPU and network operator, and *sensor* data, i.e., battery, temperature, and motion. For example, Aladdin (ID: 8) checks that the device contains more than 3 CPU cores before generating ad traffic to an invisible web page; Vilny (ID: 108) checks the phone temperature and battery level before mining cryptocurrency, presumably to not alarm the user by a sharp change of these parameters.

Environment. These checks, commonly known as time and logic bombs, activate malicious payloads only at a certain *time* and/or *location*. For example, PletorD (ID: 80) checks the user's location to ensure that the user is not from Russia or Ukraine, before locking the phone in silent mode and asking for a ransom.

Application. Application-specific checks include *permissions*, *data format*, and *app version*. For example, SMSAndroidOSWesp (ID: 91) uses regular expressions to check that the received SMS contains a PIN, before leaking it to the malware developer's server. Moreover, we identified a number of samples that decide whether to activate the malicious payload *probabilistically*. For example, HiddenAdHRXH (ID: 64) uses the *java.util.Random* library to roll a number between one and 100 and automatically click on ads only if the obtained value is less than 25.

Comparison with earlier reports. Previous work reported conditions based on commands retrieved from an external server, installed apps, time, and location. Our detailed categorization reveals new checks: hardware specifications, sensors, and random numbers.

3. **Hiding Techniques** (Figure 4c). The most prominent technique in our dataset is the inclusion of *rich benign functionality*, which is employed by more than half of our samples. This is done to convince the user that the app is legitimate, e.g., by presenting a weather forecast (ID: 99) or providing a YouTube-like video interface (ID: 2). Unlike repackaging, these are unique apps and we are not aware of other reports that explicitly identified such behavior.

Another frequent hiding technique is *icon manipulation*. In addition to the previously observed case of hiding the app icon, samples in our dataset also change the icon, pretending to be another app. For example, Reputation1 (ID: 86) changes the icon and name to resemble Google Maps. Moreover, when the user clicks on the icon, the application redirects to Google Maps. Meanwhile, it periodically pushes full-screen ads to the user without the user being aware of the origin of these ads.

Obtaining *device admin privileges* can make the malware harder to install. It can also allow the malware to perform privileged operations, such as changing the lock-screen password or locking the screen, to further increase the difficulty of uninstallation. For example, SpyBankerHU (ID: 99) locks the screen when the user attempts to disable its admin privileges. Samples also employ *information blocking*, i.e., hiding information produced by the payload from the user, and *screen locking*, to prevent the user from noticing suspicious behaviors on the device.

We also observed a new hiding approach via *self-uninstallation*. For example, BanBraA (ID: 19) can receive commands from the malware developer's server to uninstall itself. This confuses the user about which app performed the malicious behavior and reduces the chances of the app receiving a bad review. Yet another new technique is abusing accessibility services to perform *automated gesture input* to prevent the user from performing certain actions. For example, SpyBankerAJZ (ID: 98) presses the back button whenever the user opens an antivirus.

Comparison with earlier reports. Similar to Malware Genome and AMD, we observed icon hiding, information blocking, device admin privileges, and device screen locking hiding techniques. We also observed new trends: implementing legitimate benign functionality, changing the app icon, self-uninstalling, and automated gesturing. In addition, the AMD report described techniques which did not occur in our dataset: cleaning system logs, killing antiviruses, removing the app from the device administrator list, and preventing uninstallation of the app.

4. **Payloads** (Figure 4d) are divided into nine sub-categories:

Information Stealing. The most frequent payload in our dataset is *information stealing*, e.g., of SMS, accounts, phone numbers, contacts, and stored files. Some malware also explicitly tricks the user into entering sensitive information. For example, Reputation1 (ID: 86) presents a web page claiming that the user won a prize. To "redeem" the prize, the web page provides a survey that steals sensitive information from the user.

Ad Abuse. Samples with *ad abuse* are also frequent in our dataset. We identified three ad abuse patterns: (a) Aggressive advertisement forces advertisements when the user performs a certain action or periodically. For example, FraudApp (ID: 51) acts as a fake cryptocurrency miner while pushing full-screen ads. (b) Hidden ads loads web pages that contain ads to gain additional revenue. For example, Aladdin (ID: 8) opens an invisible *WebView* to increase the number of visits to the malicious developer's website that contains ads. (c) Ad-click fraud automate clicks on ads. For example, AdClickerBN (ID: 2) retrieves a list of targeted websites and a JavaScript payload from a C&C server and performs ad clicks.

Premium Charge. Apps also impose *premium charges* on the user by subscribing them to premium services without user consent, e.g.,

SMSAndroidOSWesp (ID: 90) and Sonyvpay (ID: 94). To perform the subscription, the malicious app obtains up-to-date information about the service, such as its subscription number, subscription message, and subscription account. It then sends a request to the service over SMS, call, or internet. A confirmation message is sent back to the device over SMS and is intercepted and confirmed by the malware.

Cryptomining. Several samples commence *cryptocurrency mining* without the user's consent, e.g., CoinMinerQ (ID: 30) and CPUMiner (ID: 31). To perform cryptomining, a miner receives a hash puzzle and its result; they need to find the puzzle input value that leads to the given result. The first miner to do so is rewarded with cryptocurrency. To increase the chances of getting the monetary reward, malicious developers leverage mobile devices of many unsuspecting users, synchronizing between the users via a configuration obtained from a remote server. Interestingly, while mining cryptocurrency is a popular monetization tactic, earlier studies did not identify such apps. This could be because of the sharp rise in bitcoin prices in 2017 [9], which falls outside of the date range of earlier reports.

Root Exploit. Malware exploits vulnerabilities in the Android operating system to grant the app unrestricted access to the device's memory space. Root exploits are typically performed in native code, as bytecode runs in a virtual machine that is isolated from the operating system. To perform root exploits, the malware retrieves native binaries stored locally or over the internet. It then executes the binaries and performs unauthorized actions, such as silently installing other malicious apps. For example, Godless (ID: 53) executes root exploits then downloads and installs an app with system privileges to perform additional payloads.

Clipboard Hijacking. Yet another technique not reported in previous studies is a *clipboard hijacking* attack, when the malware changes the text clipboard so that the user pastes unintended text. For example, the ClipperC malware described earlier (ID: 28) uses this mechanism to replace a wallet address copied by the user with the malware developer's wallet address.

