An Analysis of the Mobile Apps' Third-party Resources Loading

Anonymous Author(s)

ABSTRACT

Like websites, mobile apps import a range of external resources from various third-party domains. In succession, the third-party domains can further load resources hosted on other domains. For each mobile app, this creates a dependency chain underpinned by a form of implicit trust between the app and transitively connected third-parties. Hence, such implicit trust may leave apps' developers unaware what resources are loaded within their apps. In this work, we perform a large-scale study of dependency chains in 7,048 free Android mobile apps. We characterize the third-party resources used by apps, and explore the presence of potentially malicious resources loaded via implicit trust. We find that around 99% of apps (with number of installs ≥ 500K) have dependency chains compared to 98% of apps with number of installs ≤ 100K. We find many different types of resources, most notably JavaScript codes, which may open the way to a range of exploits. These JavaScript codes are implicitly loaded by 92.3% of Android apps. Using VirusTotal, we classify 1.18% of third-party resources as suspicious. Our observations raise concerns for how apps are currently developed, and suggest that more rigorous vetting of in-app third-party resource loading is required.

ACM Reference Format:

Anonymous Author(s). 2020. An Analysis of the Mobile Apps' Third-party Resources Loading. In *Proceedings of the WWW'20, May 20–24, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3308558

1 INTRODUCTION

Mobile apps have become extremely popular [90], however, recently there has been a flurry of research [41, 48, 78, 89] exposing how many of these apps carry out misleading or even malicious activities. These acts range from low-risk (*e.g.,* usage of services and inter process communication which may drain the battery, CPU or memory) to high-risk (*e.g.,* harvesting data and ex-filtrating to third-parties [44, 48, 70, 83]).

We are interested in understanding the root source of this suspicious (or *malicious*) activity. Past work has treated this question as trivial—naturally, the root source of suspicious activity is the app's developer [40, 41, 73]. However, in this paper, we counter this assumption and shed light on the true complexity of suspicious app activity. We focus on the use of dynamically loaded *third-party* resources within apps. Mobile apps often load these resources from a range of third-party domains which include, for example, ad providers, tracking services, content distribution networks (CDNs) and analytics services. Although loading these resources is a well known design decision that establishes *explicit trust* between mobile apps and the domains providing such services, it creates complexity in terms of attribution. For example, it is not clear whether an app developer *knows* the third party resources are suspicious. This is further complicated by the fact that certain third-party code can further load resources from other domains. This creates a *dependency chain*, where the first-party app might not even be aware of the resources being loaded during its execution. This results in a form of *implicit trust* between mobile apps and any domains loaded further down the chain.

Consider the example `BBC News` [10] Android mobile app (see Figure 1b) which loads JavaScript code from the `widgets.com` domain, which, upon execution loads additional content from another third-party, `ads.com`. Here, `BBC News` as the first-party, *explicitly* trusts `widgets.com`, but *implicitly* trusts `ads.com`. This can be represented as a simple dependency chain in which `widgets.com` is at level-1 and `ads.com` is at level-2 (see Figure 1b). Past work tends to ignore this, instead, collapsing these levels into a single set of third-parties [37, 67]. This, however, overlooks a vital security aspect for resources loaded by mobile apps. For instance, it creates a significant security challenge, as mobile apps lack visibility on resources loaded further down their domain's dependency chain. The dynamic nature of the content being loaded and the wide adoption of in-path traffic alterations [31, 76] further complicates the issue. The potential threat should not be underestimated as errant active content (*e.g.,* JavaScript code) opens the way to a range of further exploits, *e.g.,* Layer-7 DDoS attacks [71] or massive ransomware campaigns [79].

In this work, we study dependency chains in Android apps. We use static and dynamic analysis to extract the URLs requested by apps and leverage our distributed crawling framework to retrieve apps' resource dependency chains. We then use VirusTotal API [50] to augment apps' dependency chains to characterize any suspicious resource loading. The explore several key topics in this paper:

Characterizing the Dependency Chains(§4): We analyze 7,048 apps' dependency chains, and answer the following unexplored questions regarding the implicit dependency of mobile apps on third-parties:

- *Do mobile apps rely on implicit trust?* We find that over 98.2% of apps have dependency chains > 1, and therefore rely on an implicit trust model (§ 4). Although the majority (84.32%) of these have short chains of 4 and below levels, a notable minority (5.12%) have chains exceeding 5 levels.
- *What objects exist in the dependency chain?* We analyze different types of resource types, and interestingly find JavaScript codes to be implicitly loaded by less than 92.3% of Android apps. This perhaps is because of the risk awareness among apps developers about implicitly trusting active content like JavaScript codes imported in WebView (cf. § 2.1).
- *What type of third-party exist in mobile apps resource dependency chains?* We inspect the categories of third-parties and find the predominance of "Business" category across all dependency levels *i.e.,* 39.34% of all loaded resources at level 1, which increases to 40.54% at level 3, then to 51.4%, and so on. We also investigate the most occurring implicit third-parties and find `google-analytics.com` and `doubleclick.net` to be imported by 83.8% and 79.41%, respectively.

WWW'20, April 20–24, 2020, Taipei, Taiwan
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/3308558

Finding Suspicious Domains (§5): Although the above findings expose the analyzed Android apps to a new attack surface (as implicit trust makes it difficult for Android apps' owners or developers to vet third-parties), arguably, this alone does not create a security violation. Hence, we proceed to test whether or not these chains contain any malicious or suspicious third parties, and answer several questions.

- *Do mobile apps' resource dependency chains contain suspicious parties?* Exploiting the VirusTotal (VT) service, we classify third-party domains into innocuous vs suspicious. We show a fraction of third-parties that are classified as suspicious using several VTscore thresholds. These perform suspicious activities such as requesting sensitive resources and sending HTTP(S) request to known malicious domains. We find that 1.18% of third-parties are suspicious with a VTscore $\geq$ 10. This fraction decreases if we increase the VT threshold; for example, with the VTscore of $\geq$ 40 the number of suspicious websites are 0.16% only. We particularly investigate JavaScript code and find that 51% of them, loaded at trust level 2, have a VTscore of $\geq$ 30 .
- *How widespread are malicious parties?* We inspect the amount of malicious resources imported by Android apps and find that 21.48% of these apps load resources from third-parties with the VTscore $\geq$ 10. A closer look on the results shows that `github.com` is the most frequently occurred third-party with VTscore of 11 providing resources to 11.08% of the analyzed apps.
- *At which level do suspicious third-parties occur?* We finally analyze the Android apps' categories that are most impacted by importing suspicious resources. We find that the most vulnerable category is Games which implicitly imports 23.34% of suspicious resources.

Finally, considering the importance of open science and reproducibility, we publicly release code and data.

2 BACKGROUND

We start by briefly explaining android apps components and show how resources are loaded by mobile apps.

2.1 Overview of app components

Android apps are distributed on marketplaces (such as Google Play) as `.apk` files. Usually, each android app consists of a set of navigable screens or `Activities` [1]. An `Activity` takes up the entire screen and consists of GUI elements or `Views` that contains things like text boxes, buttons, lists, ad-banners, and images. Depending upon the intended app's functionality, these components may launch HTTP(S) fetches in the following two ways:

- **Native Mode:** Upon execution, an app invokes functions provided by third-party libraries thus triggering HTTP(S) requests to third-party services (as shown by *Case (ii)* in Figure 1a).
- **WebView Mode:** WebView [6] is a subclass of `View` which allows an app to embed a powerful browser inside itself. Once a given HTML code is loaded in the `WebView`, the rendering engine (such as WebKit[5] or Gecko [3]) interprets the HTML and executes JavaScript code within `script` tags. Note that by default JavaScript rendering in WebView is disabled and an app needs to call the method `webview.getSettings().setJavaScriptEnabled(true)` to enable JavaScript rendering. JavaScript code execution further triggers requests for retrieving additional resources (illustrated by *Case (i)* in Figure 1a).

As well as displaying web content, the `WebView` enables apps to interact with web content in two-way: (*i*) apps can invoke JavaScript code within webpages or insert their own JavaScript code into webpages, as well as intercepting events that occur within webpages; and (*ii*) apps can register interfaces to `WebView`, so JavaScript code in the embedded webpages can invoke the interfaces. The reader can refer to [6, 64] for a more comprehensive overview on Android `WebView`. Note that although apps can use both of these modes, by scanning the source code of apps (see § 3) in our dataset, we find use `WebView` mode to access HTML content.

2.2 Third-party ecosystem

Third-parties extend an app's capabilities by providing useful content (*e.g.,* video, audio) and ways (*e.g.,* libraries and codes) to track users and deliver advertisements. Third-party services such as content delivery networks (CDNs), advertisers, and trackers have been around for years [62]. Recent years have seen apps relying on a wide range of third-party mobile ad and tracking services [84], typically fetched from ad aggregators such as `doubleclick.net` and AdMob through the ad libraries embedded in apps. Generally, an app developer registers with an ad aggregator, who provides the developer with a developer ID and an ad library which will be embedded in the app to fetch ads from other third-parties advertisers. The app developer is then paid by the ad provider based on the number of ad clicks or impressions, or both.

