Who's left behind?
Measuring Adoption of Application Updates at Scale

John P. Rula
Akamai
jrula@akamai.com

Philipp Richter
Akamai / MIT
prichter@akamai.com

Georgios Smaragdakis
TU Berlin
georgios.smaragdakis@tu-berlin.de

Arthur Berger
Akamai / MIT
awberger@akamai.com

ABSTRACT

This work presents a large-scale, longitudinal measurement study on the adoption of application updates, enabling continuous reporting of potentially vulnerable software populations worldwide. Studying the factors impacting software currentness, we investigate and discuss the impact of the platform and its updating strategies on software currentness, device lock-in effects, as well as user behavior. Utilizing HTTP User-Agent strings from end-hosts, we introduce techniques to extract application and operating system information from myriad structures, infer version release dates of applications, and measure population adoption, at a global scale. To deal with loosely structured User-Agent data, we develop a semi-supervised method that can reliably extract application and version information for some 87% of requests served by a major CDN every day. Using this methodology, we track release and adoption dynamics of some 35,000 applications. Analyzing over three years of CDN logs, we show that vendors' update strategies and platforms have a significant effect on the adoption of application updates. Our results show that, on some platforms, up to 25% of requests originate from hosts running application versions that are out-of-date by more than 100 days, and 16% more than 300 days. We find pronounced differences across geographical regions, and overall, less developed regions are more likely to have out-of-date software versions. Though, for every country, we find that at least 10% of requests reaching the CDN run software that is out-of-date by more than three months.

CCS CONCEPTS

• **Networks** → Network measurement;

ACM Reference Format:
John P. Rula, Philipp Richter, Georgios Smaragdakis, and Arthur Berger. 2020. Who's left behind? Measuring Adoption of Application Updates at Scale. In *ACM Internet Measurement Conference (IMC '20), October 27–29, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3419394.3423656

ACM ISBN 978-1-4503-8138-3/20/10…$15.00
https://doi.org/10.1145/3419394.3423656

1 INTRODUCTION

Internet penetration has reached an all-time high. With more than 4 billion users online [42], we are observing a once in a generation revolution in the way we socially interact, have access to education and information, entertain, and do business. Moreover, the Internet is becoming more performant and increasingly mobile, allowing for innovation in both well-developed economies as well as in developing ones. Websites as well as increasingly mobile applications (apps) continue to offer new services, increasing user subscriptions and engagement [31]. Traffic from mobile applications has increased dramatically in the last years [9, 15].

However, an increasing diversity of applications, software versions, and devices also brings new security challenges. Vulnerabilities in software or end-user device hardware can be used as an attack vector to orchestrate denial-of-service attacks [5] or expose personal information [33–35, 47]. The industry's response is to regularly release software updates to fix bugs and patch vulnerabilities [26] as well as to provide platforms to make applications and their updates secure and easily available to end users [2]. For example, Google maintains the Google Play Store (formerly known as Android Market) [22] for Android OS updates as well as Android application updates, and Apple maintains the App Store [6] for Apple iOS and MacOS, and applications in Apple products. Software vendors can push their updates to these platforms and the platforms make them available to millions of end users. All these platforms support automatic updates, but default settings vary, and in some instances, users have to opt-in to enable automatic updates.

Unfortunately, despite the effort from software and platform vendors to release application updates frequently and make them easily available to users, there are many factors that may slow down the adoption of a software update. There is the human factor, the user, who decides to download and install the latest update or not. Some users operate outdated handsets running unsupported operating systems (e.g., smartphones running Android versions that are not supported any longer). Moreover, some updates, especially OS updates, may not be supported by individual device vendors, especially for mobile devices that were purchased as part of a package with mobile operators [30]. There are also parameters that may affect the adoption of updates that have to do with the network speed and cost. For example, Apple iOS recommends to download updates to the iPhone only when connected to a WiFi network, and not when connected via a cellular network to speed up the download and reduce mobile data charges. In the absence of

WiFi connectivity, mobile data charges may prompt users to not use their bandwidth allowance for software updates.

Despite the importance of application updates, we have a rather limited understanding of how "current" is the software of large user populations. Today, we can not answer simple questions such as, "what is the fraction of mobile users that run the latest version of an application", or "what is the fraction of mobile users that have not updated a given application for more than a month, or run a version with a known vulnerability". There are many challenges to answering these questions at scale. First, active measurements that have been used to infer the software versions and related vulnerabilities that run on the server-side, e.g., with active scanning [12, 13, 48], are not applicable to all end user devices, especially if these are mobile or behind Carrier-Grade or home NATs [37]. Telemetry can provide useful insights but only for a limited number of applications [29, 39]. Tracking the currentness of end-user software hence requires passive measurements using a vantage point that can illuminate version adoption at scale. Second, even if access to such a vantage point is possible, determining useful information, such as the current version of a particular application, is a complex task [11]. While HTTP(S)-based communication has long emerged as the preferred way for applications to communicate, there is no standardized or universally adopted method that applications use to convey their current version to the server they communicate with. Third, with new versions of thousands of applications released every day to fix bugs, patch vulnerabilities, and improve user engagement, it is very challenging to maintain a detailed and current list that tracks version release dates, and consequently their adoption, for each application [31].

In this work, we tackle some of these challenges. The contributions of this paper can be summarized as follows:

(1) We develop an adaptive clustering-based methodology to extract useful information from loosely structured HTTP User-Agent strings about the requesting application, its version, operating system, and device type of the host that initiated the HTTP request. We showcase the efficiency of our method by applying it on Web server logs from a major Content Delivery Network (CDN). We show that our approach scales well with millions of unique User-Agents on a daily basis, and is able to extract useful information for 87% of requests that reach the CDN platform.
(2) We devise a method that allows us to infer the release dates of new versions and updates of some 35,000 applications that interact with the CDN over the course of 3.5 years. Leveraging this information, we devise metrics that can capture software currentness, version adoption, and allow us to present broad and detailed statistics on version adoption of applications, globally.
(3) We study the impact of the platform on software adoption: our analysis shows significant differences when it comes to the adoption of application updates across different platforms, an indication that software update strategies by software and platform vendors have a profound effect on the applications run at the edge. We report that on some platforms, up to 25% of requests originate from hosts running application versions that have not been updated 100 days after the release of a new software version, and 16% from hosts still not updated over 300 days after a new release. We investigate and discuss the impact of the platform and its updating strategies on software currentness, device lock-in effects, as well as the impact of user behavior.
(4) We find pronounced differences across geographical regions, and overall find that less developed regions are more likely to have out-of-date software versions. Though, for every country, we find that a notable percentage, at least 10%, of requests that reach the CDN platform run software that is out-of-date by more than three months. The associated hosts can potentially be compromised and exploited.

The remainder of this paper is structured as follows: we provide a brief introduction on HTTP User-Agent strings in Section 2 and introduce our dataset in Section 3. We present our method to parse User-Agent strings in Section 4. Applying our methodology, we present broad and detailed statistics on adoption trends in Section 5. We then assess currentness of applications across platforms and regions, and study factors impacting currentness in Section 6, and discuss implications of our work in Section 7.

2 BACKGROUND

The User-Agent string (User-Agent or UA in the following) is a request-header field that is present in HTTP requests and contains information about the *User-Agent* (i.e., the application) that originates the request [18]. While historically, agents, applications, were limited to a small number of classical Web browsers, today agents comprise of "any software that retrieves, renders and facilitates end user interaction with Web content, or whose user interface is implemented using Web technologies" [46]. Historically, UA strings used to follow a well-structured pattern to identify Web browsers, their version, as well as their capabilities. Consider the following example UA string taken from [3]:

```
Mozilla/5.0 (iPhone; CPU iPhone OS 10 3 3 like Mac OS X)
AppleWebKit/602.1.50 (KHTML, like Gecko) Version/10.0
Mobile/14G60 Safari/602.1
```

The above example presents us with a UA string in a classical format, and we can infer that the request comes from an iPhone with iOS 10.3.3 (build 14G60) that runs the Apple Safari 10 browser with browser engine (WebKit) version 602.1.50. Over time, however, UA strings became more complex, less-standard, and included more information about the connecting host, the device type, and its operating system. Especially with the emergence of mobile applications, new types of UA strings emerged. Consider the following two example UA strings taken from [3]:

```
Mozilla/5.0 (iPhone; CPU iPhone OS 13_3_1 like Mac OS X)
AppleWebKit/605.1.15 (KHTML, like Gecko) Mobile/15E148
[FBAN/FBIOS;FBDV/iPhone12,1;FBMD
/iPhone;FBSN/iOS;FBSV/13.3.1;FBSS/2;FBID/phone;
FBLC/en_US;FBOP/5;FBCR/]
```

```
Outlook-iOS/711.2237063.prod.iphone (3.25.0)
```

The above strings show examples of UAs coming from the Facebook app browser, and from the Outlook app running on iOS. The Facebook app browser string follows a partially well-defined format but includes additional information on the device the application is running on, as well as detailed information on the version of the application. The second example, Outlook on iOS, does not provide information on the device used, but reveals the platform (iOS), as well as a detailed version number. These strings do not follow a well-defined pattern and may include additional pieces of information that have to be communicated to the server-side of the application for advanced optimizations of the content. User-Agents contain relevant information but parsing and processing this information becomes increasingly cumbersome due to increasing fragmentation of UA strings, especially for User-Agents of mobile applications.