Port Forwarding. Our samples also perform *port forwarding*, a special form of information stealing where malware attempts to steal data from the user's internal network. To this end, malware creates an outbound connection to the malicious developer's server, retrieves a target address and network command, forwards the network command to the internal network to collect data, and relays the response back to the developer's server. For example, MilkyDoor (ID: 77) tunnels using SSH into the user's internal network to steal enterprise data.

Ransom. For monetization, some apps ask for a *ransom*, where the malware encrypts data or disables device functionalities to coerce the victim into paying a fee, e.g., using a bitcoin transaction, to release device functionalities that are being held captive. For example, PletorD (ID: 80) locks the device into silent mode and opens a *WebView* to ask for a ransom to unlock it.

Unknown Payload. There are 16 samples in our dataset that download code dynamically from a remote server. Such code might contain malicious behavior, adding to other payloads we observed in these apps. As we could not confirm the malicious behavior of the downloaded code, we mark these payloads with *unknown*.

Comparison with earlier reports. Prior work did not report on the cryptocurrency mining, clipboard hijacking, and port forwarding payloads, which were observed in our datasets. The AMD study reported the ad abuse payload, but did not elaborate on the different types of ad abuse that we found in our study: aggressive advertisement, hidden ads, and ad-click fraud.

5. **Code Properties** (Figure 4e) are divided into two sub-categories:

Location. Check conditions, hiding techniques, and payloads can be implemented *directly* in the APK .dex file, *downloaded (remote)* from a server, or *hidden (resource)*, where the code is packed in the app's resources. For example, HiddadBZ (ID: 61) hides its payload in a text font file, which is converted to APK and executed at runtime.

Interestingly, around half of the analyzed samples download code from a third-party server. The samples typically use *DexClassLoader* to load the downloaded bytecode and invoke it via reflection. Two samples, DroidPlugin (ID: 34) and AsiaHitGroup (ID: 15), abuse the virtual installation technology [67], originally designed to install multiple copies of the same app, to silently install malicious apps. Other samples attempt to install the downloaded APKs directly, using social engineering to convince the user to allow installation. For example, AnubisDropper (ID: 12) claims that the system is out of date and the user should install a secondary app to "update" it.

Language. Check conditions, hiding techniques, and payload can also be implemented in different languages: Java *bytecode*, *Web* (JavaScript), or in *native* code (C/C++). We found that while most of the analyzed samples implement the malicious functionality in Java bytecode, some malicious behaviors, mostly related to ad abuse, are implemented in JavaScript code loaded into *WebViews*. Finally, a few samples also employ native code.

Comparison with earlier reports. The analysis performed in earlier work was more coarse-grained and only considered the location of the payload. Instead, our work maps each of the malicious activities – check conditions, hiding techniques, and payloads – based on their code/location. The detailed map is in our online appendix [35].

To answer RQ1, we observed that a high fraction of the analyzed samples implement information stealing and ad abuse. We also identified new payloads: cryptomining, clipboard hijacking, and port forwarding. Malware shifts its payloads off the main execution path and hides them under a number of difficult-to-trigger checks, complicating the detection of these samples. Furthermore, samples also split their malicious paths across multiple languages. Finally, we observed that malware employs new hiding techniques, such as self-uninstallation and automated gesturing.

4 FLOW-BASED MALWARE SIGNATURE

The malware categorization in Section 3 provides information about event and check conditions, hiding mechanisms, and payloads employed by current malware; understanding how these actions are combined together is an essential step towards building efficient tools for detecting such malware. For the SpyBankerHU sample in Figure 3, presenting web-based content in a *WebView* is, by itself, a legitimate application behavior, common to many benign apps. However, this behavior is executed only when a certain app, i.e., cepbank, is installed on the device (lines 29-32). Discovering such contextual information can help a technique deem this sample malicious. Likewise, understanding the interplay between deleting SMS

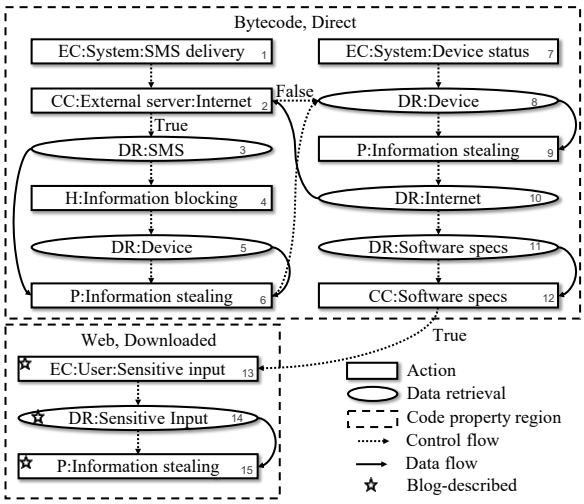

Figure 5: Flow-based malware signature of SpyBankerHU

messages and the condition received from the server which enables this behavior (lines 25-28 and 5-17) can help distinguish between benign apps that manage users' SMS messages and apps that have a malicious intent. Understanding the precise order of events and conditions that activate a malicious payload, as well as implementation mechanisms, such as information flows between bytecode and JavaScript code in WebViews, and code-level obfuscation techniques, such as String manipulations and app- and phone-specific content management code (lines 5-17 in Figure 3) is also essential to build efficient techniques that can detect realistic malware.

To capture such contextual and path-level information, we manually created a *flow-based malware signature* graph for each of the analyzed samples. These signatures go beyond the categorization in Section 3; they rather capture control- and data-flow dependencies for paths related to each malicious behavior. They also capture information about how each malicious component is implemented, including implementation location and code language.

Figure 5 shows an example of such a graph for SpyBankerHU. The nodes of the graph are the four actions from the malware categorization schema in Figure 2: *Event Conditions (EC)*, *Check Conditions (CC)*, *Hiding Techniques (H)*, and *Payload (P)*. These nodes are shown as rectangles with the corresponding event type in their names. We add one additional type of node, which we call *data retrieval (DR)*; it is shown as an ellipse in Figure 5. Each such node corresponds to the action of retrieving data, e.g., from an SMS message, the Internet, or user-sensitive input. The retrieved data is used either in evaluating the check condition or in a payload.

