2016 IEEE Security and Privacy Workshops

Target Fragmentation in Android Apps

Patrick Mutchler
Stanford University
pcm2d@stanford.edu

Yeganeh Safaei
Arizona State University
ysafaeis@asu.edu

Adam Doupé
Arizona State University
doupe@asu.edu

John Mitchell
Stanford University
mitchell@cs.stanford.edu

***Abstract*—Android apps declare a *target* version of the Android run-time platform. When run on devices with more recent Android versions, apps are executed in a compatibility mode that attempts to mimic the behavior of the older target version. This design has serious security consequences. Apps that target outdated Android versions disable important security changes to the Android platform. We call the problem of apps targeting outdated Android versions the *target fragmentation problem*.**

We analyze a dataset of 1,232,696 free Android apps collected between May, 2012 and December, 2015 and show that the target fragmentation problem is a serious concern across the entire app ecosystem and has not changed considerably in several years. In total, 93% of current apps target out-of-date platform versions and have a mean outdatedness of 686 days; 79% of apps are already out-of-date on the day they are uploaded to the app store. Finally, we examine seven security related changes to the Android platform that are disabled in apps that target outdated platform versions and show that target fragmentation hamstrings attempts to improve the security of Android apps.

I. Introduction

Android has become the most popular smartphone platform worldwide, with more than one billion active devices [1]. Android faces two major security challenges in delivering secure code to users, fragmentation within devices and fragmentation within apps. Device fragmentation is a well known concern [2]. Google does not control the distribution of Android devices or Android software updates. Google instead relies on a network of other businesses to deliver up-to-date Android software to users, in the form of new devices or software updates. Because of the distributed nature of this process, a large number of Android devices are running out-of-date versions of the Android platform. Critical security patches do not reach millions of Android users, extending the lifetime of security vulnerabilities. In comparison to device fragmentation, fragmentation within apps has received little attention despite carrying similar security consequences.

Android exposes numerous essential library features to apps through the Android API. New versions of the Android platform can introduce changes to the behavior of existing library features. For example, Android 4.4 changed the behavior of `AlarmManager.set` to batch alarms set for similar times. This change helps improve battery life at the expense of inexact alarms. Behavioral changes such as this present a problem for apps, which might suddenly break when a device updates to a new Android version.

Google assigns each Android platform version an integer called an API level. To maintain a degree of forwards compatibility and prevent apps from dramatically changing their behavior without warning, every app has a *target API level*. Apps that are run on devices with a higher API level than their target API level are executed in a compatibility mode that attempts to match the behavior of devices with the target API level as closely as possible. For example, apps that set their target API level to 18 (Android 4.3) will use the unbatched alarm behavior even when run on up-to-date Android devices.

Android platform changes can include important new security features that either resolve known problems with the Android APIs or provide extra protection against attack. Any of these features disabled by the compatibility mode will be unavailable to apps that target outdated Android levels. This design means that, even when running an up-to-date device, *an app that targets an outdated API level will not have access to the most current security features*. Google does not publish an app's target API level on the Google Play store so there is no simple way for users to know if an app targets an outdated API level. Instead, users are entirely at the mercy of app developers. We call this problem of apps targeting outdated API levels the *target fragmentation problem*.

The most well-known consequence of the target fragmentation problem relates to a remote code execution vulnerability in Android WebView [3]. API level 17 added new behavior to resolve this vulnerability, however this change is only applied to apps that target API level 17 or higher. Years after the vulnerability was disclosed and fixed in the Android platform, apps can still be vulnerable, even when run on the new platform, by targeting outdated API levels. Several studies examining this vulnerability have reported the percentage of apps that target levels 16 or below [4, 5]. But this vulnerability is just one example of how the target fragmentation problem makes Android apps less secure and no research has studied target fragmentation in its own right. A more complete analysis of the target fragmentation problem is essential to understanding the security of the Android ecosystem.

To the best of our knowledge, this paper reports the first study to identify and measure the target fragmentation problem and its security implications at a broad scale. In this paper we study target fragmentation in five datasets of free Android apps collected from the Google Play store over almost four years. In total, these datasets include 1,232,696 apps. We measure trends in the target fragmentation problem using metadata obtained from the Google Play store. Finally, we study the security implications of the target fragmentation problem with respect to several concrete vulnerabilities. The major research questions and results of our study are as follows:

DOI 10.1109/SPW.2016.31

What is the state of the target fragmentation problem in the Android app ecosystem? We examine a dataset of 60,086 free apps collected from the Google Play store in December, 2015 and find that 93% of apps target out-of-date API levels. We define a measure of "outdatedness" and find that apps, on average, target API levels that are 686 days out-of-date.

Do developers choose to target outdated Android versions or is fragmentation caused by developers abandoning their apps? To account for unmaintained apps, we define a measure of "negligent outdatedness" that measures outdatedness from the date an app was uploaded to the app store and show that target fragmentation is not just caused by stagnant apps. We find that apps collected in December, 2015 have a mean negligent outdatedness of 536 days.

Is target fragmentation a problem in the most popular apps? We examine the target fragmentation problem with respect to app popularity and conclude that target fragmentation is a serious problem even among the most popular apps. We find that 88% of apps collected in December, 2015 and installed more than one million times target out-of-date API levels. These apps have a mean outdatedness of 607 days and a mean negligent outdatedness of 493 days, only slightly lower than the general population.

Is the target fragmentation problem becoming less severe over time? We compare the target fragmentation results from the December, 2015 dataset with four other datasets collected from the Google Play store between May, 2012 and July, 2014. These datasets combined contain 1,232,696 apps. We find that, other than a growing tail of extremely out-of-date apps, outdatedness distributions among the four most recently collected datasets are very similar, suggesting that the severity of the target fragmentation problem has not changed considerably in several years.

What are the specific security implications of the target fragmentation problem today? We expand the discussion of the target fragmentation problem by examining seven security relevant changes in the Android platform and provide the first quantitative analysis of the broad implications of target fragmentation on the security of the Android ecosystem.