Indeed, we found existing open source solutions for parsing User-Agents performed poorly when identifying non-browser applications (common examples shown above). This motivates us to develop our own methodology to handle these applications and their corresponding non-traditional User-Agent formats.

Related work. Several studies have investigated the diversity and characteristics of HTTP User-Agents. These have ranged from general characteristics of User-Agent strings [23], to studying User-Agent uniqueness [14], it's implications in device profiling [40, 47], and abuse of UAs [49]. Similar to our approach, information extracted from User-Agent headers has been used to track software adoption [11, 21]. Our work greatly expands on the scope and scale of these related works, providing a significantly longer measurement period, covering orders of magnitude more clients and applications.

Given their importance in functional security policy, software update adoption has been the subject of much prior work. Recent incidents of widespread server vulnerabilities have prompted studies of the global response of server operators to these highly publicized events. These include Heartbleed [12], Spectre [24] and Meltdown [27]. Other studies have looked at adoption rates of client software updates by analyzing traces collected at a University campus [11] and by using an antivirus on-host vantage point to track application updates for millions of hosts over several years, but are limited to a small set of popular applications [29, 39]. A very recent study utilizes Facebook as a vantage point to characterize out-of-date browsers [25]. Tracking currentness of software also caught the attention of policymakers. In May 2016, the US Federal Trade Commission (FTC) issued identical orders to file special reports to eight mobile device manufacturers to gather information about their security update procedures and practices [10]. In February 2018, a report by the FTC summarized this investigation and made recommendations [43]. While this effort sheds light on the importance of mobile security updates, it also shows the complexity of gathering information from different vendors and understanding the state of the mobile ecosystem.

Our work provides a global and detailed analysis of the state of application update adoption by relying on requests that arrive at the CDN platform. To the best of our knowledge, our work represents the largest study of global software adoption dynamics to date.

3 DATASET

We obtain HTTP User-Agents from access logs of a large content delivery network (CDN). The CDN processes trillions of HTTP connections daily, and serves clients from over 290,000 servers across 1,500 networks in 135 countries. Of course, not all devices connect to the CDN, however the CDN sees activity from 1.2B IPv4 addresses in a year, i.e. the vast majority of the active address space [36]. Traffic observed is for content hosted by current customers of the CDN, and may be biased towards customers with a large footprint, and may change over time as business relationships shift. User-Agents were sampled at approximately 1 out of 4K HTTP and HTTPS requests across the CDN platform. The proliferation of HTTP as the standard network protocol across all connected devices means that the CDN has visibility into mobile applications, API traffic [45], embedded devices, and operating system services in addition to browsers. We conducted daily measurements for 3.5 years, between April 17, 2016 and January 20, 2020, capturing a total of over 2 trillion User-Agent strings.

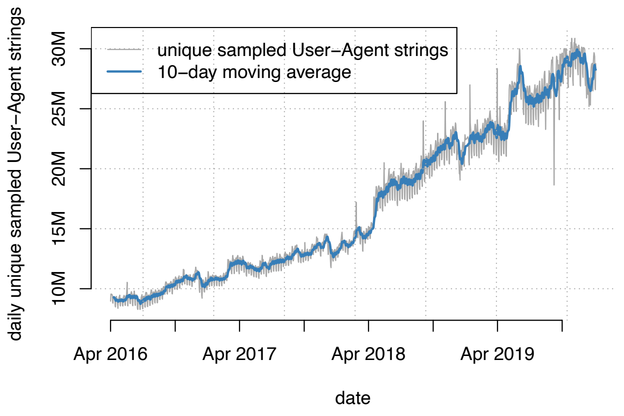


Figure 1: Unique HTTP User-Agents sampled per day.

Some User-Agent strings relate to popular applications, and we see many requests carrying the same User-Agent. Figure 1 shows the daily *unique* number of User-Agent strings collected. Throughout our measurement window, we see an increase in the number of unique strings collected each day, highlighting the increasing number of applications, and versions of applications, that interact with the CDN platform. We also find significant churn in collected User-Agents between consecutive days, between 6 to 13 million unique User-Agents observed on day d_{t+1} were not observed on day d_t. This represents more than half of unique User-Agents sampled each day.

4 MINING USER-AGENT STRINGS

In this section we present our methodology for parsing structured information from User-Agent strings.

4.1 Methodology Overview

To tackle the problem of extracting information from UA strings, we develop a semi-automated methodology that takes as input the User-Agent strings and extracts information about OS, applications, and device. Our approach needs to be adaptive, as throughout our measurement window, a period of 3.5 years, new applications have appeared and major vendors, browsers, and application providers change the format of UA strings or introduce new UA strings.

Our methodology, summarized in Figure 2, involves three steps: (i) partitioning of the User-Agent space, (ii) development of parser classes for each partition, and (*iii*) information extraction. Our approach combines these into an effective solution for maximizing the coverage of collected UAs, with minimal custom rules. We utilize a clustering approach, grouping a subset of popular UAs by their edit distance similarity (Section 4.2). We found that initial clustering of a subset of popular UAs by their edit distance similarity can partition the UA space into a tractable set of clusters. Strings in these individual clusters share a common structure, allowing them to be simply parsed from common logic. We create a parser class for each cluster consisting of regular expressions and token processing logic to match UAs and extract information of interest (Section 4.3). Given that UAs change over time, we develop a methodology to periodically identify new clusters of UAs that appear, or existing UAs that change structure. Our approach provides a scalable approach to deal with the billions of UAs we sample daily.

4.2 Partitioning the Space: Clustering

Unique UA strings often differ by only a few characters from similar strings. For example, UAs of the same applications running on different devices may only differ by the device name (e.g. iPad vs. iPhone), or operating system version. We leverage this similarity to make the problem of parsing billions of UAs tractable: performing clustering on UA strings to partition and reduce the dimensionality of the problem. We found that most UAs can be clustered into a few dozens of clusters, and that similar parsing strategies can be tailored to each cluster, greatly reducing the scope of the problem.

Similarity metric. Since we deal with textual data, i.e., strings, an obvious option is to use the Levenshtein distance to perform an initial comparison and clustering of similar UA strings. The Levenshtein distance takes two strings as input and expresses their distance by the minimum number of single-character edits required to convert one of the two strings into the other. This distance is also commonly referred to as *edit distance*. The Levenshtein distance is naturally dependent on the length of the compared strings. For our approach, we leverage the inverse Levenshtein ratio as the distance metric, which ranges between 0 and 1, where a value of 0 expresses equality of two strings and 1 expresses maximum string distance.

Clustering algorithm. For clustering, we use density-based spatial clustering (DBSCAN) [16]. In a nutshell, DBSCAN takes as input a set of points in a space, and groups points that are closely packed together. In our approach, each UA string represents one point, and we leverage the above introduced string similarity metric for our clustering. DBSCAN takes two parameters as input, *(i)* the minimum cluster size, *(ii)* and the parameter *epsilon*, which determines the maximum distance between two points to be considered in the same neighborhood. A higher epsilon value results in fewer and larger clusters, whereas a lower epsilon value yields a larger number of smaller clusters and a larger number of outlier (non-clustered) points. To determine a sensible epsilon value, we use HDBSCAN, the hierarchical variant of DBSCAN which does not rely on a fixed epsilon value, and inspect the resulting cluster hierarchy. Figure 3 shows a dendrogram of the top 20k User-Agent strings in our dataset for one day during our observation period. Manual inspection of the resulting clusters (see annotations for iOS/Android apps, as well as browsers) led us to choose an epsilon value of 0.3 for clustering, since it effectively clusters UA strings

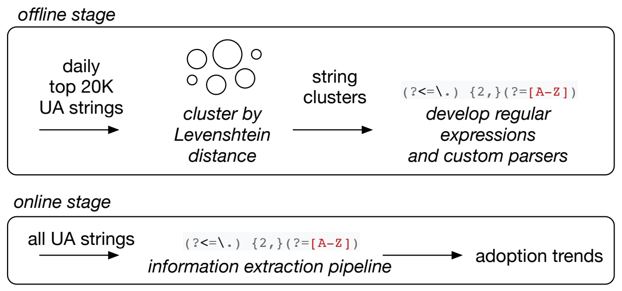


Figure 2: Our methodology's pipeline: (1) clustering, (2) regular expression generation, (3) parsing of UAs and extraction of information.

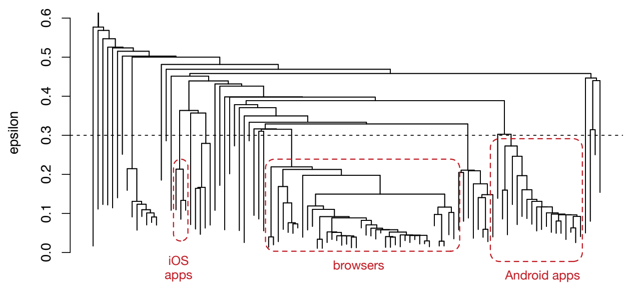


Figure 3: Dendrogram: HDBSCAN clustering process of the top 20k most common User-Agent strings based on string similarity in a single day. With decreasing value of epsilon, fragmentation in more and smaller clusters increases. We chose epsilon=0.3 as global cutoff for cluster forming.

into a manageable number of clusters with high similarity. For our values (epsilon set to 0.3, minimum cluster size set to 25), there are 33 clusters on that day.