Figure 1a provides an overview of mobile app resource loading procedures. Here, an app accesses, in *step 1*, a first-party domain in WebView. Depending on the third-party libraries and the third-party JavaScript codes included in the webpage's HTML source code, the app sends additional requests, in *step 2*, to third-party ad services. The app then displays ads from the ad services either in in-app ads-banner, *Case ii*, in a browsed webpage, *Case i*, or both.

In the context of mobile advertising, the advertisers are parties who wish to advertise their products, the publishers are mobile applications (or their developers) that bring advertisements to the users. Ad networks or aggregators link the publishers to the advertisers, being paid by the latter and paying the former. Tapping on advertisements may lead users to content on Google Play or to web links. This often happens through a chain of several webpage redirections [47, 65]. We generally refer to all these URLs in the webpage redirections as the redirection chain and the final webpage as the landing page. Ad networks themselves may participate in complex relationships with each other [47]. Certain parties, such as ad networks, run so-called ad exchanges where a given ad space is auctioned among several bidding ad networks so as to maximize profits for the publishers [28]. Ad networks also have syndication relationships with each other: an ad network assigned to fill a given ad space may delegate that space to another network. Such delegation can happen multiple times through a chain of ad networks and is visible in the redirection chains.

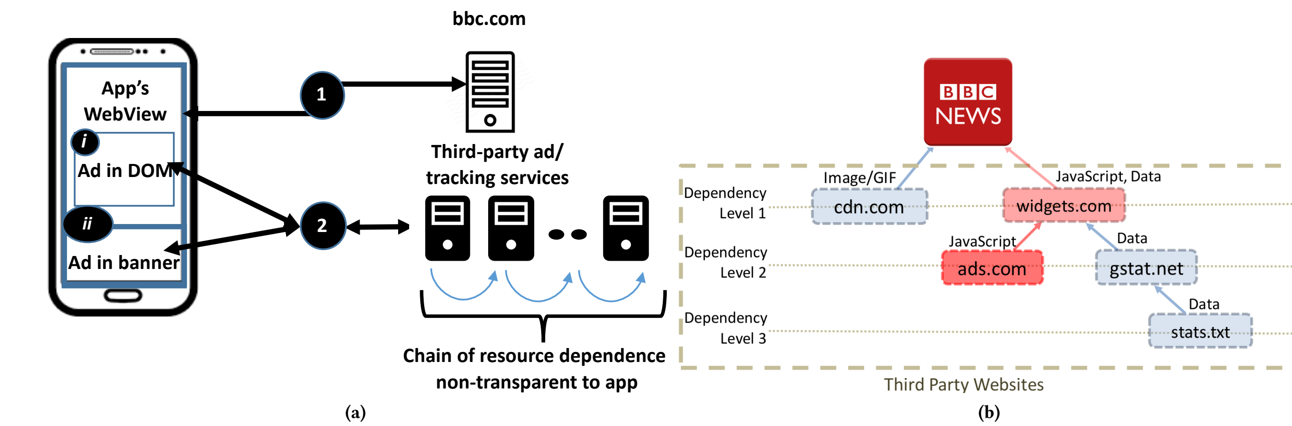


Figure 1: Overview of resource (i.e., ads) loaded from third-party services (Figure 1a) and, in Figure 1b, example of Android App dependency chain, including malicious third-party (in red).

3 METHODOLOGY AND DATASET

In this section, we describe the employed methodology used to collect our dataset. Figure 2 presents an overview of the steps involved in analyzing apps' resource dependency.

3.1 Collecting app metadata from Google Play

It is first necessary to collect a representative sample of mobile apps. As we wish to study the infiltration by suspicious third-parties, we strive to obtain a set of 'mainstream' apps (rather than fringe or malware related apps). Thus, we implement a Google Play crawler. This first obtains the app ID (or package name) for the top 50 apps listed within 10 Google Play categories: game, entertainment, business, communication, finance, tools, productivity, personalization, news & magazines, and education. Our crawler follows a breadth-first-search approach for any other app considered as *"similar"* by Google Play and for other apps published by the same developer. For each (*free*) app, we collect its metadata and executables (apks). We collect all metadata: number of downloads, category, average rating, negative/positive reviews, and developer's website.

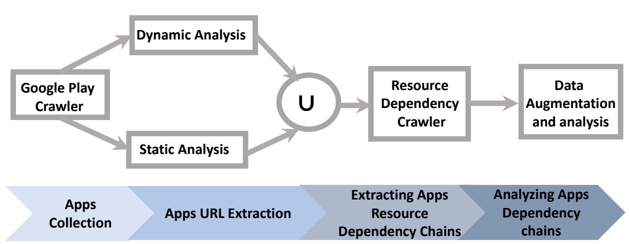


Figure 2: Overview of our data-collection and analysis methodology.

We collected 10,000 *free* apps from Google Play of which we successfully analyze 7,048 apps. Due to obfuscation and protection against source code our static analysis tools (further detailed bellow in § 3.2) failed to decompile 2,469 apps. Similarly, 483 require *login* to interact with the apps' declared activities, thus our dynamic analysis tools failed to analyze them. Overall, our corpus consists of 7,048 apps distributed across 27 different categories (collected in Dec 2018). For context, Figure 3 presents the number of apps we have with differing ratings and number of downloads. Our dataset also consists of apps that receive high user ratings: 70.7% of the apps have more than 4-star ratings and 72% of them have 500K+ downloads as depicted in Figure 3. These apps have 364,999,376 downloads (the sum of lower values of the installs). We argue this constitutes a reasonable sample of apps considered to be both mainstream and non-malicious in nature.

3.2 Extracting third-party resources

It is next necessary to extract the third-party (web) resources utilized by each app in our dataset. As depicted in Figure 2, we extract this set of URLs/domains in two ways.

First, for each app, we use the Google Play Unofficial Python API [4] to download each app's executable, and use `ApkTool` [2]–a static analysis tool–to decompile them. We then leverage regular expressions to comprehensively search and extract embedded URLs/domains in the decompiled source code.

Second, we use a dedicated testbed, composed of a smartphone that connects to the Internet via a computer configured as a WiFi access point (AP) with dual-stack support. The WiFi AP runs `MITMProxy` [17] to intercept all the traffic being transmitted between the mobile device and the Internet. This allows us to observe the resources loaded (or URLs/domains requested) by each app. To automate the execution of apps in our corpus, we leverage `MonkeyRunner` [18] to launch an app in our test mobile phones and to recursively interact with app by emulating user interactions such as clicking and swapping on all *activities* defined in the `AndroidManifest.xml` files.[1] To complete the execution and rendering of each activity, we enforce a 20 second waiting time and 400 seconds runtime session per app executing, on average, 35 different activities. For each app, we combine the list of URLs/domains extracted from the app's source code and app's network traffic. The

[1]We exclude 483 apps from our analysis as they have only *login* activities defined in they AndroidManifest.xml files.

above two techniques result in 414,387 URLs and 89,787 domains that correspond to 16,069 second-level domains.

3.3 Resource dependency dataset

Once we have the third-party resources for each analyzed app, we next strive to reconstruct the dependency chain. To this end, we build a crawling framework to collects apps' resource dependency chains. As mobile browsers have limited automation options and instrumentation capabilities, we modify the Chrome Headles crawler, detailed in [58], to imitate Google apps' WebViews. To ensure that we would see the correct mobile WebViews, leveraging previous work [33], our instrumentation involves: overriding the navigator object's user agent, OS platform, appVersion and appCodeName strings; and screen dimensions. Specifically, we emulate Chrome on Android, as it uses the same WebKit layout engine as the desktop Chrome used in the crawls. Recall that this covers the sequence of (JavaScript) resources that trigger further fetches.

For each of the 414,387 URLs identified, we then load and render it using our Chromium Headless crawler, detailed in [58]. This Chromium based Headless [39] crawler renders a given URL/domain and tracks the resource dependencies by recording network requests sent to third-party domains. The requests are then used to reconstruct the dependency chain between each app and its requested URLs. Essentially, a dependency chain is constructed by analyzing each parent and child domain duple. We then extract the URL of the parent domain, the URL of the child domain, and the URL of the referrer. If the referrer differs from the parent, we add a branch from the parent to the referrer and then from the referrer to the child. Otherwise if the parent is the referrer, we add a branch from the parent to the child. This is done for every parent and child tuple returned from our crawler. Note that each app can trigger the creation of multiple dependency chains. This process results in 414,387 dependency chains extracted (one per URL), creating a total set of 4,670,741 URLs.

Figure 1b presents an example of a dependency chain with 3 levels. *level 1* is loaded directly by the app, and is therefore explicitly trusted by it (*i.e.,* `BBC News`). *level 2* and *3*, however, are implicitly (or indirectly) trusted as the BBC News app may not necessarily be aware of their loading. For simplicity, we consider any domain that differs from the domain owned by the analyzed app[2] to be a third-party. More formally, to construct the dependency chain, we identify third-party requests by comparing the second level domain of the page (*e.g.,* bbc.com) to the domains of the requests (*e.g.,* cdn.com and `ads.com` via widgets.com).

