Lost in the Mists of Time: Expirations in DNS Footprints of Mobile Apps

Johnny So
Stony Brook University

Iskander Sanchez-Rola
Norton Research Group

Nick Nikiforakis
Stony Brook University

Abstract

Compared to the traditional desktop setting where web applications (apps) are live by nature, mobile apps are similar to binary programs that are installed on devices, in that they remain static until they are updated. However, they can also contain live, dynamic components if they interface with the web. This may lead to a confusing scenario, in which a mobile app itself has not been updated, but changes in dynamic components have caused changes in the overall app behavior.

In this work, we present the first large-scale analysis of mobile app dependencies through a dual perspective accounting for time and version updates, with a focus on expired domains. First, we detail a methodology to build a representative corpus comprising 77,206 versions of 15,124 unique Android apps. Next, we extract the unique eTLD+1 domain dependencies — the "DNS footprint" — of each APK by monitoring the network traffic produced with a dynamic, UI-guided test input generator and report on the footprint of a typical app. Using these footprints, combined with a methodology that deduces potential periods of vulnerability for individual APKs by leveraging passive DNS, we characterize how apps may have been affected by expired domains throughout time. Our findings indicate that the threat of expired domains in app dependencies is nontrivial at scale, affecting hundreds of apps and thousands of APKs, occasionally affecting apps that rank within the top ten of their categories, apps that have hundreds of millions of downloads, or apps that were the latest version. Furthermore, we uncovered 41 immediately registrable domains that were found in app footprints during our analyses, and provide evidence in the form of case studies as to their potential for abuse. We also find that even the most security-conscious users cannot protect themselves against the risk of their using an app that has an expired dependency, even if they can update their apps instantaneously.

1 Introduction

The mobile platform is responsible for a significant portion of web traffic; according to a recent statistic, mobile usage was responsible for 62% of all web traffic in 2023, as compared to 36% for desktop usage and 2% for tablets [64]. Although web browsing contributes most of the overall mobile web traffic, reports have found that the contribution of mobile applications (apps) has been steadily increasing. On average, a person has more than 80 apps installed, and uses nine apps every day or 30 apps every month [42], and the total number of app downloads per year continues to increase [20].

These mobile apps are binary programs that are distributed via app marketplaces and installed on user devices. Apps that communicate over the Internet to provide their services can be perceived to contain a live, dynamic component, in addition to its static, installed component. Regardless of whether the static, installed component is outdated, the dynamic component can change at any moment if the online dependencies change. This is particularly worrisome if such changes are caused by the expiration or re-registration of critical domains that may be responsible for key functionality, interface with sensitive data, or affect app content. An orthogonal question is whether this is mitigated if a user keeps their apps up to date. From prior surveys and studies, we know that a significant portion of users do not enable automatic app updates [18, 49, 68], and there is usually a delay before users choose to update their software in general [41, 46, 56]. Thus, the security of older app versions becomes much more important.

Currently, industry best practices recommend that app developers attempt to balance user experience with app updates, and force updates if necessary by integrating version-control logic [13, 34, 51]. For example, a simple method can be to implement minimum and maximum versions supported by the backend, and upon discovering that the local app version is outdated, block access to the app with a dialog that recommends — or requires — an app update. Surprisingly, Android app developers needed to implement this functionality ad-hoc until the release of the Google Play Core `AppUpdate` library in 2020 [6], and it appears that there is still no iOS counterpart in the standard library as of 2024.

In this work, we conduct the first large-scale dynamic analysis of Android apps to characterize their dependency usage

across time and app updates, with a focus on identifying expired domains. For each APK, we track its DNS dependencies and report on temporal characteristics of app stability, particularly on time windows of vulnerability when app dependencies can be hijacked and influence app behavior. We frame our key goals through several research questions (RQ).

RQ0: What is the DNS footprint of a typical app? In the rest of this work, we use the term "DNS footprint" to refer to the set of eTLD+1 domains that are contacted by an app during execution, where eTLD+1 domains are the registrable names that account for public suffixes. Additionally, we use the concepts of first-order domains and second-order domains to distinguish domains that are automatically contacted, from domains of links that are present on webpages loaded by apps. We extract the footprint of each APK, and find that the supply chain attack surface of apps consists of 7.4 eTLD+1 first- and second-order domains, on average. We discovered that 309 versions of 149 unique apps relied on 41 first-order domains that were immediately registrable at the time of analysis. Some affected apps were ranked within the top ten of their categories and were downloaded more than 100 million times, and some versions were the latest of their app.

RQ1: How stable are DNS footprints over time? We perform a longitudinal analysis of app dependencies by leveraging a historical, passive DNS database to determine critical times for every extracted footprint. On average, 2.5% of all APKs, or 4.2% of all apps, in our data use at least one expired domain at any time. Furthermore, 0.7% of APKs are at risk of a domain in its footprint expiring within one month of release.

RQ2: How do DNS footprints change across updates? We track how DNS footprints change from one version to the next, and focus on how such changes affect the attack surface of apps. Updates can drastically alter the footprint of an app: over the course of 9 updates, the number of changes to the footprints of 29% of apps is approximately equal to the overall average app footprint size. Furthermore, we uncover evidence that updates do not always address problems with the use of expired domains in previous versions, even for highly-ranked apps with more than 10 million downloads. Additionally, our analyses suggest that the latest version of an app is more secure than its older versions, confirming expectations.

2 Background

In this section, we outline the Domain Name System (DNS) and dynamic analysis of Android apps. Additionally, Appendix A describes how DNS records *may exist even for expired domains* and Appendix B details over-the-air updates.

2.1 The Domain Name System

The Domain Name System (DNS) is responsible for translating human-readable domain names to their machine IP addresses. Each top-level domain (TLD) has its own governing authority; for example, generic TLDs (gTLDs) are managed by ICANN, whereas specialized TLDs such as country-code TLDs are managed by their respective countries. Domain names are registered for one or several years, after which they expire and must be renewed. Registrars are required to send multiple expiration notifications to domain owners to provide buffer time to renew their domains before they are released to the public, for domains in gTLDs that are governed by the ICANN Expired Registration Recovery Policy (ERRP) [30]. If a domain expires and is not renewed during the 30-day Redemption Grace Period, it enters the Pending Delete status, and will become available for registration after five days [29].

Public Suffixes. The actual names that are registrable are found at different levels in a full domain name. The term "top-level domain" refers to the ending suffix of a domain — for example, `.uk` in `example.ac.uk`. However, `ac.uk` is not a registrable domain; instead, `example.ac.uk` is. The community maintains the Public Suffix List (PSL) [47] to make this easier — the PSL is a list of "public suffixes", or "effective top-level domains" (eTLDs, or sometimes referred to as e2LDs), under which users can directly register domain names. These registrable domains are colloquially termed "eTLD+1" domains because they comprise the entire eTLD and the neighboring, registered name [43, 69].

Passive DNS. Passive DNS refers to a technique that strategically places DNS sensors to capture live DNS records observed in the wild [67], building partial replicas of DNS zones, and storing them in a database. Typical DNS records are annotated with the first and last times they were observed, and how many times they were observed, essentially providing a maximal time window during which a DNS record was live, with a representation of its request volume. The resulting data can be used to answer questions that would otherwise be impossible, such as historical resolution information (e.g., IP addresses that a domain resolved to in the past).

2.2 Mobile App Analysis

Program analysis techniques are broadly categorized as static or dynamic, depending on whether the approach involves actually executing a program: whereas static techniques analyze the source code of a program, dynamic ones extract data from the program at runtime. Each approach has their own advantages: static analysis techniques are lightweight, performant, and scalable, whereas dynamic ones are slower, more resource-intensive, but can provide different coverage.

Dynamic analysis of mobile apps can be particularly challenging because of the ecosystem; as compared to its desktop counterpart, mobile devices are limited in variation, and are more difficult to automate, regardless if they are emulators or actual devices. Whereas emulators can be freely spawned on existing resources with a flexible device and system image configuration, they are often much slower and unstable as compared to physical devices. In addition, although physical

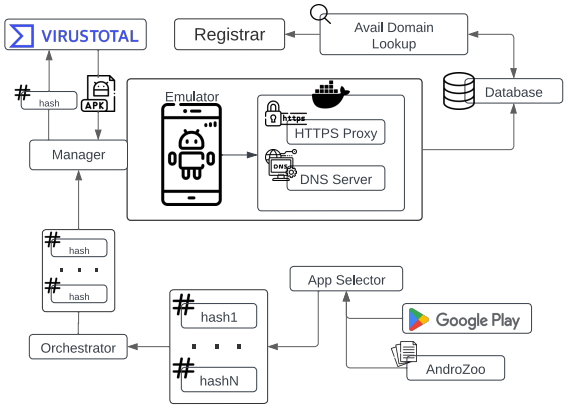

Figure 1: *Footprint extraction infrastructure.*

devices present higher fidelity testing environments, maintaining them for long experiments will require extended physical access to reboot devices in case of crashes or hardware failures. Thus, we use the Android Studio Emulator.

3 Methodology

In this section, we present the rationale behind our experimental and analytical design. First, we discuss our threat, and subsequently analytical, model. Then, we detail how we build our APK corpus from multiple, disparate data sources. Next, we describe the supporting infrastructure for data collection and dynamic analysis. Finally, we present our approach to detect periods in time when domains were expired, by using passive DNS and the domain lifecycle.

3.1 Threat & Analytical Models

We consider the scenario where Android users have app installations that may be outdated (i.e., not the latest version). Regardless of the update status, each application contacts a set of online services. We seek to understand the threat of expired domains in these dependencies, and simultaneously characterize how each application's set of dependencies changes with respect to time and app updates.

We primarily consider our analyses on the eTLD+1 part of the full domains, which presents a more accurate perspective in assessing the risk of expiring app dependencies as compared to the use of the full domains. We use the following terminology: (a) an *app* as an Android application, a grouping of APKs with the same package name; (b) a *version* of an app as an APK with a particular package name; (c) the *DNS footprint of an APK* to refer to the set of eTLD+1 domains that it contacts; and (d) the *DNS footprint of an app* to refer to the union of the DNS footprints of each of its versions. When referring to DNS footprints, we use the terms "domains," "eTLD+1 domains," and "eTLD+1s" interchangeably.

Higher-Order Links. In-app web content rendering presents web content in the context of an app itself; thus, such content can be considered as a live, dynamic component of the app and content changes may be perceived as app changes.

We broadly categorize web content URLs in apps, and the resulting domains in footprints, as *first-order* and *second-order*. The initial URL that is directly loaded by the in-app browser upon instantiation, and the URLs that automatically fire from loading this initial URL, are considered first-order links. Changes in first-order links have the greatest potential for damage, because they directly influence the initial webpage that is loaded by the in-app browser. In contrast, second-order links — the URLs present in the anchor tags of the initial webpage, as well as the URLs of their subsequent requests that fire — represent the URLs that are just "one click away." These clicks may have "dirty" hygiene, causing users who click on their webpages to suffer from privacy- and security-harming behaviors, including passing through invisible layers of scripts, visiting unexpected domains, encountering mixed content, and receiving undesired cookies, resulting in an overall increase in the risk faced by users [59]. In the context of this study, this differentiation amounts to the inclusion of one more party in the supply chain.

Accordingly, we introduce the notions of first-order footprints, which are the set of first-order eTLD+1s, and second-order footprints, which also adds second-order eTLD+1s, to account for the difference between the severity of a first-order and a second-order eTLD+1 expiring, in Sections 4 and 5.

3.2 Data Sources

The primary sources for our app dataset are the Google Play Store, the AndroZoo dataset [1], and the VirusTotal database [66]. Google Play is the official Android app marketplace and is easily accessible from the web. It provides app metadata without requiring authentication, but *only for the latest version of applications*, which can be downloaded with a Google account and an Android device. In contrast, the AndroZoo APK list is a *collection of app metadata*, spanning multiple years, resulting from prior work that continues to crawl the entirety of multiple app marketplaces, including Google Play, on a daily basis. Similarly, VirusTotal, a Google-owned service that leverages tens of malware detection engines to determine whether a file is malicious, provides API endpoints that return metadata and file downloads for hash digests in their database, which continually grows from its free and commercial user base.

Our dataset collection process begins by crawling Google Play to amass metadata for the 500 most popular apps in all 49 Google Play app categories. Afterwards, we filter the AndroZoo APK list for apps that match the metadata from Google Play — in particular, we filter for those with a matching package name as these are guaranteed to be unique within each marketplace [22]. Some apps have hundreds of versions, so we limit the number of versions of each app to the most recent 10 in the aggregated dataset. Finally, we query the VirusTotal

API to download the APKs using their hash digests obtained from the AndroZoo list. We note that VirusTotal is integrated into enterprise systems for automated malware scanning, resulting in large numbers of both benign and malicious samples in its database [25].

In summary, the dataset comprises the VirusTotal APKs of the most recent 10 AndroZoo versions of the 500 most popular apps in each of the 49 Google Play categories.

Passive DNS. We also depend on Farsight DNSDB [21], a commercial passive DNS database, to provide historical DNS resolution data for the domains that are contacted during execution of an app version. According to their product website, their database contains over 100 billion DNS records, starting from 2010 [21]. We describe our domain expiration time window analysis using this passive DNS data in Section 3.4, and discuss the results in Sections 4 and 5. We provide a detailed empirical verification of our approach in Appendix C.

