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 We begin by demonstrating that the majority of sampled apps target outdated API levels.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Apps that target more outdated API levels are more likely to miss crucial security changes. We are not just interested in if an app is outdated but also in how outdated an app is.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 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By examining the low end of the curve we see that the large majority of apps target outdated API levels. The long tail at the top of the curve shows us that a considerable number of apps target API levels that are many years out of date.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normal distribution, with the percentage of apps targeting each API level decreasing as API levels get more and more out-of-date.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?

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 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-daC. 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 times have a mean outdatedness of 607 days and a mean negligent outdatedness of 493 days.te API level.

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 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.

四. 评价

在本节中, 我们将量化目标碎片问题的程度。我们首先证明, 大多数取样的应用程序都以过时的 API 级别为目标。我们定义了一个 outdatedness 度量, 用于测量单个应用程序以及跨应用程序的总体目标碎片问题的严重性。我们表明, 目标碎片问题主要是由开发者疏忽造成的, 而不是在 app 商店闲置的应用程序。我们比较流行和不受欢迎的应用程序之间的 outdatedness 度量和证明目标碎片甚至是最流行的应用程序中的一个问题。最后, 我们表明, outdatedness 曲线在四最近收集的数据集中是相似的。这一结果表明, 除非重新审视目标碎片问题, 否则将来可能会以相同的规模继续进行。

由于我们的数据集规模很大, 我们认为这些结果广泛应用于整个 Google Play商店生态系统。

A. 今天的目标碎片

图1显示了数据集 A (包含最新应用程序的数据集) 中的应用程序的目标 API 级别的分布。立即清楚的是, 绝大多数应用程序都不针对 api 级别 23, 这是收集数据集 A 时最新的 api 级别。更确切地说, 我们发现数据集中的93% 应用程序是目标 API 级别22或更低。

针对更过时的 API 级别的应用程序更有可能错过关键的安全更改。我们感兴趣的是, 如果一个应用程序是过时的, 但也在如何过时的应用程序。我们定义一个称为 "outdatedness" 的量化度量值, 作为应用程序的目标 API 级别的发布日期和收集应用程序时最新 API 级别的发布日期之间的差异 (以天为例)。我们发现一个中值 outdatedness 704 天和平均 outdatedness 686 天的应用程序在数据集 a。我们可以使用累计分布的 outdatedness (图 2) 作为衡量目标碎片问题的严重性在一个应用程序的入口。

通过检查曲线的底端, 我们看到大多数应用程序都以过时的 API 级别为目标。曲线顶端的长尾显示了大量的应用程序目标 API 级别, 这是多年过时的。

我们可能期望目标 API 级别的分布类似于一个倾斜的正态分布, 随着 api 级别越来越过时, 针对每个 api 级别的应用程序的百分比都在下降。但是, 如图1所示, 我们发现, 针对 API 级别7到16的应用程序的百分比相对持平。针对 api 级别7的应用程序几乎和 api 级别16一样多。API 级别9特别明确, 在数据集 A 中几乎有3% 的应用程序被瞄准, 尽管它不提供任何关键的兼容性功能。为什么应用程序针对这样一个过时的 API 级别？

如果我们分析数据集中的原始目标 api 级别, 我们发现8.2% 的应用程序要么不包括原始目标 api 级别, 要么设置一个低于其最小 api 级别的值, 要么具有其他无效的原始目标 api 级别。这些应用程序使用其最低 api 级别作为其目标 api 级别。这些应用程序占了目标 API 级别分布的大部分长尾。例如, 63% 针对 api 级别15或更低的应用程序这样做是因为它们使用其最小 api 级别作为目标 api 级别。通过说服开发人员正确使用 targetSdkVersion 属性, 这是消除目标 API 级别分布的长尾的一个主要机会。

B. 陈旧的应用程序

并不是所有 Google Play商店的应用程序都是由开发者定期维护的。图3显示了应用程序集合之间的天数分布以及它的开发人员将其上传到 Google Paly 商店的时间。请注意, 此日期可能是首次发布应用程序或最新的应用程序更新被推送的日期。在2015年10月发布 API 级别23之后, 数据集 A 中仅有38% 的应用程序被上传。显然, 在 GooglePlay上停滞不前的应用程序不会在发布时针对新的 API 级别。未维护应用程序的存在会扭曲目标 API 的分布, 并且错误地暗示那些积极维护其应用程序的开发人员无法更新其应用程序以达到新 api 级别的目标。在这里, 我们试图区分 outdatedness 引起的未维护应用程序和 outdatedness, 坚持通过应用程序维护。