Those with different second level domains are considered third-party. We ignore the sub-domains so that a request to a domain such as player.bbc.com is not considered as third-party. Due to the lack of purely automated mechanism to disambiguate between site-specific sub-domains (*e.g.,* player.bbc.com) or country-specific sub-domains (*e.g.,* bbc.co.uk), we leverage Mozilla Public Suffix list [82] and `tldextract` [59] for this task. Moreover, we distinguish between first-party second-level domains, in which case the developer of an app also owns the domain, and third-party domains, which include ad networks, trackers, social networks, and any other party that an app contacts. For instance, `twitter.com` is a first-party to the Twitter App but it is a third-party to BBC News.

[2] For each Android app, we obtained the domain belonging to the app from Google Play store.

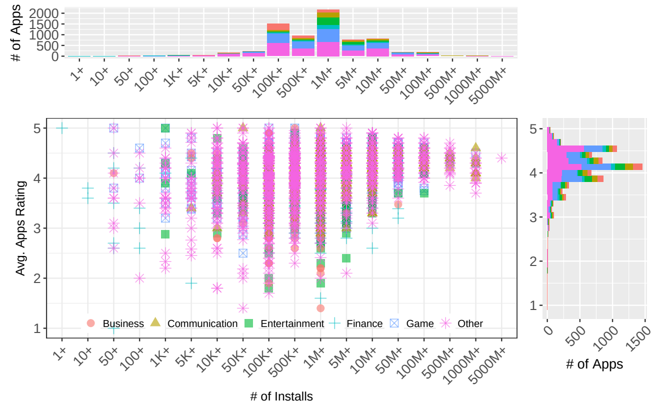


Figure 3: An overview of analyzed apps' install and ratings on Google Play.

3.4 Meta-data collection from VirusTotal

The above steps result in a dependency chain being created for each URL loaded by an app. As a major goal within our work is to identify potentially suspicious third-party resources, it is necessary to annotate these dependency chains with data about potential risk.

To achieve this, we leverage the VirusTotal public API to automatize our classification process. VirusTotal is an online solution which aggregates the scanning capabilities provided by 68 AV tools, scanning engines and datasets. It has been commonly used in the academic literature to detect malicious apps, executables, software and domains [48, 51, 52]. Upon submitting a URL, VirusTotal provides a list of scans from 68 anti-virus tools. We use the report APIs to obtain the VTscore for each third-party URL belonging to mobile apps in our dataset. Concretely, this score is the number of AV tools that flagged the website as *malicious* (max. 68). We further supplement each domain with their WebSense [92] category[3] provided by the VirusTotal's *record* API. During the augmentation, we eliminate invalid URLs (1.7%) in each dependency chain. Thus, we collect the above metadata for each second level domain in our dataset. This results in a final sample of 4,675,173 URLs consisting of 89,787 unique domains from which we extract 16,699 unique second level domains.

4 ANALYSIS OF APPS' RESOURCE DEPENDENCY CHAINS

We begin by characterizing the resources imported by apps. We seek to determine if apps do, indeed, rely on implicit trust.

4.1 Characterizing apps' implicit trust

We analyze the resource loaded per app (resp. per category of apps) and measure the "depth" of implicit trust, *i.e.,* how many levels in the dependency chain an app loads resources from. Collectively, the 7,048 apps in our dataset make 4,670,901 calls to 414,387 unique external resources, with a median of 509 external resources per app.

[3] For details on the websites or domains classification, we refer the reader to WebSense's, also known as ForcePoint [49], domains classification repository.

To dissect this, Table 1 presents the percentage of apps that both explicitly and implicitly trust third-party resources. We separate apps into their popularity, based on their number of downloads on the Play store. Table 1 shows that the use of third-party resources is extremely common. 98.2% of explicitly trust third-parties at least once, with 22.1% importing externally hosted JavaScript code. Moreover, around 95% of the apps do rely on *implicit* trust chains, *e.g.,* they allow third-parties to load further third-parties on their behalf. This trend is already well-known [47] in the web context; here we confirm it for mobile apps. Note, the propensity to form dependency chains is marginally higher in more popular apps; for example, 99% of apps with number of installs ≥ 500K have dependency chains compared to 98% of apps with number of installs ≤ 100K.

	Number of Installs						
	1-5B (7048)	1-10K (119)	10K-100K (391)	100K-500K (1456)	500K-5M (3069)	5M-50M (1588)	50M-5B (425)
Apps that trust at least one third-party which loads:							
Any Resources:							
Explicitly (Lvl. 1)	98.2%	89.9%	93.6%	97.1%	98.2%	99.0%	99.3%
Implicitly (Lvl. ≥ 2)	95%	82%	86%	93%	94%	96%	98%
JavaScript:							
Explicitly	22.1%	26.7%	25.3	23.1%	20.6%	21.7%	18.1%
Implicitly	92.3%	65.5%	79.3%	90.9%	92.9%	94.3%	92.0%

Table 1: Overview of the dataset for different ranges of number of apps' install. The rows indicate the proportion of number of apps installs that explicitly and implicitly trust at least one third-party (*i*) resource (of any type); and (*ii*) JavaScript code. It shows that 98.2% of apps import external resources, with 22.1% importing externally hosted JavaScript codes. Moreover, around 95% of the apps do rely on *implicit* trust chains,i.e.,they allow third-parties to load further third-parties on their behalf.

We next inspect the *depth* of the dependency chain. Intuitively, long chains are undesirable as they typically have a deleterious impact on resource load times [91] and increase attacks surface, *e.g.,* drive-by downloads [32, 34], malware and binary exploitation [75, 81, 86, 87], or phishing attacks [93].

Figure 4a presents the CDF of chain level for all apps in our dataset. For context, apps are separated into their sub-categories.[4] It shows that 84.32% of the analyzed apps create chains of trust of level 4 or below. Overall, we find that all mobile apps import ≈5.12% of their external resources from level 5 and above. However, there is also a small minority that dramatically exceed this level. In the most extreme case, we see `AntiVirus 2019` [8], having 1M+ downloads and average rating 4.2, with a chain containing 7 levels, consisting of mutual calls between `pubmatic.com` (online marketing) and `mathtag.com` (ad provider). Other notable examples include `RoboForm Password Manager` [22] (productivity app with 500K+ downloads and average rating 4.3), `Borussia Dortmund` [11] (sport, 1M+, 4.5), and `Cover art Evite: Free Online & Text Invitations` [15] (social, 1M+, 3.9) have a maximum dependency level of 7. We argue that these complex configurations make it extremely difficult to reliably audit such apps, as an app cannot be assured of which objects are later loaded. Briefly, we also note that

[4]We include the most popular categories, and group subcategories (Arcade, Action, Adventure, Board, Card, Casino, Casual, Educational, Music, Puzzle, Racing, Role Playing, Simulation, Strategy, Sports, Trivia, and Word) to 'Game'.

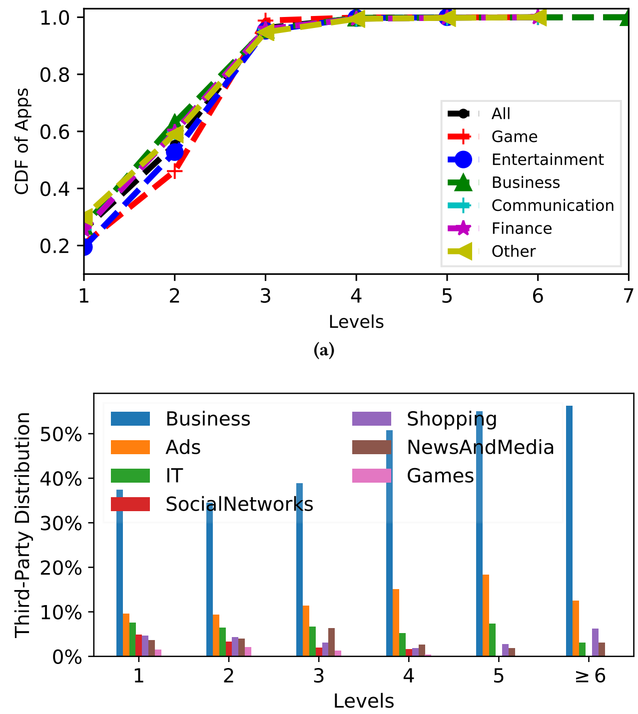


Figure 4: (a) CDF of dependency chain levels (broken down into categories of apps); and (b) distribution of third-parties across various categories and levels.

Figure 4b reveals subtle differences *between* different categories, according to WebSense categorization (cf. § 3.4), of third-party domains. For example, those classified as Business and Adverts are most likely to be loaded at level 1; this is perhaps to be expected, as many ad brokers naturally serve and manage their own content. In contrast, Social Network third-parties (*e.g.,* Facebook plug-ins) are least likely to be loaded at level 1.

4.2 Characterizing the types of resources

The previous section has confirmed that a notable fraction of apps create dependency chains with (up to) 7 levels. Next, we inspect the types of imported resources within these dependency chains. For analyzed (categories of) apps at each level of resource dependency chain, we classify the types of loaded resources into six main types: Data (consisting of HTML, XML, JSON, plain text and encoded files), Image, JavaScript code, CSS/Fonts, Audio, and Video. We were unable to classify 5.28% of resources loaded by the analyzed apps. On a closer look, we find that 98% of these uncategorized resources were imported from 242 unique, static IP addresses via WebSockets while 2% of the uncategorized resources were requested from localhost (127.0.0.1).