APK Release Dates. To answer **RQ1** and **RQ2**, it is crucial that our dataset includes the dates when each APK was released. Unfortunately, there is no readily-available, reliable data for this: Google Play only tracks the date of the latest update of apps, and the APK build process was changed in 2016 to explicitly clear the timestamp in the metadata [24]. We adopt a best-effort estimate by collecting the dates when individual APKs were first seen from various sources, and taking the earliest date among these to be the release date of the APK. In particular, we compare the following dates: 1) the first seen date in VirusTotal, 2) the first seen date in telemetry data of app installations, identified by hash digests, provided by telemetry data from an industry partner, and 3) the upload date on the third-party sites APKPure [11] and APKMirror [10], which store copies of APKs.

3.3 Infrastructure

We present an overview of the main components of the associated infrastructure for this study in Figure 1 — an app selector, a batch orchestrator, and analysis node managers. The app selector bootstraps the pipeline by building the dataset of apps as previously described in Section 3.2 and sending it to the orchestrator, which batches the workload and assigns analysis nodes for distributed processing. Each manager is responsible for assigning jobs to each of the multiple workers and execution pipelines running in parallel on the node.

Pipelines are centered around an Android Studio Emulator controlled through `adb`, the Android Debug Bridge [3]. Emulators were created with a default device profile using Android version 11 (API level 30). Each emulator uses a unique HTTPS man-in-the-middle proxy based on `mitmproxy` [19] and a DNS server configured as a `bind9` [31] forwarding resolver; they are also equipped with a hook that checks non-resolving domains to see if they are registrable in real time. This one-to-one mapping between an emulator and its HTTPS proxy and DNS server allows us to cleanly isolate and attribute the traffic from each emulator. In addition, emulators run in rooted mode to import the Certificate Authority (CA) of the HTTPS proxy as a trusted CA in the Android OS in order to decrypt traffic. Lastly, each emulator installs a Frida [57] server used to dynamically instrument functions to disable certificate pinning and grant permissions at runtime.

Each app is scheduled to be processed three times to minimize variability in the extracted footprints. Upon receiving a task to process an app version, the worker schedules it to the next available pipeline which marks the start and the end of an analysis session, before and after an app is launched. This involves: 1) installing the actual APK file on the emulator, 2) coordinating database state so that the HTTPS proxy, which is already connected, can attribute any observed traffic to an APK, 3) instructing the DNS server to begin a DNS packet trace, and 4) querying a special, non-existent domain that encodes an identifier for the app analysis session. The last step provides traffic delimiters to increase confidence in associating traffic to the running app by marking a window of time during which the app was active, which is used to exclude any background requests.

We use a custom version of DroidBot [36] to dynamically explore apps for three minutes, modified to launch apps with Frida to intercept certain function calls. In particular, Frida is configured to disable certificate pinning [54] by instrumenting the standard method for programmatically creating a `javax.net.ssl.SSLContext` that manages trusted authorities [8], to automatically grant requested runtime permissions with `androidx.core.ContextCompat` [4], and to disable the `FLAG_SECURE` flag in views [9]. Instrumenting these methods enables the analysis pipeline to correctly intercept HTTPS traffic even if developers attempted certificate pinning, proceed with app exploration in the face of runtime permission dialogs, and ensures DroidBot can take screenshots.

3.4 Identifying Expirations from Passive DNS

This section details the approach for identifying expirations from passive DNS data based on knowledge of the domain lifecycle. Algorithm 1 in Appendix C presents the same information in the form of pseudocode.

As discussed in Section 2, passive DNS data comprises DNS records annotated with the first and last time they were observed, and how many times they were observed. We summarize the passive DNS records into non-overlapping time intervals during which DNS records were, or were not, configured for an individual domain. These intervals were obtained by extracting the first seen and last seen timestamps for individual A, AAAA, CNAME, NS, SOA, MX, or TXT records (Algorithm 1, Line 25), merging any overlapping time windows within each category (Line 30), and then merging the resulting time windows once more among all categories (Line 32). While processing these intervals, we err on the side

of caution and truncate all timestamps to the granularity of dates, and ignore gaps between successive intervals that are under 5 days to account for transient errors or misconfigurations. With this, we can determine whether a domain d was configured with a DNS record at any time t, assuming t is after the launch of the database in 2010.

Next, our approach looks for domains that have gaps of at least 35 days in the merged passive DNS intervals. We assume that such long gaps exist because the domains had expired, and passed both the 30-day Redemption Grace Period and 5-day Pending Delete durations, as we are not aware of any scenarios that enable an individual to remove *all* DNS records for a domain that they own. After such gaps (if any) are identified for a domain (Line 36), we adjust them so that the start of a gap is at least one year after the end of the prior gap (Line 37). If any adjusted gap interval does not meet the previous duration requirement of at least 35 days, the gap period is excluded. This additional step reduces false positives in detecting expirations because domain registration periods are in integer intervals of (up to 10) years. Thus, this approach does not account for phenomena such as early domain deletions [14], as it inherently limits the number of expirations to at most one per year.

We empirically verified the accuracy and granularity of the passive DNS data from our provider, finding zero false positives, and some false negatives that were caused by registrar behavior. However, false negatives do not hamper our results — our findings present a lower bound, and worst-case, analysis on the exploitability of the DNS footprints of apps. See Appendix C for a more detailed discussion.

4 DNS Footprints

In this section, we describe the collected dataset, and characterize high-level trends to describe a typical footprint. We focus on studying the hijackability of the domains that comprise each footprint to quantify the feasibility of abuse.

4.1 Dataset

We scraped the 500 highest ranked apps in each of the 49 Google Play categories and filtered for the latest 10 versions of these apps from the AndroZoo list downloaded on December 20, 2022, as described by the input generation process described in Section 3. This resulted in 24,166 unique packages from Google Play. Accounting for the latest 10 versions, we found 185,685 versions for 23,332 packages in the AndroZoo list. We were able to successfully download 83.7% of all versions, or 92.6% of all apps, from VirusTotal, and there remain analysis sessions for 49.7% of versions, or 70.0% of apps, downloaded from VirusTotal after filtering.

Apps that were not successfully executed could have failed at any step — installation, app launch, or app exploration — for a number of reasons, including compatibility (e.g., apps expecting different device architecture or Android API level, or launching apps with Frida) and reliability in the emulators and the apps (e.g., crashing or unresponsiveness). Furthermore, it may be the case that apps were not explored "sufficiently", or at all, depending on the interactions between DroidBot and the app. Reasons for inadequate exploration include insufficient runtime and splash screens (e.g., login screens) that prevent further interaction with the app. However, these do not hinder the goal of our study, which is to provide a quantified lower bound on the threat posed by expired domain dependencies. We argue that inadequate interactions with apps result in smaller footprints and only contribute to false negatives in general, but discuss this issue in more depth in Section 7.

We use the presence of extracted traffic as an indicator of successful execution to preprocess the data before our analyses. One such filter ensures that execution behavior was as expected, and increases the confidence of attributing domains to apps, by querying non-existent domains to mark the beginning and ending of an app execution session, as previously mentioned in Section 3; we exclude these and only keep domains that were queried between these markers. Another filter excludes domains found in DNS traffic for 99.35% of apps, or 96.69% of versions, which we suspect or confirm to be contributed by the Android operating system instead of the apps. These domains include `time.android.com`, which is used as an NTP server, `connectivitycheck.gstatic.com` which is used to check Internet access with frequent heartbeats, `*.googleapis.com`, and `www.google.com`.

In the rest of this work, we present results for 15,124 apps, or 77,206 app versions, which are 70.0% apps, or 49.7% app versions, of those found on VirusTotal. See Table 1 for a summary of the number of apps and versions at each step of the process, and Section D for confirmation that the resulting dataset is representative of the Android app landscape.

4.2 Footprint Composition

After excluding domains for which we had no passive DNS data, we found that the 77,206 versions of 15,124 apps query for 22,695 unique domains across 10,279 unique eTLD+1s. Consistent with real world expectations, the distribution of the number of apps that use each domain has a long tail, as do other distributions such as the DNS footprint size of apps and the number of unique subdomains per eTLD+1. On average, the size of a first-order footprint of an individual APK is 10.6 unique domains, or 6.7 eTLD+1s. Second-order links contribute a significant additional 11.0% more domains and 10.6% more eTLD+1s. The risk of compromise and perceived app functionality breakage may be greater when users navigate webpages that are rendered in-app, because the size of the supply chain only increases, which we confirm in Section 5. Table 2 breaks down the average footprint size.

Table 1: *App dataset; percentages reference the prior step.*

Source	Packages	APKs
Google Play	24,166 (100%)	-
AndroZoo	23,332 (96.5%)	185,685 (100%)
VirusTotal	21,612 (92.6%)	155,442 (83.7%)
Filtered	15,124 (70.0%)	77,206 (49.7%)

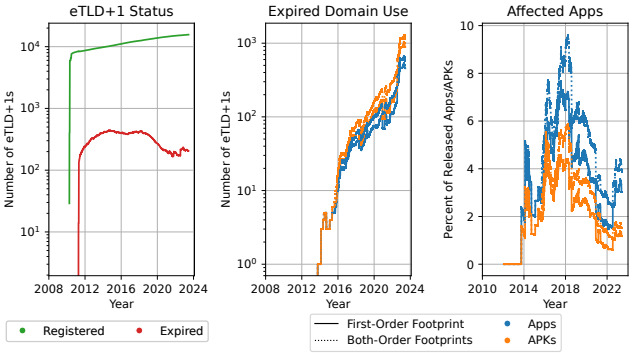

Figure 2: *Registration status of eTLD+1s, focusing on expirations.*

4.3 Expirations Throughout Time

In this section, we quantify the threat of expired domains in the mobile ecosystem by measuring the percentage of domains, apps, and APKs that are at risk throughout time. To this end, we used the computed expiration periods for all extracted eTLD+1 domains for apps in our dataset, and deduced the domains that were expired on any given day d. Following this, we used our DNS footprint data to extract the set of APKs that depended on such domains, and filtered for those that were released on or before d. The resulting data represents the set of APKs that relied on domains which were expired at d, and is represented in Figure 2.

The leftmost plot reports the registration status of domains in our dataset at any given time. As our passive DNS provider started collecting data in 2010, there is a perceived spike in registrations at the beginning of 2010, and a perceived spike in expirations at the beginning of 2011. After this, we observe that there is a steady increase in the number of registrations, implying that new eTLD+1s continue to be added to the DNS footprints of apps. In contrast, the number of expired domains remained relatively constant in the last 11 years, and is now approximately two orders of magnitude smaller than the number of footprint domains ever registered.

The center and right plots represent the apps (in blue) and APKs (in orange) that use at least one expired domain at any given time, with a breakdown that illustrates the contribution of additional domains that are contributed by second-order links. We count an APK at time t if at least one domain in its footprint is expired, and we count an app at t if at least one of its APKs counts at t. The middle plot, which reports raw numbers of apps and APKs, shows an expected, non-decreasing trend because the number of outdated APKs continues to grow.

Table 2: *Average footprint size in our dataset. Second-order percentages are percentage increases over the first-order size.*

# vs. Order	First-Order	Second-Order
Domains / APK	10.6	11.8 (+11.0%)
eTLD+1s / APK	6.7	7.4 (+10.6%)
Domains / App	25.6	28.4 (+10.9%)
eTLD+1s / App	13.9	15.4 (+11.1%)
Total Domains	22,695	33,323 (+46.8%)
Total eTLD+1s	10,279	16,758 (+63.0%)

Interestingly, the lines representing the apps and APKs slowly diverge, with the number of APKs pulling ahead of the number of apps, suggesting that when common domains expire, all versions with those domains are affected. The right plot represents the same data as percentages of the number of affected apps affected at time t out of the number of apps that had been released by t. Although the raw number of affected apps is growing, we find that their percentage of all apps and APKs ever released fluctuate, but remain under 10% at any point in time. Overall, the average percentage of APKs using an expired domain is 2.0% and 2.5% for first-order and second-order footprints respectively, and the average percentage of apps respectively is 3.3% and 4.2%.

4.4 Immediately Registrable Domains

In contrast to the previous temporal analysis based on the computed expiration periods, we discovered 41 expired domains — which we verified to be immediately registrable by asking a registrar — in the dependencies of apps at runtime. Based on the network traffic and DEX to Java decompilation [62], we summarize the 32 domains whose purpose we were able to identify in Table 3, which exclude the 9 whose purpose we could not identify. This section discusses their usage patterns and how such domains can be abused when expired, and Section 4.5 examines several as case studies.