The graph contains two types of edges: control-flow (dotted lines) and data-flow (solid lines). Similar to the control- and data-flow concepts from the program analysis domain [45], control-flow edges in our graph describe the order in which events are executed; data-flow edges describe the flow of information from data retrieval to either check condition or payload nodes. For the example in Figure 5, the *P:Information stealing* node (node #6 in the figure) is executed after *DR:SMS* (node #3), *H:Information blocking* (node #4), and *DR:Device* (node #5); the data-flow edges show that the *P:Information stealing* node leaks SMS and device-related information.

Another important detail captured by the graph is that the info stealing payload is triggered only after an SMS is received (node #1) and only if the condition *CC:External server:Internet* (node #2) evaluates to *True*. The condition itself is dependent on the information obtained from the Internet (node #10), which is retrieved when the user unlocks the phone: *EC:System:Device status* (node #7).

Finally, we group nodes into *regions* capturing the code properties of the nodes – location and language. The regions are indicated by dashed lines in Figure 5. In this example, there are two regions: the first contains nodes corresponding to the bytecode directly loaded by the app from the .dex file (nodes #1-#12) and the second contains nodes corresponding to Web code (HTML/JavaScript) downloaded from the remote server (nodes #13-#15).

When we analyzed this app, the remote server was down and, thus, we have no access to the code downloaded from the server. We describe the behavior of that code based on the description in the blog post: upon the user entering their credentials (node #13), this info gets leaked to the malware developer's server (nodes #14 and #15). As discussed in Section 2.2, we mark nodes for which we followed the post description rather than observed the malicious payload directly with a star (☆).

A *path* in our graph is a sequence of nodes connected by control-flow edges. A path starts from a node with no incoming edge, which is an Event Condition (e.g., node #1 in Figure 5). A path ends with a node with no outgoing control-flow edges, which could be a payload, a data retrieval node, or a hiding technique node (e.g., node #15). The example in Figure 5 has three paths: P_1 is the path from node #1 to #15, where the condition in node #2 evaluates to *True*. P_2 is a (shorter) path from node #1 to #15 where the condition evaluates to *False*. Finally, P_3 is the path from node #7 to #15.

A path can include multiple event conditions. For example, P_3 requires the user to unlock the device (node #7) and provide sensitive input (node #13). Likewise, a path can include multiple check conditions, e.g., P_1 checks a server command (node #2) and installed apps (node #12). Finally, a path can have multiple hiding techniques and payloads, e.g., P_2 contains two information stealing payloads (nodes #9 and #15).

Signatures of GP malware. By analyzing the malware signatures of our samples, we make three main observations:

1) Overall, the samples have 14.9 paths on average (for this calculation, we excluded 5 outlier samples with more than 100 paths as the median number of paths is 9). Each path has multiple event and check conditions, hiding techniques, data retrieval nodes, and payloads. There are 2.8 event conditions and 4.8 check conditions per path, on average. The relatively high number of conditions per path shows that malware conditions the execution on specific external and internal settings (see Figure 4 for the list of conditions). For example, Vilny (ID: 108), which mines cryptocurrency on the device, checks a command from the server, phone temperature, battery level, and that the screen is off, all before starting the payload. Malware analysis tools that are based on dynamic and symbolic execution should account for all these conditions of different types to fully analyze / trigger the malware.

2) There are 5.3 data retrieval nodes per path, on average. The majority of these data retrievals are for validating the check conditions

and the rest are for retrieving sensitive data for the information leakage payloads. Interestingly, around 40% of data retrievals happen on a path different from the target check condition or payload. That is, the applications retrieve and store data in, say, a file, and then uses it on another path, triggered by a different event. We identified several cross-path data storage locations: global variables, files, and shared preferences. For example, Bahamut (ID: 18) records the call audio into a file when the call is started. Later, in a different event – on network change – it sends the stored file to a third-party server. This implies that malware analysis tools should be able to consider more than one path simultaneously, e.g., to trigger multiple paths during dynamic analysis and to analyze data flows across multiple paths during information flow analysis. Unfortunately, some of the existing tools, especially those based on formal methods and complex program analysis, currently only consider one callback at a time, e.g., [75, 90, 98]. Furthermore, tools should consider information flows through multiple data types given above, in addition to fields and local variables.

3) Samples have 3 payloads on average, each of which is reachable via 12.4 paths on average. Multiple paths lead to the same payload as malware typically triggers the payload from multiple events, to maximize the likelihood and frequency of the payload activation. For example, a MilkyDoor (ID: 77) payload which is not reachable by simply opening the app can instead be reached from 60 different paths originating from phone boot, restart, unlock, Wi-Fi connectivity change, and cellular connectivity change events. These events are more likely to occur on a user device rather than on an emulator, as an emulator is typically on, unlocked, and has a stable connection. We believe malware uses this mechanism to avoid detection while maximizing the likelihood to execute the payload. In fact, more than 70% of the paths in our samples do not start on app opening.

To answer RQ2, our samples have more than 14 paths on average, many of which originate from different events. Each path contains numerous conditions, which need to be satisfied for the payload to execute. Paths also contain numerous data retrieval nodes to access data that is validated in the conditions and that flows to payloads. Some data retrieval operations cross multiple paths by "temporarily" storing data in global variables, files, and shared preferences. Understanding the set and order of these behaviors is needed to build accurate malware detection approaches.

5 DISCUSSION AND IMPLICATIONS

In an attempt to understand why such malware still makes its way to the Google Play store, we analyze how existing tools deal with the different characteristics of malware in our dataset. We focus on academic tools only because intellectual property (IP) rights, "security by obscurity" (to prevent malware authors from attacking the tools), and other restrictions prevent us from analyzing commercial offerings. Yet, as all malware collected in our dataset bypassed Google Play defenses, we observe that commercial tools are also limited in detecting such malware.

To identify relevant academic tools, we systematically analyzed proceedings of 20 top conferences and journals in software engineering and security, focusing on publications between January 2010 and July 2021: ICSE [6], FSE [10], ASE [5], ISSTA [8], TSE [91], Empirical SE Journal [39], IEEE Software [3], JSS [12], Information and Software Technology [4], TOSEM [18], CCS [2], S&P [15], USENIX Security [19], ACSAC [1], NDSS [13], TIFS [17], Computers & Security [11], Information Security [7], Security and Privacy Journal [14], and Trans. on Dependable and Secure Computing [16]. This search resulted in 414 papers. We further augmented our list of papers with techniques from seven recent surveys on Android security [26, 31, 43, 73, 74, 84, 88], to cover papers that were published in other venues, identifying additional 106 papers.