Table 2 presents the volume of each resource type imported at each level in the trust chain. We observe that the make-up of resources varies dramatically based on the level in the dependency chain. For example, the fraction of images imported tend to increase—this is largely because third-parties are in-turn loading images (*e.g.,* for adverts). In contrast, the fraction of JavaScript codes decreases as the level in the dependency chain increases: 27.2% of resources at level 2 are JavaScript codes compared to just 11.92% at level 5. This trend is caused by the fact that new levels are typically created by JavaScript execution (thus, by definition, the fraction of JavaScript codes must deplete along the chain). However, it remains at a level that should be of concern to app developers as this confirms a significant fraction of JavaScript code is loaded from potentially unknown implicitly trusted domains.

Lev.	Total	Data	Image	JS	CSS/Font	Audio	Video	Uncat.
1	315,217	91.76%	3.74%	1.4%	0.06%	0.21%	0.07%	2.76%
2	4,040,882	10.22%	45.55%	27.2%	13.12%	0.06%	0.17%	3.53%
3	171,035	8.13%	33.11%	23.62%	5.36%	0.03%	0.01%	29.75%
4	63,179	1.6%	24.16%	14.32%	0.48%	0%	0%	59.43%
5	6,116	14.34%	18.35%	11.92%	8.19%	0%	0%	47.2%
≥ 6	383	7.31%	26.11%	1.04%	0%	0%	0.52%	65.01%

Table 2: Breakdown of resource types requested by the analyzed apps across each level in the dependency chain. Total column refers to the number of resource calls made at each level. Here *JS* represents JavaScript code category of imported resources.

To build on this, we also inspect the *categories*, taken from WebSense (see § 3.4 for details), of third-party domains hosting these resources. Figure 4b presents the make-up of third-party categories at each level in the chain. It is clear that, across all levels, Business and Advertisement domains make up the bulk of third-parties. We also notice other highly demanded third-party categories such as Business, Ads, and IT. These are led by well known providers, *e.g.,* `google-analytics.com` (web-analytics[5]) provides resources to 83.78% of the analyzed apps. This observation is inline with the fact that 81.4% of the analyzed apps embed Google ads and analytic service libraries. The figure also reveals that the distributions of categories vary slightly across each dependency level. For example, 37.7% of all loaded resources at *level 1* come from Business domains compared to 39.1% at *level 3*, *i.e.,* overall, the proportion increases across dependency levels. We also observe similar trends for resources loaded from Ads and IT (*e.g.,* web hosting) domains. In contrast, social network third-parties (*e.g.,* Facebook) are mostly presented at *level 1* (4.89%) and 2 (3.26%) with a significant drop at *level 3*. The dominance of Business and Advertisements is not, however, caused by a plethora of Ads domains: there are far fewer Ads domains than Business (see Table 3). Instead, it is driven by the large number of requests to advertisements: even though Ads domains only make-up 9.01% of third-parties, they generate 13.58% of resources. Naturally, these are led by major providers. Importantly, these popular providers can trigger further dependencies; for example, 79.41% of apps leverage `doubleclick.net` which imports 11% of its resources from further implicitly trusted third-party domains. This makes such third-parities means for online fraudulent activities and ideal propagator of "malicious" resources for any other domains having implicit trust in it [27, 63].

4.3 Characterizing protocol use

Next, we briefly profile the use of HTTP(S) at each level of dependency chain. Table 4 shows the results of our analysis. Overall, 75.51% of resources are loaded via HTTPS. Although several standardization bodies such as IETF, W3C, and IAB have called for efforts to secure the web by means of HTTPS, worryingly, approximately 20% of resources are loaded via HTTP. This might facilitate malicious actors and censors to intercept funnel resource chains from a vulnerable third-party at level ≥ 2 to one which provides actively malicious content in the form of drive-by downloads [32, 34], malware and binary exploitation [75, 81, 86, 87], or phishing attacks [93].

5 ANALYZING MALICIOUS RESOURCE DEPENDENCY CHAINS OF APPS

The previous section has shown that the creation of dependency chains is widespread, and there is therefore extensive implicit trust within the mobiles and third-party app ecosystem. This, however, does not shed light on the activity of resources within the dependency chains, nor does it mean that the implicit trust is abused by third-parties. Thus, we next study the existence of *suspicious* third-parties, which could lead to abuse of the implicit trust. Within this section, we use the term *suspicious* (to be more generic than malicious) because VirusTotal covers activities ranging from low-risk (*e.g.,* sharing private data over unencrypted channels) to high-risk (malware).

5.1 Do apps load suspicious third-parties?

First, we inspect the fraction of third-party domains that trigger a warning by VirusTotal. From our third-party domains, in Table 3, 14.95% have a VTscore of 1 or above, *i.e.,* at least one virus checker classifies the domain as suspicious. If one treats the VTscore as a ground truth, this confirms that popular websites do load content from suspicious third-parties via their chains of trust. However, we are reticent to rely on VTscore = 1, as this indicates the remaining 67 virus checkers[6] did not flag the domain[7]. Thus, we start by inspecting the presence of suspicious third-parties using a range of thresholds.

Table 3 shows the fraction of third-parties that are classified as suspicious using several VTscore thresholds. For context, we separate third-parties into their respective categories. If we classify any resource with a VTscore of ≥ 10 as suspicious, we find that 1.18% (188) of third-party domains are classified as suspicious with 1.36% of all resource calls in our dataset going to these third-parties. Notably this only drops to 0.59% with a *very* conservative VTscore of ≥ 20. We observe similar results when considering thresholds in the [5..50] range. We therefore conservatively refer to domains with a VTscore ≥ 10 as suspicious in the rest of this analysis.

5.2 Do apps' dependency chains contain suspicious parties?

The above has shown that apps do load suspicious resources. We next inspect where in the dependency chains these resources are loaded at. Additionally, we inspect apps that inherit suspicious JavaScript resources from the explicit and various implicit levels. We focus on JavaScript codes as active web content that poses great

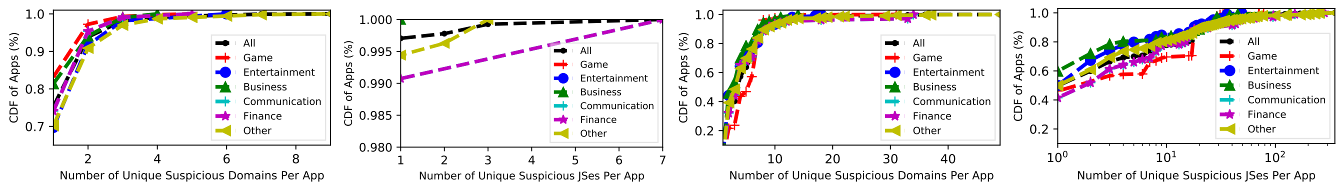
[5] Grouped as in business category as per VirusTotal reports.

[6] VirusTotal provides a list of scans from 68 anti-virus tools.

[7] Diversity is likely caused by the databases used by the various virus checkers [30, 72].

Category	Third-parties	Total Calls	Suspicious JS	VTScore ≥ 1		VTScore ≥ 5		VTScore ≥ 10		VTScore ≥ 20		VTScore ≥ 40	
				Num.	Vol.	Num.	Vol.	Num.	Vol.	Num.	Vol.	Num.	Vol.
Business	5,073 (33.43%)	1,030,635	63,970 (6.21%)	14.75%	47.59%	1.70%	2.75%	1.13%	1.17%	0.61%	0.22%	0.12%	0.09%
Ads	1,367 (9.01%)	623,261	100,843 (16.18%)	24.58%	60.65%	2.93%	5.36%	1.54	5.03%	0.59%	0.08%	0.08%	0.01%
IT	1,173 (7.73%)	41,841	887 (2.12%)	13.98%	14.54%	1.62%	3.42%	0.68%	1.61%	0.26%	0.09%	0%	0%
Shopping	607 (4.0%)	137,686	990 (0.72%)	13.51%	12.01%	1.98%	0.37%	1.32%	0.17%	1.15%	0.13%	0.66%	0.12%
NewsAndMedia	549 (3.62%)	76,566	1,205 (1.57%)	15.12%	28.86%	3.28%	0.94%	2.37%	0.93%	1.09%	0.14%	0.18%	0.03%
Social Networks	246 (1.62%)	160,789	5,033 (3.13%)	19.51%	85.77%	1.63%	0.59%	0.81%	0.59%	0.81%	0.59%	0%	0%
Games	244 (1.61%)	27,419	358 (1.30%)	16.39%	16.40%	2.46%	3.11	1.64%	1.96%	1.23%	1.93%	1.23%	1.93%
Others	5,916 (38.99%)	2,656,419	213,604 (8.04%)	12.98%	89.83%	1.81%	1.066%	1.12%	0.65%	0.50%	0.60%	0.15%	0.027%
Total	15,175 (100%)	4,670,741	386,890 (8.28%)	14.95%	73.69%	1.93%	2.03%	1.18%	1.36%	0.59%	0.44%	0.16%	0.06%

Table 3: Overview of suspicious third-parties in each category. Col.2-4: number of third-party websites in different categories, the number of resource calls to resources, and the proportion of calls to suspicious JavaScript code. Col.5-9: Fraction of third-party domains classified as suspicious (*Num.*), and fraction of resource calls classified as suspicious (*Vol.*), across various VTscores (i.e., ≥ 1 and ≥ 40).