我们将应用程序的 "疏忽 outdatedness" 定义为其目标 API 级别的发布日期与在将其上传到 GooglePlay商店时最新 api 级别的发布日期之间的差异 (以天为间隔)。疏忽 outdatedness 措施错失了开发商将其应用目标对准当前 API 水平的机会。图4描述了一个具体的例子来澄清 outdatedness 和疏忽 outdatedness 的区别。我们发现一个中位疏忽 outdatedness 为377 天和平均疏忽 outdatedness为 536 天之间的应用程序在数据集 A。

我们可以更慷慨的定义疏忽 outdatedness, 包括一些滞后时间, 允许开发人员在新的 API 级别发布后重新将他们的应用程序。在应用程序被上传到 app 商店时, 我们选择的是最新的 api 级别, 而不是在应用程序被上传到应用程序商店时, 至少可以使用 N 天。我们把这个滞后时间称为 "领养窗口"。增加领养窗口30天只轻微地影响疏忽 outdatedness, 减少中位疏忽 outdatedness 到327天和平均疏忽 outdatedness 到496天。

图5显示了使用30天的收养窗口的疏忽 outdatedness 的累积分布。即使使用了一个慷慨的收养窗口, 应用程序仍然无法针对适当的 API 级别。79% 的应用程序被疏忽地针对过时的 API 级别, 这意味着这些应用程序一旦上传到 app商店, 就会过时。开发人员在发布新的 Android 平台之前就可以访问它, 因此, 在新的 API 级别发布时间为0时, 应用程序就有可能是最新的。从这些结果可以清楚地看出, 目标碎片问题不能由陈旧的应用程序解释, 而是由于忽视了针对过期 API 级别的结果, 或者由于故意选择目标过时的 API 级别。

C. 热门应用

大多数用户都没有下载 GooglePlay商店的大部分应用程序。在数据集 a 中仅有2% 的应用程序安装了至少100万次, 但这些应用程序在数据集 a 的应用程序中占总安装的74%。

了解应用程序流行与目标碎片问题之间的关系是很重要的。图6比较了不同 最流行应用程序之间的累计 outdatedness 分布。我们看到, outdatedness 曲线是非常相似的, 最流行的应用程序 (安装至少100万次) 的目标只是少量的过时 API 水平。比较不同 最流行应用程序之间的疏忽 outdatedness 分布给出了类似的结果 (我们不包括在这里的空间原因)。安装了至少100万次的应用程序平均的 outdatedness 为607 天，平均疏忽 outdatedness 为493 天。

D. 随着时间的推移, 目标碎片化

从数据集的分析中清楚地看出, 目标碎片是当今的一个严重问题。即使是最热门的应用程序也以过时的、通常非常过时的 API 级别为目标。通过分析在不同日期收集的数据集, 我们可以看到问题是如何随时间变化的。我们对其余四个数据集重复前面的分析, 并将这些结果与数据集 A 的结果进行比较。

图7比较了我们的五个数据集的 outdatedness 分布。随着时间的推移, 只有一个明显的趋势: 越来越多的应用程序的尾部以非常过时的 API 级别为目标, outdatedness 的第九十位数超过了数据集 E 和 A之间的加倍。这是目标碎片问题的自然属性, 因为最大 outdatedness 随时间而增长。只要有针对最低 API 水平的应用程序, 我们希望看到这条尾巴继续增长。

比较 outdatedness 曲线的底端不显示明显的模式。我们看到, 在每个数据集中, 绝大多数应用程序的目标都是过时的 API 级别, 而且, 除了数据集 E 之外, outdatedness 曲线的下端似乎没有显著的差异。我们注意到, 数据集 A 和 C 分别在平台发布后两个月收集, 数据集 B 和 D 在平台发布之后七和九月收集。这种差异对 outdatedness 曲线的低端的影响比任何模式都要大, 在数据集 B 和 D 中有超过20% 的应用程序针对当前的 api 级别, 在数据集 a 和 C 中针对当前 api 级别的应用程序少于10%。

我们的数据集之间有一个非常有希望的趋势。我们发现没有指定原始目标 api 级别 (图 8) 的应用程序百分比的明显下降趋势, 因此将目标 api 级别设置为其最小 api 级别。由于开发人员经常希望支持尽可能多的设备, 因此应用程序通常具有非常低的最低 API 级别, 这使得无法指定目标 api 级别非常危险。这一趋势表明, 开发人员对目标 api 功能的认识越来越高, 针对其最低 API 级别的开发人员数量将随着时间的推移而消失。

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 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. 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 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-URL5 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.