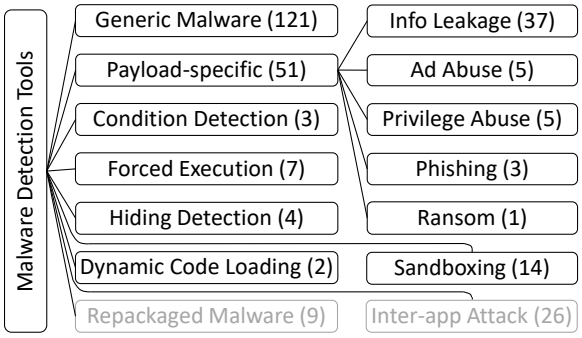

Figure 6: The space of malware detection tools

Two authors of this paper read the abstract/introduction of the identified papers, selecting those that propose approaches for flagging apps as malicious vs. benign. The disagreements (7%) were discussed and resolved either in a mutual discussion or with the help of a third arbitrator. At the end of this process, we identified 237 relevant reports which we further categorized by the type of malware analysis they perform. The resulting categorization is shown in Figure 6 and discussed below; the full list of papers and their categories is available online [35].

Generic Malware. The majority of papers in our list (121) are in this category. They introduce techniques that aim to detect malware by learning from a large set of malicious and benign apps. The techniques vary by the type of features they extract, the way of extracting these features (statically, dynamically, or using a hybrid approach), and the type of machine learning technique used for creating the resulting malware classification model. Features extracted from the apps include permissions, e.g., [40, 77], Java/Android API calls, e.g., [33, 66, 105], system calls, e.g., [78, 87, 109], properties extracted from control- and data-flow graphs of the application, e.g., [66] or combinations of the above, e.g., [30, 49, 57].

These techniques rely on the assumption that malware apps have similar features, which are different from the features of benign applications. Our analysis shows that 65% of the analyzed malware implement rich benign functionality to penetrate into the store and further avoid detection by the user. In such situations, features extracted from the benign portion of the apps may outweigh features from the malicious portion, tipping the classification outcome towards the benign class. Future research could benefit from identifying more targeted features that highlight specific malicious behaviors. Approaches for pre-processing apps, to remove known benign code, e.g., using library or repackaged code detection [37, 62, 86, 104] could also be used in tandem with generic malware detection techniques.

Payload-specific. Another major line of work (51 papers) focuses on looking for a specific payload in an app. Out of those, most approaches (37 papers) identify information leakages. While information leakage is indeed the most prevalent payload in our analysis, other payloads do not seem to receive similar attention. In particular, only a few techniques, e.g., [56], focus on ad abuse detection – the second most common payload in our analysis (69 samples). Moreover, while these techniques work on the Java bytecode level, 29 samples in our analysis implement the payload in JavaScript and 11 – in native code.

A handful of approaches have also been proposed to detect privilege abuse, overlay phishing, and ransom. Yet, these approaches cannot detect the full range of related malicious behaviors identified in our dataset. For example, ransomware detection [36] focuses on identifying apps that perform encryption, while samples in our dataset rather force the device into silent mode before requesting ransom. Finally, no payload-specific techniques exist for other payloads in our dataset, such as cryptomining, clipboard hijacking, account stealing via accessibility services, and more.

Condition Detection. A few papers focus on detecting the check conditions that lead to activating malicious payloads. This line of work assumes that code that is executed under narrow conditions is indicative for malicious behavior. For example, TriggerScope [46] identifies sensitive APIs conditioned by checks constrained by a constant value, mostly targeting logic and time bombs. HsoMiner [69] observes that malicious and benign branches under a narrow condition usually exhibit different characteristics and thus trains a classifier to learn the difference between malicious and benign branches. EnMobile [101] detects data- and control-flows between commands received from the Internet and sensitive APIs used by the app, targeting command-and-control malware.

We believe these are promising directions. Yet, our analysis shows that these techniques might miss existing samples, e.g., because both conditioned branches can be malicious, as in our SpyBankerHU example in Figure 5. Moreover, around 30% of the samples we analyzed employ new check conditions that were not considered before; such conditions would be difficult to integrate into existing approaches, e.g., checking for the Internet or screen status is a common behavior of benign apps as well. Finally, to avoid false positives, techniques in this category often approximate malicious behaviors by the usage of sensitive APIs executed under conditioned checks. Yet, the precise list of sensitive APIs is hard to define; we found samples that use even the most "naïve" APIs for malicious purposes, e.g., an API to set clipboard data. At the same time, apps can deliver payloads without relying on any API at all, e.g., they can use mathematical operations only to mine cryptocurrency on the device. We thus believe it would be beneficial to combine this line of work with techniques that look at specific payloads and also identify strategies for incorporating additional condition checks that are common in benign apps as well.

Forced Execution. Another interesting line of work aims at dynamically forcing the app execution into a malicious path. For example, IntelliDroid [98] and AppIntent [102] use symbolic execution to find the path constraints that lead to executing sensitive APIs, then solve the constraints and provide the appropriate input that triggers the path. FuzzDroid [75] and DualForce [90] use concolic execution to simplify unsolvable constraints. In addition to the problem of triggering the right conditions in the right order discussed in Section 4, the main limitation of these techniques is that, to scale condition resolution via symbolic or concolic execution, they limit the scope of the analysis, e.g., to analyze each application event (entry point) separately. Our analysis shows that malicious behaviors cross entry points in 40% of the analyzed samples; to detect such behaviors, it is essential to analyze paths that cross multiple entry points, including cases where data is transferred from one path to another. Moreover, constraints for some conditions are hard to infer statically. For example, in the Bahamut sample (ID: 18), the response from the server is processed in a loop, which extracts and parses each response line separately, towards retrieving a command. Analyzing such code requires "guessing" the right structure of the response and the desired number of loop iterations, which is a nontrivial task.

Hiding Detection. Several papers in this category mostly focus on flagging the difference between the behavior of the app and its presented user interface [52, 68]. Shan et al. [80] propose scanning the apps for identifying, "self-hiding behavior": blocking phone calls/text messages or removing calls and messages from logs. Yet, the authors notice that benign apps also sometimes employ these behaviors. Thus, hiding techniques on their own do not provide sufficient evidence of maliciousness and can be combined with other techniques, e.g., for trigger and payload detection.