(a) Level 1 **(b) Level 1** **(c) Level ≥ 2** **(d) Level ≥ 2**

Figure 5: CDFs of number of unique suspicious domains contacted and JavaScript codes downloaded by apps (broken down into apps' categories) at explicit level (Level = 1) and implicit level (Level ≥ 2).

Lev.	Requests	HTTPS	HTTP	HTTP/1.0	HTTP2	HTTP/1.1	Data
1	315,217	65.39%	0.01%	0.12%	0.02%	34.46%	0%
2	4,031,695	76.17%	0.57%	0.0%	0.7%	16.74%	4.78%
3	170,867	79.43%	0.97%	0.04%	2.19%	14.64%	2.73%
4	63,069	72.56%	0.46%	0%	5.89%	3.69%	17.4%
5	6,106	88.72%	0.64%	0%	0.05%	5.03%	5.57%
≥ 6	337	71.81%	9.5%	0%	0.3%	18.4%	0%
All	4,587,291	75.51%	0.54%	0.03%	0.78%	17.69%	4.55%

Table 4: Breakdown of resources via different HTTP(S) protocols and WebSockets (Data) requested by the analyzed apps across each level in the dependency chain. Total column refers to the number of resource calls made at each level.

# App	Cat.	Rat.	Insta.	Dom.	JSes	Chain len.
Unique Suspicious Domains (and JSes) at Level = 1						
1 Dashlane Pass. Manag. [13]	Prod.	4.6	1M+	9	3	7
2 BPI [12]	Fina.	4.3	1M+	7	4	5
3 Korean Dictionary [16]	Educ.	4.2	100K+	7	5	6
4 RoboForm Pass. Manag. [23]	Prod.	4.3	500K+	7	4	7
5 Bane Voice Changer [9]	Enter.	3.4	1M+	6	2	4
Unique Suspicious Domains (and JSes) at Level ≥ 2						
# App	**Cat.**	**Rat.**	**Insta.**	**Dom.**	**JSes**	**Chain len.**
1 Package Tracker [21]	Prod.	4.6	1M+	37	34	4
2 SGETHER Live Stream. [24]	Vid. Play.	4.0	1M+	36	252	5
3 Opera Browser [20]	Comm.	4.4	100M+	34	64	5
4 Adrohelm Antivirus [8]	Comm.	4.2	1M+	34	48	7
5 NFL Game Centre [19]	Game	4.1	50M+	31	34	3

Table 5: Top 5 most exposed apps (with VTscore ≥ 10) ranked by the number of unique suspicious domains.

threats with significant attack surfaces consisting of vulnerabilities related to client-side JavaScript when executed in apps WebView mode (§ 2.1), such as cross-site scripting (XSS) and advanced phishing [60, 93].

Figure 5 depicts the cumulative distributions (CDFs) of number of unique suspicious domains and JavaScript codes per (different categories of) apps. Although we do not observe significant differences among the various apps' categories, however, from the trends in the sub-figures, interestingly, we find that the majority of (resp. JavaScript codes) resources classified as suspicious are located at level 2 in the dependency chain (*i.e.,* implicitly trusted by the app).

Overall, we find that 21.46% (1,513) of the analyzed apps import at least one resource from suspicious domain with VTscore ≥ 10. Table 5 shows well-known apps, ranked according to the number of unique suspicious third-parties in their chain of dependency. We note that the popular (most vulnerable) apps belong to various categories such as Productivity, Finance, Education, and Communication. This indicates that there is no one category of domains that inherits suspicious JavaScript codes. However, we note that first mobile apps categorized as "Productivity" represent the majority of most exposed domains at level ≥2, with 16% of the total number of apps implicitly trusting suspicious JavaScript codes belonging to the Business Category, with distant second being the "Communication" Category and third the "Finance" category. The number of suspicious JavaScript codes loaded by these apps ranges from 3 to 25 JavaScript codes. We note the extreme case of 35% app *implicitly* importing at least 6 unique suspicious JavaScript programs from 3 unique suspicious domains. Moreover, we observe at most 7 unique third-parties (combining both explicit and implicit level) that is a cause of suspicious JavaScripts in mobile apps. This happens for Package Tracker [21], with over 1M install and 4.6 average rating

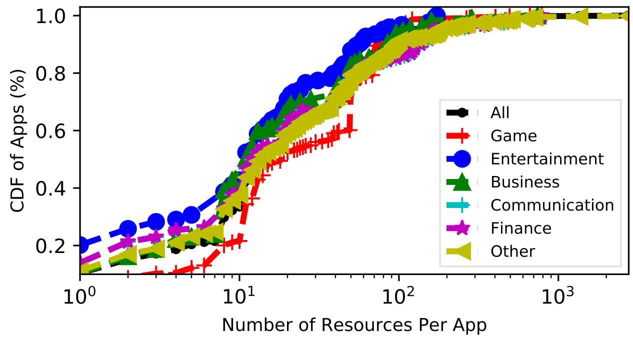


Figure 6: CDF of resources loaded per app from various categories of third-parties. Here 'All' refers to *total* number of third-party resources.

on Google Play, having third-party domains such as `nrg-tk.ru`, `yw56.com.cn`, `bitrix.info`, `fundebug.com`, and `ghbtns.com`.

5.3 How widespread are suspicious parties?

We next inspect how "popular" these suspicious third-parties are at each position in the dependency chain, by inspecting how many Android apps utilize them. Figure 6 displays the CDF of resource calls to third-parties made by each app in our dataset. We decompose the third-party resources into various groups (including total *vs.* suspicious). As mentioned earlier, we take a conservative approach and consider a resource suspicious if it receives a VTscore ≥ 10.

The figure reveals that suspicious parties within the dependency chains are commonplace: 12.76% of all apps contain at least 3 third-parties classified as suspicious in their dependency chain. Remarkably, 21.48% of apps load resources from third-parties at least once. Hence, even though only 9.01% of third-party domains are classified as suspicious, their reach covers nearly one fifth of the apps (indirectly via implicit trust).

This is a product of the power-law distribution of third-party "popularity" across Android apps: The top 20% of third-party domains cover 86% (3,650,582) of all resource calls. Closer inspection shows that it is driven by prominent third-parties: `github.com` and tapjoy.com, and `baidu.com` obtaining, during the measurement period, VTscore of 11, 18, and 21 suggesting a high degree of certainty of being suspicious. For instance, in the case of "Egypt News Moment by Moment" [14] which loaded JavaScript resources from github.com, it was actually caused by github.com loading another third-party, `ghbtn.com`, which is known to distribute *adwares*. It is unclear why ghbtn.com was performing this.

5.4 Which suspicious third-parties are most prevalent?

Next, we inspect in, Table 6, the top 10 most frequently encountered suspicious third-party domains that are providing suspicious JavaScript resources to first-parties (as opposed to the most exposed Android apps domains shown earlier in Table 5). We rank these suspicious third-party domains according to their prevalence in the Web ecosystem and further decompose our analysis at explicit and implicit levels in the table. We found `github.com` is the most called domain. Interestingly, we find several suspicious third-party domains from the Top 100 Alexa ranking. For instance, `baidu.com`, a search engine website mostly geared toward East-Asian countries has been used by 253 apps and is ranked 4 by Alexa. This website is found to be one of the most prevalent suspicious third-party domains at both level 1 (140 apps) and levels ≥ 2 (113 apps). An obvious reason for this domain's presence is because of other infected (malware-based) apps that try to authenticate users from such domains [74]. Others such as `topjoy.com` and `baidu.com` are also among the most prevalent third-party domains at level 1. These websites were reported to contain malware in their JavaScript codes [36] and suggesting users to promote [83] and installs potentially unwanted programs [7].

While it is *not shown in the table*, we also note the presence of `qq.com`, a Chinese Search Engine ranked high by Alexa. This is among the top 10 most encountered suspicious third-party domains, as defined by 13 AV tools within VirusTotal. Closer inspection reveals this is likely due to repeated instances of insecure data transmission, use of `qq.com` fake accounts for malware manifestation and for data encryption Trojans [42, 53, 88].