II. Background

Packaged with each Android app is a *manifest* file. The manifest file is an XML document that contains information about an app such as the list of application components, requested permissions, and system events to which the app responds [6]. The manifest contains two attributes relevant to this study: a minimum API level (`minSdkVersion`) and a target API level (`targetSdkVersion`)[1]. The meaning of the minimum API level is very straightforward. An app cannot be installed on a device with an Android level below the minimum API level. This design ensures that apps are not installed on devices that lack essential functionality.

[1]Manifests can also include a maximum API level, but since Android 2.0.1 the maximum API level is no longer used for anything other than filtering searches on the Google Play store.

API Level	Version Code	Codename	Release Date
14	4.0–4.0.2	Ice Cream Sandwich	Oct, 2011
15	4.0.3–4.0.4		Dec, 2011
16	4.1–4.1.2	Jelly Bean	Jul, 2012
17	4.2–4.2.2		Nov, 2012
18	4.3–4.3.1		Jul, 2013
19	4.4–4.4.4	KitKat	Nov, 2013
20	4.4W–4.4W.2		Jun, 2014
21	5.0–5.0.2	Lollipop	Nov, 2014
22	5.1–5.1.1		Mar, 2015
23	6.0–6.0.1	Marshmallow	Oct, 2015

TABLE I: An abbreviated Android version history.

An app's target API level is used to maintain forwards compatibility with new Android platforms. If the API level of a device is higher than an app's target API level, the device will enable compatibility features to match the behavior of the target API level as closely as possible. The set of compatibility features enabled in each API level can be found in the Android documentation [7]. Note that apps can be safely installed on devices running lower API levels than the target API level. Developers can target the most current API level without making their app incompatible with older Android devices.

If an app does not declare a target API level or if the target API level is lower than the minimum API level, the target API level is set to the minimum API level. For the remainder of this paper we distinguish between the *raw target API level*, which is the target level listed in the manifest file, and the *target API level*, which is the target level computed using this rule and actually used by the Android operating system. Approximately 8% of apps today do not declare a valid raw target API level.

The `targetSdkVersion` and `minSdkVersion` attributes take integer values called "API levels" that correspond to the different Android version codes. For the remainder of this paper we use the API level values (e.g., 17, 18, 19) rather than the version codes (e.g., 4.2, 4.3, 4.4) when discussing different API versions. The correspondence between recent API levels and Android version codes is presented in Table I.

A. Security Concerns

It is not immediately obvious from the Android documentation that targeting outdated API levels can have security implications. On the documentation page for the `targetSdkVersion` attribute [8], Google suggests that developers should "increase the value of [the targetSdkVersion] attribute to match the latest API level," but there is no mention of the security consequences of targeting outdated API levels. On the "Security Tips" page [9] there is no mention of target API level. One might assume that the security consequences of targeting outdated API levels are minimal or nonexistent.

However, there are important security changes in recent Android versions that are not applied to apps that target outdated API levels. For example, API levels 17 and 19 both contain changes that prevent code injection vulnerabilities in widely used features. Several other API levels change popular features to have safer default behaviors, providing an extra layer of protection. Table II lists the major security changes to the Android platform that can be disabled by targeting

API	Platform Change
16	Access to file URLs from JavaScript is disabled by default
17	Content Providers are no longer exposed to foreign apps by default
17	Unannotated app methods are not callable from JavaScript code
19	`isValidFragment` is added to prevent Fragment Hijacking
19	JavaScript URLs are executed in a separate WebView context
21	Context.bindService no longer accepts Implicit Intents
21	WebView blocks mixed content by default

TABLE II: Selected security-relevant changes to the Android platform. Apps that target API levels below the listed levels do not have the benefit of any security protection provided by these changes.

outdated Android levels. The details of these changes and the vulnerabilities they close are discussed in Section V. If a large number of apps target out-of-date API levels then these changes, no matter how well intentioned, are made ineffective and apps are put at unnecessary risk.

III. Methodology

Our study analyzes a dataset of 1,232,696 free apps collected from the Google Play app store between May, 2012 and January, 2016. To collect these apps we developed a system to crawl the Google Play store to identify new apps, scrape metadata from the Google Play store, and download actual app files. This system was operational during five brief time windows, naturally separating our dataset into five smaller datasets (Datasets A, B, C, D, and E, in reverse chronological order) that correspond to these time windows. Table III describes the details of these datasets and lists the most current API level at the time each dataset was collected.

A. Collecting Apps

Our system first crawls the Google Play store for apps to download. We consider an app unique if it has a unique app id[2]. To crawl the Google Play store, we use the following four techniques: (1) crawl the Google Play designated categories for popular apps and collections, (2) crawl random known developer pages to look for new apps, (3) search on the Google Play store using words from known app descriptions, and (4) extract all app ids from URLs on crawled pages.

Google publishes some metadata about apps on the store. We scrape and collect metadata about each crawled app including the date the most recent version of that app was uploaded to the app store and the number of devices on which the app has been installed[3].

To download apps we use a method similar to the one described by Viennot, Garcia, and Nieh [10]. In an attempt to efficiently collect as diverse a set of applications as possible, we only download apps with never before seen app ids. For each dataset except Dataset A, we attempted to download *every* app with a never before seen app id identified during crawling. Due to time constraints (and technical challenges), Dataset A is only a subset of the available apps. We discuss the effect of this collection method in Section VI-B.

[2]The value of the `id` query parameter of the app's Google Play URL, for example `com.instagram.android` in the URL play.google.com/store/apps/details?id=com.instagram.android. Note that this is not necessarily the same as the app's package name

[3]Precise install counts are not available. Google Play reports a range of possible install counts with two buckets per order of magnitude. For example, an app might have a reported install count between 10,000 and 50,000.

Dataset	App Count	Crawl Date	Most Current API
A	60,086	December, 2015	23
B	219,115	June, 2014	19
C	165,489	January, 2014	19
D	645,862	July, 2013	17
E	142,144	May, 2012	15

TABLE III: An overview of the five datasets used in this study and the most current API level at the time each dataset was collected.

B. Analysis

The Google Play store publishes each app's minimum API level but does not publish target API levels. We use `apktool` [11], a static analysis tool that converts packaged apps into human readable files, to extract manifests and record their target API levels. Because our database of apps is extremely large, it is impractical to perform complex static analysis. All of the static analysis used in this study is purely syntactic, which we perform by processing the `smali` representation of app bytecode extracted by `apktool`.