Initial daily clustering. We found that clustering *all* unique User-Agents to be too computationally expensive. In light of the long tailed popularity of User-Agents, we use a subset of popular User-Agents for the initial clustering. We considered different popular sets, from top-10k to top-100k unique UAs, and ultimately settled on the top 20 thousand unique UAs. [1] We note, however, we restrict only our initial clustering to the top 20k UAs which for a more efficient development of matching algorithms on relevant strings. The eventual signatures that we develop to process individual strings are not restricted to this set of UAs and can effectively parse a much greater number of UA strings.

Aggregating clusters over time. Recall that our goal is to perform clustering to partition the UA space captured over a period of 3.5 years into manageable sets of strings that show common patterns, which we leverage in the next section to extract relevant information from the strings. The process described above outlines the clustering process for a single snapshot, which we conducted daily for over 3 years, ultimately yielding thousands of clusters overall. In light of the similarity of many of these clusters over time (e.g., common browser UA strings are present throughout the entire time period), and to maintain a set of manageable size, we

[1]The top 20 thousand unique UAs account between 85-90% of all UAs over the course of our study.

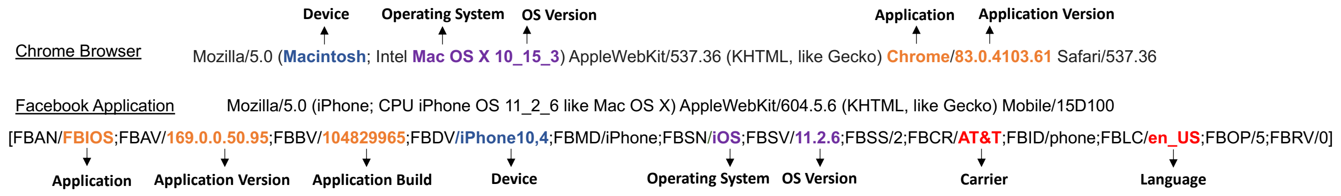

Device
Operating System
OS Version
Application
Application Version
Chrome Browser
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.61 Safari/537.36
Facebook Application
Mozilla/5.0 (iPhone; CPU iPhone OS 11_2_6 like Mac OS X) AppleWebKit/604.5.6 (KHTML, like Gecko) Mobile/15D100
[FBAN/FBIOS;FBAV/169.0.0.50.95;FBBV/104829965;FBDV/iPhone10,4;FBMD/iPhone;FBSN/iOS;FBSV/11.2.6;FBSS/2;FBCR/AT&T;FBID/phone;FBLC/en_US;FBOP/5;FBRV/0]
Application
Application Version
Application Build
Device
Operating System
OS Version
Carrier
Language

Figure 4: Extracting information from User-Agents. Samples for Chrome for OS X (top) and Facebook mobile application for iOS (bottom) highlight the variety and breadth of host information.

aggregated similar sets of clusters across days, yielding an aggregated set of meta clusters in the end. To merge similar clusters over time, we rely on the overlapping occurrence of popular UA strings across days. We first transform each cluster set into a graph, with nodes identified by their UA string and edges interconnecting UAs within the same cluster. Each cluster therefore becomes an independent connected component in this graph. This transformation is necessary since the output of our clustering is unlabeled and inconsistent between days. To merge a new clustering graph, we add edges between nodes in the different graphs that have the same UA string, effectively connecting the components from the two snapshots. Our two-step method of first clustering strings on a daily basis, combined with aggregation across days, ensures that the final number of clusters stays maintainable, but also ensures that we catch clusters of UA strings that were visible only for part of our measurement window, e.g., newly emerging clusters as result of new applications.

Final cluster output. Merging the clusters from all daily snapshots, thus representing 3.5 years worth of captured User-Agent strings, yielded a total of 107 unique clusters. We find that our clustering approach returned aggregated clusters with remarkable similarity. Most clusters can be grouped into common categories, such as Webkit browser strings (i.e Chrome, Safari), iOS applications, Android applications, and many cases of popular but unique UAs for individual applications.

4.3 Classification and Information Extraction Pipeline

The next challenge is to develop a parser that can accurately identify a given UA class, and can extract all given information regarding the OS, application, device, etc. For the clusters generated in the previous section, we manually craft a regular expression to capture the features of the cluster's particular structure. In light of the structural similarity between certain clusters, a single regular expression often captures information from several clusters. We crafted only 35 parsers to capture all 107 clusters. While we were able to extract a portion of UA information from regular expression groups alone, for some clusters we found it necessary to use ad-hoc rules, such as token processing, to extract the required information. In all we found that custom parsing functions were quite rare: of the 35 parsers we created, only 6 required custom parsing functions to supplement regular expression extraction.

For our custom parsing rules, we found that simple manual intervention was sufficient to cover the most common cases. Figure 4 (top) shows an example of our process for popular web browsers.

Metric	Description
Application Name	Name of the User-Agent application, either browser or application (e.g. Safari, Facebook)
Application Version	Version of User-Agent application
Operating System Name	Name of the User-Agent operating system
Operating System Version	Version of the User-Agent operating system
Operating System Build	Specific build of the User-Agent operating system
Device Name	Identifier of User-Agent device

Table 1: Classes of information parsed from User-Agents.

The common structure is very similar, yet the interior tokens vary in their information given, the ordering of the information, and the manner it is displayed. In this example, the parenthesis-enclosed area presents multiple semi-colon delimited tokens. We are able to easily parse the details of these tokens with a simple logic – much easier than attempting to match the multiple combinations through regular expressions alone. The example in Figure 4 (bottom) presents a common User-Agent from the Facebook application. In this UA, there exists a series of tokens enclosed in brackets that are also semi-colon delimited. Each token contains a key-value pair separated by a forward slash. Our parsing rules simply decoded each key's label to our corresponding categories to extract the information. For example, the token `FBDV/iPhone10,4` maps the key `FBDV` to our device category, and `FBSV/11.2.6` maps the key `FBSV` to our operating system version category. This User-Agent is a good example where regular expression parsing was problematic, since it was common to encounter different combinations and orderings of the given keys.

We match a UA to a particular parser class using each class's regular expression. In cases where multiple classes match a UA, we select the class which matches the longest substring within a UA. While we discovered certain UAs to expose additional device and application information, we settled on a set of common structured information available in most UAs, shown in Table 1.

4.4 Classification Efficacy

In this section, we first provide an overview of the information we harvested from over 3.5 years of daily User-Agent strings, leveraging our semi-automated information extraction method.

In all, our process matched 87.3% of total requests carrying a UA string, and 58.6% of the unique UAs in our dataset. We found that most UAs have similar structures and patterns and are able to be captured by a small number of parser classes. We find that, e.g., on January 1, 2020 (other days similar), the two largest parser classes account for roughly 50% of all matched UAs, and the top 5 account for nearly 75%. In this case, the first parser class is designed to match the Webkit UA, used by popular browsers such as Chrome, Safari, and Edge, and the second to match common iOS application UAs. The remaining UAs have much more variability in their structure,

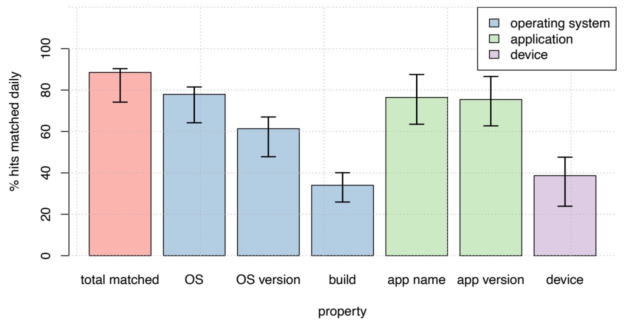


Figure 5: Median daily UAs matched and percentage of property inferred using our User-Agent classification approach. Bars show the median daily matched requests and properties, error bars show max and min over our entire time frame (3.5+ years).

requiring custom parser classes to match and extract, and have diminishing returns for overall matches. In fact, the bottom 25 parser classes only account for some 4% of User-Agents on that day.

To assess the effectiveness of the extraction of information using the regular expressions we derived, we plot the percentage of daily requests from which we could extract information in Figure 5. For the large majority of these UA hits we can extract both the OS and the OS version, and for half of them the OS build. In the majority of the cases we can also extract the application name and version. For devices, the coverage is lower, but still significant, i.e., for about one third of the requests we can infer the device. Then, we examine the variability over time of successfully extracting the information. To this end, we plot the error bars in Figure 5 that represent the minimum and maximum percentage of requests that our methodology can extract information from for each type, for each day in our measurement window. We notice that the range is rather stable, staying within a 20-25% window across all types for the entirety of our dataset.

4.5 Classification Results and Filtering

Our processing pipeline extracts application and operating system information from User-Agents. The largely unstructured nature of User-Agents means that we observe noise in our extracted results. We filtered out these ephemeral applications, which only occur a few times throughout our collection period, ignoring those which were observed fewer than 10,000 times. This filtering step leaves us with **35.2 thousand applications**, from the original 2.5 million, for the remainder of this analysis. We also found application versions to contain large numbers of low hit values, with 99% of all detected versions seeing fewer than 10 hits over our observation period. We further restrict the set of individual versions of applications we analyze to those with more than 1,000 hits, leaving us with **165 thousand distinct application versions**.

We found that Web browsers represent a minority of User-Agents in our dataset, with popular browsers,[2] comprising only 38.6% of detected UAs. The remaining 61.4% of UAs represent the growing fraction of non-browser HTTP traffic on the Internet. While mobile apps make up a large fraction of this set, operating system services such as software update daemons, and desktop applications also exist. With regards to the popularity of Web browsers, we find a clear divide between desktop and mobile platforms. Desktop operating systems show browser User-Agents predominantly, with 96.6% and 82.2% for Windows and Mac OS X respectively. On mobile platforms, however, we find that browsers make up only 29.4% of User-Agents on Android, and only 13.2% of iOS User-Agents.