Domain Usage. We identified four different types of expired domains based on how they were used by their respective apps, from the network traffic and decompiled APK code: APIs, Mirrors, Ads/Tracking, and Web Content. API was the most common, with 32 of them — these domains host web servers that respond to requests for app functionality, and interface with the app database. The next type, Mirrors (N=7), are pre-defined sets of domains that are used interchangeably in first-party app code. They can be considered API servers, but are separated because their expiration does not pose the same risk as a regular API server, as a mirror domain would only be responsible for a portion of the app's userbase. The third type, Ads/Tracking (N=3), is responsible for serving advertisements or tracking user statistics. If the associated ads/tracking libraries fail gracefully in the event that these domains cannot be reached, app functionality should not be affected. The last type, Web Content (N=1), is a rare type of domain that serves web content inside the app, and thus

Table 3: *Immediately-registrable domains that were successfully identified. Rankings were taken during data collection, but number of downloads were taken from Google Play at the time of writing. The value "-" in "Downloads" means that the app is no longer available.*

Domain(s)	Usage	Location	# Apps	# APKs	Top App	Rank	Category	Downloads	In Latest
voodoo-ads.io	Ads/Tracking	Library	49	111	io.voodoo.holeio	6	Game/Arcade	100M+	No
melaniapps.com	Ads/Tracking	First-Party	3	3	best.beard.photo.editor	91	Beauty	500K+	No
wecareapps.net	Web Content	First-Party	2	6	com.meshref.pregnancy	232	Parenting	-	-
goneorbital.com	API	First-Party	1	6	com.gamehouse.slingosolitairegp	377	Game/Casino	500K+	**Yes**
cneplant.cn	API	First-Party	1	5	com.mini.wificam	454	Video Players	-	-
kok{lion,joy}.com	API	Library	1	5	alpha.hd.anime.girls.wallpapers	90	Comics	-	-
iphonequran.org	Mirror	First-Party	1	3	com.guidedways.iQuran	466	Books and References	10M+	No
islamicsoftwares.com	Mirror	First-Party	1	3	com.guidedways.iQuran	466	Books and References	10M+	No
klyapps.com	API	First-Party	1	3	com.kalay.equalizer	232	Music and Audio	5M+	No
ozzmogvt.com	API	Dynamic	1	3	com.free.vpn.ozzmo	307	Tools	1M+	No
badrniaimi.xyz	API	First-Party	1	2	com.manga.library.reader	97	Comics	-	-
bdjobstutor.com	API	First-Party	1	2	com.americabangla.nailpolish	402	Beauty	-	-
c2s4.xyz	API	First-Party	1	2	chapter2.fortnite.wallpapers.season6	82	Books and References	500K+	No
freegtw.com	API	Dynamic	1	2	com.free.vpn.ozzmo	307	Tools	1M+	No
amazon-node.com	Mirror	First-Party	1	1	com.free.vpn.ozzmo	307	Tools	1M+	No
cloudozzmo.com	Mirror	First-Party	1	1	com.free.vpn.ozzmo	307	Tools	1M+	No
cuebiqxxx.com	API	Library	1	1	com.weather.radar.forecast.livemaps	458	Weather	-	-
datingxa.com	Ads/Tracking	First-Party	1	1	comg.aml.cre.ati.vepl.ay	62	Art and Design	-	-
demo1json.shop	API	First-Party	1	1	com.AinoTdemon.slayer.game.guide	311	Books and References	-	-
firebaseu.com	Mirror	First-Party	1	1	com.free.vpn.ozzmo	307	Tools	1M+	No
firstbankmellat.com	Mirror	First-Party	1	1	com.free.vpn.ozzmo	307	Tools	1M+	No
freeozzmo.com	Mirror	First-Party	1	1	com.free.vpn.ozzmo	307	Tools	1M+	No
jatisulistiyo.com	API	First-Party	1	1	com.ajegalways.behelBeautyeditor	361	Beauty	-	-
lionosur.com	API	First-Party	1	1	com.lionosur.ironmanarcreactor	209	Comics	-	-
livefoothd.com	API	First-Party	1	1	com.pikaterapp.pikaterlivefoot	329	Libraries and Demo	10K+	**Yes**
masterbinary.trade	API	First-Party	1	1	com.zaunz.latestoutfitsideasforteenagegirls	301	Beauty	50K+	No
nexshome.com	API	First-Party	1	1	com.altec.shsm	371	House and Home	-	-
ozzmovpn.com	API	First-Party	1	1	com.free.vpn.ozzmo	307	Tools	1M+	No
sourtimeprime.com	Mirror	First-Party	1	1	com.free.vpn.ozzmo	307	Tools	1M+	No
tekoiaiot.com	API	First-Party	1	1	com.tekoia.sure.activities	107	House and Home	10M+	No
unbwebgvt.com	API	Dynamic	1	1	com.free.vpn.ozzmo	307	Tools	1M+	No

control of such domains would enable arbitrary web content in affected apps.

Domain Location. In addition to identifying how each domain was used, Table 3 also identifies in which parts of the apps these domains were used, marking whether the domains were found in first-party app code (`First-Party`), third-party libraries (`Library`), or not at all (`Dynamic`). Categorization for each domain was performed by finding exact matches for the eTLD+1 domains for each app. Then, after considering the app structure, package names, and similarities with other apps that use the domain, we labeled each domain. Most of the domains (25) were found directly inside first-party code or resources (e.g., `strings.xml`), with 4 found in app-external library code, and 12 not found at all. As there is no context for the 12 domains, we were unable to identify their purpose, except for a select few used by one app as API servers. We discuss several `Library` domains in greater detail in Section 4.5.

Root Cause. With the proliferation of the number of apps on the market and the long-tail distribution in app popularity and usage, there exist many domains that are registered and configured for specific apps. Thus, when these less-popular apps are abandoned by their developers, the accompanying domain will be left to expire. This hypothesis is supported by Table 3 — 11 domains are used by apps that are no longer in the Google Play Store — and the phenomenon contributes the largest portion of expired domains by number. On the other hand, domains that are integrated with SDKs that are used by many apps can also be left to expire if the SDKs change. The latest versions of the Voodoo apps that were affected by the expired domain `voodoo-ads.io` no longer use the same SDK, and the Cuebiq SDK that relied on `cuebiqxxx.com` has been deprecated. Similarly, apps that overhaul their codebase can change their dependencies, and developers do not continue to re-register domains that were used in older versions. Of the 11 apps that are listed in "Top App" and still available on Google Play, 9 of them have changed their code in their latest versions such that the domains are no longer statically defined ("In Latest" value is "No"). Interestingly, the app `chapter2.fortnite.wallpapers.season6` is now a racing car game in its latest iteration. In contrast, the latest versions of `com.pikaterapp.pikaterlivefoot` and `com.gamehouse.slingosolitairegp` contain the expired domain. In fact, the former app has only one version on AndroZoo, and was the latest on Google Play at the time of writing; the latest version of the latter app was part of our dataset but failed to execute on our infrastructure. Thus, these two apps are not included in the analysis described in Section 5.2 (they were also recently removed from Google Play). Overall, it appears that the domains were left to expire because of app abandonment, SDK changes, and app changes.

4.5 Immediately Registrable Case Studies

This section provides concrete details of how several immediately registrable domains from Table 3 can be abused as supply chain attacks from manual investigation of the identified app versions that was performed at a later date.

Usage: APIs. We highlight several API domains used by the remote camera viewer `com.mini.wificam`, IoT device managers `com.altec.shsm` and `com.tekoia.sure.activities`, and live stream viewer `com.pikaterapp.pikaterlivefoot`. The first app, ver-

sion 25.0.0.0.20, contacts p.cneplant.cn at the path api.php?type=6&version=<vers>, to determine if an app update is available. We verified that control of this domain can disarm this built-in version control logic by sending responses that indicate no updates exist. From inspecting the decompiled code, com.altec.shsm, version 0.0.8, uses the domain app.nexshome.com for its login functionality and IoT device management. However, we found that the login functionality was actually moved into a third-party SDK (which *sends login credentials in plaintext* to a currently non-expired domain) and a significant portion of the API class that contacts app.nexshome.com is dead code. It does send one request to the domain at /smart/share/getShare to retrieve devices, but it is not properly populated with the values for userId and pw_session, even if authenticated.

In contrast, com.tekoia.sure.activities relies on tekoiaiot.com for its primary API and user database. On startup, it fetches a user database from the domain geo-service-userdb.tekoiaiot.com and prevents further interaction with the app if no valid zip file is downloaded. We verified that control of the tekoiaiot.com domain can push an arbitrary zip file to user devices on app startup — which the app will attempt to read as a database — and respond to several database API requests. Another app com.pikaterapp.pikaterlivefoot, for sports video streaming, version 2.0.9 depends on the domain livefoothd.com. The app requests video content from mobtv.livefoothd.com on startup at the path api/get_category_posts, and also links to its privacy policy on this domain. We verified that this endpoint grants the ability to change the content (e.g., name via the value in channel_name and preview images via channel_image) and the video stream source via channel_url in the app.

Usage: Mirrors. This next category describes domains that are pre-defined and used interchangeably inside first-party app code, and we detail the two apps com.free.vpn.ozzmo and com.guidedways.iQuran that used domains in this manner. The former app, version 1.0.6, is a VPN service app, and contains a class that contains a set of IP addresses and domain names. When the class is instantiated, it adds the domain api.ozzmovpn.com, one randomly-selected IP address, and three randomly-selected domain names, to an instance variable. The servers in this variable are then used as the destinations for API requests, with the API key and function names such as signup and admob in the request URL. We verified that control of some of these domains would grant the ability to interact with a portion of these requests. In contrast, the latter app com.guidedways.iQuran version 2.5.4 provides a digital religious text, and offers recorded readings of the text that can be downloaded and played. The app is bundled with several domains that act as download mirrors for redundancy, and the expired domains iphonequran.org and islamicsoftwares.com are mirrors for audio data. When a user opts to download audio files, the app requests the path /downloads/iphone/audiov2.2/0/<num>.tar.gz, caches it in local storage, and plays it aloud. We verified that control of these domains enables pushing arbitrary audio to user devices. Audio content can agitate audiences, or can be intentionally crafted to exploit vulnerabilities in libraries such as audio decoders [65].

Usage: Ads/Tracking. The third category includes domains that are used for advertisements or tracking. Control of the most notable domain in this category, voodoo-ads.io, would position an adversary to expand their influence and maximize their cost efficiency by gaining a foothold into an advertising network used by 111 versions of 49 apps with hundreds of millions of downloads, one of which was even ranked sixth in its arcade game category. We verified that the responses from this domain are used by the app to determine which advertisements are shown to app users: the domain is used as the primary backend in an SDK from the developers, with subdomains such as splash-screen, data-collector, addelivery-engine-api, front-logs, and paths such as /api/.../request-campaign and /request-ad.

Another case involves the app best.beard.photo.editor version 1.0 and the domain melaniapps.com. This app fetches advertisements for other Google Play apps it wishes to advertise from its own domain at /iApp/AdService/getAd.php before integrating with Google Ads to render them. In particular, it expects a JSON response from getAd.php with the keys list and url. The values in list are populated with objects with keys id (app id), nm (app name), and ic (icon to display). Finally, a click listener is attached to the ad view so that it opens the Google Play URL for the app. We verified that the JSON response determines the ads that will be displayed, but discovered that the ads are not actually displayed because of an error in the Google Ads integration.

Usage: Web Content. In contrast to the prior category, obtaining control over domains in this category would enable adversaries to directly change the content inside apps. The single domain we found, wecareapps.net, is used by the apps com.meshref.engpregnancy version 47 and com.meshref.pregnancy version 46, to host web content that is rendered inside the apps. In particular, this app directly loads the URL wecareapps.net/pregnancytracker/... in an embedded WebView that can be accessed from the main navigation menu for the app. In addition to this, the app also directly includes images from the site in the default posts that are generated when pressing the "Share" button for a social media platform. We verified that control of the wecareapps.net domain enables an attacker to send arbitrary HTML content and JavaScript code to this WebView.

Location: `Library`. Aside from the domain voodoo-ads.io which was previously highlighted, the domain cuebiqxxx.com hosts the primary API server for Cuebiq's location data and analytics SDK, and was used in

the app com.weather.radar.forecast.livemaps version 3.0. The domain has since expired, with Cuebiq deprecating their SDK [61], but control of this domain would grant access to user traffic from older apps that still use the SDK. In contrast to these domains used for ads and analytics, the app alpha.hd.anime.girls.wallpapers directly embeds the expired domains koklion.com and kokjoy.com in a library at resources/lib/<arch>/libtest.so. The domain kokmob.com is also found in the same files, although it is not expired. The three domains are labeled with the symbols "mainServer", "mainServerIDx2", and "myADsServer" respectively, and the first two domains are APIs.

5 Footprint Stability

In this section, we present insights on the stability of DNS footprints of apps with respect to time, updates, and emulators.

5.1 App and APK Lifespans

As can be seen from the case studies we presented in Section 4.5, even if only one domain in an app's footprint has expired, the entire app is at risk of compromise. By determining at what time after release apps become at risk of compromise through expired domains in their footprints, we can estimate the lifespan of any particular app or APK.

Figure 3 presents the result of this analysis as a cumulative distribution of all apps and APKs we analyzed, with respect to their DNS footprint lifespan — the number of days, from release, until an eTLD+1 in an app's or APK's footprint expires. Similarly to other figures, the contributions of domains in second-order footprints are represented as another series in the plot. As previously discussed in Section 3, annotations for release dates were aggregated from multiple sources and the earliest was chosen, and expirations were detected by scanning for long gaps in the passive DNS configuration history of the domain. If all domains in an app or APK's footprint never expired after they were first registered, the app or APK contributes to the remaining percentage that totals to 100.

On the one hand, this analysis shows that 98.3% of all APKs, across 95.1% of all apps, that we analyzed are not at risk of compromise via an expired domain in their DNS footprint. On the other hand, even though the percentages may not be large, this means that 1,232 of all APKs, across 738 of all apps, are at risk. Concretely, 0.7% of all APKs, across 2.4% of all apps, have at least one domain in their footprint expire after one month, and this increases to 1.1% of all APKs, across 3.6% of all apps, after twelve months.