IV. Evaluation

In this section we quantify the extent of the target fragmentation problem. We begin by demonstrating that the majority of sampled apps target outdated API levels. We define an outdatedness metric that measures the severity of the target fragmentation problem for individual apps as well as across a population of apps. We show that the target fragmentation problem is primarily caused by developer negligence rather than apps that lie fallow on the app store. We compare our outdatedness metric between popular and unpopular apps and prove that target fragmentation is a problem even among the most popular apps. Finally, we show that outdatedness curves are similar in the four most recently collected datasets. This result suggests that, unless the target fragmentation problem is reexamined, it may continue with the same scale in the future. Due to the large sizes of our datasets, we believe that these results apply broadly to the entire Google Play ecosystem.

A. Target Fragmentation Today

Figure 1 shows the distribution of target API levels for apps in Dataset A, the dataset containing the most current apps. It is immediately clear that the huge majority of apps do not target API level 23, the most current API level at the time Dataset A was collected. More precisely, we find that 93% of apps in Dataset A target API levels 22 or lower.

Apps that target more outdated API levels are more likely to miss crucial security changes. We are not just interested in *if* an app is outdated but also in *how* outdated an app is. We define a quantitative measure called "outdatedness" as the difference (in days) between the release date of an app's target API level and the release date of the most current API level at the time of the app was collected. We find a median outdatedness of 704 days and a mean outdatedness of 686 days

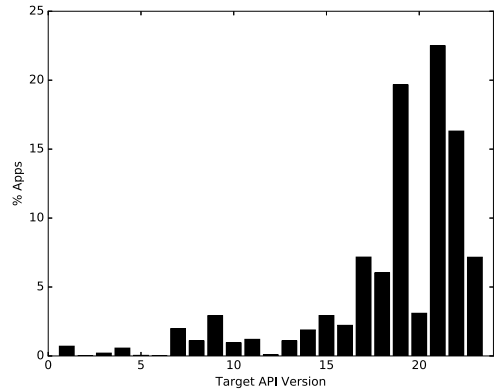

Fig. 1: The distribution of target API levels in dataset A, collected two months after the release of API level 23. API level 20 is specifically intended for wearable devices, explaining why few apps target it.

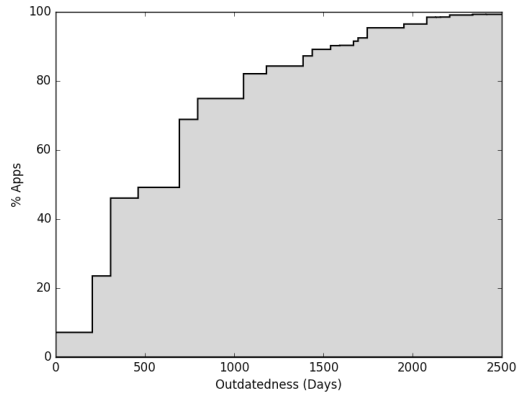

Fig. 2: The cumulative distribution of outdatedness in Dataset A. A point at (X,Y) means that Y% of apps have an outdatedness of X days or less. The low resolution of outdatedness values causes the jagged nature of this curve.

for apps in Dataset A. We can use the cumulative distribution of outdatedness (Figure 2) as a measure of the severity of the target fragmentation problem over a population of apps. By examining the low end of the curve we see that the large majority of apps target outdated API levels. The long tail at the top of the curve shows us that a considerable number of apps target API levels that are many years out of date.

We might expect the distribution of target API levels to resemble a skewed normal distribution, with the percentage of apps targeting each API level decreasing as API levels get more and more out-of-date. But instead, as seen in Figure 1, we find that the percentage of apps targeting API levels 7 through 16 is relatively flat. Nearly as many apps targeting API level 7 as API level 16. API level 9 sticks out in particular, being targeted by nearly 3% of all apps in Dataset A even though it does not offer any critical compatibility features. Why are apps targeting such an out-of-date API level?

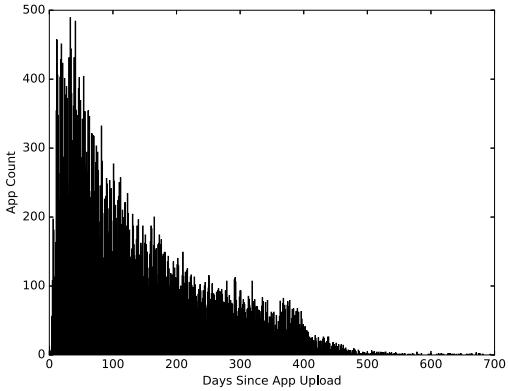

Fig. 3: The distribution of the number of days between when an app was uploaded to the Google Play store and when that app was collected.

If we analyze the *raw* target API levels in Dataset A we find that 8.2% of apps either do not include a raw target API level, set a value that is lower than their minimum API level, or have an otherwise invalid raw target API level. These apps use their *minimum* API level as their target API level. These apps account for the majority of the long tail in the distribution of target API levels. For example, 63% of apps that target API levels 15 or lower do so because they use their minimum API level as their target API level. This represents a major opportunity for eliminating the long tail in the distribution of target API levels by convincing developers to use the `targetSdkVersion` attribute correctly.

B. Stale Apps

Not all apps on the Google Play store are regularly maintained by their developers. Figure 3 shows the distribution of the number of days between an app's collection and when it was uploaded to the Google Play store by its developers. Note that this date could either be the first time the app was published or the date the most recent app update was pushed. Only 38% of apps in Dataset A were uploaded after the release of API level 23 in October, 2015. Clearly, apps that have stagnated on the Google Play store will not target the new API levels as they are released. The presence of unmaintained apps can skew the target API distribution and falsely imply that developers who actively maintain their apps fail to update their apps to target new API levels. Here, we attempt to distinguish outdatedness caused by unmaintained apps and outdatedness that persists through app maintenance.