[2]For popular browsers we selected User-Agents for Chrome, Safari, Firefox, Opera, Internet Explorer and Edge.

5 A SOFTWARE ECOSYSTEM IN FLUX

With our method to extract viable information from User-Agent strings in hand, we now analyze the release and adoption of Internet-connected applications. We describe our methodologies for inferring software release dates from our dataset, introduce metrics that can capture and describe the distribution of software updates across the device eco-system, their timing aspects, update fidelity, and study dominant trends in software adoption across different applications and platforms.

An illustrative example. Figure 6 presents an example overview of what we find via the analyses in the following sections. For each day over 3.5 years, the figure shows the fraction of requests seen from different versions of the Chrome browser running on Android. Each contiguous color area corresponds to a unique version. The dashed vertical lines denote detected release dates for new versions. The two, key points are: *(i)* new versions quickly reach adoption of about 50% to 60% and then the adoption increases more slowly, and *(ii)* 22% of the requests are from versions that are more than 30 days out-of-date and 6% are over 300 days out-of-date, which suggests that there is significant opportunity for malicious actors to exploit vulnerable software. In the remainder of this section, we develop metrics to capture and study such adoption trends more broadly.

Request counts vs. device population. The counts of requests seen by the CDN for different versions of an OS, browser, or application are related to the number of devices/users making these requests, and we can make some general inferences about the devices/users, even though we do not know the actual counts of devices with a given version at a given time. The counts of requests for different versions can vary by 6 orders of magnitude, from which we infer the more popular versions on the devices. When we begin to see requests from a given version, we infer that some devices have it installed. We see both quick and slow ramp up of requests, from which we infer how quickly a new version has been installed on devices, for example, via automatic updates or requiring user action. If the counts from a given version dominate the counts from all prior ones, then we infer that the majority of devices are now using that version. When we present results in terms of the percentages of requests seen from a given version, the percentages should be viewed as rough approximations of the percentages of devices/users with that version.

5.1 Inferring Software Release Dates

We numerically describe the state of deployed applications. For this analysis, we utilize the subset of applications and versions described previously in Section 4.5.

Data-driven software release detection. We derive the software release date of individual versions from their appearance in our

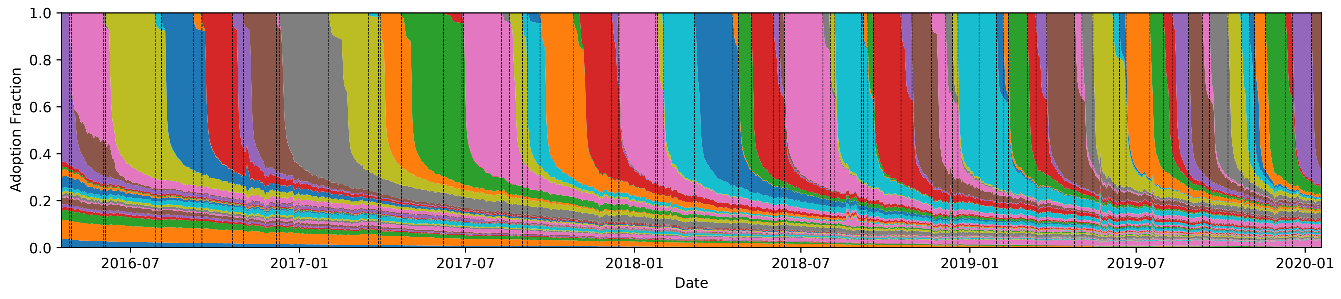


Figure 6: Version adoption of Chrome Browser on Android. In the above stacked area chart, the colors are used to identify adoption of individual browser versions. Each contiguous color area corresponds to a unique version. The dashed vertical lines denote detected release dates for new versions. Note that new versions quickly reach adoption of about 50% to 60% and then the adoption increases more slowly, and throughout the 3.5 years, lingering, older versions remain active for months to years. We find 22% of the requests are from Chrome versions that more than 30 days out-of-date and 6% are over 300 days out-of-date.

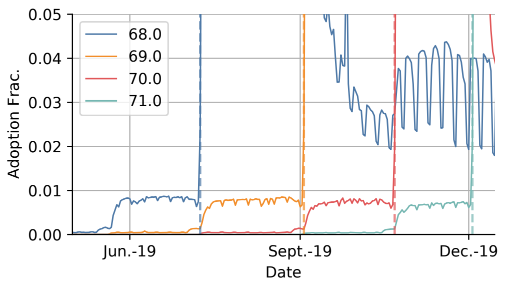


Figure 7: Observed "beta" releases for Firefox browser on Windows. Solid lines plot the adoption rate for each version, dashed lines plot the calculated release date. Beta releases consistently account for ~1% of population.

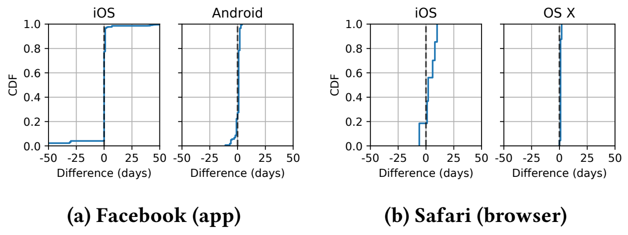


Figure 8: Validation of release date inference across 2 popular applications in our dataset, and across different platforms.

dataset. While some popular application release dates are readily available, many are not. Given the size of our application set studied (some 35,000 applications), it is infeasible to mine all available software release notifications—even if such information was widely available—and we instead opted to infer release dates for all applications from our dataset itself. We refer to an application's version by the most specific version in a given UA. Changes in major version numbers, for instance, are arguably arbitrary and hide certain update dynamics. We are interested in adoption of distinct software builds in order to best understand the behavior of software update dissemination across connected devices.

One consequence of this decision is that many individual software versions appear prior to their official release date. This is commonly due to testing, or "beta" versions of an application which are given to a small subset of active users for testing. For example, we originally discovered that Firefox releases were detected at roughly 30 to 60 days before their official release. Further investigation revealed that new versions of Firefox show adoption around 1-2% for a period of time before their official release. Mozilla maintains a separate release channel for Beta, which operates one major version ahead of the current release. Figure 7 plots this behavior, showing approximately 1% of the population shifting to a future release immediately after a new release. We ignore most of these spurious testing versions by calculating the software release date of a particular version as the first date on which its traffic share exceeds 1% of the overall application traffic share on that day. However, we manually identified a handful of applications where this sort of live trial of upcoming versions exceeds 1% adoption, and thus for these rare cases we increased the minimum threshold for adoption percentage from 1% to 5%.

Validation of release date inference. In total we inferred 148,041 release dates of application versions. We validated our release date inferences against publicly announced release dates from Google Chrome, Safari and Firefox browsers, and the Facebook mobile application. In all we were able to validate 540 application versions. We find that our inferred release dates are similar to the published releases, with 74% of versions within 2 days of inferred release dates and 97% of inferred release dates are within 20 days of the published date. Figure 8 plots this comparison for two popular applications.

Release frequency. With our ability to automatically derive software release dates, we next measure the time between an application's consecutive releases. Overall we find a very high frequency of application releases, with a median value of 10 days between releases. The 25th and 75th percentiles range between 4 and 25 days, respectively. We find that the release frequency varies widely between applications, which we illustrate with a subset of popular applications. Figure 9 plots the median time between releases, with the error bars denoting the inner quartiles, for these popular applications. We observe a wide range of behavior, from iOS Safari's median 51 days between releases, all the way to the Facebook app on Android with a median of 6 days between releases.

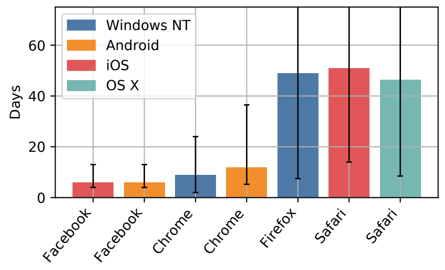


Figure 9: Median days between releases for a set of popular application. Error bars represent middle quartiles.

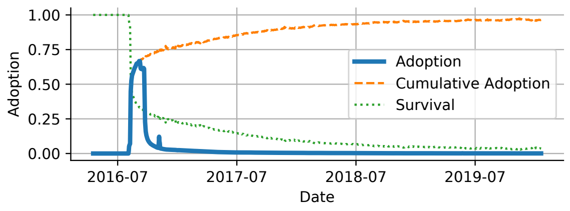


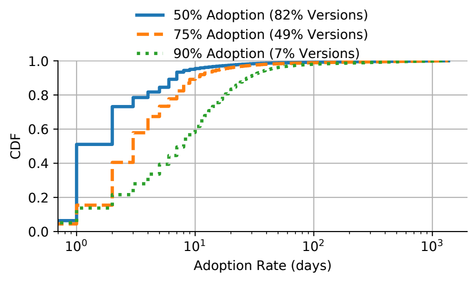
Figure 10: The blue line (which initially coincides with orange line) represents application *adoption* for Chrome on Android version 52.0.2743.98 over time. The orange line represents the *cumulative adoption*, including all versions ≥ 52.0.2743.98.