Interestingly, the lifespans of APKs and apps are stratified according to the category ranking of the app, as can be seen from the lower two plots in Figure 3. These present the same data as their top counterparts, but with an added dimension of the app category ranking. The individual sample size for each ranking is denoted in the legend, and we observe that for any

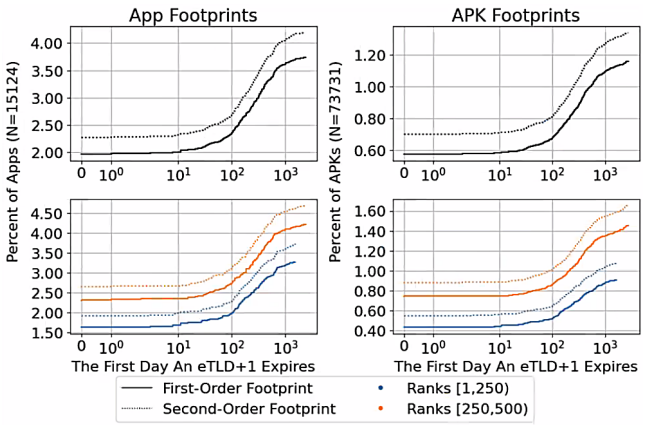

Figure 3: *Computed lifespan of apps. Missing percentages correspond to apps with domains that have never expired or domains with no passive DNS data.*

given amount of time after release, the proportion of lower-ranked apps that experience an expiration in its footprint is greater than that of higher-ranked apps. This trend holds true even with more fine-grained rank groupings (e.g., 5 groups for each 100 ranks), and is caused in part by lower-ranked apps relying on expired domains, on their update day.

Case Study: "Dead-on-Arrival" Domains. In our results, we found numerous instances of APKs contacting domains that were supposedly expired on the release day of the APK, thus contributing to the fraction of apps and APKs at $x = 0$ in Figure 3. After investigation, it appears that some of these instances were false positives, and have been excluded. One case appears to be caused by web services that trigger requests to secondary services. Affected APKs were contacting domains whose first-ever registration was several years after the release date of the APK, suggesting that a live component used by the APK caused this seemingly-contradictory and anachronistic behavior. Examples of these occurrences include the domains liftoff-creatives.io, transparency.google, and app-measurement.com, which were found for 571, 52, and 52 different apps respectively. Another case that was flagged as a false positive involved eTLDs of organizations that automatically provision eTLD+1 domains for users, resulting in a list of eTLD+1s with very high cardinality. Examples of such eTLDs include cloudfunctions.net used in Google Cloud Functions, myshopify.com used in Shopify, and github.io used in GitHub Pages. The provider of our DNS data routinely scrubs data records that are suspected to be a result of DNS wildcard configurations [60], and it appears that their scrubbing algorithm flagged the records of such eTLDs because of the high-cardinality set of subdomains. This hypothesis is supported by the fact that many of these products assign random identifiers as part of the eTLD+1. Other instances are difficult to evaluate in general, but we were able to manually confirm that some appear to be true positives from querying a historical WHOIS product and

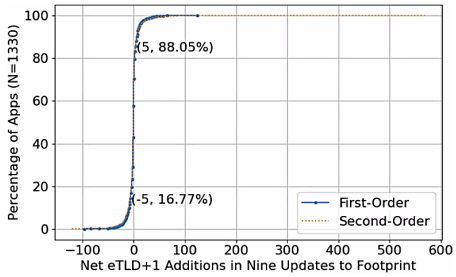

Figure 4: *Cumulative distribution of net eTLD+1 additions in footprints for apps with 10 versions.*

observing a gap during the expected release period, and our manual analysis into the registrable domains at runtime.

5.2 Stability Throughout Updates

It is imperative for developers to not only consider the security of the latest version of their apps, but also the security of older versions. A large-scale study from 2023 found that more than 20% of the users for more than 630K app versions intentionally do not update their apps, and more than 20% of users for 3.5K app versions downgraded their app versions [37]. From the perspective of developers, apps with stable footprints across updates would likely be easier to manage than those with volatile footprints. In this section, we characterize how the footprints of apps change throughout their updates.

We consider only the 1,330 apps for which we were able to successfully download and execute all the latest 10 versions as reported by AndroZoo. This sample was selected in order to present an analysis based on successive, contiguous versions of an app. The other classes of apps — apps for which we were not able to download and execute all the latest 10 versions, and apps that have fewer than 10 versions — are excluded because their versions may not be contiguous.

Net eTLD+1 Additions to Footprints. To start, we investigate how often domains are added to an app's footprint as it receives updates. To do so, we order each APK of an app by its version number and compare the footprint of one version against the next, successively tracking the domains that were introduced, removed, and the same from the footprint of one version to that of the next. We plot the net number of eTLD+1 additions with a cumulative distribution in Figure 4 for both first-order and second-order footprints. This net addition statistic minimizes the potential of execution variability between one version to the next, tracking the overall number of changes in footprint size in each update of an app. As seen from the annotations in the plot, for 71% of apps, they experience up to five net additions to their footprint across their latest 9 updates. Although a change of at most five may seem small, we can compare this to the average DNS footprint size found in Table 2, which is 15.4 for an app and 7.4 for an APK. Thus, for 71% of apps, they experience almost as many net additions to their footprint size as the average footprint size of an app in 9 updates; for 29% of apps, they can experience more changes than the average footprint size.

Case Study: The Guardian (com.guardian). The app responsible for the anomalous value of over 500 net eTLD+1 additions in Figure 4 is The Guardian, an app by the British daily newspaper. In one update — from version `6.90.13685` released on 2022-09-13 to `6.91.14140` released on 2022-09-27 — the app added 40 new eTLD+1 domains to its first-order footprint, resulting in the addition of 535 new eTLD+1 domains to its second-order footprint. These domains remained in the footprints of all subsequent versions we tracked, resulting in the egregious long tail in the second-order series.

Updates vs. Expirations. A major question we sought to answer is whether users can completely prevent any risk of compromise via expired domains if they could always update their apps on the day of a new release. We found several instances that empirically show that it is not possible to completely protect oneself, even if there is an unlikely guarantee of updating apps on the same day as releases. Table 4 enumerates the details for the six instances of domains in APK footprints *expiring before the next version was released, yet the expired domain was still found in the footprint of the next version*. Users of those apps would not have been protected from the risk of using expired domains, even if they immediately updated their apps. Two of the cases reference an eTLD+1 domain whose eTLD is not the same as their TLD — `funny-videos-2018-8b162.firebaseapp.com` and `vhx-notifications.herokuapp.com` — indicating that these are custom public suffixes, and they are located in first-party app code. In particular, the first domain was not found in decompiled Java code, but does appear in a URL (in a DEX file) with the path `/configs-sdk28/`, indicating it may be to fetch a configuration file. The second domain appears in a URL (in decompiled source code) with the path `/api/subscriptions` for a messaging and push notification management class. These domains are associated with the Firebase and Heroku products, where subdomains are uniquely associated with projects, and generally contain random strings in their name to prevent collisions, rendering it difficult or impossible to obtain the same subdomain for another project. However, in Heroku *it is possible to configure a custom and predictable subdomain*, which is why that domain does not have a random substring. This introduces the risk of a subdomain takeover, which is rightfully acknowledged with a warning on the Heroku documentation page [27]. Overall, these findings offer empirical evidence that version updates do not always resolve problems with using dependencies whose domains have expired.

Security of Older Versions. Another question we want to investigate is whether later versions of apps are more secure than older ones. To answer this, we revisit the set of 41 expired, immediately registrable domains that were mentioned in Section 4.3. We filter for the sample of apps that have these

Table 4: *APKs that did not fix an expired domain in the next update. "Previous Release" lists the release date for the latest version of the app when the domain expired, and "Next Release" lists the date for the next update which still used the expired domain. "Next Registration" indicates the date of the next registration after the listed expiration date; an "N/A" value indicates that the domain was still expired at the time of analysis. Downloads were taken from the Google Play Store at the time of writing.*

Package	Ranking	Downloads	Previous Release	Next Release	Domain	Location	Expiration	Next Registration
com.ringpro.popular.freerings	Personalization, 11	10M+	2022-07-06	2022-08-12	funny[...].firebaseapp.com	First-Party	2022-07-21	N/A
com.tasmanic.camtoplanfree	Business, 133	10M+	2022-07-30	2022-10-20	credebat.com	Dynamic	2022-10-10	N/A
com.olympiatv	Health/Fitness, 180	10K+	2020-12-05	2021-06-18	vhx[...].herokuapp.com	First-Party	2021-05-26	N/A
com.smarttools.compasspro	Travel, 202	500K+	2022-03-16	2022-06-19	spirituallovespells.com	Dynamic	2022-04-20	2022-09-24
com.cast.iptv.player	Video Players, 250	50K+	2022-10-11	2022-10-16	royaldocksrainbows.com	Dynamic	2022-10-14	2022-11-23
com.androidwasabi.livewallpaper.christmas	Personalization, 279	500K+	2015-07-03	2015-12-24	cognitivlabs.com	Dynamic	2015-07-26	2015-09-11

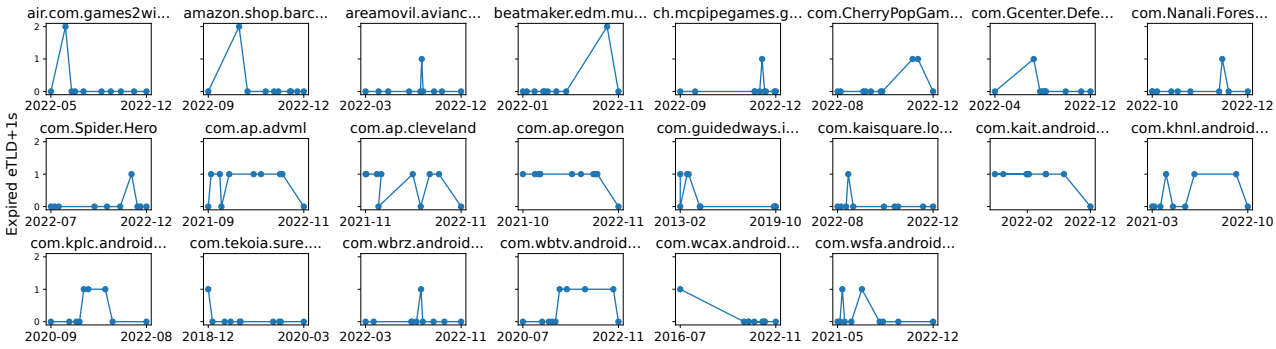

Figure 5: *Apps for which we have footprints for all 10 versions, and used at least one expired domain.*

domains in their footprints, and then filtered one more time for apps for which we successfully extracted footprints from all 10 versions. Then, for each of the remaining 22 apps, we annotate each version with the number of expired domains that were in its footprint. The results are displayed in Figure 5.

We highlight several observations. For some apps, we find that their earlier versions are affected, whereas their latest versions are not (`com.ap.oregon`, `com.kait.android...`, `com.ap.advml...`); however, in other apps, we observe that expired domains are found in the later versions, but not in their earlier ones (`com.wbtv.android...`, `beatmaker.edm...`). In all of these cases, the latest version of each app in this sample are not the ones that are susceptible to the threat of expired domains. Additionally, several apps show successive versions that are susceptible to the same expired domain, suggesting that the footprints of those contiguous versions are more similar as compared to their disjoint ones.

5.3 Footprints Across Emulators

Compared to the Android Studio Emulator used in the infrastructure (Section 3) which is intended for app development, the Cuttlefish emulator is designed to provide behavior that is functionally equivalent to a physical device [5]. We developed a smaller-scale, but equivalent, version of the main infrastructure that uses Cuttlefish to investigate any differences in footprints based on device choice.

We use this version to extract the footprints from two random samples of 500 APKs on a Cuttlefish device running Android 11. The first random sample consists of apps from which the main infrastructure was able to successfully extract footprints, to compare how the footprints extracted from APKs might differ based on whether the apps were launched on the Android Studio Emulator vs. Cuttlefish. The second consists of apps from which the main infrastructure did not successfully extract footprints, to approximate how many more apps could have been analyzed with Cuttlefish.

App Functionality Based on Emulator. First, we wanted to understand what portion of apps might function differently, with respect to the domains they contact. For each APK in the two samples, we aggregate all eTLD+1 domains in their footprints, and label each domain based on the device where it originated: `Emulator` if the domain was only present in the original dataset, `Cuttlefish` if it was only in the Cuttlefish data, or `Shared` if it was found in both datasets. Then, we compute the share of each label out of the total number of unique eTLD+1s for the app. Finally, we assign a class to the APK with threshold t, if there exists a label whose proportion is greater than t. For example, if we observe traffic to domains $\{A,B,C,D\}$, with $\{A,B,C\}$ from the original dataset, and $\{C,D\}$ from the Cuttlefish data, then the app would be labeled `Emulator` for any threshold $t \geq 0.5$. If no label is larger than the threshold, the APK will be classified as `Unclear`.