We define an app's "negligent outdatedness" as the difference (in days) between the release date of its target API level and the release date of the most current API level *at the time it was uploaded to the Google Play store*. Negligent outdatedness measures missed opportunities for developers to target their apps to current API levels. Figure 4 describes a concrete example to clarify the difference between outdatedness and negligent outdatedness. We find a median negligent

Fig. 4: A example app demonstrating outdatedness and negligent outdatedness. This app targets API level 21, was uploaded after the release of API level 22, and was collected after the release of API level 23. Outdatedness is measured from the highest API level before app collection. Negligent outdatedness is measured from the highest API level before the app was uploaded.

outdatedness of 377 days and a mean negligent outdatedness of 536 days among apps in Dataset A.

We can be even more generous with our definition of negligent outdatedness and include some lag time to allow developers to retarget their apps after a new API level is released. Instead of choosing the most current API level at the time an app was uploaded to the app store, we choose the most current API level that has been available for at least N days at the time an app was uploaded to the app store. We call this lag time an "adoption window." Adding an adoption window of 30 days only marginally affects negligent outdatedness, reducing the median negligent outdatedness to 327 days and the mean negligent outdatedness to 496 days.

Figure 5 shows the cumulative distribution of negligent outdatedness using an adoption window of 30 days. Even with a generous adoption window, apps still fail to target the appropriate API levels. 79% of apps are negligently targeted to outdated API levels, meaning these apps are out of date as soon as they are uploaded to the app store. Developers have access to new Android platforms before they are released, so it is possible for apps to be up-to-date on day zero of a new API level. It is clear from these results that the target fragmentation problem cannot be explained by stale apps but is the result of developer negligence, either due to ignorance of the consequences of targeting out-of-date API levels or due to a deliberate choice to target an out-of-date API level.

C. Popular Apps

The large majority of apps on the Google Play store are not downloaded by many users. Just 2% of apps in Dataset A have been installed at least 1,000,000 times yet these apps account for 74% of total installs among apps in Dataset A. It is important to understand the relationship between app popularity and the target fragmentation problem.

Figure 6 compares the cumulative outdatedness distribution between apps of different popularities. We see that the outdatedness curves are very similar, with the most popular apps (installed at least one million times) targeting only marginally less out-of-date API levels. Comparing the negligent outdatedness distributions between apps of different popularities gives similar results (and we do not include it here for space reasons). Apps that have been installed at least one million

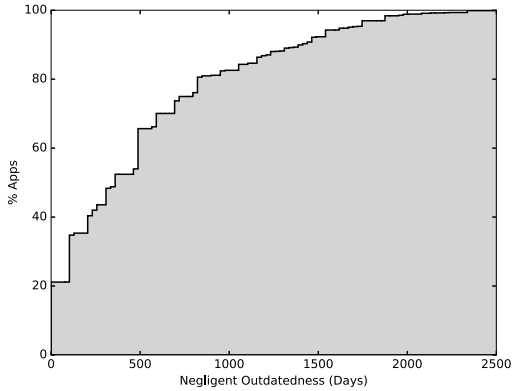

Fig. 5: The cumulative distribution of negligent outdatedness in Dataset A using an Adoption Window of 30 days.

times have a mean outdatedness of 607 days and a mean negligent outdatedness of 493 days.

D. Target Fragmentation Over Time

It is clear from the analysis of Dataset A that target fragmentation is a serious problem today. Even the most popular apps are targeting outdated, and often extremely outdated, API levels. By analyzing datasets collected at different dates we can see how the problem has changed over time. We repeat the previous analyses on the four remaining datasets and compare these results against the results from Dataset A.

Figure 7 compares the outdatedness distributions of our five datasets. There is only one clear trend over time: a growing tail of apps that target extremely outdated API levels, with the 90th percentile of outdatedness more than doubling between Datasets E and A. This is a natural property of the target fragmentation problem because the maximum outdatedness grows over time. As long as there are apps that target the lowest API levels we expect to see this tail continue to grow.

Comparing the low end of the outdatedness curves does not show an obvious pattern. We see that in each dataset the vast majority of apps target out of date API levels and that, excluding than Dataset E, there does not appear to be a dramatic difference in the lower end of the outdatedness curves. We note that Datasets A and C were each collected two months after a platform release and Datasets B and D were collected seven and nine months after a platform release, respectively. This difference appears to have a greater impact on the low end of the outdatedness curves than any pattern over time, with more than 20% of apps in Datasets B and D targeting current API levels and less than 10% of apps in Datasets A and C targeting current API levels.

There is one very promising trend between our datasets. We find a clear downward trend in the percentage of apps that do not specify a *raw* target API level (Figure 8) and therefore set their target API level to their minimum API level. Because developers often want to support as many devices as possible, apps generally have very low minimum API levels, making it

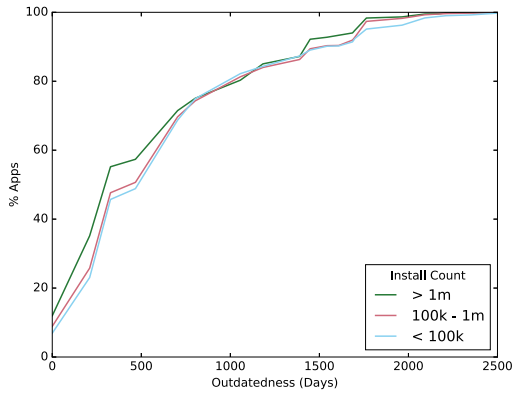

Fig. 6: A comparison of the cumulative outdatedness distributions of different app popularities showing that outdatedness statistics are robust against changes in app popularity. Apps with with 1,000,000 or more installs have an outdatedness curve that is only marginally better than the rest of the population. Data points are linearly interpolated to improve readability.