Dynamic Code Loading. A few papers focus on detecting dynamic code loading [72, 82], where code that is not in the APK's bytecode is fetched, either from local storage or from the internet, and loaded at runtime. While dynamic code loading can indeed be used maliciously, it can still be used by benign apps for legitimate reasons, such as patching or loading add-ons. Analyzing the dynamically downloaded code on-demand, as it is being loaded and executed, could be a productive direction for possible future work.

Sandboxing. Tools in this category aim to create analysis platforms to aid researchers in analyzing malware apps, e.g., [27, 63, 89]. Our analysis shows that in almost 80% of our samples, a payload is activated on receipt of a command from the application server. As the malicious developer may turn their servers off or simply not send the right command until the app passes the vetting stage, sandboxing will have limited abilities to detect such apps.

Repackaged Malware and Inter-app Attacks. Tools in these categories focus on detecting repackaged apps, i.e., apps that resemble benign apps available in the store, and apps that "collude" with each other to perform a joint attack. As our dataset does not contain such apps, we omit a discussion of papers in these categories.

A Note on Programming Languages. Most existing tools focus on Java bytecode, with a few analyzing native code as well [20, 21, 49]. While more than a quarter of our samples split the malicious execution between Java bytecode and Web/native code, none of the tools perform an end-to-end analysis of such executions, e.g., to identify conditions in bytecode that constrain Web/native payloads. As malware becomes increasingly more sophisticated, developing such tools could be a direction of possible future work.

Summary. The majority of existing tools focuses on identifying features and training machine learning classifiers to tell malicious and benign apps apart. Payload-specific tools focus mostly on information leakages, leaving a gap in identifying additional payloads. A number of approaches also look at analyzing and triggering conditions that lead to payloads, as well as at hiding techniques. A productive research direction could be to combine condition-, payload-, and hiding-specific techniques, providing a holistic detection approach that can analyze the full behavior of apps. Tools looking at cross-platform malware behaviors, as well as dynamically downloaded code, are also needed.

6 LIMITATIONS AND THREATS TO VALIDITY

The main threat to the validity of our results stems from the manual analysis we performed: when identifying blog posts describing the malware, we could have misinterpreted the findings in a post or missed some relevant blog posts. We could also have missed or misclassified some of the analyzed tools. To mitigate these threats, two authors of this paper performed the blogs and tools classification independently and then cross-validated each other's results.

When performing manual analysis of apps, we could have missed some payloads or incorrectly interpreted the findings. To mitigate this threat, two authors of this paper worked collaboratively on analyzing a number of apps, to establish a common process and methodology. The findings from all analysis steps were discussed with all authors of the paper in periodic meetings, to identify and fix misinterpretations and omissions. We make all our results publicly available to facilitate reproducibility.

We could not perform a detailed manual analysis for some of the apps due to heavy obfuscation, the use of commercial packers, missing encryption keys, and more (see Section 2.2). We excluded these apps from our analysis, to ensure the validity of our results. For apps that rely on a server that is now down, we used descriptions of malicious behaviors taken from the blog post, when available. As the posts may not describe all event conditions, check conditions, and payloads that the downloaded code uses, we could have captured the paths in these apps only partially. We explicitly mark nodes (with ☆) in our flow signatures (5.9 % of all nodes) to distinguish them from behaviors we observed directly.

Our findings might not generalize beyond the dataset that we considered. Yet, as we carefully designed the data collection process and selected a large number of apps from several top security company blogs, we believe that our results are reliable.

Finally, our discussion of implications on malware detection tools is based on paper analysis rather than concrete evaluation. As we identified 202 relevant tools, with around 50 being open-sourced, performing proper evaluation requires (1) a good tool sampling strategy, including a strategy for evaluating tools that are not open-sourced, and (2) proper training/testing methodology as the majority of the tools are based on machine learning (see Section 5). We thus leave such a study for possible future work.

7 RELATED WORK

Felt et al. [44] were likely the first to manually analyze and report on the behavior of 18 Android malware apps, as well as 28 iOS and Symbian apps, collected between 2009 and 2011. Later, Jiang and Zhou [108] collected and manually analyzed apps from antivirus company blogs between 2010 and 2012, contributing a comprehensive set of 1,260 labeled Android malware apps from 49 malware families (the Malware Genome dataset). Wei et al. [97] applied a similar methodology to collect a large set of 24,650 apps that belong to 71 malware families (the AMD dataset) and manually analyzed 405 of these apps. The dataset contains malware between 2012 and 2016. While our work is similar, we collected and manually analyzed a newer set of apps, identifying new payloads and activation mechanisms, as discussed in Section 3. We also outlined hiding techniques and code-level app properties, and provided detailed information on the end-to-end malware activation mechanisms (Section 4), which was not done before.

Similar to us, Kiss et al. [58] performed a detailed manual analysis of samples from seven malware families spanning 2011-2015, describing events and activation conditions required to trigger the malicious behavior. Yet, our dataset is substantially newer and larger, allowing us to identify a larger set of malware characteristics. We also created a schema to capture information about path-sensitive behaviors.

Recently, Xia et al. [100] and Wang et al. [94] studied payloads in cryptocurrency- and COVID-related apps from various markets. Unlike our work, these analyses did not provide activation conditions and the code-level properties of malicious behaviors, focusing on high-level characteristics of the payload. We also do not limit our analysis to apps in a particular category.

A number of authors collected large datasets of malware from various sources, mostly using antivirus scanners to distinguish malicious and benign apps [24, 30, 51, 59, 60, 93, 106]. Others collected datasets of malware barring particular payloads, e.g., piggybacking [61], adware [48, 65], ransomware [36], or banking malware [32]. While these datasets are useful for tool evaluation, the authors do not perform a detailed manual analysis to extract the exact properties of the collected apps, as we do in our work.

Several authors focused on studying the space of features used for malware classification, e.g., [22, 95, 96], as well as biases in malware classification, e.g., [23, 25, 71, 103]. Others built generic and dedicated techniques for identifying malicious applications, as discussed in Section 5. Our work is orthogonal as we do not study datasets at scale but rather perform a detailed manual analysis of apps that evaded detection by the Google Play store.