The results of this analysis, with varying thresholds from 50% to 100%, are summarized in Figure 6. First, we highlight the size of the `Cuttlefish` class, and the presence of the "Originally-Failed" line, as 422 of the 500 APKs in the originally-unsuccessful sample succeeded with Cuttlefish and contributed to the `Cuttlefish` class at every threshold. The originally-successful APKs lie above the "Originally-Failed" line. For these apps, each class is approximately the same size at the 50% threshold, and the `Unclear` class increases in size

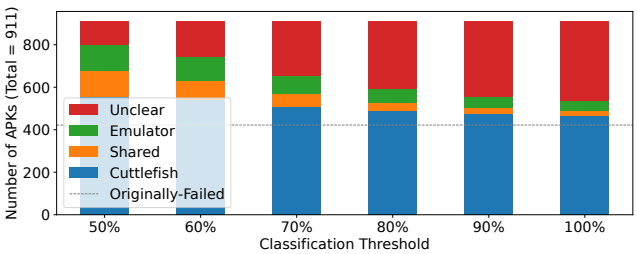

Figure 6: *Class distribution for Cuttlefish samples.*

as the threshold increases. Overall, the classes in the 100% threshold represent a group of apps that does not function on the Android Studio Emulator, another that does not function on the Cuttlefish emulator, a third that functions identically, regardless of the two emulators, and a fourth that may function differently based on the device.

Footprint Differences. Next, we investigate the differences in the footprints. We note that the Cuttlefish infrastructure failed to extract network traffic from 54 of the originally-successful APKs, but we found that 11 of these were caused by an environmental difference. Footprints extracted with the Android Studio Emulator from the main infrastructure included traffic to the domains `alt-*.mtalk.google.com`. These domains are attributed to Google Chat, Firebase Cloud Messaging, and push notifications [40, 52], and they were not filtered out because they were not as ubiquitous as the globally excluded domains described in Section 4.1. There was no traffic to these domains on Cuttlefish, and the original footprints of those 11 APKs consisted of traffic to only these domains. After accounting for this environmental difference, the number of APKs that failed to execute on the Cuttlefish emulators is 43. For the originally-failed sample, 422 of them succeeded on Cuttlefish as previously described.

Next, we quantify the value that the Cuttlefish dataset brings to the original dataset, in terms of new domains that are contacted. The originally-successful sample produced a greater number of unique domains (925) than the originally-failed one (519). Compared to the original 9,525 eTLD+1s in the DNS traffic in the original dataset, these numbers are reduced to 243 and 171 new unique domains respectively for each sample, or 401 new unique domains overall. This corresponds to contributing an additional 4.21% to the set of domains. The originally-successful sample contributed one new expired domain, and the originally-failed one contributed two, each of which were used by one APK.

Overall, the results suggest that Cuttlefish devices are able to launch many apps that the Android Studio Emulator cannot, but most of the domains that are contacted by apps on Cuttlefish are also contacted on the Android Studio Emulator. Domains in Cuttlefish footprints follow a similar long-tail distribution, with a small percentage of popular domains used by a large percentage of apps, and a majority of one-off domains used by individual APKs.

6 Related Work

In this section, we review and discuss differences in other works in broken dependencies and mobile app analysis.

Broken App Dependencies. Our work primarily relates to studies of mobile app dependencies. In the space of Android app dependency analysis, Pariwono et al. conducted a large-scale static analysis of 1.1M Android apps and uncovered that 7.3K apps were associated with 3.7K dangling domain names and hardcoded IP addresses, with 13 of them having been installed more than a million times [53]. Similarly, Mutchler et al. performed a large-scale static analysis study of 1M mobile apps that use WebView, and found that 28% of the sample contained at least one vulnerability, including some that directly load expired domains [48].

Although our study investigates a smaller sample of unique applications as compared to these two works, we do so dynamically with Android emulators, and present longitudinal analyses across time and version history, which enabled us to investigate how the risk of expired domains in DNS dependencies varies with time ad updates. We present statistics on the overall landscape of Android apps regarding the usage of expired domains at any point in time, and shed light on the lifespan of apps with respect to their dependencies. Furthermore, we quantify the additional risk imposed by higher-order links, and report on the update behaviors of apps with respect to their dependencies.

Robust App Dependency Enumeration. Another space focuses on accurate enumeration of mobile app dependencies. Zuo and Lin [75] used symbolic execution to enumerate mobile app backend URLs, and then identified malicious URLs by cross-examining public blocklists. Alrawi et al. [2] builds on the work of Zuo and Lin, and assesses the security stance of the remote servers with vetting techniques. These works study the malicious nature and vulnerabilities of backend servers, whereas our work provides insights into the influence of time and updates on app dependencies, and the extent to which expired domains of dependencies are a threat to apps.

Residual Trust in Other Contexts. Others have also studied this problem at a more general level, beyond the scope of Android apps. Lever et al. coined the term *residual trust* to mean the implicit transfer of the trust held in domain names across changes in domain ownership, and proposed a defense mechanism to identify changes in domain ownership [35]. So et al. found that there is great potential for abuse of residual trust without requiring heavy financial investment. After re-registering previously-popular expired domains and configuring honeypot services, their domains attracted requests from millions of IP addresses, and they were able to determine the type of services that were previously hosted on more than half of them [63]. Liu et al. investigated a contributor to residual trust: dangling DNS records. They identified three new attack vectors in a large-scale measurement study, and uncovered hundreds of such dangling records in the top 10K

Alexa and edu domains [38]. Similarly, Borgolte et al. found that it was time- and cost-efficient to allocate IP addresses to which dangling DNS records resolve on public clouds [17]. There are also studies of exploitability of expired domains in more specific domains: email [26], remote JavaScript [50], and Content Security Policies [58].

These works quantify the exploitability and severity of the consequences of expired domains and residual trust in general or in specific contexts. In comparison, the core of the problem that our work focuses on is the inherent incompatibility because of the dual nature of Android apps that contain both locally-installed static and Internet-accessing dynamic components, with respect to time. If an app is not kept up to date, the dynamic components may attempt to access services that change or no longer exist, thus manifesting as a similar problem that is studied in these related works.

Mobile App Analysis. Another major line of related work is analysis of mobile applications. Program analysis techniques can be broadly categorized as static or dynamic, depending on whether they execute the source code. Both categories have their strengths and weaknesses, and one may do better than the other in certain contexts, typically leading to efforts to design approaches that leverage both types of analysis. In this work, we relied on Li et al.'s DroidBot [36], which generates state transition models on the fly to produce UI-guided inputs, to execute the mobile apps in a sandboxed environment to extract DNS dependencies at runtime. Other similar approaches include SmartDroid [74], a hybrid static and dynamic approach to reveal UI-based trigger conditions, IntelliDroid [72], a generic Android input generator that can be tailored to specific dynamic analysis tools, and DroidMate 1 [32] or 2 [16], input generation tools that were designed to be robust, extensible, and easy to use.

Higher-Order Links. The approach that we use in this work is based on extracting the network footprints of APKs, and we include domains found in HTML responses to our apps as second-order links. There is a large body of prior work that investigates the security risks posed by techniques such as WebView, which is the source of our second-order links. Luo et al. conducted one of the first studies on the security implications of using WebViews as lightweight, in-app browsers in 2011, discovering that malicious WebView pages can attack benign apps, and vice versa [39], and this line of WebView vulnerability analyses continues today, with a recent study focusing on identity confusion in WebView-based app-in-app ecosystems [73]. Another work that suggests the importance of higher-order links in the attack surface of an app is a study conducted by Sanchez-Rola et al. which focuses on the opaque interactions that occur when clicking web elements, often performing undesired actions [59].

Passive DNS. Another main focus of our approach is in deducing periods of expiration for the extracted domains using passive DNS data. Prior works have relied on passive DNS data for many tasks, such as discovering malicious domains [15, 33] and identifying IoT devices [55]. Similar in nature, Barron et al.'s study of early domain deletions [14] is similar in nature to our methodology of identifying expiration periods and face similar challenges, except with historical WHOIS data at its base. Additionally, other works use the (in)stability of DNS infrastructure for security-related detections, such as in the design of the Alembic algorithm by Lever et al. [35] to detect domain ownership changes.

Our work focuses on quantifying the impact of the threat of expired domains on the mobile app ecosystem via longitudinal analyses. We do not extensively focus on the vulnerabilities, and types of security risks, that can occur from rendering web content in an app, or characterize domain expirations in depth; instead, we centered our analyses on quantifying the fraction of the mobile app ecosystem that is at risk of compromise via supply chain attacks on their network dependencies.

7 Discussion

In this work, we systematically quantified the risk of the use of expired domains in a large-scale analysis of the dependencies of 77,206 versions of 15,124 unique Android apps that were well-distributed across categories, ranks, and versions, using a scalable and efficient dynamic analysis runtime and temporal analysis methodology. We reported on insights regarding expired domain dependencies both at a snapshot in time, and longitudinally throughout time and app updates by leveraging passive DNS data.

Summary of Key Findings. Our analyses demonstrate that, on average, several hundred footprint domains are expired, and such domains are used by 2.5% of APKs or 4.2% of apps at any point in time. We uncovered that 41 domains in the footprints of 309 APKs across 149 apps were publicly registrable at runtime. Such apps included those that were ranked within the top ten of their categories, with more than one hundred million downloads, and occasionally were the latest versions. Furthermore, by leveraging passive DNS data, we estimate that 0.7% of APKs have at least one domain in their footprint expire within one month of their release. The footprints of apps can experience drastic change across updates: 29% of apps experience approximately the same number of changes to their footprint as the mean footprint size of all apps, in 9 updates. Finally, we find that updates do not always fix problems with expired domains in previous versions, even for highly-ranked apps with over ten million downloads. In addition, the latest app version appears more secure than any of its older versions.

7.1 Limitations & False Positives

We acknowledge that our methodology is constrained by several factors, but we mitigate their impact and argue that our

high-level findings stand true regardless of these limitations. The main limitations of our work are the data sources and footprints, and that we do not have ground truth with which to evaluate our proposed methodology for detecting expirations.

App Sample. Although we did not use different emulator configurations to increase app coverage, or account for update frequency in the app sample, the resulting dataset is well-distributed across categories, rankings, and number of versions, suggesting that there is no bias to a specific class of apps, and the possibility for larger footprints only frames the findings reported in our study as lower bounds.

Dynamic Analysis. Another limitation is that our approach only uses dynamic analysis, and we are unable to bypass splash screens or login screens, which may have prevented us from seeing more "interesting" domains that may have been uncovered with static analysis. Additionally, dynamic analysis of older versions of apps can result in DNS footprints that are different from their footprints at release. However, our footprints are accurate representations of the supply chain of apps, at their time of extraction. It is well-known that users do not always update their apps, and footprints of older app versions are representative of this subset.

Capturing Traffic. A different point of concern may be about the traffic we captured. Our HTTPS proxy was configured via an emulator setting, instead of via transparent proxying (e.g., `iptables`) so it does not intercept HTTPS traffic from custom network clients that disregard system-level proxy configurations, but the captured traffic was sufficient for our analyses. Also, we disabled certificate pinning, and although it is possible for certificate pinning to defend against domain re-registration attacks, its disadvantages outweigh its benefits, which has caused it to be deprecated in multiple contexts, including in Android [7, 44]. Furthermore, our results are obtained from the Android Studio Emulator instead of physical devices, but we are confident in our overall results because of the results from the Cuttlefish experiments (§5.3).

Deducing Expirations. Our approach that deduces expirations from passive DNS records does not have ground truth. However, Farsight is one of the leading commercial providers in the industry, we empirically verified the accuracy and granularity of their data (described in depth in Appendix C), and accounted for "dead-on-arrival" cases in Section 5.1. In general, because we incorporate the context of the domain lifecycle into our fundamental approach by scanning for long gaps in DNS configurations that are at least one year apart, we limit the number of detected expirations to one per year. Furthermore, the high-level results from our passive DNS approach aligns with the results we obtained from checking for re-registrable, expired domains at runtime. Appendix E further discusses the shortcomings of historical WHOIS data.

7.2 Looking Forward

Our work demonstrates that the threat of expired domains is ever-present, particularly given the nature of the mobile app ecosystem in which outdated versions are commonplace among end users. Dependency management should be an integral part of the development and maintenance lifecycle. Ideally, with the appropriate infrastructure and code logic, developers should verify the integrity of dependencies before attempting to connect and send data. Unfortunately, past works that have studied the impact of expired domains found that outdated software tend to blindly attempt to connect to a static list of remote dependencies with potentially private information of end users [63]. Furthermore, developers often use third-party libraries that dynamically pull in additional dependencies (e.g., advertising libraries for in-app monetization), making it difficult to map out all dependencies.

In light of our finding that updates do not always prevent an app from using an expired domain, we argue that the introduction of version control logic is a step in the right direction, but instead of merely ensuring that end users have installed the latest version of an app, mechanisms that can thoroughly map all dependencies, and ensure their integrity, are necessary to safeguard against supply chain attacks.

8 Conclusion

In this paper, we quantified the threat of expired domains with a series of systematic analyses derived from dynamically executing numerous APKs and apps. We extracted the eTLD+1 domain dependencies of each APK by capturing its DNS and HTTPS traffic generated from its interaction with a UI-guided test input generator, and studied the stability of such dependencies across time and version updates. During app execution, we uncovered expired domains that were used by hundreds of APKs and apps. By devising a methodical approach to deduce domain expirations from passive DNS data, we performed longitudinal analyses corroborating this initial finding. Overall, we found that expired domains can affect thousands of APKs, and hundreds of apps, at any given point in time, that app updates do not always prevent this from occurring, or fix this problem from a previous version, and that older app versions are equally insecure. Additionally, we reported evidence in the form of concrete case studies that demonstrate the potential for abusing expired domains used by mobile apps. We argue that new robust mechanisms that can manage dependencies and provide integrity guarantees — as opposed to merely forcing updates — are necessary.

Acknowledgments. We thank the anonymous reviewers and shepherd for their help, as well as Farsight and WhoisXML API for providing research access to their products. This work was supported by the National Science Foundation (NSF) under grants CNS-2211575, CNS-2126654, and CNS-1941617.