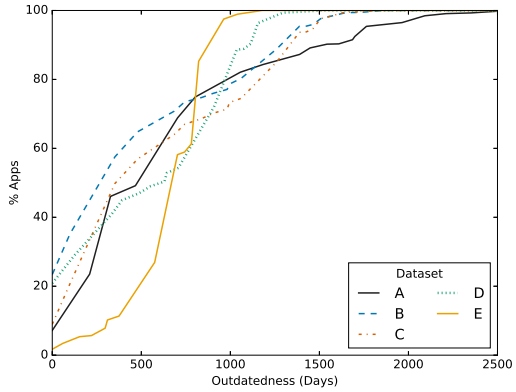

Fig. 7: The cumulative distribution of outdatedness for all five datasets. The four most recently collected datasets (A through D) follow similar curves but with growing tails. Data points are lineraly interpolated to improve readability.

extremely dangerous to fail to specify a target API level. This trend suggests that developers have become more aware of the target API feature and that the number of developers targeting their minimum API level will vanish over time.

V. Security Implications

The target fragmentation problem means that any security change to the Android platform will be less effective, so long as the change is disabled in compatibility mode. In this section we explore the practical consequences of the target fragmentation problem on seven security changes to the Android platform. Apps targeting outdated API levels may not necessarily be vulnerable but instead are at a heightened risk for security vulnerabilities. In all but one case we only show that apps are unnecessarily at a heightened risk. Because Google has deemed the outdated behavior unsafe enough to

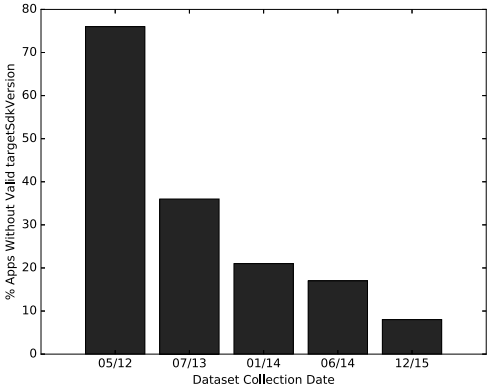

Fig. 8: The percentage of apps in each dataset that do not include a valid `targetSdkVersion` attribute in their manifest and therefore set their target API version to be equal to their minimum API version.

deserve a change to the Android platform, widespread use of outdated behavior is troubling on its own. We also cite research showing that the conditions necessary for these apps to be exploitable are frequent. The statistics in this section are computed on Dataset A, the most recently collected dataset.

A. WebView Defaults

WebView [12] is a UI element that acts as an embedded web browser within Android apps. In Dataset A, 91% of apps include at least one WebView instance[4]. There are three major security changes to the default behavior of WebView that are disabled by targeting outdated API levels. The following sections describe these changes and show that apps that target current API levels are less likely to use the unsafe behaviors.

1) File Scheme Same Origin Policy: According to the Same Origin Policy, JavaScript code only has access to content loaded from the same origin. Treating all `file:` URLs as belonging to the same origin is a security risk in systems with mutually distrusting files. If a WebView loads any untrusted content using a `file:` URL, then that content has full access to every other file accessible by the app. In API level 16, the default behavior of WebView was changed to treat all `file:` URLs as belonging to separate origins, however this change only applies to apps targeting API levels 16 or higher. Apps that wish to override this behavior and treat all `file:` URLs as belonging to the same origin can use the methods `setAllowFileAccessFromFileURLs` and `setAllowUniversalAccessFromFileURLs`.

We find that 82% of apps that target API levels 15 or lower and 18% of apps that target API levels 16 or higher use the unsafe policy for `file:` URLs. Because apps that target API levels 16 or higher and want to use the unsafe policy must do so explicitly, we can use the percentage of these apps that include calls to `setAllowFileAccessFromFileURLs`

[4]Because we are unable to ensure that ad libraries do not contain vulnerabilities, this figure includes WebViews from ad libraries as well as primary app functionality.

or `setAllowUniversalAccessFromFileURLs` as an upper bound on apps that want to use the unsafe policy. If we assume this percentage is uniform across both app populations then we conclude that 64% of apps that target API level 15 or lower unnecessarily use the unsafe policy and would become more secure if they were retargeted to API level 16. These apps represent 9.6% of all apps in Dataset A. We discuss the validity of this assumption in Section VI-A.

Apps that allow unsafe file access are only at a heightened risk. To be exploited, an app must load untrusted content using a `file:` URL. Two ways an app might be exploited are by loading `file:` URLs received from other apps or by navigating to untrusted web pages, which could drop a file on disk and redirect the WebView to that file. See Chin and Wagner [13] for a detailed description of this exploit. Research has shown that 23% of web browser apps unsafely load `file:` URLs from foreign apps [14] and that WebView apps frequently load untrusted web pages [13, 15, 4].

2) JavaScript URLs: Apps can load web content in a WebView by calling `loadUrl`. In apps that target API levels 18 or lower, calling `loadUrl` on a JavaScript Pseudo-URL[5] executes the script in the currently rendered web page. This behavior is exploitable in apps that load unfiltered URLs retrieved from foreign apps. A malicious app can send a request to load a URL at `foo.com` and then send a script to be executed in the context of `foo.com` to read private content. This attack is known as Cross-Application Scripting [16].

Apps that target API levels 19 or higher load JavaScript Psuedo-URLs in an empty context and must instead use the method `evaluateJavascript` to execute JavaScript code in the current WebView context. We find that 60% of apps that target API levels 19 or higher contain at least one call to this method. Again, we can use this as an upper bound on the number of apps that intentionally execute JavaScript code in this manner. Because 90% of apps that target API levels 18 or lower include a WebView and enable JavaScript, we conclude that 30% of apps that target API levels 18 or lower have no need for evaluating JavaScript code in this manner and would become more secure if they were retargeted to API level 19. These apps represent 9.4% of all apps in Dataset A.

3) Mixed Content: A web page that includes web elements retrieved over HTTP when it is loaded over HTTPS is said to include *mixed content*. Loading mixed content is a security risk, and several major browsers block mixed content [17, 18]. WebView blocks mixed content by default in apps that target API levels 21 or higher. Apps that wish to override this behavior and allow mixed content can use the method `setMixedContentMode` to specify a custom policy.

We find that 76% of apps that target API levels 20 or lower and 42% of apps that target API levels 21 or higher allow mixed content. If we assume that the percentage of apps that want to allow mixed content is uniform across both app populations then we conclude that 34% of apps that target API levels 20 or lower unnecessarily allow mixed content and would become more secure if they were retargeted to API level 21. These apps represent 18% of all apps in Dataset A.