8 CONCLUSION

In this paper, we systematically built a dataset of 1,238 malware apps that penetrated the official Google Play app store between January 2016 and July 2021. We manually analyzed samples from 105 distinct families, collecting detailed information about activation conditions, hiding techniques, payloads, and code properties (original location and language) that malware employs. We compared our characterization to prior malware analysis reports and further produced detailed malware signatures that accurately capture malicious execution paths of each sample app. We discussed the properties of existing malware detection tools in the context of our analysis and identified gaps and possible future research directions. We believe our detailed analysis of malware behaviors can help develop more efficient malware detection tools.

REFERENCES

[1] [n.d.]. Annual Computer Security Applications Conference (ACSAC).
[2] [n.d.]. Conference on Computer and Communications Security (CCS).
[3] [n.d.]. IEEE Software.
[4] [n.d.]. Information and Software Technology (IST).
[5] [n.d.]. International Conference on Automated Software Engineering (ASE).
[6] [n.d.]. International Conference on Software Engineering (ICSE).
[7] [n.d.]. International Journal of Information Security (IJIS).
[8] [n.d.]. International Symposium on Software Testing and Analysis (ISSTA).
[9] [n.d.]. Investopedia. https://www.investopedia.com/articles/forex/121815/bitcoins-price-history.asp.
[10] [n.d.]. Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).
[11] [n.d.]. Journal of Computers & Security.
[12] [n.d.]. Journal of Systems and Software (JSS).
[13] [n.d.]. Network and Distributed System Security Symposium (NDSS).
[14] [n.d.]. Security & Privacy Journal.
[15] [n.d.]. Symposium on Security and Privacy (S&P).
[16] [n.d.]. Transactions on Dependable and Secure Computing (TDSC).
[17] [n.d.]. Transactions on Information Forensics and Security (TIFS).
[18] [n.d.]. Transactions on Software Engineering and Methodology (TOSEM).
[19] [n.d.]. USENIX Security Symposium.
[20] Vitor Monte Afonso, Matheus Favero de Amorim, André Ricardo Abed Grégio, Glauco Barroso Junquera, and Paulo Lício de Geus. 2015. Identifying Android Malware Using Dynamically Obtained Features. *Computer Virology and Hacking Techniques* 11, 1 (2015), 9–17.
[21] Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. 2017. DroidNative: Automating and Optimizing Detection of Android Native Code Malware Variants. *Computers and Security (CS)* 65 (2017), 230–246.
[22] Kevin Allix, Tegawendé F. Bissyandé, Quentin Jérome, Jacques Klein, Radu State, and Yves Le Traon. 2016. Empirical Assessment of Machine Learning-Based Malware Detectors for Android. *Empirical Software Engineering (EMSE)* 21, 1 (2016), 183–211.
[23] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2015. Are Your Training Datasets Yet Relevant?. In *Proc. of the International Symposium on Engineering Secure Software and Systems (ISESSS)*. 51–67.
[24] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proc. of the Working Conference on Mining Software Repositories (MSR)*. 14–15.
[25] Kevin Allix, Tegawendé François D Assise Bissyande, Jacques Klein, and Yves Le Traon. 2014. *Machine Learning-Based Malware Detection for Android Applications: History Matters!* Technical Report. University of Luxembourg, SnT.
[26] Ebtesam J. Alqahtani, Rachid Zagrouba, and Abdullah Almuhaideb. 2019. A Survey on Android Malware Detection Techniques Using Machine Learning Algorithms. In *Proc. of the International Conference on Software Defined Systems (SDS)*. 110–117.
[27] Cosimo Anglano, Massimo Canonico, and Marco Guazzone. 2020. The Android Forensics Automator (AnForA): A tool for the Automated Forensic Analysis of Android Applications. *Computers and Security (CS)* 88 (2020), 1–15.
[28] APKMonk. [n.d.]. APKMonk. https://www.apkmonk.com.
[29] APKPure. [n.d.]. APKPure. https://apkpure.com.
[30] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*. 1–15.
[31] Saba Arshad, Munam Ali Shah, Abid Khan, and Mansoor Ahmed. 2016. Android Malware Detection & Protection: A Survey. *International Journal of Advanced Computer Science and Applications (IJACSA)* 7, 2 (2016), 463–475.
[32] Chongyang Bai, Qian Han, Ghita Mezzour, Fabio Pierazzi, and VS Subrahmanian. 2019. Dbank: Predictive behavioral analysis of recent android banking trojans. *Transactions on Dependable and Secure Computing (TDSC)* (2019).
[33] Haipeng Cai, Na Meng, Barbara Ryder, and Daphne Yao. 2018. Droidcat: Effective Android Malware Detection and Categorization Via App-Level Profiling. *Transactions on Information Forensics and Security (TIFS)* 14, 6 (2018), 1455–1470.
[34] Michael Cao, Khaled Ahmed, and Julia Rubin. 2022. *GooglePlayMalware. https://doi.org/10.5281/zenodo.5376011.*
[35] Michael Cao, Khaled Ahmed, and Julia Rubin. 2022. *Supplementary Materials. https://resess.github.io/artifacts/GooglePlayMalwareAnalysis.*