5.2 Metrics to Capture Version Adoption

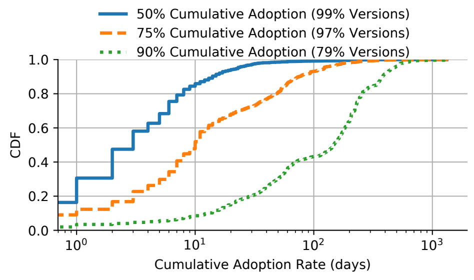
For our update analysis below, we discard any versions which existed on the first day of our data collection, as our method to infer their release date does not apply.

Version adoption. In order to characterize a software's population at any point in time, we define *version adoption* to be the fraction of an application's requests from any particular version. Understanding the characteristics of version adoption are important for modeling software and security patches, and understanding the extent to which existing update mechanisms influence software distribution. To capture temporal properties of updates, we measure version adoption over time—the *adoption rate*. The blue line in Figure 10 represents the individual version adoption for version 52.0.2743.98 of Chrome for Android. Common to many applications, the version grows quickly after being released, reaching 50% of global adoption after only 8 days. Then, the growth rate slows. The version reaches a maximum adoption of 63% at 31 days after release, after which it incurs a steep drop. The drop starts when a new version is released. This indicates that there is a population of application users which maintain very up-to-date software.

Cumulative adoption. Due to the fast pace of software release versions, the scope of any one particular version may not be descriptive of the state of a software population. We introduce another metric—*cumulative adoption* —which describes the fraction of a population which have adopted a given version or a more recent one. We use the expression $y \geq x$, where x is the given software version, and y is the versions the same as or newer than x, to express this adoption. Similar as above, the time taken to reach a particular cumulative adoption is the *cumulative adoption rate*.



(a) Adoption rate for all application versions.



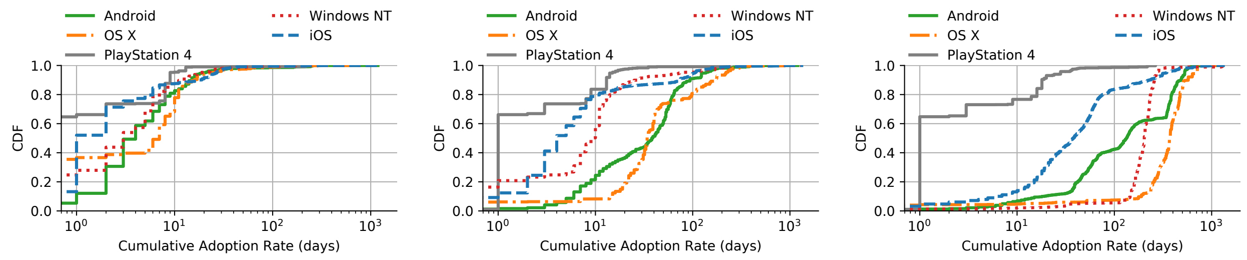
(b) Cumulative adoption rate for all application versions.

Figure 11: Temporal properties for version adoption across all operating systems. The adoption rates shown are conditional on individual versions reaching the specified percentile of adoption. Only 7% of all application versions reach 90% adoption. 21% of versions never reach a cumulative adoption of 90%.

The orange line in Figure 10 represents the cumulative adoption rate, which measures the adoption of all versions ≥ 52.0.2743.98. The slow, steady increase reveals that there is a significant portion of the application's population—around 35% in this case—which update their software at a lower frequency than the application's release cycle. That the cumulative adoption rate never reaches 100% means that there exists a small fraction of the population which maintains software older than this version for the three year duration of our measurements. Cumulative adoption is the inverse function of the Survival function $S(t)$, which is used in related work to model vulnerable application populations [29]. We chose cumulative adoption for this work instead of the Survival function because we believe it is more intuitive for measuring overall update adoption. For comparison, we also plot the Survival function in Figure 10.

5.3 Tracking Version Lifecycles

Time to adoption. To measure the rate of software adoption of application versions, we calculate the time taken from its release to reach 50%, 75%, and 90% of an application's population. However, note that many versions fail even to reach these percentiles. Previous studies [29, 39] have also reported that the adoption of updates can be low for the small set of applications they studied. While 72% all versions reach 50% adoption, only 56% reach 75% adoption and only 27% reach 90% adoption. If we weight versions by hits, which gives more weight to the popular versions, then fewer reach the higher percentiles: 49% reach 75% adoption and only 7% reach 90%. Figure 11a plots the distribution, in days, of the time taken



(a) Time to 50% cumulative adoption. **(b) Time to 75% cumulative adoption.** **(c) Time to 90% cumulative adoption.**

Figure 12: Platform is a strong determinant of application update behavior.

for individual software versions, weighted by hits, to reach a given percentile of the hits from the given application, conditioned on the version actually reaching that percentile. On the whole, we find that software versions that do reach 50% population share do so relatively quickly, with the mode occurring at one day, and 90% of versions reaching 50% share within 2-3 days. Those few versions that do reach 75% and 90% also do so expediently, with modes at 5 and 12 days, respectively.

Adoption vs. cumulative adoption over time. We believe that the high frequency of application releases limits the overall version adoption. We found that, in general, version adoption rates were higher when the version was the latest version for longer periods. Intuitively, as newer versions replace the latest, there is less time for users to adopt any individual version. When a given software version has corrected vulnerabilities that were present in a prior version, the adoption rate of that individual given version provides only a partial view into the extent that these vulnerabilities have been removed from the application's user base. As an example, suppose the given version of Chrome on Android in Figure 10 corrected some vulnerability, then the correction would have 75% cumulative adoption after about 81 days and 90% after 423 days.

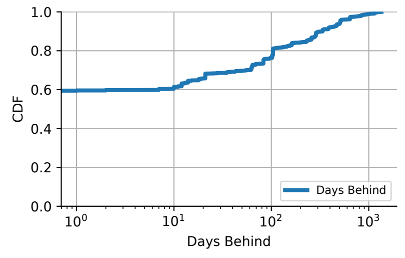
Adoption rates of individual versions are do not reveal the full picture, due to the low number of individual versions which actually reach high levels of adoption. We believe cumulative adoption is more representative of population behavior, especially when measuring high adoption fractions: 97% and 79% of versions reach 75% and 90% cumulative adoption respectively, which is qualitatively higher than the percentages for (individual version) adoption of 49% and 7%. Figure 11 plots the adoption and cumulative adoption rate for all application versions in our dataset. We find that cumulative adoption rates are substantially longer than individual version adoption rates. The increased coverage of cumulative adoption captures the complexity of software updates which have high release velocities, showing that these require more time to achieve high levels of adoption. The figure highlights the challenges for high levels of update adoption. Of the 79% of versions which reached 90% cumulative adoption, half required 145 days to achieve, and 10% required more than 383 days.

5.4 The Platform Effect

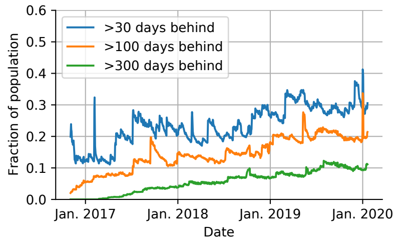
We find that application update behavior differs significantly across platforms, especially at the higher percentiles of adoption. Figure 12 plots the 50th, 75th and 90th percentiles of cumulative adoption for all applications, grouped by operating system. We include 2 desktop, 2 mobile and one set-top box as platforms for comparison. Indeed, 80% of the applications of PlayStation4, iOS, Android, Windows NT, reach 50% in 8 days as shown in Figure 12a. With the exception of OS X, 80% of all applications reach 50% cumulative adoption within 20 days. Turning our attention to 75% cumulative adoption (Fig. 12b), 80% of the applications on PlayStation4, iOS, and Windows NT reach 75% cumulative adoption rate in around 12 days. However, 80% of applications in Android and OS X reach 75% cumulative rate in around 50 and 100 days, respectively. Finally, 80% of the application of PlayStation4 reach 90% cumulative rate in around 12 days, but for the other platforms 80% of the applications reach 90% adoption rate in between 100 (iOS) and 300 days (OS X), see Figure 12c. Next, we investigate the differences between platforms.

Embedded platforms. The PlayStation 4 gaming console has the quickest and most comprehensive adoption behavior of all platforms. Nearly 70% of applications reach 50% adoption on the same day of the release, and 65% reach 90% cumulative adoption after only one day. Applications on this platform reach the 95th percentile of 90% adoption in 19 days, ten times faster than iOS, the next quickest platform. If we look at the process of software updates on this platform, there are several clues to its success, with several default options and norms which may contribute to the high rate of update compliance. There is a default mode when the device is not in use which downloads updates. When the device prompts the user to update at next use, this greatly reduces the burden of updating. More importantly are the norms surrounding game/application updates. Many applications which allow players to interact online require the latest software version. With online gameplay being a large part of many modern games, this forces update compliance for use. Relatedly, many application updates require the latest operating system updates as well, creating a halo effect for operating system compliance.

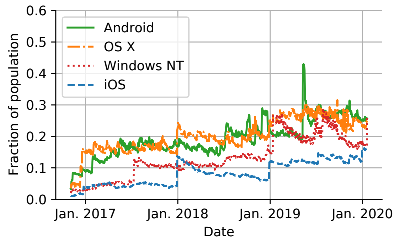
Mobile platforms. We next look at mobile platforms, comparing application update behavior between iOS and Android. Overall we find applications on Android to take 3-10 times longer to reach similar adoption when compared to iOS. The median application takes 4 days to reach 75% cumulative adoption on iOS, while taking 38 days on Android. Both Android and iOS have official software release channels, Google Play and AppStore respectively, which are managed by the operating system owners. Since the beginning



(a) Distribution of days behind for detected versions on Jan. 1, 2020. 60% of versions are fully up to date, though with a long tail.