[5]A URL that starts with the protocol `javascript` and contains the JavaScript code to execute.

B. JavaScript Interface Remote Code Execution

Android allows apps to expose app-level objects to JavaScript code running in a WebView by using a feature called the JavaScript Interface [19]. In 2012, a remote code execution attack on the JavaScript Interface was published [20]. Because JavaScript code has access to all of the public methods of objects added to the JavaScript Interface, malicious scripts could access the Java Reflection APIs by calling `getClass` (a method inherited from `java.lang.Object`) on the exposed object. From there, the malicious script could build any arbitrary Java object and execute arbitrary code.

Android addressed this vulnerability in API level 17 by forcing apps to annotate methods that should be callable from JavaScript code. Calling an unannotated method from JavaScript code does nothing. Because developers were unlikely to need to expose `getClass` to JavaScript code, this limited the damage that a malicious script could do. But this change is not applied for apps that target API levels 16 or lower. Apps that use the JavaScript Interface, target API levels 16 or lower, and load untrusted web content can be exploited.

We identify apps that use the JavaScript Interface by looking for calls to `addJavascriptInterface` in `smali` code. 50% of apps in Dataset A use the JavaScript Interface. Of these apps, 15% target API levels 16 or lower. Any of these apps that load untrusted JavaScript code in their WebView can be exploited. Identifying apps that can load untrusted JavaScript code is beyond the scope of this study so we cannot say what portion of these apps are exploitable, but we note that existing work has shown that it is not uncommon for apps to load untrusted web content in their WebViews [13, 15, 4].

C. Exported Content Providers

App components that manage access to structured data are called *Content Providers* [21]. Content Providers are declared in an app's manifest file, and they can be either made local to the app or exposed to other apps with the `exported` attribute. Unintentionally exported Content Providers that hold sensitive data are a security flaw as they are accessible to every app on the device. Yet if the `exported` attribute is not specified it falls back to a default value. API level 17 changed the default value to `false` but apps that target API levels 16 or below use a default value of `true`. Research has shown that 65% of apps that export a Content Provider leak private data [22].

9.7% of apps that target API levels 16 or lower and 8.0% of apps that target API levels 17 or higher include at least one exported Content Provider. 4.9% of apps that target API levels 16 or lower include a Content Provider that is exported due to default behavior. Assuming that the percentage of apps that want to export a Content Provider is uniform across both app populations we conclude that 1.7% of apps that target API levels 16 or lower unnecessarily export a Content Provider. These apps represent 0.3% of apps in Dataset A.

D. Fragment Injection

In 2013, security researchers identified a vulnerability in the `PreferenceActivity` class [23]. Malicious apps can send crafted messages to exported classes that inherit from `PreferenceActivity`. The messages are interpreted as `Fragment` instances and loaded dynamically using the Reflection APIs, executing arbitrary code from the malicious app.

API level 19 added the method `isValidFragment` to `PreferenceActivity` to close this vulnerability. Developers are expected to use this method to check the package name of injected fragments and reject unauthorized fragments. Apps that target API levels 19 or higher inherit an implementation of `isValidFragment` that always raises an exception and are therefore safe by default, but apps that target API levels 18 or lower inherit an implementation that always returns `true`, which offers no protection against this attack.

We find that 1.7% of apps that target API levels 18 or lower contain at least one exported class that inherits from `PreferenceActivity` and does not override the unsafe implementation of `isValidFragment`[6]. Unlike previous examples, this is sufficient information to prove that these apps are exploitable rather than just at heightened risk. The exploitable apps account for 0.5% of all apps in Dataset A.

E. Service Hijacking

A *Service* is an app component that performs operations without user interaction. UI components can interact with a Service using the method `bindService`. Apps specify which Service to interact with by using an Intent [24]. Intents can be explicit or implicit. Explicit Intents list a unique Service using its class name. Implicit Intents only specify a general action to perform and the system chooses an appropriate Service to handle the request. Communicating with a Service using an Implicit Intent is not safe. Because Services are not user facing components, users have no control over which Service responds to an Implicit Intent. If multiple Services match the Implicit Intent used in `bindService` then a *random* one of those Services is chosen. Malicious apps can create a Service that matches the Implicit Intent and impersonate trusted code. In apps that target API levels 21 or higher, passing an Implicit Intent to `bindService` throws a security exception.

83% of apps that target API levels 20 or lower contain at least one call to `bindService`. These apps make up 43% of all apps in Dataset A. Statically identifying which of these apps use Implicit Intents to bind Services is nontrivial and beyond the scope of this study. However, prior research [25] has found that 19% of apps are potentially vulnerable to Service Hijacking because they use Implicit Intents.

F. Detecting Outdated Apps

Google Play does not publish an app's target API level. Security conscious users are not empowered to make decisions based on target API levels and must instead trust developers to retarget their apps. It is essential to give users access to this information. To this end, we created an open-source tool [26] that reports the target API level of each app on a user's phone and notifies users of any security implications.

[6]We assume that all implementations of `isValidFragment` correctly validate the package name of injected fragments. Future work could expand on our results by looking for incorrect implementations of `isValidFragment`.

VI. Discussion

The data presented in this study makes two conclusions clear: that target fragmentation exists across the entire Android ecosystem and that target fragmentation has practical implications on Android security. Making new Android versions compatible with un-updated apps ensures that apps do not suddenly break, but distributing security changes in this manner has clear limitations. This approach makes security changes *optional* and mixes security changes with non-security changes. Developers cannot pick-and-choose just the security changes but must integrate *all* of the platform changes from a new API level. Developers could try to sidestep this problem by setting a maximum API level but this feature is not enforced and would exacerbate device fragmentation problem by discouraging platform updates.

With this in mind, we consider the alternative to the current system: enforcing all security changes to the Android platform regardless of target API level. Nearly four months since the release of API level 23, less than 1% of active devices have updated to version 23 [27]. This slow update process means that developers have ample time to update their apps to work with new behavior. There have also been security changes in the past that were both mandatory and breaking that did not appear to cause great pain. In Android level 21, a uniqueness requirement was added for custom permissions to ensure that malicious apps could not access protected content. This change was mandatory and could prevent apps from being reinstalled after a device update but searching on developer forums reveals few complaints. We suspect that if developers were simply forced to adjust to all security changes that it would not be a problem for the majority of apps. If this is not feasible then, at the very least, users should be informed if their apps target out-of-date API levels so that they can be empowered to make security conscious decisions.