9 Ethics considerations

Our work conducted a large-scale analysis of the dependencies of apps by downloading them and dynamically exploring them with a UI-guided test input generator. As such, all interactions with apps follow a generic behavior that does not attempt targeted execution flows for specific apps, nor generate highly-realistic, human-like interactions with backend services. We did not attempt to bypass any login or splash screens of apps, and our infrastructure interacted with each APK for a total of nine minutes. Thus, we are confident that our dynamic analysis of apps did not negatively impact the contacted backend services. We did not register any expired domains and therefore never interacted with potential users of the affected APKs (we manipulated our testing environment for the investigations presented in Section 4.5).

Disclosure of our findings was complicated by the fact that we finished our analyses more than six months after the initial data collection, and that there is no immediately obvious purpose in contacting app developers about outdated versions of their software having been susceptible to the use of expired domains in the past (particularly for the analyses with passive DNS). We instead opted to contact the developers of the apps affected by the 41 registrable domains we found at runtime. Only one affected app developer replied: most of our disclosures were redirected to automated support ticket customer support systems, and these tickets were automatically closed with no response. The one contact that responded asked for more information to relay to the relevant app developers, but never followed up.

10 Open science

The original research artifacts cannot be made available to the public as they are governed by the terms of the first author's employment, but the authors have released alternative artifacts that are not subject to the same restrictions at https://doi.org/10.5281/zenodo.14737144.

Restrictions. The source code for the infrastructure that tests apps and records their network traffic as described in Section 3.3 and Figure 1 cannot be made available to the public, as it was completed during the first author's employment in the research and development division of a cybersecurity company. In addition, the raw app dataset also cannot be released as it was compiled through the use of a licensed VirusTotal Enterprise API key, and the VirusTotal Enterprise Agreement forbids the public sharing of samples obtained from their API. The authors also used telemetry data from the company's commercial products to assist in determining the first seen date for APKs. Thus, the open-sourcing of these two artifacts (infrastructure and raw app dataset) is not permissible.

Artifacts. On the other hand, we are able to open-source artifacts that are not governed by these policies. Five artifacts that will be shared are detailed below. The first artifact will be a tabular dataset that identifies individual APKs with metadata (e.g., app name, version, and hash) and their extracted DNS footprints. The second artifact will be a testing environment used in Section 4.5 that replicates the original infrastructure shown in Figure 1 that can be used to manually interact with individual APKs and record the resultant network traffic on a headful display. This comprises Docker containers for a DNS server and a proxy to man-in-the-middle the network traffic from the emulator, and scripts to configure the emulator to trust the proxy. In addition, we will release the Jupyter notebooks used to select the input sample of APKs as the third artifact, and the analysis notebooks used to produce the figures presented in the text based on the tabular, processed dataset as the fourth artifact. We will also share the alternative setup used to extract footprints from apps on Cuttlefish builds used for Section 5.3, and the notebooks that compare them to the originally extracted footprints from the Android Studio Emulator as the fifth artifact.

References

[1] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471. IEEE, 2016.

[2] Omar Alrawi, Chaoshun Zuo, Ruian Duan, Ranjita Pai Kasturi, Zhiqiang Lin, and Brendan Saltaformaggio. The betrayal at cloud city: An empirical analysis of {Cloud-Based} mobile backends. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 551–566, 2019.

[3] Android. Android debug bridge (adb). https://developer.android.com/tools/adb.

[4] Android. Contextcompat. https://developer.android.com/reference/androidx/core/content/ContextCompat.

[5] Android. Cuttlefish virtual android devices | android open source project. https://source.android.com/docs/devices/cuttlefish.

[6] Android. In-app updates. https://developer.android.com/guide/playcore/in-app-updates.

[7] Android. Security with network protocols. https://developer.android.com/privacy-and-security/security-ssl.

[8] Android. Sslcontext. https://developer.android.com/reference/javax/net/ssl/SSLContext.

[9] Android. Windowmanager.layoutparams. https://developer.android.com/reference/android/view/WindowManager.LayoutParams.

[10] APKMirror. Apkmirror - free apk downloads - free and safe android apk downloads. https://www.apkmirror.com/.

[11] APKPure. Download apk on android with free online apk downloader - apkpure. https://m.apkpure.com/.

[12] Apple. Apple developer program license agreement. https://developer.apple.com/support/terms/apple-developer-program-license-agreement/.

[13] Waqar Aziz. 7 developer best practices for app updates. https://developer.amazon.com/apps-and-games/blogs/2023/02/app-updates-best-practices, 2023.

[14] Timothy Barron, Najmeh Miramirkhani, and Nick Nikiforakis. Now you see it, now you {Don't}: A large-scale analysis of early domain deletions. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 383–397, 2019.

[15] Leyla Bilge, Engin Kirda, Christopher Kruegel, and Marco Balduzzi. Exposure: Finding malicious domains using passive dns analysis. In *Ndss*, pages 1–17, 2011.

[16] Nataniel P Borges Jr, Jenny Hotzkow, and Andreas Zeller. Droidmate-2: a platform for android test generation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 916–919, 2018.

[17] Kevin Borgolte, Tobias Fiebig, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Cloud Strife: Mitigating the Security Risks of Domain-Validated Certificates. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS)*.

[18] Laura Ceci. Frequency of application updates among smartphone owners in the united states, as of 2016. https://www.statista.com/statistics/747569/united-states-survey-smartphone-users-app-update-frequency/.

[19] Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. mitmproxy: A free and open source interactive HTTPS proxy, 2010–.

[20] Data.ai. Number of mobile app downloads worldwide from 2016 to 2022 (in billions). https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/, 2023.

[21] Farsight. Passive dns historical internet database: Farsight dnsdb. https://www.farsightsecurity.com/solutions/dnsdb/.

[22] Google. Configure the app module. https://developer.android.com/build/configure-app-module.

[23] Google. Device and network abuse - play console help. https://support.google.com/googleplay/android-developer/answer/9888379?sjid=9339651112128996414-NA.

[24] Google. Last modified timestamp on all files in apk default to 'fri, nov 30 1979 00:00:00' [37116029] - issue tracker. https://issuetracker.google.com/issues/37116029, 2016.

[25] Google. Virustotal enterprise. https://assets.virustotal.com/vt-360-outcomes.pdf, 2021.

[26] Daniel Gruss, Michael Schwarz, Matthias Wübbeling, Simon Guggi, Timo Malderle, Stefan More, and Moritz Lipp. Use-after-freemail: Generalizing the use-after-free problem and applying it to email services. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 297–311, 2018.

[27] Heroku. Review apps - selecting the url pattern. https://devcenter.heroku.com/articles/github-integration-review-apps.

[28] ICANN. Centralized zone data service. https://czds.icann.org/home.

[29] ICANN. Epp status codes | what do they mean, and why should i know? https://www.icann.org/resources/pages/epp-status-codes-2014-06-16-en, 2014.

[30] ICANN. Expired registration recovery policy. https://www.icann.org/en/resources/registrars/consensus-policies/errp, 2024.

[31] ISC. Bind 9 - isc. https://www.isc.org/bind/.

[32] Konrad Jamrozik and Andreas Zeller. Droidmate: a robust and extensible test generator for android. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, pages 293–294, 2016.

[33] Issa Khalil, Ting Yu, and Bei Guan. Discovering malicious domains through passive dns data graph analysis. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 663–674, 2016.

[34] InConcept Labs. How to force update a mobile app when a new version is available. https://www.linkedin.com/pulse/how-force-update-mobile-app-when-new-version-available-/.

[35] Chaz Lever, Robert Walls, Yacin Nadji, David Dagon, Patrick McDaniel, and Manos Antonakakis. Domain-z: 28 registrations later measuring the exploitation of residual trust in domains. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 691–706. IEEE, 2016.

[36] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 23–26. IEEE, 2017.

[37] Fuqi Lin, Xuan Lu, Wei Ai, Huoran Li, Yun Ma, Yulian Yang, Hongfei Deng, Qingxiang Wang, Qiaozhu Mei, and Xuanzhe Liu. Adoption of recurrent innovations: A large-scale case study on mobile app updates. *ACM Transactions on the Web*, 18(1):1–26, 2023.

[38] Daiping Liu, Shuai Hao, and Haining Wang. All your dns records point to us: Understanding the security threats of dangling dns records. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1414–1425, 2016.

[39] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. Attacks on webview in the android system. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 343–352, 2011.

[40] Detlef M. Mtalk.google.com managing my play services, what can i do to end this? https://support.google.com/pixelphone/thread/213101183/mtalk-google-com-managing-my-play-services-what-can-i-do-to-end-this?hl=en.

[41] Arunesh Mathur and Marshini Chetty. Impact of user characteristics on attitudes towards automatic mobile application updates. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, pages 175–193, 2017.

[42] Lauren McCormack. Mobile app download. https://buildfire.com/app-statistics/, 2023.

[43] MDN. etld. https://developer.mozilla.org/en-US/docs/Glossary/eTLD.

[44] MDN. Http public key pinning (hpkp). https://developer.mozilla.org/en-US/docs/Web/HTTP/Public_Key_Pinning.

[45] Microsoft. Codepush. https://github.com/microsoft/code-push.

[46] Andreas Möller, Florian Michahelles, Stefan Diewald, Luis Roalter, and Matthias Kranz. Update behavior in app markets and security implications: A case study in google play. In *Research in the Large, LARGE 3.0: 21/09/2012-21/09/2012*, pages 3–6, 2012.

[47] Mozilla. Public suffix list. https://publicsuffix.org/, 2005.

[48] Patrick Mutchler, Adam Doupé, John Mitchell, Chris Kruegel, and Giovanni Vigna. A large-scale study of mobile web app security. In *Proceedings of the Mobile Security Technologies Workshop (MoST)*, volume 50, 2015.

[49] Maleknaz Nayebi, Bram Adams, and Guenther Ruhe. Release practices for mobile apps – what do users and developers think? In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 552–562, 2016.

[50] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 736–747, 2012.

[51] Gergely Orosz. Force upgrading for mobile apps. In *Building Mobile Apps at Scale: 39 Engineering Challenges*. Primedia E-launch LLC, April 2021.

[52] p1r473. mtalk.google.com required by many android apps - issue 2 - th3m3/blocklists. https://github.com/Th3M3/blocklists/issues/2.

[53] Elkana Pariwono, Daiki Chiba, Mitsuaki Akiyama, and Tatsuya Mori. Don't throw me away: Threats caused by the abandoned internet resources used by android apps. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 147–158, 2018.

[54] pcipolloni. Frida codeshare: Universal android ssl pinning bypass with frida. https://codeshare.frida.re/@pcipolloni/universal-android-ssl-pinning-bypass-with-frida/.

[55] Roberto Perdisci, Thomas Papastergiou, Omar Alrawi, and Manos Antonakakis. Iotfinder: Efficient large-scale identification of iot devices via passive dns traffic analysis. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 474–489. IEEE, 2020.

[56] Prashanth Rajivan, Efrat Aharonov-Majar, and Cleotilde Gonzalez. Update now or later? effects of experience, cost, and risk preference on update decisions. *Journal of Cybersecurity*, 6(1):tyaa002, 2020.

[57] Ole André V. Ravnås. Frida - a world-class dynamic instrumentation toolkit. https://frida.re/.

[58] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis, and Ben Stock. Complex security policy? a longitudinal analysis of deployed content security policies. In *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS)*, 2020.

[59] Iskander Sanchez-Rola, Davide Balzarotti, Christopher Kruegel, Giovanni Vigna, and Igor Santos. Dirty clicks: A study of the usability and security implications of click-related behaviors on the web. In *Proceedings of The Web Conference 2020*, pages 395–406, 2020.

[60] Joe St Sauver. Enhancing dnsdb to better handle dns wildcard names. https://www.domaintools.com/resources/blog/enhancing-dnsdb-to-better-handle-dns-wildcard-names/.

[61] Allison Schiff. Location intel provider cuebiq is shutting down its sdk in the name of privacy. https://www.adexchanger.com/mobile/location-intel-provider-cuebiq-is-shutting-down-its-sdk-in-the-name-of-privacy/.

[62] skylot. skylot/jadx: Dex to java decompiler. https://github.com/skylot/jadx.

[63] Johnny So, Najmeh Miramirkhani, Michael Ferdman, and Nick Nikiforakis. Domains do change their spots: Quantifying potential abuse of residual trust. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2130–2144. IEEE, 2022.

[64] statcounter. Desktop vs mobile vs tablet market share worldwide. https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide/2023, 2023.

[65] Check Point Research Team. Largest mobile chipset manufacturers used vulnerable audio decoder, 2/3 of android users' privacy around the world were at risk. https://blog.checkpoint.com/security/largest-mobile-chipset-manufacturers-used-vulnerable-audio-decoder-2-3-of-android-users-privacy-around-the-world-were-at-risk/.

[66] VirusTotal. Virustotal api v3 overview. https://developers.virustotal.com/reference/overview.

[67] Florian Weimer. Passive dns replication. In *FIRST conference on computer security incident*, volume 98, pages 1–14, 2005.

[68] Jimmy Westenberg. We asked, you told us: 65 percent of you throw caution to the wind and auto-update apps. https://www.androidauthority.com/auto-update-apps-google-play-store-poll-results-1077403/, 2020.

[69] WhatWG. Url standard. https://url.spec.whatwg.org.

[70] WHOISDataCenter. Whoisdatacenter.com. https://whoisdatacenter.com/.