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 JavaScriptcode 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 isValidFragment6. 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.

五. 所涉安全问题

目标碎片问题意味着, 只要在兼容模式下禁用更改, 对 Android 平台的任何安全更改都将更有效。在本节中, 我们将探讨目标碎片问题对 Android 平台更改的实际影响的七个安全影响。针对过时 API 级别的应用程序可能不一定是易受攻击的, 但会对安全漏洞造成更高的风险。在所有情况下, 我们只表明应用程序不必要地处于更高的风险中。因为谷歌认为过时的行为不安全, 足以让 Android 平台得到改变, 因此, 广泛使用过时的行为本身就会令人不安。我们还引用研究表明, 这些应用程序所需的条件被利用是频繁的。本节中的统计信息是在数据集 A (最近收集的数据集) 上计算的。

A.WebView默认值

WebView [12] 是一个 UI 元素, 它充当 Android 应用程序中的嵌入式 web 浏览器。在数据集 A 中, 91% 的应用程序至少包含一个 WebView 实例。通过针对过时的 API 级别来禁用 WebView 的默认行为, 有三个主要的安全更改。以下各节描述这些更改, 并显示针对当前 API 级别的应用程序不太可能使用不安全的行为。

1) 文件方案同源策略: 根据同源策略, JavaScript 代码只能访问从同一原点加载的内容。处理所有文件: url 属于同一原点是具有相互不文件的系统的安全风险。如果 WebView使用文件 (URL) 加载任何不受信任的内容, 则该内容可以完全访问应用程序可访问的其他所有文件。在 api 级别16中, WebView的默认行为被更改为处理所有文件: url 属于单独的来源, 但是此更改仅适用于针对 api 级别16或更高的应用程序。希望重写此行为并处理所有文件的应用程序: 属于同一来源的 url 可以使用 setAllowFileAccessFromFileURLs 和 setAllowUniversalAccessFromFileURLs 的方法。

我们发现, 82% 的应用程序的目标 api 级别15或更低, 18% 的应用程序的目标 api 级别16或更高的使用 "不安全的文件: url" 策略。由于目标 API 级别为16或更高且希望使用不安全策略的应用程序必须显式执行, 因此, 我们可以使用这些应用程序的百分比, 包括调用 setAllowFileAccessFromFileURLs 或 setAllowUniversalAccessFromFileURLs 作为应用程序的上限要使用不安全策略的如果我们假设这个百分比在两个应用程序群体中都是一致的, 那么我们得出结论: 64% 的应用程序的目标 api 级别15或更低不必要地使用不安全策略, 如果它们被重到 api 级别 16, 将变得更加安全。这些应用程序代表了数据集 A 中所有应用程序的9.6%。我们在第六 A 节讨论这一假设的有效性。

允许不安全文件访问的应用程序只会有更高的风险。要被利用, 应用程序必须使用文件加载不受信任的内容: URL。应用程序可能被利用的两种方式是加载文件: 从其他应用程序接收的 url 或导航到不受信任的网页, 这可能会删除磁盘上的文件并将 WebView重定向到该文件。可以参考 Chin和 Wagner [13] 为这个利用的详细的描述。研究表明, 23% 的 web 浏览器应用程序安全加载文件: 来自外部应用程序的 url [14] 和 WebView应用程序频繁加载不受信任的网页 [13, 15, 4]。

2) JavaScript url: 应用程序可以通过调用 loadUrl 在 WebView中加载 web 内容。在目标 API 级别18或更低的应用程序中, 在 JavaScript Pseudo-URL5 上调用 loadUrl 执行当前呈现的网页中的脚本。在加载从外部应用程序检索到的未过滤的 url 的应用程序中, 这种行为是可利用的。恶意应用程序可以发送一个请求, 在 foo.com 加载一个 URL, 然后发送一个脚本, 在 foo.com 的上下文中执行以读取私有内容。