The JavaScript Interface vulnerability described in Section V-B is a good case study to support our argument. After failing to fix the problem in API level 17 with an optional change, API level 19 banned all access to `getClass` from the JavaScript Interface. This change is *not* made optional for apps targeting lower API levels and would be unneeded if not for the existence of apps that target API levels 16 or lower. The vulnerability was only truly addressed with a change that is applied to all apps, regardless of their target API version.

A. Alternative Explanations

Ignorance is not the only reason why a developer might not target the most current API level. One alternative is that developers do not retarget their apps to the most current API level because few devices run the most current API level. If this were the case we would expect most apps to be no more than one API level out of date. Although 16% of devices were

running API levels 22 or 23 when Dataset A was collected, only 23% of apps in Dataset A targeted these levels.

Another possibility is that developers choose not to retarget apps to higher API levels unless there is a security concern. Because API levels 22 and 23 do not include security changes, developers may have chosen not to retarget their apps. However, we show in Section V that numerous apps target outdated API levels even though they miss relevant security changes (54% of apps target API levels below 21). If we look at the Fragment Injection vulnerability we find that apps that use a `PreferenceActivity` are not more likely to target API levels 19 or higher (66%) than the rest of the app population (69%). This suggests that developers, as a whole, do not consider security when deciding what API level to target.

A final option is that developers choose not to target the most current API level to avoid breaking critical app functionality. This is a real possibility but, if true, shows that Google's "all or nothing" design is flawed because it forces developers to make an impossible choice and sacrifice security.

B. Threats to Validity

We only downloaded a subset of available apps for Dataset A so there is some possibility of selection bias in our results. However, a dataset size of 60,086 apps is within the normal range for studies like ours. Because these apps were selected randomly from the available apps we believe that the dataset is large enough to smooth out any selection bias.

Our dataset is not a uniform snapshot of the apps available on the Google Play store because we do not download updated versions of previously downloaded apps. We do not have longitudinal statistics on individual apps and we tend to have young versions of apps. If apps that have been present on the app store for a very long time are considerably more likely to target current API levels then the statistics presented in this paper may overestimate the problem from the user's perspective. However, there is an 18 month gap between the collection of Dataset A and Dataset B. If long-lived apps target outdated API levels less frequently then we would expect to see some indication in the results for Dataset A.

Because we only study free apps, it is possible that our results do not apply to non-free apps. We have no reason to suspect that development practices are significantly different for non-free apps. However, even if there is a considerable difference between free and non-free apps, free apps comprise 89% of apps on the Google Play store [28] so any problem present in free apps is critical and should be addressed.

Much of the analysis done in Section V assumes that apps use certain dangerous features at uniform rates no matter which API level they target. If this assumption is not true then our conclusion that many apps unnecessarily use these dangerous features because they target outdated API levels might not be valid. In particular, it is possible that behavioral changes discourage developers from retargeting their apps, and we cannot assume that usage of dangerous features is uniform across the app population. However, the differences we observe in the usage of dangerous features are so great that it would be extremely surprising for these differences to be caused by different intended behavior in apps.

VII. Related Work

The most similar work to this study is by McDonnell, Ray, and Kim [29], who study the use of deprecated API methods and the adoption of updated API methods in ten open source Android apps. Unlike our study, which focuses on target API levels, they focus on the usage of methods changed in new API levels. Targeting an API level is not dependent on using methods added in that platform version and apps should target the most current API level even if they do not use any newly added methods. Their results say nothing about the security consequences of outdated apps but do show that developers are slow to adapt to the changing Android platform.

The target fragmentation problem has been discussed in relation to *specific* vulnerabilities in several studies. Thomas et al. [5] study the changes in Android 17 that closed the JavaScript Interface vulnerability in depth. Their study focuses on the slow adoption of new *devices* and its effect on the lifetime of the vulnerability but they also find that 22% of studied apps use the JavaScript Interface and targeted API levels below 17. Mutchler et al. [4] identify apps that load untrusted content in WebView and note that the JavaScript Interface vulnerability puts these apps at risk.

The vulnerabilities mentioned in this paper have been studied without mention of target fragmentation. Lu et al. [30] build a static analyzer to identify vulnerabilities including Service Hijacking. Georgiev, Jana, and Shmatikov [15] provide a tool to prevent attacks through the JavaScript Interface. Chin and Wagner [13] statically analyze apps and find unsafe use of `file:` URLs. Jin et al. [31] build a tool to detect a variety of XSS-like vulnerabilities in WebView apps.

Because *both* apps and devices must be updated in order to take advantage of new security features, target fragmentation and device fragmentation are linked. Thomas, Beresford, and Rice [2] study device fragmentation using volunteers who install their device monitor app. They find that 88% of devices are vulnerable to at least one of selected vulnerabilities and that devices are updated infrequently (1.26 times per year on average). Zhou et al. [32] find that more than 1,000 of 2,423 factory images can be exploited through misconfigurations of device drivers. Xing et al. [33] identify how apps can exploit the OS update process to obtain sensitive system permissions. Mulliner et al. [34] provide a scalable method for applying third party patches to vulnerable Android devices.

The app update process has also been studied. Möller et al. [35] investigate the update patterns of Android *users* and find that only half of users install an app update within one week of the update being published. McIlroy, Ali, and Hassan [36] mine update data from 10,713 apps and find that only 1% of apps receive at least one update per week.

Several other studies analyze large datasets of Android apps. Viennot, Garcia, and Nieh [10] crawl, download, and analyze 1,100,000 apps to obtain statistics about permission and library distributions as well as identify apps that unsafely embed

credentials. Other studies [37, 38] also study permission and library usage. Kavaler et al. [39] compare usage of Android classes and questions asked on StackOverflow.