(b) Days behind over time for all platforms and applications for multiple thresholds.



(c) Fraction of population that are at least 100 days behind, by platform.

Figure 13: Days behind measurements across multiple time frames and platforms.

of our measurement period, both Android and iOS platforms provide options for automatic application updates. Multiple studies have previously outlined how the fractured nature of the Android ecosystem contributes to poor adoption of operating system updates [1, 30]. It is possible that a similar effect is occurring with applications as well. Android also has wider use of App Stores other than the official Google Play, such as the Amazon App Store [4] along with many others. It may be that Android applications on these alternative marketplaces are released at different times, and since we have no way of knowing the provenance of detected applications, this may account for some of the lag in application updates.

Desktop platforms. Last we compare application update behavior between the Windows and Mac OS X desktop platforms. In all we found applications on Mac OS X to have the lowest cumulative adoption rate, across all percentages. Applications running on Windows show comparable rates to the above mobile platforms, yet have a long tail reaching 90% cumulative adoption. Again, in the case of Mac OS X, the user has to agree before downloading updates and this potentially slows down adoption rates.

6 WHO'S LEFT BEHIND?

The currentness of running software is dependent on a variety of factors, and our tools allow us to identify and track these factors. In this section, we track software currentness on a macroscopic level. In particular, we are interested in the population of users that run an outdated version of a given application. There exists a fraction of each application's population which are running outdated versions. In Figure 6 we exemplified this using versions of the Chrome browser on Android, where each version is represented in a distinct color. We observe that while new versions typically reach high adoption rates of some 70%, we observe an increasing number of old versions that remain "out there" and in use.

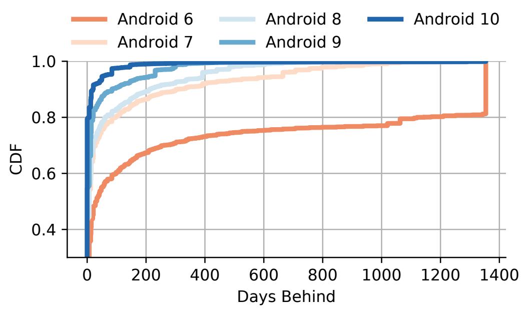
6.1 Capturing Behindness

The age of any version is the number of days between the current date and the release date of that version. A large age of any version may not indicate that a population is outdated or vulnerable, and instead depends on the release decisions of application owners.

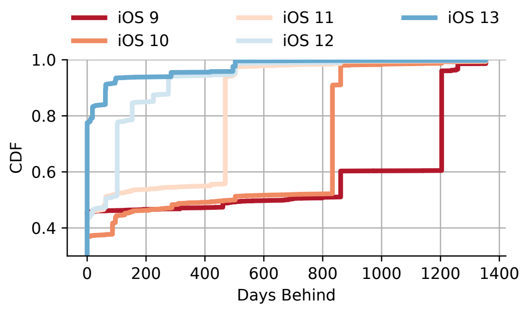
Behindness metric. A more appropriate metric for understanding this vulnerable client population is to measure the number of days that an application version is out-of-date, which we refer to as *days behind*. We calculate an application version's days behind as the number of days which have passed since a newer version of the application has been released. For example, clients running the latest version of an application, v_i, are considered zero days behind. If a newer application version v_{i+1} was released 10 ago, then all clients running version v_i would be 10 days behind, and those now running v_{i+1} are 0 days behind. As an example, in Figure 6, the first detected version is at April 26, 2020 (pink area), which 6 months after release is at 75% cumulative adoption, and thus all of the other versions that existed prior to its release are behind by at least 6 months, yet still account for 25% of the requests. Figure 13a plots the distribution of the days behind for all applications from the viewpoint of January 1, 2020. On this date, we find that 60% of all applications are running the latest software versions – that is 0 days behind. The remaining 40% exhibit a long tail, with 20% of the population at least 100 days behind their latest software version, and 10% over 400 days behind.

Behindness over time. To view this metric over time, Figure 13b plots the fraction of our population which are 30, 100 and 300 days behind. A complication is that when the viewpoint is at start of the observation period, we have not yet inferred the software release dates of the various applications, Section 5.1, and thus we can not compute the number of days behind. Thus we need a warm-up period. In principle, we wait until the first release date after the start of observation period and then another 300 days, for example, to compute the percentage of requests that are that far behind. A further complication is that these release dates vary for the different applications. Based on the statistics for days between releases, Figure 9, we judge that ball-park warm-up periods of 150, 250, and 450 days, respectively, are reasonable to get estimates of the percentage of requests that are 30, 100, and 300 days behind. Thus, looking at the results in Figure 13b after the warm-up periods, we see an increasing trend of the percentage of requests that are behind.

Behindness per platform. With our metric of days behind in hand, we next study the populations of different platforms. Figure 13c shows, for four popular operating systems, the fraction of daily requests (from all applications running on the given OS) that are more than 100 days behind. Between these platforms, Android and OS X have the largest fraction of the population with applications greater than 100 days behind. In contrast, iOS has fewer than

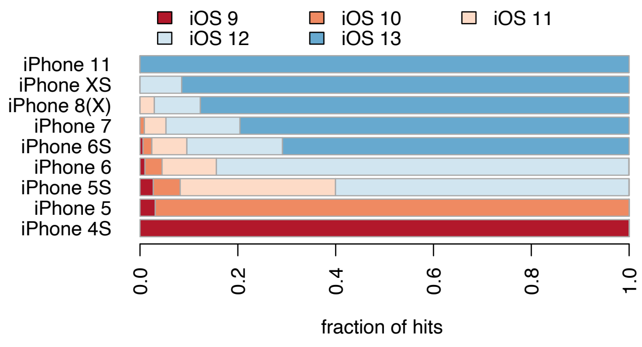


(a) Android.

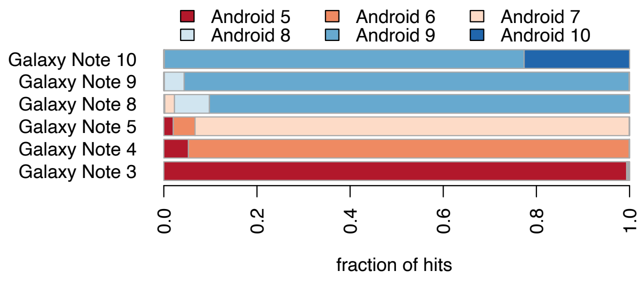


(b) iOS.

Figure 14: Distribution of days behind of requests partitioned per operating system version for Android and iOS on January 1, 2020. Application behindness increases in older operating systems.



(a) iOS version on iPhone devices.



(b) Android versions on Galaxy Note devices.

Figure 15: Fraction of requests matching a specific OS version on January 1, 2020 running on select devices. Older devices tend to run older OS versions, in part due to non-availability of OS updates for older devices, and in part due to a lack of action by users.

15% of its population with versions more than 100 days behind. The sharp increases in Figure 13c) are due to highly adopted versions, of very popular applications, first reaching 100 days behind. For example, the sudden increase observed for Windows NT in July 2017 is due to Microsoft's Edge browser version 13.10586 first becoming 100 days behind the latest version 14.14393. At the time of this transition, version 13 occupies roughly 40% of Edge's population, and Edge's popularity increases Window's behindness by 7% overall.

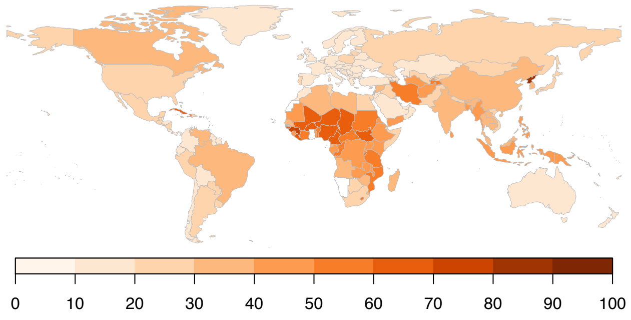
6.2 Understanding Root Causes of Behindness

Our results show that platforms have a profound impact on the behindness of applications. In this section we investigate further the root causes by analyzing the effect of the age of the operating system and the device on application behindness. While the motivations surrounding out-of-date applications are complex, we found several correlations between out-of-date applications and the operating systems and devices they run on. Using a single day snapshot from January 1, 2020, we investigate these causes.

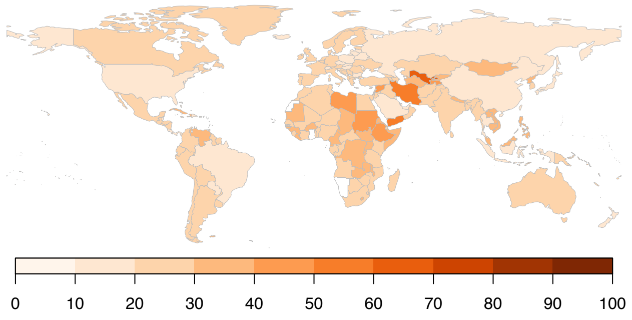
Impact of operating system aging. We discovered that application behindness was higher on older operating systems. Figure 14 plots the cumulative distribution of application behindness for major versions of Android and iOS. The figure illustrates this trend clearly, with the latest mobile operating systems having applications that are highly up-to-date (typically within 10 days), and older operating systems showing greater application behindness—increasing with the oldest OS versions. While this does not provide direct causation for outdated applications, since that may be dependent both on user behavior and/or application availability for different operating system versions, it does show that the older one's operating system the more likely for applications to be behind the latest updates.