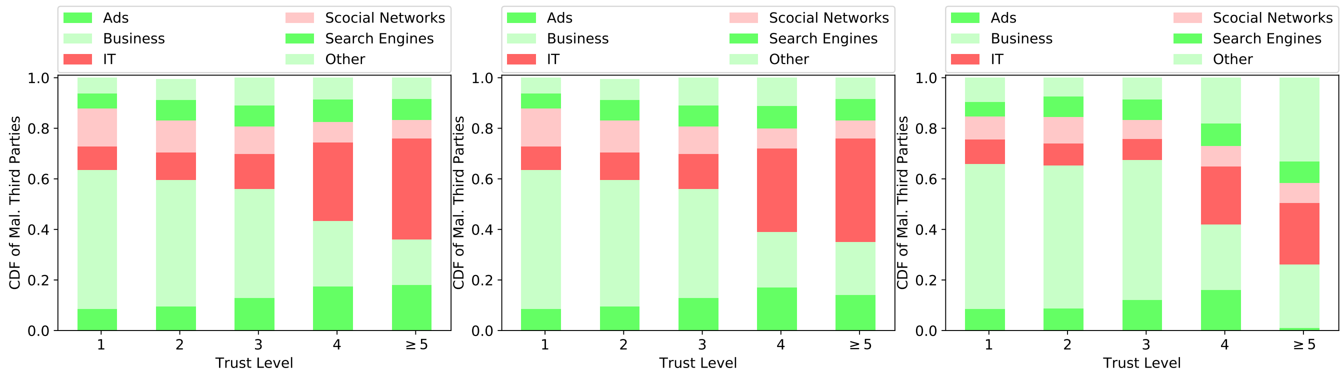
Prevalence of Third-parties at Level = 1					
#	**Third-party Domain**	**Alexa Rank**	**VTscore**	**# Apps**	**Category**
1	`github.com`	50	11	769	IT
2	`tapjoy.com`	47,720	18	199	IT
3	`baidu.com`	4	21	140	SearchEngine
4	`oracle.com`	825	10	39	IT
5	`dominionenergy.com`	16,757	12	31	Business
Prevalence of Third-parties at Level ≥ 2					
1	`bit.ly`	4	21	113	SearchEngine
2	`sil.org`	64,483	16	17	Ads
3	`comeet.co`	87,766	13	117	Business
4	`cloudfront.net`	264	11	12	WebHosting
5	`amazonaws.com`	1,597	10	8	WebHosting

Table 6: Top 5 most prevalent suspicious third-party domains (VTscore ≥ 10) on level 1 and level ≥ 2 providing resources to Apps. The number of apps (# Apps) having the corresponding suspicious third-party domain in their chain of dependency.

5.5 At which level do suspicious third-parties occur?

Next, by inspecting the location(s) in the dependency chain where the malicious third-parties are situated and the *types* of apps that load them, we analyze the impact of suspicious resource loaded on mobile apps. This is vital as implicitly trusted (level ≥ 2) resources are far more difficult for an app developer (or owner) to remove—they could, of course, remove the intermediate level 1 resource, but this may disrupt their own business activities.

Table 7 presents the proportion of apps that import at least one resource with a VTscore ≥ 10. We separate resources into their level in the dependency chain. Interestingly, the majority of resources classified as suspicious are located at level 1 in the dependency chain (*i.e.,* they are explicitly trusted by the app). 41.2% of the analyzed apps containing suspicious third-parties are "infected" via level 1. This suggests that the app developers are not entirely diligent in monitoring their third-party resources and may purposefully utilise such third-parties [45].



(a) All Apps (b) Games Apps (c) Entertainment Apps

Figure 7: Distribution of calls to suspicious third-party domains (VT score ≥ 10) per category at each level, for all analyzed apps (Figure 7a) and most vulnerable apps categories (Figures 7b, 7c).

4.24% of the analyzed apps import at least 11 resources from suspicious via *implicit* trust (*i.e.,* level ≥ 2). In these cases, the Game developers are potentially unaware of their presence. The most vulnerable category is Games: 23.34% of Game apps import *implicitly* trusted resources from level 2 with a VTscore ≥ 10. Notably, among the 78 Game apps importing suspicious JavaScript resources from trust level 2 and deeper, we find 41 apps loading advertisements from `adadvisor.net`. One possible reason is that ad-networks could be infected, victimized with malware to perform malvertising [63, 85] or binary exploitation [75, 81, 86, 87].

Lv.	All Apps All	All Apps JS	Games Apps All	Games Apps JS	Entert. Apps All	Entert. Apps JS	Business Apps All	Business Apps JS	Comm. Apps All	Comm. Apps JS
1	41.20%	37.37%	55.40%	43.43%	53.61%	47.20%	49.23%	45.38%	47.41%	45.25%
2	4.24%	1.29%	23.34%	4.53%	10.09%	3.50%	7.53%	3.21%	8.09%	3.05%
3	1.01%	0.13%	1.59%	0.40%	3.26%	0.18%	1.070%	0.29%	2.20%	0.10%
4	0.11%	≤ 0.1%	0.51%	≤ 0.1%	0.80%	≤ 0.1%	0.60%	≤ 0.1%	0.40%	≤0.001%
≥ 5	≤ 0.10%	0	≤0.001%	≤ 0.1%	≤0.001%	≤ 0.1%	≤0.001%	≤ 0.1%	≤0.001%	0.00%

Table 7: Proportion of apps importing resources classified as suspicious (with VTscore ≥ 10) at each level.

Similar, albeit less extreme, observations can be made across Entertainment (abbreviated as Entert. in Table 7) and Business apps. Briefly, Figure 7 displays the categories of (suspicious) third-parties loaded at each level in the apps' dependency chains — it can be seen that the majority are classified as Business according to WebSense domain classification (cf. Section 3.4). This is, again, because of several major providers classified as suspicious such as `comeet.co` and `dominionenergy.com`. Furthermore, it can be seen that the fraction of advertisement resources also increases with the number of levels due to the loading of further resources (*e.g.,* images).

We next strive to quantify the level of suspicion raised by each of these JavaScript programs. Intuitively, those with higher VTscores represent a higher threat as defined by the 68 AV tools used by VirusTotal. Hence, Figure 8 presents the cumulative distribution of the VTscores for all JavaScript resources loaded with VTscore ≥ 1. We separate the JavaScript programs into their location in the dependency chain. Clear difference can be observed, with level 2 obtaining the highest VTscore (median 28). In fact, 51% of the suspicious JavaScript resources loaded on trust level 2 have a VTscore > 30 (indicating *very* high confidence).

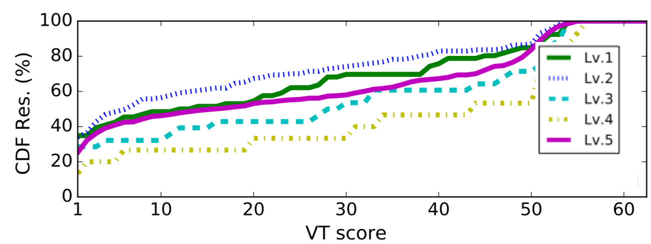


Figure 8: CDF of suspicious JavaScript programs (VTscores ≥ 1) at different levels in the chain.

Figure 9 also presents the breakdown of the domain categories specifically for suspicious JavaScript resources. Clear trends can be seen, with IT (*e.g.,* `dynaquestpc.com`), News and Media (*e.g.,* `therealnews.com`), Entertainment (*e.g.,* `youwatchfilm.net`) and Business (*e.g.,* `vindale.com`) are dominating. Clearly, suspicious JavaScript resources cover a broad spectrum of activities. Interestingly, we observed that 63% and 66%, respectively, of IT and News & Media JavaScript codes are loaded from level ≥ 2 in contrast to 17% and 25% of JavaScript code from Social Networks and Streaming loaded at *level 1.*

We next strive to quantify the level of suspicion raised by each of these JavaScript programs. Intuitively, those with higher VTscores represent a higher threat as defined by the 68 AV tools used by VirusTotal. Hence, Figure 8 presents the cumulative distribution of the VTscores for all JavaScript resources loaded with VTscore ≥ 1. We separate the JavaScript programs into their location in the dependency chain. Clear difference can be observed, with level 2 obtaining the highest VTscore (median 32). In fact, 78% of the suspicious JavaScript resources loaded on trust level 2 have a VTscore > 52 (indicating *very* high confidence).

This is a critical observation since as mentioned earlier, while suspicious third-parties at level 1 can be ultimately removed by apps' developers if flagged as suspicious, this is much more difficult for *implicitly* trusted resources further along the dependency chain.

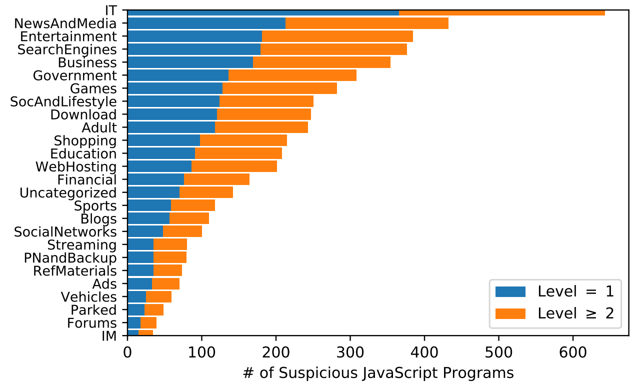


Figure 9: Breakdown of suspicious JavaScript resources based on category of domain.

If the intermediate (non-suspicious) *level 1* resource is vital for the webpage, it is likely that some operators would be unable or unwilling to perform this action. The lack of transparency and the inability to perform a vetting process on implicitly trusted loaded resources further complicates the issue. It is also worth noting that the VTscore for resources loaded further down the dependency chain is lower (*e.g., level 4*). For example, 92% of level 2 resources receive a VTscore below 3. This suggests that the activity of these resources is more contentious, with a smaller number of AV tools reaching consensus.