VIII. Conclusion

Android apps specify a target API level and run in a compatibility mode on devices with higher API levels. The compatibility mode can disable important security changes in the Android platform. We call the problem of apps targeting outdated API levels the target fragmentation problem. In this study we analyze a dataset of more than one million Android apps collected over four years and show that the large majority of collected apps target outdated API levels. We examine the practical implications of target fragmentation on seven security changes to the Android platform and show that target fragmentation hamstrings new security features.

We believe that applying security changes in this optional manner is a flawed approach that sacrifices security at the altar of compatibility. Developers become a new obstacle to securing apps and users have no means of ensuring that their apps target the most current API levels. The target fragmentation problem is further compounded by the coupling of security changes and non-security changes. We hope that by shedding light on this problem, developers can become more aware of the consequences of targeting outdated API levels and this flawed design can be reexamined and changed so that there is less opportunity for Android apps to operate without access to important security features.

References

[1] I. Lunden. Android breaks 1b mark for 2014, 81% of all 1.3b smartphones shipped. [Online]. Available: http://techcrunch.com/2015/01/29/android-breaks-1b-mark-for-2014-81-of-the-1-3b-smartphones-shipped-in-total

[2] D. R. Thomas, A. R. Beresford, and A. Rice, "Security metrics for the android ecosystem," in *Proceedings of the 5th ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2015.

[3] Webview addjavascriptinterface remote code execution. [Online]. Available: http://labs.mwrinfosecurity.com/blog/2013/09/24/webview-addjavascriptinterface-remote-code-execution

[4] P. Mutchler, A. Doupé, J. Mitchell, C. Kruegel, and G. Vigna, "A large-scale study of mobile web app security," in *Mobile Security Techologies*, 2015.

[5] D. R. Thomas, A. R. Beresford, T. Coudray, T. Sutcliffe, and A. Taylor, "The lifetime of android api vulnerabilities: Case study on the javascript-to-java interface," *Security Protocols XXII*, 2015.

[6] App manifest. [Online]. Available: http://developer.android.com/guide/topics/manifest/manifest-intro.html

[7] Build.version_codes. [Online]. Available: http://developer.android.com/reference/android/os/Build.VERSION_CODES.html

[8] <uses-sdk>. [Online]. Available: http://developer.android.com/guide/topics/manifest/uses-sdk-element.html

[9] Security tips. [Online]. Available: http://developer.android.com/training/articles/security-tips.html

[10] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," in *Proceedings of the 2014 ACM Conference on Measurement and Modeling of Computer Systems*, 2014.

[11] Apktool. [Online]. Available: http://code.google.com/p/android-apktool

[12] Webview. [Online]. Available: http://developer.android.com/reference/android/webkit/WebView.html

[13] E. Chin and D. Wagner, "Bifocals: Analyzing webview vulnerabilities in android applications," in *Information Security Applications*, 2014.

[14] D. Wu and R. K. Chang, "Analyzing android browser apps for file://vulnerabilities," in *Information Security*, 2014.

[15] M. Georgiev, S. Jana, and V. Shmatikov, "Breaking and fixing origin-based access control in hybrid web/mobile application frameworks," in *Proceedings of the 2014 Network and Distributed System Security Symposium*, 2014.

[16] M. Backes, S. Gerling, and P. von Styp-Rekowsky, "A local cross-site scripting attack against android phones," 2011. [Online]. Available: http://infsec.cs.uni-saarland.de/projects/android-vuln/android_xss.pdf

[17] What is mixed content? [Online]. Available: http://developers.google.com/web/fundamentals/security/prevent-mixed-content/what-is-mixed-content

[18] Mixed content blocking in firefox. [Online]. Available: http://support.mozilla.org/en-US/kb/mixed-content-blocking-firefox

[19] Binding javascript code to android code. [Online]. Available: http://developer.android.com/guide/webapps/webview.html#BindingJavaScript

[20] N. Bergman. Abusing webview javascript bridges. [Online]. Available: http://50.56.33.56/blog/?p=314

[21] <provider>. [Online]. Available: http://developer.android.com/guide/topics/manifest/provider-element.html

[22] Y. Z. X. Jiang, "Detecting passive content leaks and pollution in android applications," in *Proceedings of the 20th Network and Distributed System Security Symposium*, 2013.

[23] R. Hay. A new vulnerability in the android framework: Fragment injection. [Online]. Available: http://securityintelligence.com/new-vulnerability-android-framework-fragment-injection

[24] Intents and intent filters. [Online]. Available: http://developer.android.com/guide/components/intents-filters.html

[25] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, 2011.

[26] Api parser. [Online]. Available: http://github.com/yeganehs/API-Parser

[27] Dashboards. [Online]. Available: http://developer.android.com/about/dashboards/index.html

[28] Google Play Stats. [Online]. Available: http://appbrain.com/stats

[29] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *Proceedings of the 29th IEEE Conference on Software Maintenance*, 2013.

[30] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[31] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[32] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in android device driver customizations," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

[33] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang, "Upgrading your android, elevating my malware: Privilege escalation through mobile os updating," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

[34] C. Mulliner, J. Oberheide, W. Robertson, and E. Kirda, "Patchdroid: Scalable third-party security patches for android devices," in *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013.

[35] A. Möller, F. Michahelles, S. Diewald, L. Roalter, and M. Kranz, "Update behavior in app markets and security implications: A case study in google play," in *Proceedings of the 3rd Workshop on Research in the Large.*, 2012.

[36] S. McIlroy, N. Ali, and A. E. Hassan, "Fresh apps: An empirical study of frequently-updated mobile apps in the google play store," in *Empirical Software Engineering*, 2015.

[37] M. L. Dering and P. McDaniel, "Android market reconstruction and analysis," in *Proceedings of the 2014 IEEE Military Communications Conference*, 2014.

[38] R. Johnson, Z. Wang, C. Gagnon, and A. Stavrou, "Analysis of android applications' permissions," in *Proceedings of the 2012 IEEE Conference on Software Security and Reliability Companion*, 2012.

[39] D. Kavaler, D. Posnett, C. Gibler, H. Chen, P. Devanbu, and V. Filkov, "Using and asking: Apis used in the android market and asked about in stackoverflow," in *Social Informatics*, 2013.