此攻击称为跨应用程序脚本 [16]。目标 API 级别19或更高的应用程序在一个空上下文中加载 javascript Psuedo-URLs必须使用方法 evaluateJavascript 在当前的 WebView 上下文中执行 javascript 代码。我们发现, 60% 的应用程序的目标 API 级别19或更高, 至少有一个调用此方法。同样, 我们可以以这种方式使用作为一个上限的数量的应用程序故意执行 JavaScript 代码。因为90% 的应用程序的目标 api 级别18或更低, 包括一个 WebView 和启用 javascript, 我们得出的结论是, 30% 的应用程序的目标 api 级别18或更低, 没有必要以这种方式评估 javascript 代码, 并会变得更安全, 如果他们被更新到 API 级别19。这些应用程序代表了数据集 A 中所有应用程序的9.4%。

3) 混合内容: 包含通过 HTTP 进行检索的 web 元素 (当它通过 HTTPS 加载时) 被认为包含混合内容。加载混合内容是一种安全风险, 一些主要浏览器会阻止混合内容 [17、18]。在目标 API 级别21或更高的应用程序中, 默认情况下, web 会阻止混合内容。希望重写此行为并允许混合内容的应用程序可以使用方法 setMixedContentMode 指定自定义策略。

我们发现76% 的应用程序的目标 api 级别20或更低, 42% 的应用程序的目标 api 级别21或更高允许混合内容。如果我们假设希望允许混合内容的应用程序的百分比在两个应用程序群体中都是一致的, 那么我们得出的结论是, 34% 的应用程序的目标 API 级别20或更低不必要地允许混合内容, 如果它们被重到API 级别21。这些应用程序代表了数据集 A 中所有应用程序的18%。

B. JavaScript 接口远程代码执行

Android 允许应用程序通过使用称为 javascript 接口 [19] 的功能将应用程序级对象暴露在 WebView运行的 javascript 代码中。在 2012年, 在 JavaScript 接口上发布了远程代码执行攻击 [20]。

因为 javascript 代码可以访问添加到 javascript 接口的所有公共方法, 所以恶意脚本通过调用 getClass (从 java. lang 对象继承的方法) 对已公开的对象访问 java 反射 api。从那里, 恶意脚本可以生成任意的 Java 对象并执行任意代码。

Android 通过强制应用程序对应该从 JavaScript 代码中调用的方法进行注释来解决 API 级别17中的此漏洞。从 JavaScript 代码调用 unannotated 方法不做任何事情。因为开发人员不太可能需要向 JavaScript 代码公开 getClass, 这就限制了恶意脚本所能造成的损害。但此更改不适用于目标 API 级别16或更低的应用程序。可以利用 JavaScript 接口、目标 API 级别16或更低的应用程序以及加载不受信任的 web 内容。

我们通过在 smali 代码中查找 addJavascriptInterface 的调用来识别使用 JavaScript 接口的应用程序。50% 的应用程序在数据集 A 使用 JavaScript 接口。在这些应用程序中, 15% 目标 API 级别16或更低。这些在 web 中加载不受信任的 JavaScript 代码的应用程序都可以被利用。识别可以加载不受信任的 JavaScriptcode 的应用程序超出了本研究的范围, 因此我们不能说这些应用程序的哪一部分是可利用的, 但我们注意到, 现有的工作已经表明, 应用程序在其 WebViews 中加载不受信任的 web 内容并不少见。[13, 15, 4]。

c. 导出的内容提供商

管理对结构化数据访问的应用程序组件称为内容提供程序 [21]。内容提供程序是在应用程序的清单文件中声明的, 并且它们可以是本地应用程序或使用导出的属性公开给其他应用程序的。

无意间导出的内容提供程序持有敏感数据, 这是一个安全缺陷, 因为它们可由设备上的每个应用程序访问。但是, 如果未指定导出的属性, 则返回默认值。api 级别17将默认值更改为 false, 但目标 api 级别16或以下的应用程序使用默认值 true。研究表明, 65% 的应用程序导出内容提供程序泄漏私有数据 [22]。