6 RELATED WORK

Nikiforakis *et al.* demonstrated that large proportions of websites rely on third-party JavaScript libraries hosted on ill-maintained external web servers making exploitation via JavaScript trivial [67]. Lauinger *et al.* led a further study, classifying sensitive libraries and the vulnerabilities caused by them [61]. Gomer *et al.* analyzed users' exposure to third-party tracking in the context of search queries, showing that 99.5% of users are tracked by popular trackers within 30 clicks [38]. Hozinger *et al.* found 61 third-party JavaScript exploits and defined three main attack vectors [43].

Our work differs quite substantially from these studies in that we are not interested in the third-party JavaScript code itself, nor the simple presence of third-party tracking domains embedded in tweets or in a webpage. Instead, we are interested in *how* mobile apps' users are exposed to third-parties and the presence of third-parties in the redirect chain. In contrast to our work, these prior studies ignore the presence of chains of resource loading and treat all third-parties as "equal", regardless of where they are loaded when users click on a given URL embedded in a tweet or webpage.

Several prior work focus into the analyzing and quantifying different kinds of third-party online tracking various types of methods such as browser fingerprinting [26, 68], cookies [55, 77], JavaScript codes [35, 46], and flash cookies [25, 80]. Another domain of work studies the personal identifiable information leakages from first-parties via third-party referrals [54–57] and cookies sharing or syncing among third-parties [25, 69]. With a mix of these tracking techniques, third-party trackers and advertisers have the abilities to garner and compile information exposing users to serious security and privacy risks [29, 66]. While these studies illuminate on the abilities of third-party trackers and advertisers, however, they do not analyze the chain of resource loading where cookies and personal identifiers are exchanged implicitly among third-parties. Our work characterizes the chain of resource loading of mobile apps.

Kumar *et al.* [58] characterized websites' resource dependencies on third-party services. In-line with our work, they found that websites' third-party resource dependency chains are widespread. This means, for example, that 55% of websites, among Alexa top 1M, are prevented from fully migrating to HTTPS by the third-parties that provide resources to them. More related work is Ikram *et al.* [47], who perform a large-scale study of suspicious resource loading and dependency chains in the Web, and around 50% of first-party websites render content that they did not directly load. They also showed that 84.91% of websites have short dependency chains (below 3 levels). The study reported that 1.2% of these suspicious third-parties have remarkable reach into the wider Web ecosystem.

To the best of our knowledge, we are the first to characterize the chains of resource loading of mobile apps. Moreover, we also characterize the role of apps' suspicious resources loading. We suggests that more rigorous vetting of in-app third-party resources is required.

7 CONCLUDING REMARKS

This paper has explored dependency chains in the Android apps. Inspired by the lack of prior work focusing on how resources are loaded by mobile apps, we found that over 98.2% of apps *do* rely on implicit trust. Although the majority (70.91%) of the analyzed apps have short chains, we found found apps with chains upto *level* 7. Of course, the most commonly *implicitly* trusted third-parties are well known operators (*e.g.,* google-analytics.com and doubleclick.net), but we also observed various less known implicit third-parties. We hypothesized that this might create notable attack surfaces. To confirm this, we classified the third-parties using VirusTotal to find that 1.18% of third-parties are classified as potentially malicious (with VTscore ≥ 10). These resources have remarkable reach — largely driven by the presence of highly central third-parties, *e.g.,* `github.com`.

In our future work, we wish to perform longitudinal measurements to understand how these metrics of maliciousness evolve over time. We are particularly interested in understanding the (potentially) ephemeral nature of threats. Another line of work is understanding how *level* ≥ 1 JavaScript content creates inter-dependencies between apps and third-party domains. This is particularly noteworthy among hypergiants (*e.g.,* google-analytics.com, doubleclick.net and github.com), who are providing resources to a large number of apps. To provide apps' users better control of their privacy and to facilitate secure resource loading, we also aim to investigate ways to automatically identify and sandbox suspicious parties in the resource dependency chains to alert users to security vulnerabilities (resp. HTTPS downgrades) of at each level of dependency chains. Finally, by opening our collected datasets and experiments code and scripts to the wider research community, we hope this paper has shed some light on an important security consideration of today's Web and we call for further collaboration and research to help address this.

REFERENCES

[1] Activity | Android Developers. https://developer.android.com/reference/android/app/Activity.html.

[2] Apktool - a tool for reverse engineering 3rd party, closed, binary android apps. https://ibotpeaches.github.io/Apktool/.

[3] Gecko - mozilla | mdn. https://developer.mozilla.org/en-US/docs/Mozilla/Gecko.

[4] Google play unofficial python 3 api library. https://github.com/alessandrodd/googleplay_api.

[5] Webkit. https://webkit.org.

[6] Webview | android developers. https://developer.android.com/reference/android/webkit/WebView.html.

[7] Android.tapjoy | symantec. https://www.symantec.com/security-center/writeup/2014-052619-4702-99, 2019.

[8] AntiVirus 2019. https://play.google.com/store/apps/details?id=com.androhelm.antivirus.free2, 2019.

[9] Bane voice changer btvc batman. https://play.google.com/store/apps/details?id=com.stereomatch.banetalk, 2019.

[10] BBC News. https://play.google.com/store/apps/details?id=bbc.mobile.news.ww, 2019.

[11] Borussia Dortmund. https://play.google.com/store/apps/details?id=de.bvb.android, 2019.

[12] Bpi. https://play.google.com/store/apps/details?id=com.fronde.mbanking.android.pckg.bpi, 2019.

[13] Dashline password manager. https://play.google.com/store/apps/details?id=com.dashlane, 2019.

[14] Egypt news moment by moment. https://play.google.com/store/apps/details?id=com.egy.new, 2019.

[15] Evite: Free Online & Text Invitations. https://play.google.com/store/apps/details?id=com.evite, 2019.

[16] Korean dictionaries. https://play.google.com/store/apps/details?id=com.kdictionaries, 2019.

[17] mitmproxy - an interactive HTTPS proxy. https://mitmproxy.org, 2019.

[18] monkeyrunner | Android Developers. https://developer.android.com/studio/test/monkeyrunner/, 2019.

[19] Nfl. https://play.google.com/store/apps/details?id=com.gotv.nflgamecenter.us.lite, 2019.

[20] Opera browser. https://play.google.com/store/apps/details?id=com.opera.browser, 2019.

[21] Package tracker. https://play.google.com/store/apps/details?id=de.orrs.deliveries, 2019.

[22] RoboForm Password Manager. https://play.google.com/store/apps/details?id=com.siber.roboform, 2019.

[23] Roboform password manager. https://play.google.com/store/apps/details?id=com.siber.roboform, 2019.

[24] Sgether live streaming. https://play.google.com/store/apps/details?id=com.sgrsoft.streetgamer, 2019.

[25] G. Acar, C. Eubank, S. Englehardt, M. Juárez, A. Narayanan, and C. Díaz. The web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 674–689, 2014.

[26] G. Acar, M. Juárez, N. Nikiforakis, C. Díaz, S. F. Gürses, F. Piessens, and B. Preneel. Fpdetective: dusting the web for fingerprinters. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 1129–1140, 2013.

[27] S. A. Alrwais, A. Gerber, C. W. Dunn, O. Spatscheck, M. Gupta, and E. Osterweil. Dissecting ghost clicks: Ad fraud via misdirected human clicks. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 21–30. ACM, 2012.

[28] M. A. Bashir, S. Arshad, W. Robertson, and C. Wilson. Tracing information flows between ad exchanges using retargeted ads. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 481–496, 2016.

[29] C. Budak, S. Goel, J. Rao, and G. Zervas. Understanding emerging threats to online advertising. In *Proceedings of the 2016 ACM Conference on Economics and Computation*. ACM, 2016.

[30] J. Canto, M. Dacier, E. Kirda, and C. Leita. Large scale malware collection: lessons learned. In *IEEE SRDS Workshop on Sharing Field Data and Experiment Measurements on Resilience of Distributed Computing Systems*. Citeseer, 2008.

[31] J. Chen, X. Zheng, H.-X. Duan, J. Liang, J. Jiang, K. Li, T. Wan, and V. Paxson. Forwarding-loop attacks in content delivery networks. In *NDSS*, 2016.

[32] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 281–290, New York, NY, USA, 2010. ACM.

[33] A. Das, G. Acar, N. Borisov, and A. Pradeep. The web's sixth sense: A study of scripts accessing smartphone sensors. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1515–1532. ACM, 2018.

[34] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 88–106. Springer, 2009.

[35] S. Englehardt and A. Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1388–1401, 2016.

[36] I. X. Exchange. Statcounter session hijack. https://exchange.xforce.ibmcloud.com/vulnerabilities/20506, 2005.

[37] M. Falahrastegar, H. Haddadi, S. Uhlig, and R. Mortier. Anatomy of the third-party web tracking ecosystem. *Traffic Measurements Analysis Workshop (TMA)*, 2014.

[38] R. Gomer, E. M. Rodrigues, N. Milic-Fraying, and M. Schrafel. Network analysis of third party tracking: User exposure to tracking cookies through search. In *WI-IAT*, 2013.

[39] Google. Headless chromium. https://chromium.googlesource.com/chromium/src/+/lkgr/headless/README.md, 2018.