Impact of device aging. We now consider the age of the devices. Figure 15 plots the fraction of requests seen from different operating system versions, partitioned by different generations of the device. We selected the Galaxy Note running Android and the iPhone running iOS, both for their popularity and long history of devices within the same line. The figure illustrates that newer devices tend to run newer operating systems, and thus, as we showed in the previous section, newer application versions. Moreover, note that older Android and iOS devices have not been updated to recent operating system versions at all, which suggests causation, and which we investigate next.

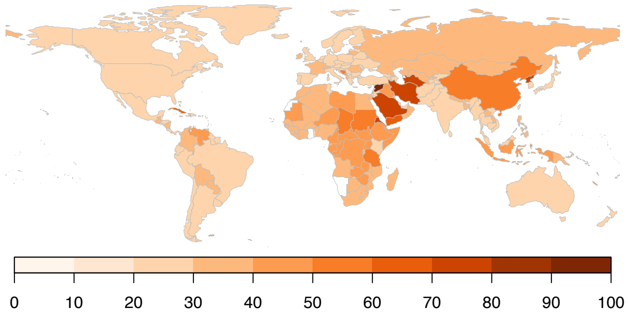
Device lock-in effects. For both studied devices, the iPhone and the Galaxy note, it is evident that older models tend to run an older version of the OSes. A closer investigation shows that none of the iPhone 4 or 5 devices run iOS 13. Moreover, none of the Samsung Galaxy Note 3, 4, and 5 runs Android 8 or later. Indeed, the latest version of operating systems are not supported by some older devices of Apple [7] and Samsung [38]. Thus, the owners of such devices can not update them to the latest version. This is true either because these devices are not anymore supported or because the support is delayed, e.g., the Galaxy Note 9 could be updated



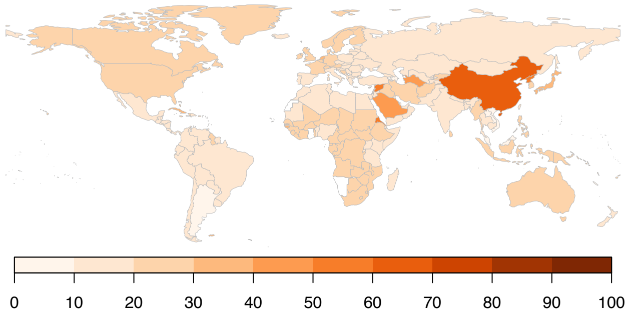
(a) Android.



(b) iOS.



(c) Mac OS.



(d) Windows.

Figure 16: Per country: % of requests that are behind more than 100 days.

to Android 10 only in late January 2020, although Android 10 was released in September 2019.

We investigate further whether these outdated operating systems on devices are restricted by the devices themselves, through a lack of support for later OS versions, or by intentional non-updates. We find that devices are strongly associated with a single major version of an operating system. For the top 1, 000 Android devices on January 1, 2020, we observe on average 89% of a device's hits are from the same OS major version. In particular, for 40% of the top devices, we see only a *single* major OS version, and those devices for which we observe at least two major OS versions have only 16% of hits in the next largest version.

User fatigue. Figure 15 clearly shows that some newer devices, both iPhone as well as Samsung Galaxy Note devices, that could have been updated to the latest version of the OS, in fact, are not. This is a strong indication that the willingness of a user to update its device plays an important role. It also shows that even if automatic update mechanisms are in place by default, some of the users may even opt-out, and thus, reduce the effectiveness of such update mechanisms.

Device lock-in vs. user fatigue. To compare the impact of device lock-in effect to user fatigue, we look at the population fraction and device support of the latest operating system release for both iOS and Android. With the latest iOS release, we find the effects of device lock-in and user fatigue to be roughly equivalent. We find that 88.8% of iOS requests come from devices which support iOS version 13, yet only 76% of iOS requests come from this latest version. This reveals 11.2% of requests originate from active devices which are locked-out of future OS releases, and 12.8% of requests from devices which have been deliberately un-updated. With Galaxy Note, the device lock-in effect is much more pronounced. We find that only 8.2% of Android requests are from devices which support Android 10, leaving 91.8% of devices locked-out of the latest OS version. Even including Android 9, we still find that 42.7% of requests come from devices which support neither version. While user fatigue exists for the latest version – only 3.4% of requests are from Android 10, while 8.2% are from devices that support it – the effect is greatly overshadowed by the device limitations.

6.3 Geographic View

We next study the impact of geography on application currentness. Figure 16 shows per country the fraction of requests from applications whose software version is behind at least 100 days, partitioned per platform. Across all platforms, we observe that less-developed countries tend to show a higher share of outdated requests when compared to industrialized nations. We see this trend particularly pronounced for Android devices, see Figure 16a, where some central African countries show up to 90% of requests coming from applications that are more than 100 days behind. We speculate that this high level of behindness relates to the use of cheap devices that often run outdated Android versions, hence not giving the user the possibility to update applications to current versions. We note that this trend is less-pronounced for iOS (Figure 16b), possibly a result of Apple's policy to also provide iOS updates for older hardware. Eyeballing desktop platforms, we find that Mac OS shows a similar pattern, with most industrialized countries showing high levels of currentness, and African countries showing high levels of outdated requests, but also high shares of outdated requests from Syria, Saudi Arabia, and Iran. Windows, on the other hand, shows a rather even distribution globally, with the exception being China that shows the highest levels of outdated Windows requests. The phenomenon of old Windows versions in China has been reported several times [8]. To illustrate the variability for a few, non-randomly selected countries, the percent of requests from Android versions that are out-of-date by more than 100 days is: 17% Germany, 20% Russia, 21% Japan, 22% Egypt, 24% USA, 27% Argentina, 32% Canada, 36% India, 40% China, and 63% Nigeria. An independent very recent study [25], also confirms that devices, particularly mobile ones, in developing countries often have legacy browsers. Overall, we find that the adoption of current software is highly unevenly distributed geographically, potentially putting less-developed regions at a greater risk for cyber threats as a result of vulnerabilities in outdated software.

7 DISCUSSION

In this section we discuss implications of our work for the research community, software vendors, and policy makers.

Timely reporting of the state of application updates. Timely and comprehensive updates are essential for security strategies, yet we uncovered multiple challenges in this area. Our study illuminates the limitations inherent to any application updating strategy at scale. We found that only 79% of all application versions ever reach a cumulative adoption rate of 90%. Moreover, adoption of new software versions varies widely between applications and across platforms, which our analysis strongly reflects. Our work presents a first step towards a principled comparison of these different approaches, allowing comparison of adoption practices across applications, operating systems, and devices. For example, our study unveils that devices with updated OSes tend to adopt newer application versions. Thus, it is imperative to increase the user awareness of OS updates as this can have a domino effect on adopting newer application versions. Moreover, our longitudinal analysis allows for comparison of the effectiveness of strategies in the wild and also helps in assessing the effectiveness of efforts by vendors to improve the adoption of new versions of applications over short as well as long observation periods. It also allows for the estimation of the population with potentially vulnerable software over time.

Device lifetimes. There is an increasing debate across policy makers regarding the sustainable consumption and device durability of end user products, including personal computers and smartphones. Regulatory bodies, e.g., the European Union Parliament, are in the process to extend consumer protection laws to eliminate "incompatibility obsolescence", and discuss introducing a "right to repair" [19, 32] for older smartphones and laptops. Our results show that any legislation aiming to extend product lifetimes (e.g., the "right to repair") must take upgradeability of the software running on devices into account. Today, older devices often cannot be updated to the most recent OS versions. Potential legislation that ensures continued support, even for older devices, could not only extend device lifetimes, but also reduce software behindness and ease related cybersecurity issues.

Target regions for improvements. Our results show pronounced differences across geographical regions when it comes to the currentness of applications. These insights can inform direct efforts towards improving update adoption in populations with out-of-date software. In this study, we identified that populations in parts of Africa and Asia lag behind the rest of the world in active software versions. While we do not comprehensively analyze the root causes of software behindness in specific regions, we identified that older and non-upgradeable devices strongly impact software behindness. Our results also show that continued availability for support for older devices could, in turn, significantly improve software behindness in these regions. Improving software behindness may also require additional investments in network infrastructure and broadband connectivity. The vast majority of Internet users in Africa use the cellular network for access [41], and charges for cellular traffic might discourage users from updating. Pricing models that discount traffic (even offer it as a free service, similar to FB free basics program [17]) related to the operating system and application updates could improve software behindness and security in these regions. Using longitudinal analysis it is possible to assess the impact of pilot investments on the currentness of applications as well as inform policy making and future investments. As part of our future agenda, we would like to detect populations of particular networks that lag behind and understand the root causes, e.g., specific types of networks and businesses that inhibit updates.