[71] WhoisXMLAPI. Whois history. https://whois-history.whoisxmlapi.com/.

[72] Michelle Y Wong and David Lie. Intellidroid: a targeted input generator for the dynamic analysis of android malware. In *NDSS*, volume 16, pages 21–24, 2016.

[73] Lei Zhang, Zhibo Zhang, Ancong Liu, Yinzhi Cao, Xiaohan Zhang, Yanjun Chen, Yuan Zhang, Guangliang Yang, and Min Yang. Identity confusion in {WebView-based} mobile app-in-app ecosystems. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1597–1613, 2022.

[74] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 93–104, 2012.

[75] Chaoshun Zuo and Zhiqiang Lin. Smartgen: Exposing server urls of mobile apps with selective symbolic execution. In *Proceedings of the 26th International Conference on World Wide Web*, pages 867–876, 2017.

Appendices

This section provides several detailed excerpts for interested readers. Appendix A details DNS record configurations; Appendix B describes over-the-air updates; Appendix C explains the empirical verifications of the approach described in Section 3.4; Appendix D discusses the even distribution of the final app sample; and Appendix E outlines the shortcomings of historical WHOIS databases.

A DNS Configurations

When a domain is purchased, a registrar typically provisions default DNS records. During the registration period, owners are free to reconfigure the DNS records as they please. The NS record type is one of the records that are automatically provisioned, and it specifies the authoritative name servers for the domain, which is furnished to the gTLD servers as *DNS glue records* to prevent circular dependencies if the authoritative name servers are subdomains of the root domain. The managing authorities publish these glue records in their daily zone files, essentially serving as the ground truth of whether a

particular domain had their DNS settings configured for that day. Participating gTLDs publish their *current zone files every day* on the Centralized Zone Data Service [28].

After a domain expires, control is passed back to the registrar. For gTLDs under the ERRP discussed in Section 2, it is possible for a domain to have DNS records configured while it is expired: registrars may "interrupt" the existing DNS resolution of a domain after it has passed the expiration date. This interruption can take various forms, including modifying the DNS resolution path to point to a web page with domain renewal instructions. However, upon deletion by the registrar, the domain enters the 30-day Redemption Grace Period before being released to the public, and there must not exist any DNS records during this time [30]. Thus, we can make the following assertion about any domain at time t: if a domain was configured with a DNS record at time t, it was not publicly registrable at that time, either because it was registered and in good standing, or it was expired but not yet deleted by the registrar. Further, if a domain was not configured, this can be because: the domain was registered and in good standing, but there was no active DNS record at time t, or the domain was not registered at time t, and thus could not have had an active DNS record at time t.

B Mobile Apps: Over-the-Air Updates

Android apps are written in a programming language, and then bundled into APK files for marketplaces to distribute, and for devices to install. For every installed app, there exists a static set of source code files and assets on the local device. Depending on the app structure, it may be possible for apps to perform certain updates without interacting with the app marketplace. The CodePush service [45] for React Native apps provides this feature as *over-the-air updates*: the ability to update certain assets (e.g., bundled JavaScript) over the air without interacting with the app marketplace. However, this feature is restricted to assets and not core source code files: official app marketplaces such as the iOS App Store and Google Play explicitly prohibit apps from downloading or installing executable code to maintain the integrity of apps after they have been reviewed [12, 23]. At its core, this ability is implemented in the same manner as the ad-hoc version-control logic as described earlier: by checking local assets against the latest assets stored on a server. In the context of this work, we consider the version-control server as another dependency of the app (i.e., part of its DNS footprint).

C Additional Discussion for Identifying Expirations from Passive DNS

To support our methodology of identifying expirations in passive DNS data, we conduct several small-scale, empirical analyses on a set of 214 domains for which we previously registered, and thus have access to at least one registration period and two expiration periods. The goal is to evaluate the accuracy and reliability of the passive DNS data, and whether Algorithm 1 incurs false positives.

Algorithm 1 Inferring domain expirations (see Section 3.4)

```
RRTYPES ← [A,AAAA,CNAME,NS,SOA,MX,TXT]
MinGapDays ← 35
procedure UnionIntervals(itvls)
                                   ▷ Union of all overlapping intervals
    sorted ← sort(itvls)                          ▷ Sorted by start time
    stack ← []
    if len(itvls) ≥ 1 then
        stack.append(itvls[0])
        for itvl ∈ itvls[1:] do
            if Overlaps(stack[−1], itvl) then
                ProcessOverlap(stack, itvl)
            else
                stack.append(itvl)
            end if
        end for
    end if
    stack ← sort(stack)
    return stack
end procedure
procedure ProcessPDNS(d)
                                   ▷ Passive DNS configuration intervals
    itvls ← {}
    raw ← QueryPDNS(d)
    for r ∈ raw do
        if r.rrtype ∉ RRTYPES then continue
        end if
        itvl ← [r.day_first, r.day_last]
        itvls[r.rrtype].append(itvl)
    end for
    final ← []
    for rrtype ∈ itvls do
        final.extend(UnionIntervals(itvls[rrtype]))
    end for
    return UnionIntervals(final)
end procedure
procedure InferExpirations(d)
    itvls ← ProcessPDNS(d)
    gaps ← FilterForGaps(itvls, MinGapDays)
    gaps ← AdjustGapsMinOneYearApart(gaps)
    return gaps
end procedure
```

Data Reliability. To verify the reliability of the data, we verify that the beginning of passive DNS intervals correspond to registration times — or when DNS records are configured — and that the end correspond to the lack of DNS records. Regarding the former, all 214 domains had passive DNS records that began within one day of registration. For the latter, we cross-referenced identified gap periods from passive DNS against the DNS zone files published by their registrars. We were able to download the daily zone files from the TLDs of 160 domains starting from a few days before their expiration dates. We confirmed that all 160 domains disappeared from their zone files during the identified gap periods in their passive DNS data. In short, this empirically confirms that identified gaps in passive DNS data are caused by a lack of DNS configuration in the specified time windows, and not because of poor data collection infrastructure from our provider.

False Positives. False positives would occur if our methodology detected domain expirations that never happened. We

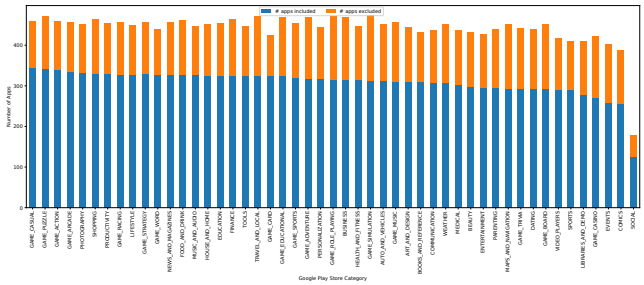

Figure 7: *Number of apps found, and successfully executed, in each of the 49 categories on the Google Play Store.*

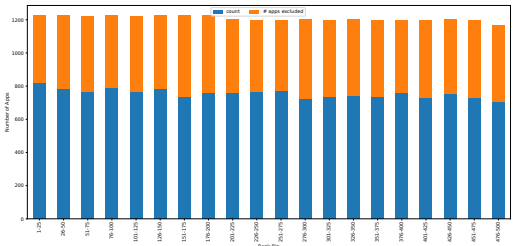

Figure 8: *Number of apps found, and successfully executed, in each of the 49 categories on the Google Play Store.*

found that all detected expirations from the passive DNS data for the 214 domains align with their actual expirations.

False Negatives. False negatives would occur if Algorithm 1 fails to detect actual expirations. After computing the gaps in the DNS configuration intervals for the 214 domains, we observe that there is a gap within the 43 days following expiration for 89.7% of the domains. The exact length of this delay depends on the policies for each registrar and its exact value is not important, but it does show that identified expirations via gap periods in passive DNS data occur soon after the actual expiration date. The remaining 22 domains manifested as false negatives with unexpected, large values (over 60 days). Upon investigation of these domains, we found default NS records configured by the registrar (Dynadot) with years-long durations of time between their last seen and first seen timestamps, well beyond the domain registration and expiration window, resulting in the inability of our approach to detect a gap at the time of expiration. The time windows provided by these NS records subsumed the shorter and more fine-grained time windows that aligned with the registration period, resulting in long registration intervals that did not correspond to our actual registration and expiration dates for these domains. In the context of this study, we argue that false negatives do not take away from our results — our findings present a lower bound, and worst-case, analysis on the exploitability of the DNS footprints of apps.

D Data Sample Representation

Although the fraction of initial apps and of initial APKs may seem low, the apps (and corresponding versions) which are included are nearly-uniformly distributed across category, and across rankings within each category, so the dataset remains a representative sample for the purposes of our analyses. See Figure 7 for a visual representation of the number of apps included, and excluded, in the final dataset for every category of the Google Play Store. Our approach successfully extracted the footprints of more than 300 apps for each category — except for the Social category which had less than 300 to begin with, but has a similar inclusion proportion — so our sample is still representative of each category. For a similar distribution of apps across ranking bins, see Figure 8, which indicates that our dataset comprises a representative sample across ranks.

Regarding distribution of versions for apps, there is a significant portion of apps with at least 10 different versions in the original set of apps downloadable from VirusTotal. Although the final dataset does not include all 10 latest versions of such apps, it retains as many versions of them as possible, resulting in increases in the number of apps with other numbers of versions. In the final dataset, the number of apps with x versions is over 1,000 for every x in [1,10]. Furthermore, we found no bias towards more popular apps having a higher number of versions.

E Historical WHOIS

Historical WHOIS databases periodically query WHOIS records of domains, which contain registration and expiration dates. Regardless if a historical WHOIS service uses a one-time fee pricing model to provide a download of a snapshot of their database, or a pay-as-you-go model with API access, it would cost thousands of dollars to retrieve data for the number of expired domains we discovered [70, 71]. Furthermore, similar concerns of reliability exist: failed queries induce false positives when identifying domain expirations.

We obtained limited research access to WhoisXML API [71] and evaluated 25 of the domains from Appendix C. We found that our automated analyses were not sufficient; manual review was required because registrars preemptively renew expiring domains with their registries, and accordingly extend the expiration date of the WHOIS record. If the domain owner does not renew their domain, the domain is deleted and the registrar resets the WHOIS expiration date. This behavior results in temporary records with extended expiration dates persisting in historical WHOIS databases, and obstructs our methodology similarly to the long NS records. Thus, in addition to its costly nature, WHOIS analysis is not scalable without automatically excluding these WHOIS records, which vary by registrar.

Lost in the Mists of Time: Expirations in DNS Footprints of Mobile Apps

Johnny So
Stony Brook University

Iskander Sanchez-Rola
Norton Research Group

Nick Nikiforakis
Stony Brook University

A Artifact Appendix

This artifact appendix is meant to be a self-contained document which describes a roadmap for the evaluation of your artifact. It should include a clear description of the hardware, software, and configuration requirements. In case your artifact aims to receive the functional or results reproduced badge, it should also include the major claims made by your paper and instructions on how to reproduce each claim through your artifact. Linking the claims of your paper to the artifact is a necessary step that ultimately allows artifact evaluators to reproduce your results.

Please fill all the mandatory sections, keeping their titles and organization but removing the current illustrative content, and remove the optional sections where those do not apply to your artifact.

A.1 Abstract

In this work, we present the first large-scale analysis of mobile app dependencies through a dual perspective accounting for time and version updates, with a focus on expired domains. First, we detail a methodology to build a representative corpus comprising 77,206 versions of 15,124 unique Android apps. Next, we extract the unique eTLD+1 domain dependencies — the "DNS footprint" — of each APK by monitoring the network traffic produced with a dynamic, UI-guided test input generator and report on the footprint of a typical app. Using these footprints, combined with a methodology that deduces potential periods of vulnerability for individual APKs by leveraging passive DNS, we characterize how apps may have been affected by expired domains throughout time. Our findings indicate that the threat of expired domains in app dependencies is nontrivial at scale, affecting hundreds of apps and thousands of APKs, occasionally affecting apps that rank within the top ten of their categories, apps that have hundreds of millions of downloads, or apps that were the latest version. Furthermore, we uncovered 40 immediately registrable domains that were found in app footprints during our analyses, and provide evidence in the form of case studies as to their potential for abuse. We also find that even the most security-conscious users cannot protect themselves against the risk of their using an app that has an expired dependency, even if they can update their apps instantaneously.

As part of the artifact evaluation, we release datasets and analysis notebooks that enable reviewers to reproduce the figures and tables that are presented in the text. Additionally, we release a version of the main app analysis infrastructure to enable future work.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There are no security, privacy, and ethical concerns associated with running these artifacts. In terms of running processes, running the Jupyter notebooks requires a running Jupyter server, and running the Cuttlefish infrastructure spawns various disposable components. However, we do recommend using a local firewall (e.g., ufw) if running the Cuttlefish infrastructure, as it will spawn Cuttlefish virtual devices that can be manipulated over the network via adb. The only host-level settings that are modified come from the Cuttlefish infrastructure, which provides scripts to manipulate iptables and ufw rules (`iptables.sh` and `ufw_modify_cvd.sh` in the `/cuttlefish/scripts` directory, respectively). These scripts enable all traffic from the Cuttlefish virtual devices to correctly pass through ufw (if it is enabled), and route DNS traffic from the Cuttlefish devices to a specified DNS server.

A.2.2 How to access

We provide access to our artifact on Zenodo at the following link: https://doi.org/10.5281/zenodo.14737144. The artifact will be updated with new versions at this URL according to the discussion of the artifact evaluation period.