[40] P. L. Gorski, L. L. Iacono, D. Wermke, C. Stransky, S. Möller, Y. Acar, and S. Fahl. Developers deserve security warnings, too: On the effect of integrated security advice on cryptographic {API} misuse. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, pages 265–281, 2018.

[41] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 101–112. ACM, 2012.

[42] A. GreenBerg. Hack brief: malware hits 225,000 (jailbroken, mostly chinese) iphones. https://www.wired.com/2015/08/hack-brief-malware-hits-225000-jailbroken-mostly-chinese-iphones/, 2015.

[43] P. Holzinger, S. Triller, A. Bartel, and E. Bodden. An in-depth study of more than ten years of java exploitation. In *CCS*, 2016.

[44] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *CCS*, 2011.

[45] D. Ibosiola, I. Castro, G. Stringhini, S. Uhlig, and G. Tyson. Who watches the watchmen: Exploring complaints on the web. In *Web Conference*, 2019.

[46] M. Ikram, H. J. Asghar, M. A. Kâafar, A. Mahanti, and B. Krishnamurthy. Towards seamless tracking-free web: Improved detection of trackers via one-class learning. *PoPETs*, 2017(1):79–99, 2017.

[47] M. Ikram, R. Masood, G. Tyson, M. A. Kaafar, N. Loizon, and R. Ensafi. The chain of implicit trust: An analysis of the web third-party resources loading. In *The Web Conference (WWW)*, 2019.

[48] M. Ikram, N. Vallina-Rodriguez, S. Seneviratne, M. A. Kaafar, and V. Paxson. An analysis of the privacy and security risks of android vpn permission-enabled apps. In *IMC*, 2016.

[49] F. Inc. Forcepoint master database url categories. https://www.forcepoint.com/master-database-url-categories, 2019.

[50] V. Inc. Virustotal public api. https://www.virustotal.com/en/documentation/public-api/, 2019.

[51] A. Kantchelian, M. C. Tschantz, S. Afroz, B. Miller, V. Shankar, R. Bachwani, A. D. Joseph, and J. D. Tygar. Better malware ground truth: Techniques for weighting anti-virus vendor labels. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*, pages 45–56. ACM, 2015.

[52] A. Kharraz, W. Robertson, D. Balzarotti, L. Bilge, and E. Kirda. Cutting the gordian knot: A look under the hood of ransomware attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer, 2015.

[53] J. Knockel, A. Senft, and R. Deibert. Wup! there it is privacy and security issues in qq browser. https://citizenlab.ca/2016/03/privacy-security-issues-qq-browser/, 2016.

[54] B. Krishnamurthy, K. Naryshkin, and C. E. Wills. Privacy leakage vs. protection measures: the growing disconnect. In *In Web 2.0 Workshop on Security and Privacy*, pages 2–11, 2011.

[55] B. Krishnamurthy and C. E. Wills. Privacy diffusion on the web: a longitudinal perspective. In *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*, pages 541–550, 2009.

[56] B. Krishnamurthy and C. E. Wills. On the leakage of personally identifiable information via online social networks. *Computer Communication Review*, 40(1):112–117, 2010.

[57] B. Krishnamurthy and C. E. Wills. Privacy leakage in mobile online social networks. In *3rd Workshop on Online Social Networks, WOSN 2010, Boston, MA, USA, June 22, 2010*, 2010.

[58] D. Kumar, Z. Ma, A. Mirian, J. Mason, J. A. Halderman, and M. Bailey. Security Challenges in an Increasingly Tangled Web. In *WWW*, 2017.

[59] J. Kurkowski. Accurately separate the TLD from the registered domain and subdomains of a url, using the public suffix list. https://github.com/john-kurkowski/tldextract, 2018.

[60] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. In *NDSS*. The Internet Society, 2017.

[61] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. In *NDSS*, 2017.

[62] A. Lerner, A. K. Simpson, T. Kohno, and F. Roesner. Internet jonesa and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016.

[63] Z. Li, K. Zhang, Y. Xie, F. Yu, and X. Wang. Knowing your enemy: understanding and detecting malicious web advertising. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 674–686. ACM, 2012.

[64] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on webview in the android system. In *ACSAC*, 2011.

[65] MalwareDontNeedCoffee. A doubleclick https open redirect used in some malvertising chain. http://malware.dontneedcoffee.com/2015/10/a-doubleclick-https-open-redirect-used.html, 2015.

[66] G. Merzdovnik, M. Huber, D. Buhov, N. Nikiforakis, S. Neuner, M. Schmiedecker, and E. R. Weippl. Block me if you can: A large-scale study of tracker-blocking tools. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, pages 319–333, 2017.

[67] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: Large-scale evaluation of remote javascript inclusions. In *CCS*, 2012.

[68] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, 2013.

[69] L. Olejnik, M. Tran, and C. Castelluccia. Selling off user privacy at auction. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[70] E. Pan, J. Ren, M. Lindorfer, C. Wilson, and D. Choffnes. Panoptispy: Characterizing audio and video exfiltration from android applications. *Proceedings on Privacy Enhancing Technologies*, 2018(4):33–50, 2018.

[71] G. Pellegrino, C. Rossow, F. J. Ryba, T. C. Schmidt, and M. WÃďhlisch. Cashing out the great cannon? on browser-based ddos attacks and economics. In *USENIX*, 2015.

[72] P. Peng, L. Yang, L. Song, and G. Wang. Opening the blackbox of virustotal: Analyzing online phishing scan engines. In *Proceedings of the Internet Measurement Conference 2019*. ACM, 2019.

[73] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *NDSS*, volume 14, pages 23–26, 2014.

[74] B. Popa. 85 infected android apps stealing social network passwords found on play store. https://news.softpedia.com/news/85-infected-android-apps-stealing-social-network-passwords -found-on-play-store-518984.shtml, 2017.

[75] V. Rastogi, R. Shao, Y. Chen, X. Pan, S. Zou, and R. Riley. Are these ads safe: Detecting hidden attacks through the mobile app-web interfaces. In *NDSS*, 2016.

[76] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver. Detecting in-flight page changes with web tripwires. In *NSDI*, 2008.

[77] F. Roesner, T. Kohno, and D. Wetherall. Detecting and defending against third-party tracking on the web. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 155–168, 2012.

[78] J. Seo, D. Kim, D. Cho, I. Shin, and T. Kim. Flexdroid: Enforcing in-app privilege separation in android. In *NDSS*, 2016.

[79] J. Sequa. Large angler malvertising campaign hits top publishers. https://blog.malwarebytes.com/threat-analysis/2016/03/large-angler-malvertising-campaign-hits-top-publishers/, 2016.

[80] A. Soltani, S. Canty, Q. Mayo, L. Thomas, and C. J. Hoofnagle. Flash cookies and privacy. In *Intelligent Information Privacy Management, Papers from the 2010 AAAI Spring Symposium, Technical Report SS-10-05, Stanford, California, USA, March 22-24, 2010*, 2010.

[81] O. Starov, J. Dahse, S. S. Ahmad, T. Holz, and N. Nikiforakis. No honor among thieves: A large-scale analysis of malicious web shells. In *Proceedings of the 25th International Conference on World Wide Web, WWW*, pages 1021–1032, 2016.

[82] M. P. Suffix. View the public suffix list. https://publicsuffix.org/list/, 2018.

[83] R. Unuchek.

[84] N. Vallina-Rodriguez, J. Shah, A. Finamore, Y. Grunenberger, K. Papagiannaki, H. Haddadi, and J. Crowcroft. Breaking for commercials: characterizing mobile advertising. In *Proceedings of the 2012 Internet Measurement Conference*, pages 343–356. ACM, 2012.

[85] A. VANCE. Times web ads show security breach. https://www.nytimes.com/2009/09/15/technology/internet/15adco.html, 2009.

[86] E. Vanrykel, G. Acar, M. Herrmann, and C. Diaz. Leaky birds: Exploiting mobile application traffic for surveillance. In *International Conference on Financial Cryptography and Data Security*, pages 367–384. Springer, 2016.

[87] G. Vigna, F. Valeur, D. Balzarotti, W. Robertson, C. Kruegel, and E. Kirda. Reducing errors in the anomaly-based detection of web-based attacks through the combined analysis of web requests and sql queries. *Journal of Computer Security*, 17(3):305–329, 2009.

[88] Q. R. Virus. How to remove nintendonx@qq.com virus completely. https://quickremovevirus.com/how-to-remove-nintendonxqq-com-virus-completely7, 2017.

[89] H. Wang, H. Li, L. Li, Y. Guo, and G. Xu. Why are android apps removed from google play?: a large-scale empirical study. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 231–242. ACM, 2018.

[90] H. Wang, Z. Liu, Y. Guo, X. Chen, M. Zhang, G. Xu, and J. Hong. An explorative study of the mobile app ecosystem from app developers' perspective. In *Proceedings of the 26th International Conference on World Wide Web*, pages 163–172. International World Wide Web Conferences Steering Committee, 2017.

[91] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystify page load performance with wprof. In *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[92] Websense. Real-time threat analysis with csi: Ace insight. https://csi.websense.com/, 2018.

[93] C. Whittaker, B. Ryner, and M. Nazif. Large-scale automatic classification of phishing pages. In *Proceedings of the Network and Distributed System Security Symposium,NDSS*, 2010.