Human factors. Despite the effort by the industry to increase the adoption of application updates and deal with bugs and vulnerabilities, our study shows that a significant fraction of applications are not updated, even if the device, as well as the software running on it, could be updated. Yet, our analysis also shows that different updating strategies employed by different vendors on different platforms indeed show a measurable effect on software behindness, boding well for advanced updating strategies that include easy opt-in to automatic updates, and other means to encourage end-users to keep software current. There are important human factors, including the awareness of the public about the danger of not updating software, and the individual decision, or the lack thereof, of end-users to either download and install updates, or to enable automatic updates. Previous studies conducted surveys to understand how user characteristics affect attitudes towards mobile application updates [20, 28, 44]. Although these studies are very useful, e.g., to understand why users turn off automatic updates based on their previous experience or to avoid disruption of settings, unnecessary reboots, compatibility issues, or changes in the user interface, they typically involve only a few hundred participants. We believe that studying these factors at the scale that our data can support is a fundamental step to better understand the matter and increase security in the Internet. Our analysis can assess the impact of vendors' campaigns to increase user awareness of these issues and provide useful feedback on raising awareness based on empirical data.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Diego Perino, for their very constructive feedback. Georgios Smaragdakis was supported by the European Research Council (ERC) Starting Grant ResolutioNet (ERC-StG-679158).

REFERENCES

[1] 2018. Android Security 2017 Year In Review. Android Open Source Project, https://source.android.com/security/reports/Google_Android_Security_2017_Report_Final.pdf.

[2] 2019. Android Security 2018 Year In Review. Android Open Source Project, https://source.android.com/security/reports/Google_Android_Security_2018_Report_Final.pdf.

[3] 2020. WhatIsMyBrowser.com Developers. https://developers.whatismybrowser.com/useragents/explore/hardware_type_specific/phone/.

[4] Amazon. 2020. Amazon Appstore App For Android. https://www.amazon.com/gp/mas/get/amazonapp.

[5] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou. 2017. Understanding the Mirai Botnet. In *USENIX Security Symposium*.

[6] Apple. 2020. Apple App Store. https://www.apple.com/ios/app-store/.

[7] Apple. 2020. Apple security updates. https://support.apple.com/en-us/HT201222.

[8] K. Chiu. 2020. Windows 7 is gone, but China's dedicated users aren't ready to let go. https://www.abacusnews.com/culture/windows-7-gone-chinas-dedicated-users-arent-ready-let-go/article/3046210.

[9] CISCO. 2020. Cisco Annual Internet Report (2018–2023) White Paper. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html.

[10] US Federal Trade Commission. 2016. FTC To Study Mobile Device Industry's Security Update Practices. https://www.ftc.gov/news-events/press-releases/2016/05/ftc-study-mobile-device-industrys-security-update-practices.

[11] L. F. DeKoven, A. Randall, A. Mirian, G. Akiwate, A. Blume, L. K. Saul, A. Schulman, G. M. Voelker, and S. Savage. 2019. Measuring Security Practices and How They Impact Security. In *ACM IMC*.

[12] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman. 2014. The Matter of Heartbleed. In *ACM IMC*.

[13] Z. Durumeric, E. Wustrow, and J. A. Halderman. 2013. ZMap: Fast Internet-Wide Scanning and its Security Applications. In *USENIX Security Symposium*.

[14] P. Eckersley. 2010. How Unique Is Your Web Browser?. In *Privacy Enhancing Technologies (PETS)*.

[15] Ericsson. 2020. Mobile data traffic outlook. https://www.ericsson.com/en/mobility-report/reports/june-2020/mobile-data-traffic-outlook.

[16] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. 1996. A Density-based Algorithm for Discovering Clusters a Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. (1996).

[17] Facebook. 2020. Facebook: Free Basics. https://connectivity.fb.com/free-basics/.

[18] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. 1999. *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616. IETF. http://tools.ietf.org/rfc/rfc2616.txt

[19] Natasha Lomas for techcrunch.com. 2020. European lawmakers propose a 'right to repair' for mobiles and laptops. [online] March 11, 2020, https://techcrunch.com/2020/03/11/european-lawmakers-propose-a-right-to-repair-for-mobiles-and-laptops/.

[20] A. Forget, S. Pearman, J. Thomas, A. Acquisti, N. Christin, L. Faith Cranor, S. Egelman, M. Harbach, and R. Telang. 2016. Do or Do Not, There Is No Try: UserEngagement May Not Improve Security Outcomes. In *USENIX SOUPS*.

[21] S. Frei, T. Duebendorfer, and B. Plattner. 2009. Firefox (In)Security Update Dynamics Exposed. *ACM CCR* 39, 1 (2009).

[22] Google. 2020. Google Play Store. https://play.google.com/store.

[23] J. Kline, A. Cahn, P. Barford, and J. Sommers. 2017. On the Structure and Characteristics of User Agent Strings. In *ACM IMC*.

[24] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (Jan 2018). arXiv:1801.01203

[25] F. Li. 2020. Shim Shimmeny: Evaluating the Security and Privacy Contributions of Link Shimming in the Modern Web. In *USENIX Security*.

[26] F. Li and V. Paxson. 2017. A Large-Scale Empirical Study of Security Patches.

[27] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. 2018. Meltdown. *ArXiv e-prints* (Jan 2018). arXiv:1801.01207

[28] A. Mathur and M. Chetty. 2017. Impact of User Characteristics on Attitudes Towards Automatic Mobile Application Updates. In *USENIX SOUPS*.

[29] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras. 2015. The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching. In *IEEE Symp. on Security and Privacy*.

[30] K. Nohl and J. Lell. 2018. Mind the Gap: Uncovering the Android Patch Gap Through Binary-Only Patch Level Analysis. HITB Conference 2018.

[31] T. Petsas, A. Papadogiannakis, M. Polychronakis, E. P. Markatos, and T. Karagiannis. 2013. Rise of the Planet of the Apps: A Systematic Study of the Mobile App Ecosystem. In *ACM IMC*.

[32] Scientific Policy Department for Economic and Quality of Life Policies Directorate-General for Internal Policies. 2020. Sustainable Consumption and Consumer Protection Legislation.

[33] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, C. Kreibich, and P. Gill. 2018. Apps, Trackers, Privacy, and Regulators: A Global Study of the Mobile Tracking Ecosystem. In *NDSS*.

[34] J. Ren, D. J. Dubois, D. Choffnes, A. M. Mandalari, R. Kolcun, and H. Haddadi. 2019. Information Exposure From Consumer IoT Devices: A Multidimensional, Network-Informed Measurement Approach. In *ACM IMC*.

[35] J. Ren, M. Lindorfer, D. Dubois, A. Rao, D. Choffnes, and N. Vallina-Rodriguez. 2018. Bug Fixes, Improvements, ... and Privacy Leaks - A Longitudinal Study of PII Leaks Across Android App Versions. In *NDSS*.

[36] P. Richter, G. Smaragdakis, D. Plonka, and A. Berger. 2016. Beyond Counting: New Perspectives on the Active IPv4 Address Space. In *ACM IMC*.

[37] P. Richter, F. Wohlfart, N. Vallina-Rodriguez, M. Allman, R. Bush, A. Feldmann, C. Kreibich, N. Weaver, and V. Paxson. 2016. A Multi-Perspective Analysis of Carrier-Grade NAT Deployment. In *ACM IMC*.

[38] Samsung. 2020. What version of Android can I upgrade my Samsung phone to? https://www.samsung.com/au/support/mobile-devices/android-version-availability/.

[39] A. Sarabi, Z. Zhu, C. Xiao, M. Liu, and T. Dumitras. 2017. Patch Me If You Can: A Study on the Effects of Individual User Behavior on the End-Host Vulnerability State. In *PAM*.

[40] R. Sen, S. Ahmad, A. Phokeer, Z. A. Farooq, I. A. Qazi, D. Choffnes, and K. P. Gummadi. 2018. Inside the Walled Garden: Deconstructing Facebook's Free Basics Program. *ACM CCR* 47, 5 (2018).

[41] International Telecommunication Union. 2020. ICT Data and Statistics. https://www.itu.int/ITU-D/ict/statistics/ict/.

[42] International Telecommunication Union. 2020. Individuals Using the Internet Statistics. https://www.itu.int/en/ITU-D/Statistics/Pages/stat/default.aspx.

[43] US Federal Trade Commision. 2018. Mobile Security Updates: Understanding the Issues. https://www.ftc.gov/news-events/press-releases/2018/02/ftc-recommends-steps-improve-mobile-device-security-update.

[44] K. Vaniea and Y. Rashidi. 2016. Tales of Software Updates: The Process of Updating Software. In *ACM CHI*.

[45] S. Vargas, U. Goel, M. Steiner, and A. Balasubramanian. 2019. Characterizing JSON Traffic Patterns on a CDN. In *ACM IMC*.

[46] W3C. 2020. Definition of User Agent. https://www.w3.org/WAI/UA/work/wiki/Definition_of_User_Agent.

[47] N. Xia, H. H. Song, Y. Liao, M. Iliofotou, A. Nucci, Z-L. Zhang, and A. Kuzmanovic. 2013. Mosaic: quantifying privacy leakage in mobile networks. In *ACM SIGCOMM*.

[48] L. Zhang, D. Choffnes, D. Levin, T. Dumitraş, A. Mislove, A. Schulman, and C. Wilson. 2014. Analysis of SSL Certificate Reissues and Revocations in the Wake of Heartbleed. In *ACM IMC*.

[49] Y. Zhang, H. Mekky, Z-L Zhang, R. Torres, S-J Lee, A. Tongaonkar, and M. Mellia. 2015. Detecting Malicious Activities with User-Agent Based Profiles. *Int. J. Network Management* (2015).