9.7% 的应用程序的目标 api 级别16或更低, 8.0% 的应用程序的目标 api 级别17或更高, 包括至少一个出口内容提供商。4.9% 目标 API 级别16或更低的应用程序包括由于默认行为而导出的内容提供程序。假设要导出内容提供商的应用程序的百分比在两个应用程序群体中都是一致的, 我们得出结论: 1.7% 的应用程序的目标 API 级别16或更低不必要地导出内容提供程序。

这些应用程序代表了数据集 A 中0.3% 的应用程序。

d. 碎片注射

在 2013年, 安全研究人员发现了 PreferenceActivity 类 [23] 中的一个漏洞。恶意应用程序可以向从 PreferenceActivity 继承的导出类发送精心编制的消息。这些消息被解释为片段实例, 并使用反射 api 动态加载, 从恶意应用程序执行任意代码。

API 级别19将方法 isValidFragment 添加到 PreferenceActivity 以关闭此漏洞。开发人员应使用此方法检查注入片段的包名, 并拒绝未经授权的片段。目标 api 级别19或更高的应用程序继承 isValidFragment 的实现, 它始终会引发异常, 因此默认情况下是安全的, 但目标 api 级别18或更低的应用程序继承始终返回 true 的实现, 它不提供防范此攻击。我们发现1.7% 的应用程序的目标 API 级别18或更低, 至少包含一个从 PreferenceActivity 继承的导出类, 并且不覆盖 isValidFragment6 的不安全实现。与以前的例子不同, 这是充分的信息, 以证明这些应用程序是可利用的, 而不是只是在高风险。

可开发的应用程序占数据集 A 中所有应用程序的0.5%。

e. 服务劫持

服务是一个应用程序组件, 在没有用户交互的情况下执行操作。UI 组件可以与服务交互, 方法是使用 bindService 应用程序指定用意图 [24] 与之交互的服务。意图可以是显式或隐式的。显式的意图使用其类名列出唯一的服务。隐式意图仅指定要执行的常规操作, 系统选择适当的服务来处理请求。使用隐式意图与服务进行通信不安全。由于服务不是面向用户的组件, 因此用户无法控制哪个服务响应隐式意图。如果多个服务与 bindService 中使用的隐含意图相匹配, 则选择这些服务中的任意一个。恶意应用程序可以创建与隐式意图相匹配的服务, 并模拟受信任的代码。

在目标 API 级别为21或更高的应用程序中, 向 bindService 传递隐式意图会引发安全异常。83% 的应用程序的目标 API 级别20或更低, 至少有一个调用 bindService。这些应用程序组成了43% 的所有应用程序的数据集 a. 静态地确定哪些应用程序使用隐含的意图绑定服务是不平凡的, 而且超出了本研究的范围。然而, 先前的研究 [25] 发现, 19% 的应用程序可能容易受到服务劫持, 因为它们使用隐式意图。

f. 检测过时的应用程序

谷歌play商店不发布应用程序的目标 API 级别。

安全意识的用户没有权力使 decisionsbased 在目标的 API 级别, 而必须信任开发者升级他们的应用程序。为用户提供对此信息的访问是非常重要的。为此, 我们创建了一个开源工具 [26], 它报告用户手机上每个应用程序的目标 API 级别, 并通知用户任何安全隐患

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 obshat it would be extremely surprising for these differences to

be caused by different intended behavior in apps.erve in the usage of dangerous features are so great.

I. 讨论

本研究提供的数据有两个结论: 目标碎片存在于整个 android 系统中, 目标碎片对 android 安全有实际意义。使新的 Android 版本与联合国更新的应用程序兼容, 确保应用程序不会突然中断, 但以这种方式分发安全更改具有明显的局限性。此方法使安全更改可选, 并将安全更改与非安全更改混合在一起。开发人员不能只选择安全更改, 但必须将所有平台更改从新的 API 级别集成。开发人员可以尝试通过设置最大 API 级别来避开此问题, 但此功能不会强制执行, 并通过阻止平台更新来加剧设备碎片问题。