A.2.3 Hardware dependencies

The only hardware requirements are imposed by the Android Cuttlefish infrastructure components, which require CPU support for `kvm`.

For convenience, we provide reviewers with access to a VM which we have pre-configured with all necessary dependencies, and installed the artifacts at `/home/ubuntu/artifacts`. In addition, this VM comes with the sample of APKs used for the Cuttlefish experiments (which unfortunately cannot be publicly shared in the artifact itself). We recommend using the provided VM as a reference as there are various dependencies required by the Cuttlefish infrastructure components.

A.2.4 Software dependencies

These artifacts were exclusively developed and tested on an Ubuntu 20.04 machine. The Jupyter notebooks should work on any platforms that support Python, and the Cuttlefish-based infrastructure should work on most Linux distributions. If you would like to run the analysis notebooks, the dependencies are fairly simple. However, if you would like to set up your own testing environment with the Cuttlefish virtual devices, there are additional dependencies that need to be installed, and certain components may not be cross-platform compatible (e.g., the script that uses `iptables` to route DNS traffic from Cuttlefish devices).

To run the Jupyter Notebooks, the only requirement is a Python environment with all the dependencies in `/requirements.txt`.

To run the Cuttlefish app analysis infrastructure, there are additional requirements:

1. Docker Engine for the DNS servers and HTTPS proxies
2. Android Cuttlefish - which can be installed using `cuttlefish_setup.sh` and `cuttlefish_download_images.sh` in the `/cuttlefish/scripts` directory
    - **NOTE:** `kvm` needs to be supported by your CPU
3. The fork of DroidBot in `droidbot.tar.gz` installed in the Python environment
    - After inflating the `droidbot` folder, it can be installed via `pip install -e /path/to/droidbot`
4. Frida server - which can be installed by `download_frida_server.sh` in the `/cuttlefish/scripts` directory
5. `iptables` to route DNS traffic from the Cuttlefish devices
6. **[recommended]** `ufw` as Cuttlefish devices expose additional ports
7. **[recommended]** a desktop environment / VNC server (e.g., `turbovnc`) to interact with the Cuttlefish devices

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

As Zenodo records only support flat files, the artifacts have been compressed.

1. Download the data artifacts from Zenodo at https://doi.org/10.5281/zenodo.14737144 and ensure they are all in the same directory.
2. `chmod +x inflate.sh && chmod +x deflate.sh`
3. `./inflate.sh` to inflate the artifacts

Depending on whether you would like to only run the analyses with Python or run the Cuttlefish infrastructure, you will need to install different dependencies. The steps below outline the installation steps for both cases.

Python Environment [Jupyter and Cuttlefish]. The official downloads can be found on the Python site, but Python is likely to have been pre-installed on your machine. We recommend creating a separate virtual environment for the required Python dependencies for this artifact (see this for a primer on virtual environments).

1. After setting up your Python virtual environment, please install the packages in `/requirements.txt` with the corresponding commands for your environment manager (e.g., `pip install requirements.txt`).

The below dependencies are only required to test the Cuttlefish infrastructure.

Python Environment [Cuttlefish, required]. After installing your Python environment as described above, you will also need to install the provided fork of DroidBot to use in the Cuttlefish infrastructure. To do so, please inflate the artifacts, and run:

```
pip install -e /path/to/droidbot-fork
```

inside your Python virtual environment.

Docker [Cuttlefish, required]. The official setup instructions can be found at the Docker docs site.

1. Docker Engine can be installed on Linux with `docker_setup.sh` in the `/cuttlefish/scripts` directory.
2. Log out and log back in so that your group membership is re-evaluated to run `docker` commands without `sudo`.

Android Cuttlefish [Cuttlefish, required]. The official setup instructions can be found at the Android Open Source site.

1. Cuttlefish packages can be set up on Linux with `cuttlefish_setup.sh` in the scripts directory.
2. Download the following Cuttlefish Android 11 artifacts from aosp-android11-gsi@11718355 to some directory (e.g., `$HOME/cf-images/11718355` which is created by the prior script):

(a) `aosp_cf_x86_64_phone-img-11718355.zip` for the Android image

(b) `cvd-host_package.tar.gz` for the host cuttlefish utilities

3. Extract the downloaded artifacts by running:

```
tar -xvf cvd-host_package.tar.gz
unzip aosp_cf_x86_64_phone-img-11718355.zip
```

4. Then, add the absolute path of the `cf-images/android11/bin` to your PATH (it is recommended to add to do so in a persistent manner)

Desktop Environment/VNC [Cuttlefish, recommended]. If you are working in a remote/headless environment, it is recommended to install a desktop environment and a VNC server so that you can visually monitor and control the Cuttlefish virtual devices with a browser and WebRTC.

It should be possible to connect to a remote WebRTC process from your local computer without installing a desktop environment on the machine running the Cuttlefish infrastructure (e.g., with SSH port tunneling), but we encountered problems with this and found that installing the desktop environment was simpler.

If you need help, here is an example guide on how to install TightVNC with Xfce4 on Ubuntu.

Configuration. After installing the required dependencies, make sure to update the configuration for the desired components. For the Jupyter notebooks, the main configuration file is `/analysis/.parameters.py`, but this should not require any changes as long as the directory structure was not modified after inflating the artifacts.

For the Cuttlefish infrastructure, the main configuration file is `/cuttlefish/.env`. In particular, make sure to change the following variables:

1. `DIR_BASE` to the absolute file path to the inflated `/data/cuttlefish` directory
2. `DIR_ADB_FILES` to the absolute file path to the inflated `/cuttlefish/adb_files` directory
3. `MITMPROXY_CACERT_FILENAME` to be the name of the created mitmproxy CA certificate produced by the `generate_mitmproxy_cert.sh` script. See Section A.3.2 for more details on what to put for this setting.
4. `NUM_APPS_PER_SAMPLE` to the number of apps per sample group desired
5. `NUM_CUTTLEFISH_DEVICES` to the number of Cuttlefish devices

Additionally, configure `sudo` to allow executing the script `cuttlefish/scripts/iptables.sh` without requiring a password. This script is executed for each Cuttlefish virtual device in the current user, so it normally requires elevated privileges. After verifying the contents of the script, you can do this by running `sudo visudo` and adding the following line:

```
your_username ALL=(ALL) NOPASSWD: /path/to/cuttlefish/scripts/iptables.sh
```

A.3.2 Basic Test

To test functionality of the Jupyter notebooks, activate the Python environment (with `poetry shell` in the artifact directory in the provided VM), and launch a Jupyter server by running `jupyter lab` in the artifact directory. Then, configure SSH port forwarding to your local machine over the default Jupyter port 8888, and navigate to the URL from the output of the Jupyter command in your browser of choice.

To test functionality of the Cuttlefish devices after installing all dependencies, perform the following:

1. Navigate to your `cf-images` directory where you downloaded the Android Cuttlefish images (e.g., `$HOME/cf-images/11718355`), and run the following:
```
HOME=$PWD ./bin/launch_cvd -num_instances=2
-resume=false -start_webrtc=true
-start_vnc_server=false
```
2. Wait until you see a message in the output (colored green) asking you to navigate to `https://localhost:8443`
3. Connect to the desktop environment, launch a browser, and navigate to `https://localhost:8443`.

Next, bootstrap the mitmproxy CA certificate creation that will be injected into the Android devices by doing the following:

1. In `/cuttlefish/compose.yaml`, change the file path of the `hardumps` volume to the absolute path of the inflated `data/cuttlefish/hardumps` folder.
2. Navigate to the `/cuttlefish` directory and run `docker compose up -build -d`.
3. After the containers have been created, run the `generate_mitmproxy_cert.sh` script from the same directory.
4. The CA certificate should then be in the `/cuttlefish/adb_files` directory (i.e., `c8750f0d.0`)
5. Make sure to change the environment variable `MITMPROXY_CACERT_FILENAME` in `/cuttlefish/.env` to the name of the certificate.

Then, test that the Cuttlefish virtual device is appropriately configured by the infrastructure by running `python analyze.py -test`. You should see that the Cuttlefish devices are set up before the script exits.

Finally, modify the input APKs to run with the Cuttlefish infrastructure by:

1. Modifying the file `samples_cuttlefish.csv` in `/data/cuttlefish/samples`, adding the SHA256 `hash`, `package` name, `version` string, and `group` name (the provided samples file uses `successful` and `unsuccessful` to refer to the originally-successful and originally-unsuccessful APKs).

2. Place the added APKs into the directory `DIR_BASE/samples/<group>` with the name `<package>_<version>_<hash>.apk`.

3. To run the actual analysis, run `analyze.py` without the test flag.

A.4 Evaluation workflow

A.4.1 Major Claims

The major claims made in the paper are as follows:

(C1): *Expired domain names are found in the DNS traffic of multiple versions of Android apps, regardless if they are out of date or the latest available versions.*

(C2): *The purpose and use of many such domains can be identified from the context of the request and inspecting the decompiled APKs, and they can be abused by malicious re-registrants to change app behavior, even if the app itself does not change.*

(C3): *All eight figures and four tables presented in the text can be produced from our extracted data. However, please note that because the commercial telemetry data we used cannot be released, this artifact produces slightly different versions of Figures 3 and 4, and Table 4. Additionally, Figure 7 is also slightly different, but because of minor differences in how the original code processed the raw data.*

(C4): *The app analysis infrastructure extracts DNS network traffic of APKs.*

A.4.2 Experiments

The following experiments can be performed to verify that the artifacts are functional and can be used to reproduce the results from the text. The Jupyter notebooks used for analysis are bundled with pre-processed data so that the raw data — which is large — does not have to be re-processed.

(E1): *[5 human-minutes + 5 compute-minutes + 0GB disk]: Run all cells in the footprints.ipynb notebook and verify that domains were checked for expirations.*
How to: *Ensure the basic test for the Jupyter notebooks has been completed, with a Jupyter server now running in a Python environment with the dependencies installed.*
Preparation: *Connect to the Jupyter server with your browser and open the `analysis/footprints.ipynb` notebook.*
Execution: *Press Run → Run All Cells.*
Results: *The `df_footprints` variable is a DataFrame that has a column named `expired_at_exec`, denoting domains that were expired during app execution. The rows that have this column as True span multiple versions of different apps.*

(E2): *[30 human-minutes + 5 compute-minutes + 0GB disk]: Run all cells in the decompilations.ipynb notebook and verify that most of the domains can be found directly inside the APK, and their purpose can be identified from inspecting the decompiled APKs.*
How to: *Ensure the basic test for the Jupyter notebooks has been completed, with a Jupyter server now running in a Python environment with the dependencies installed.*
Preparation: *Connect to the Jupyter server with your browser and open the `analysis/decompilations.ipynb` notebook.*
Execution: *Press Run → Run All Cells.*
Results: *Verify that the `Searching for Domains in Decompiled Code` launches grep commands to search the decompiled APKs for their corresponding domains. Manually confirm that files in the directory `DIR_BASE → samples-jadx-processing/expired_at_reg` contain the output of the grep commands. Randomly sample some of the identified domains to look at the decompiled APKs.*

(E3): *[10 human-minutes + 10 compute-minutes + 0GB disk]: Run all cells in each Jupyter notebook and verify that all figures and tables are produced.*
How to: *Ensure the basic test for the Jupyter notebooks has been completed, with a Jupyter server now running in a Python environment with the dependencies installed.*
Preparation: *Connect to the Jupyter server with your browser and open the `footprints.ipynb`, `decompilations.ipynb`, and `cuttlefish.ipynb` notebooks in the `analysis/` directory.*
Execution: *Press Run → Run All Cells in each notebook.*
Results: *Verify that `footprints.ipynb` produces Tables 1, 2, and 4 and Figures 2, 3, 4, 7, and 8, `decompilations.ipynb` produces Tables 3 and 4, and Figure 5, and `cuttlefish.ipynb` produces Figure 6.*

(E4): *[10 human-minutes + 3*2*N/D compute-minutes + 0GB disk]: Launch the Cuttlefish infrastructure on a sample of APKs.*

How to: *Ensure the basic test for the Cuttlefish infrastructure setup has been completed and the configuration has been updated.*
Preparation: *Modify the* `cuttlefish/.env` *file and change* `NUM_APPS_PER_SAMPLE`*, the number of apps that will be analyzed from each of the two sample groups (N), and* `NUM_CUTTLEFISH_DEVICES`*, the number of live Cuttlefish virtual devices (D).*
Execution: *Using the configured Python environment, run* `python analyze.py` *in the* `/cuttlefish` *directory.*
Results: *When analysis of an app completes, there is a pcap file in* `DIR_BASE` → `DIR_DATADUMPS` → `tcpdumps` → `<group>` → `<package>_<version>_<hash>_<timestamp>.pcap`.

A.5 Notes on Reusability

The number of Cuttlefish devices that can be launched concurrently can be adjusted by modifying the value of the `num_instances` flag provided to the `launch_cvd` command, changing the `NUM_CUTTLEFISH_DEVICES` variable in `cuttlefish/.env`, and adding additional Docker containers to `cuttlefish/compose.yaml` if necessary. Furthermore, the Cuttlefish infrastructure can be reused for different APK inputs, and can be done by modifying the input `samples_cuttlefish.csv` file that describe the APKs and placing them into the expected locations. The analysis notebooks will continue to function even with the addition of new data, although it may also be desirable to insert your own API keys for Farsight and Dynadot to analyze new domains.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.