[36] Jing Chen, Chiheng Wang, Ziming Zhao, Kai Chen, Ruiying Du, and Gail-Joon Ahn. 2017. Uncovering the face of android ransomware: Characterization and real-time detection. *Transactions on Information Forensics and Security (TIFS)* 13, 5 (2017), 1286–1300.
[37] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. 2015. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *Proc. of the USENIX Security Symposium (USENIX)*. 659–674.
[38] DrizzleRisk. 2021. DrizzleDumper. https://github.com/DrizzleRisk/drizzleDumper.
[39] Empirical Software Engineering (EMSE). [n.d.].
[40] William Enck, Machigar Ongtang, and Patrick McDaniel. 2009. On Lightweight Mobile Phone Application Certification. In *Proc. of the Conference on Computer and Communications Security (CCS)*. 235–245.
[41] ESET. [n.d.]. https://www.welivesecurity.com/2017/02/22/sunny-chance-stolen-credentials-malicious-weather-app-found-google-play/.
[42] ESET. [n.d.]. ESET. https://www.welivesecurity.com/2018/02/28/cryptocurrency-scams-android/.
[43] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. 2015. Android Security: A Survey of Issues, Malware Penetration, and Defenses. *IEEE Communications Surveys Tutorials* 17, 2 (2015), 998–1022.
[44] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. 2011. A Survey of Mobile Malware in the Wild. In *Proc. of the CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*. 3–14.
[45] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–49.
[46] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2016. TriggerScope: Towards Detecting Logic Bombs in Android Applications. In *Proc. of the Symposium on Security and Privacy (S&P)*. 377–396.
[47] Yanick Fratantonio, Chenxiong Qian, Simon P Chung, and Wenke Lee. 2017. Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. In *Proc. of the Symposium on Security and Privacy (S&P)*. 1041–1057.
[48] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. 2019. Should You Consider Adware as Malware in Your Study?. In *Proc. of the International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 604–608.
[49] Joshua Garcia, Mahmoud Hammad, and Sam Malek. 2018. Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware. *Transactions on Software Engineering and Methodology (TOSEM)* 26, 3 (2018), 1–29.
[50] Google. 2021. FirebaseCloudMessaging. https://firebase.google.com/docs/cloud-messaging.
[51] Alejandro Guerra-Manzanares, Hayretdin Bahsi, and Sven Nõmm. 2021. KronoDroid: Time-Based Hybrid-Featured Dataset for Effective Android Malware Detection and Characterization. *Computers and Security (CS)* 110 (2021), 1–38.
[52] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction. In *Proc. of the International Conference on Software Engineering (ICSE)*. 1036–1046.
[53] E. Ouellet I. McShane, A. Litan and P. Bhajanka. [n.d.]. Magic Quadrant for Endpoint Protection Platforms. https://www.gartner.com/en/documents/4001307.
[54] JetBrains. 2021. IntelliJ IDEA. https://www.jetbrains.com/idea/.
[55] Jiagu. 2021. Jiagu. https://jiagu.360.cn/.
[56] Joongyum Kim, Jung-hwan Park, and Sooel Son. 2020. The Abuser Inside Apps: Finding the Culprit Committing Mobile Ad Fraud. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*. 1–16.
[57] TaeGuen Kim, BooJoong Kang, Mina Rho, Sakir Sezer, and Eul Gyu Im. 2018. A Multimodal Deep Learning Method for Android Malware Detection Using Various Features. *Transactions on Information Forensics and Security (TIFS)* 14, 3 (2018), 773–788.
[58] Nicolas Kiss, Jean-François Lalande, Mourad Leslous, and Valérie Viet Triem Tong. 2016. Kharon Dataset: Android Malware under a Microscope. In *Proc. of the Learning from Authoritative Security Experiment Results Workshop (LASER)*. 1–12.
[59] Arash Habibi Lashkari, Andi Fitriah A Kadir, Hugo Gonzalez, Kenneth Fon Mbah, and Ali A Ghorbani. 2017. Towards a Network-based Framework for Android Malware Detection and Characterization. In *Proc. of the Annual Conference on Privacy, Security and Trust (PST)*. 233–23309.
[60] Arash Habibi Lashkari, Andi Fitriah A Kadir, Laya Taheri, and Ali A Ghorbani. 2018. Toward Developing a Systematic Approach to Generate Benchmark Android Malware Datasets and Classification. In *Proc. of the International Carnahan Conference on Security Technology (ICCST)*. 1–7.
[61] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. 2017. Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting. *Transactions on Information Forensics and Security (TIFS)* 12, 6 (2017), 1269–1284.
[62] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. 2017. LibD: Scalable and Precise Third-Party Library Detection in Android Markets. In *Proc. of the International Conference on Software Engineering (ICSE)*. 335–346.
[63] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. 2014. Andrubis–1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *Proc. of the Workshop on Building Analysis Datasets and Gathering Experience Returns*