考虑到这一点, 我们认为是当前系统的替代方案: 对 Android 平台实施所有安全更改, 而不管目标 API 级别如何。近四月以来, API 23的发布, 不到1% 的活动设备已更新到版本 23 [27]。这个缓慢的更新过程意味着开发者有足够的时间来更新他们的应用程序来处理新的行为。过去也有安全方面的变化, 既有强制性的, 也有破坏的, 似乎没有造成巨大的痛苦。在 Android API 21中, 为自定义权限添加了唯一性要求, 以确保恶意应用程序无法访问受保护的内容。此更改是强制性的, 可以防止在设备更新后重新安装应用程序, 但在开发人员论坛上搜索却很少有抱怨。我们怀疑, 如果开发人员只是被迫调整到所有的安全更改, 它不会是一个问题的大多数应用程序。如果这是不可行的, 那么至少, 应该通知用户, 如果他们的应用程序的目标是过时的 API 级别, 以便他们可以有权作出安全意识的决定。

第五-b 节中描述的 JavaScript 接口漏洞是支持我们的论点的一个很好的案例研究。在 API 17中没有用可选的更改解决问题后, api 级别19禁止从 JavaScript 接口访问 getClass。对于针对较低 api 级别的应用程序, 此更改不是可选的, 如果不是针对 api 级别16或更低的应用程序存在, 则不需要这样做。该漏洞只是真正解决的变化,

适用于所有应用程序, 无论其目标 API 版本如何。

A. 备选解释

不了解不是开发人员可能针对最新 API 级别的唯一原因。一种选择是, 开发人员不更新他们的应用程序到最新的 api 级别, 因为很少有设备运行最新的 api 级别。如果是这样的话, 我们希望大多数应用程序不超过一个 API 级别的日期。尽管在收集数据集 a 时, 16% 的设备正在运行 API 级别22或 23, 但数据集中的应用程序中只有23% 是针对这些级别的。另一种可能性是, 除非存在安全问题, 否则开发人员选择不将应用程序更新到更高的 API 级别。

因为 API 22和23不包括安全更改, 所以开发人员可能选择不重新他们的应用程序。然而, 我们在第五节显示, 许多应用程序的目标是过时的 api 级别, 即使他们错过了相关的安全更改 (54% 的应用程序目标 api 级别低于 21)。如果我们看一下片段注入漏洞我们发现使用 PreferenceActivity 的应用程序比应用程序的其余部分 (69%) 更容易目标 API 级别19或更高 (66%)。这表明,作为一个开发商不确定目标的 API 级别时, 请考虑安全性。

最后一个选择是开发人员选择不针对最新的 API 级别, 以避免破坏关键的应用程序功能。这是一个真正的可能性, 但如果真的, 表明谷歌的 "全部或没有" 的设计是有缺陷的, 因为它迫使开发商做出一个不可能的选择和牺牲安全。

B. 对有效性的威胁

我们只为数据集下载了一个可用的应用程序子集, 因此在我们的结果中存在选择偏差的可能性。然而, 60086个应用程序的数据集大小在正常范围内, 就像我们的研究一样。因为这些应用程序是从可用的应用程序中随机选择的, 所以我们认为数据集足够大, 可以避免任何选择偏差。

我们的数据集不单单来自Google play商店 ，因为我们不下载更新版本的以前的应用程序。我们没有关于单个应用程序的纵向统计数据, 我们倾向于有新版本的应用程序。如果应用程序商店已经存在了很长的时间, 更有可能的目标是当前的 API 水平, 那么本文中的统计数据可能会高估问题从用户的角度。然而,数据集 A和数据集 B 的集合，有18月之间的差距。如果长时间的应用程序的目标是过时的 API 级别, 那么我们希望在数据集 A 的结果中看到一些指示。

因为我们只研究免费的应用程序, 所以我们的结果可能不适用于非免费的应用程序。我们没有理由怀疑, 对于非免费应用程序, 开发实践有明显的不同。然而, 即使自由和非免费的应用程序之间也有相当大的区别, 免费的应用程序在 Google 游戏商店中占了89% 的总应用程序数量 [28]， 所以在免费应用中存在的任何问题都是至关重要的, 应该加以解决。

第五节中的许多分析都假定应用程序使用特定的危险特性, 无论其目标是什么 API 级别。如果这个假设是不正确的, 那么我们的结论是, 许多应用程序不必要地使用这些危险特性, 因为它们的目标是过时的 API 级别可能无效。特别是, 行为变化可能阻止开发者重他们的应用程序, 而且我们不能假设在应用程序群体中使用危险特性是统一的。然而, 我们在使用危险特性时所观察到的差异是如此之大, 以至于这些差异会因应用程序中不同的预期行为而引起。

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. Moller 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.

VII. 相关工作

这项研究最相似的工作是由McDonnell, Ray,and Kim [29],——研究使用过时的 API 方法在十个开源Android 应用程序中采用了更新的 API 方法。与我们的研究不同, 它侧重于目标 API级别, 他们侧重于在新的API 级别方法中更改的用法。针对 API 级别的目标不依赖于使用在该平台版本和应用程序中添加的方法。针对最新的 API 级别, 即使它们不使用任何新的添加的方法。他们的结果说过时的应用程序与安全无关, 但确实表明, 开发人员很慢, 无法适应不断变化的 Android 平台。

对目标碎片问题进行了讨论几个研究中的特定漏洞的关系。Thomaset al. [5] 研究了 Android API 17 关闭JavaScript 接口漏洞的深度的变化。他们重点研究新设备的缓慢采用及其对生存期的漏洞, 但他们也发现, 22% 研究的应用程序使用 JavaScript 接口和目标 API等级低于17。Mutchler et al. [4] 识别加载的应用程序web 中不受信任的内容, 并注意 JavaScript接口漏洞使这些应用程序面临风险。

本文中提到的漏洞已被研究, 而没有提到目标碎片。Lu et al.[30]。

构建静态分析器来识别漏洞, 包括服务劫持。Georgiev, Jana, 和 Shmatikov [15] 提供

一种防止通过 JavaScript 接口进行攻击的工具。 Chin 和Wagner [13] 静态分析应用程序和发现不安全的使用的文件: 网址。Jin et al. [31] 建立一个工具来检测各种web 应用程序中的类似 XSS 的漏洞。

因为必须更新应用程序和设备才能利用新的安全功能, 目标碎片和设备碎片链接。Thomas, Beresford 和Rice [2]研究设备碎片使用志愿者安装他们的设备监视器应用程序。他们发现88% 的设备容易受到至少一个选定的漏洞和该设备不经常更新 (平均每年1.26 次)。 Zhou et al. [32] 在2423工厂中超过1000家图像可以通过错误设备驱动程序。Xing et al.. [33] 识别应用程序如何利用操作系统更新过程以获取敏感的系统权限。Mulliner et al. [34]提供了一种可伸缩的应用方法通过第三方补丁到易受攻击的 Android 设备。

我们还研究了 app 更新过程。Moller et al [35] 调查 Android 用户的更新模式发现只有一半的用户在应用程序更新发布更新的一周内安装了。McIlroy Ali和Hassan [36]从10713个应用的更新数据里发现只有1% 的应用程序每周至少更新一次。

其他一些研究分析了 Android 应用的大数据采集。 Garcia和Nieh [10] 抓取、下载和分析110万应用程序获取有关权限和库的统计信息分布以及识别安全嵌入的应用程序凭据.其他研究 [37, 38] 也研究允许和库使用。Kavaler et al. [39]比较了使用了 Android在

StackOverflow 的相关课程和问题。

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.

VIII. 结论

Android 应用程序指定一个目标 API 级别, 并在具有较高 API 级别的设备上的兼容模式运行。兼容模式可以禁用Android 平台重要的安全更改。我们叫这个问题为针对过时API级别的应用程序目标碎片化问题。我们花了四年里研究收集分析了超过 100万 Android 应用，显示大部分收集的应用的目标是过时的 API 级别。我们研究目标碎片化对Android 平台七个安全变化的实际影响 , 并显示目标碎片化对新的安全功能的削弱。我们认为, 应用可选的安全更改方式是一个有缺陷的方法, 牺牲安全性来确保兼容性。开发商为了保证应用程序和用户的安全成为新的障碍，没有办法确保他们的应用程序的目标是最新的 API 级别。目标碎片化问题进一步复杂化加剧安全更改和非安全更改。我们希望通过在这个问题上, 开发者可以变得更意识到以过时的 API 级别为目标的后果，对这种有缺陷设计可以重新审视和改变, 以便Android应用运行时不会产生重要的安全功能。