for Security (BADGERS). 3–17.
[64] Hongjiu Lu. 1995. ELF: From The Programmer's Perspective. *NYNEX Science & Technology Inc* (1995), 95.
[65] Samaneh Mahdavifar, Andi Fitriah Abdul Kadir, Rasool Fatemi, Dima Alhadidi, and Ali A Ghorbani. 2020. Dynamic Android Malware Category Classification using Semi-Supervised Deep Learning. In *Proc. of the International Conference on Dependable, Autonomic and Secure Computing, International Conference on Pervasive Intelligence and Computing, International Conference on Cloud and Big Data Computing, International Conference on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. 515–522.
[66] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. 2017. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*. 1–12.
[67] Palo Alto Networks. 2021. PluginPhantom: New Android Trojan Abuses "DroidPlugin" Framework. https://unit42.paloaltonetworks.com/unit42-pluginphantom-new-android-trojan-abuses-droidplugin-framework/.
[68] Xiang Pan, Yinzhi Cao, Xuechao Du, Boyuan He, Gan Fang, Rui Shao, and Yan Chen. 2018. FlowCog: Context-aware Semantics Extraction and Analysis of Information Flow Leaks in Android Apps. In *Proc. of the USENIX Security Symposium (USENIX)*. 1669–1685.
[69] Xiaorui Pan, Xueqiang Wang, Yue Duan, XiaoFeng Wang, and Heng Yin. 2017. Dark Hazard: Learning-based, Large-Scale Discovery of Hidden Sensitive Operations in Android Apps. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*. 1–15.
[70] Parkour. [n.d.]. Contagio Malware Dump. http://contagiodump.blogspot.com/.
[71] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. 2019. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. In *Proc. of the USENIX Security Symposium (USENIX)*. 729–746.
[72] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2014. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*. 23–26.
[73] Junyang Qiu, Jun Zhang, Wei Luo, Lei Pan, Surya Nepal, and Yang Xiang. 2020. A Survey of Android Malware Detection with Deep Neural Models. *Comput. Surveys* 53, 6 (2020), 1–36.
[74] Bahman Rashidi and Carol J Fung. 2015. A Survey of Android Security Threats and Defenses. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications* 6, 3 (2015), 3–35.
[75] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. 2017. Making Malory Behave Maliciously: Targeted Fuzzing of Android Execution Environments. In *Proc. of the International Conference on Software Engineering (ICSE)*. 300–311.
[76] Hex Rays. 2021. IDA. https://hex-rays.com/ida-pro/.
[77] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Pablo Garcia Bringas, and Gonzalo Álvarez. 2013. PUMA: Permission Usage to Detect Malware in Android. In *Proc. of the Joint Conference CISIS'12-ICEUTE'12-SOCO'12 Special Sessions*. 289–298.
[78] Andrea Saracino, Daniele Sgandurra, Gianluca Dini, and Fabio Martinelli. 2016. MADAM: Effective and Efficient Behavior-based Android Malware Detection and Prevention. *Transactions on Dependable and Secure Computing (TDSC)* 15, 1 (2016), 83–97.
[79] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. 2016. Avclass: A Tool for Massive Malware Labeling. In *Proc. of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. 230–253.
[80] Zhiyong Shan, Iulian Neamtiu, and Raina Samuel. 2018. Self-Hiding Behavior in Android Apps: Detection and Characterization. In *Proc. of the International Conference on Software Engineering (ICSE)*. 728–739.
[81] Virus Share. [n.d.]. https://virusshare.com.
[82] Luman Shi, Jiang Ming, Jianming Fu, Guojun Peng, Dongpeng Xu, Kun Gao, and Xuanchen Pan. 2020. VAHunt: Warding Off New Repackaged Android Malware in App-Virtualization's Clothing. In *Proc. of the Conference on Computer and Communications Security (CCS)*. 535–549.
[83] skylot. 2021. JADX. https://github.com/skylot/jadx.
[84] Alireza Souri and Rahil Hosseini. 2018. A State-of-the-art Survey of Malware Detection Approaches Using Data Mining Techniques. *Human-centric Computing and Information Sciences* 8, 1 (2018), 1–22.
[85] Tim Strazzere. 2021. Android Unpacker. https://github.com/strazzere/android-unpacker.
[86] Guillermo Suarez-Tangil and Gianluca Stringhini. 2020. Eight Years of Rider Measurement in the Android Malware Ecosystem. *Transactions on Dependable and Secure Computing (TDSC)* (2020).
[87] Mingshen Sun, Xiaolei Li, John C.S. Lui, Richard T.B. Ma, and Zhenkai Liang. 2017. Monet: A User-Oriented Behavior-Based Malware Variants Detection System for Android. *Transactions on Information Forensics and Security (TIFS)* 12, 5 (2017), 1103–1112.
[88] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. 2017. The Evolution of Android Malware and Android Analysis Techniques. *Comput. Surveys* 49, 4 (2017), 1–41.
[89] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*. 1–15.
[90] Zhenhao Tang, Juan Zhai, Minxue Pan, Yousra Aafer, Shiqing Ma, Xiangyu Zhang, and Jianhua Zhao. 2018. Dual-Force: Understanding WebView Malware via Cross-Language Forced Execution. In *Proc. of the International Conference on Automated Software Engineering (ASE)*. 714–725.
[91] Transactions on Software Engineering (TSE). [n.d.].
[92] VirusTotal. 2021. VirusTotal. https://www.virustotal.com/home.
[93] Haoyu Wang, Junjun Si, Hao Li, and Yao Guo. 2019. RmvDroid: Towards a Reliable Android Malware Dataset with App Metadata. In *Proc. of the International Conference on Mining Software Repositories (MSR)*. 404–408.
[94] Liu Wang, Ren He, Haoyu Wang, Pengcheng Xia, Yuanchun Li, Lei Wu, Yajin Zhou, Xiapu Luo, Yulei Sui, Yao Guo, et al. 2021. Beyond the virus: a first look at coronavirus-themed Android malware. *Empirical Software Engineering* 26, 4 (2021), 1–38.
[95] Wei Wang, Meichen Zhao, Zhenzhen Gao, Guangquan Xu, Hequn Xian, Yuanyuan Li, and Xiangliang Zhang. 2019. Constructing Features for Detecting Android Malicious Applications: Issues, Taxonomy and Directions. *IEEE Access* 7 (2019), 67602–67631.
[96] Xing Wang, Wei Wang, Yongzhong He, Jiqiang Liu, Zhen Han, and Xiangliang Zhang. 2017. Characterizing Android Apps' Behavior for Effective Detection of Malapps at Large Scale. *Future Generation Computer Systems* 75 (2017), 30–45.
[97] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. 2017. Deep Ground Truth Analysis of Current Android Malware. In *Proc. of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 252–276.
[98] Wong, Michelle Y and Lie, David. 2016. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware.. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*. 1–15.
[99] Woxihuannisja. 2021. Bangcle. https://github.com/woxihuannisja/Bangcle.
[100] Pengcheng Xia, Haoyu Wang, Bowen Zhang, Ru Ji, Bingyu Gao, Lei Wu, Xiapu Luo, and Guoai Xu. 2020. Characterizing cryptocurrency exchange scams. *Computers & Security* 98 (2020), 101993.
[101] Wei Yang, Mukul R. Prasad, and Tao Xie. 2018. EnMobile: Entity-based Characterization and Analysis of Mobile Malware. In *Proc. of the International Conference on Software Engineering (ICSE)*. 384–394.
[102] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. 2013. AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In *Proc. of the Conference on Computer and Communications Security (CCS)*. 1043–1054.
[103] Yanfang Ye, Tao Li, Donald Adjeroh, and S. Sitharama Iyengar. 2017. A Survey on Malware Detection Using Data Mining Techniques. *Comput. Surveys* 50, 3 (2017), 41:1–41:40.
[104] Jiexin Zhang, Alastair R Beresford, and Stephan A Kollmann. 2019. LibID: Reliable Identification of Obfuscated Third-Party Android Libraries. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*. 55–65.
[105] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *Proc. of the conference on Computer and Communications Security (CCS)*. 1105–1116.
[106] Xiaohan Zhang, Yuan Zhang, Ming Zhong, Daizong Ding, Yinzhi Cao, Yukun Zhang, Mi Zhang, and Min Yang. 2020. Enhancing State-of-the-art Classifiers with API Semantics to Detect Evolved Android Malware. In *Proc. of the Conference on Computer and Communications Security (CCS)*. 757–770.
[107] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. 2015. Dexhunter: Toward Extracting Hidden Code From Packed Android Applications. In *Proc. of the European Symposium on Research in Computer Security (ESORICS)*. 293–311.
[108] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android Malware: Characterization and Evolution. In *Proc. of the Symposium on Security and Privacy (S&P)*. 95–109.
[109] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 2012. Hey, You, Get Off of my Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*. 50–52.