A Measurement Study of Wechat Mini-Apps

YUE ZHANG, The Ohio State University, USA
BAYAN TURKISTANI, The Ohio State University, USA
ALLEN YUQING YANG, The Ohio State University, USA
CHAOSHUN ZUO, The Ohio State University, USA
ZHIQIANG LIN, The Ohio State University, USA

A new mobile computing paradigm, dubbed mini-app, has been growing rapidly over the past few years since being introduced by WeChat in 2017. In this paradigm, a host app allows its end-users to install and run mini-apps inside itself, enabling the host app to build an ecosystem around (much like Google Play and Apple AppStore), enrich the host's functionalities, and offer mobile users elevated convenience without leaving the host app. It has been reported that there are over millions of mini-apps in WeChat. However, little information is known about these mini-apps at an aggregated level. In this paper, we present MiniCrawler, the first scalable and open source WeChat mini-app crawler that has indexed over 1,333,308 mini-apps. It leverages a number of reverse engineering techniques to uncover the interfaces and APIs in WeChat for crawling the mini-apps. With the crawled mini-apps, we then measure their resource consumption, API usage, library usage, obfuscation rate, app categorization, and app ratings at an aggregated level. The details of how we develop MiniCrawler and our measurement results are reported in this paper.

CCS Concepts: • **Network performance evaluation → Network Measurement**; • **Security and privacy → Systems security**; **Software and application security**; • **Software and its engineering → Software system structures**.

Additional Key Words and Phrases: Mini-apps, Crawler, Wechat

ACM Reference Format:
Yue Zhang, Bayan Turkistani, Allen Yuqing Yang, Chaoshun Zuo, and Zhiqiang Lin. 2021. A Measurement Study of Wechat Mini-Apps. *Proc. ACM Meas. Anal. Comput. Syst.* 5, 2, Article 14 (June 2021), 25 pages. https://doi.org/10.1145/3460081

1 INTRODUCTION

Increasingly, a new mobile-computing paradigm, dubbed *mini-app*, which was debuted by WeChat in 2017 [1] that allowed an end-user to use mobile apps directly downloaded from WeChat, is becoming more and more popular. With this paradigm, a *host app* such as WeChat, enables the execution of *mini-apps*, which are small apps built by 3rd-party developers that can run within the *host*, have significantly enriched functionalities of the *host* and elevated user experience (*e.g.*, "*WeChat is used for everything from booking doctor's appointments to hailing taxi rides, making payments, shopping, and even filing for a divorce*" [2]), thereby substantially increasing the users' *stickiness* [3–5]. It has been reported that WeChat has more than one million mini-apps and 440

Authors' addresses: Yue Zhang, The Ohio State University, Ohio, USA, zhang.12047@osu.edu; Bayan Turkistani, The Ohio State University, Ohio, USA, turkistani.3@osu.edu; Allen Yuqing Yang, The Ohio State University, Ohio, USA, yang.5656@osu.edu; Chaoshun Zuo, The Ohio State University, Ohio, USA, zuo.118@osu.edu; Zhiqiang Lin, The Ohio State University, Ohio, USA, lin.3021@osu.edu.

2476-1249/2021/6-ART14.
https://doi.org/10.1145/3460081

million monthly-active mini-app users [6–8]. Today, the mini-app paradigm is not exclusively available in WeChat. "*Following WeChat's growth in China, companies like Facebook, Google, and Apple began allowing third-party apps and services to plug into their own messaging platforms*" [2].

However, even though the mini-app paradigm has been appeared for more than four years, it still remains a myth at an aggregated level, and many research questions have not been answered. First and foremost, it would be quite important to investigate how 'mini' a mini-app really is. The mini-apps are given the name 'mini' likely because they are executed inside the *host* such as WeChat, and they are expected to be smaller when compared to native apps. Additionally, the host platform such as WeChat even limits its size to be less than 12 MB [9], compared with other native apps that require tens or hundreds MB of storage (*e.g.*, the maximum app size of an Android app is up to 4GB [10]). However, there is no study to quantify the resource usage of the mini-apps and justify the benefits of using mini-apps other than the obvious reason of increasing user's stickiness from the platform vendor's perspective.

Second, there is a need to characterize the mini-app ecosystem for developers and researchers to better understand the mini-app paradigm. There have been a large body of measurement studies focusing on traditional native apps including Android apps (*e.g.*, [11–14]) and iOS apps (*e.g.*, [15, 16]). These efforts have provided various insights of the apps such as the current status and trend of the apps. For example, PlayDrone [11] provided a measurement study of Google Play apps at scale and revealed a number of trends such as the types of apps that likely receive higher popularity and quality. Similar to traditional native apps, WeChat mini-apps are also experiencing a drastic consumer boom. As such, a comprehensive study of the status and trends of mini-apps, particularly on the category of the apps, the functionality they provided, and the current programming practices, can benefit the researchers, developers, and the ecosystem as a whole.

Motivated by the above two needs, in this paper, we present MiniCrawler, the first large-scale mini-app crawler that is able to automatically download, unpack, and index the mini-apps from WeChat. Multiple challenges have to be addressed when developing MiniCrawler. First, WeChat is a closed ecosystem. Little information has been disclosed, other than the public available mini-app development manuals and SDKs [17, 18]. We must therefore reverse-engineer WeChat to understand its interfaces and workflows. Second, WeChat is a giant software (its latest Android version has 163 MB) and it heavily uses native code and obfuscation to thwart the reverse engineering attempts. Third, WeChat also collects user's behavior and running environments to build a risk-model for each account [19] and block this account if certain behaviors have exceeded some threshold (*e.g.*, a rate limit). Finally, unlike Google Play, there is no public store, nor public interface to allow end users to query the mini-apps. Users have to do so inside WeChat, but there is also no visible app category and unfortunately users have to enter the corresponding keywords to query the available mini-apps.

We have addressed these challenges when developing our MiniCrawler. In particular, to build our crawler, we first reverse engineered WeChat binary code and identified a search interface (*i.e.*, a server API invoked by WeChat) that can reveal the mini-app IDs (*i.e.*, the unique identifiers of the mini-apps) by exhaustively feeding different keywords to this API. To automatically build our keywords list, we leverage natural language processing (NLP) techniques such as word splitting [20]. With the retrieved mini-app IDs, we developed an Xposed [21] module to continuously trigger the interface in WeChat that downloads a mini-app when provided its ID. With the crawled mini-apps (each in a packed binary file named `wxapkg`), we then developed an unpacking tool to automatically unpack the `wxapkgs`, resulting a set of files including resource declaration files, UI, and JavaScript code files. The mini-app logic is encoded in the JavaScript code. With these unpacked files, we can then perform various measurement analysis, as we have demonstrated in this paper.

Contribution. In short, we make the following contributions in this paper:

- **Practical Tool.** We design and implement MiniCrawler, the first open source tool that is able to crawl WeChat mini-apps automatically and at scale. Its source code has been made public available at https://github.com/OSUSecLab/MiniCrawler.
- **Large Scale Evaluation.** With MiniCrawler, we have crawled over 1,333,308 mini-apps in a 5-month period since August 2020. These wxapkgs have consumed 2.14 TB storage space. After the unpacking, it yields a total of 193,651,923 files, and 21,127,129,690 lines of JavaScript code.
- **Empirical Results.** With these collected mini-apps, we have measured both their meta-data and their JavaScript code. Our experimental results show that a mini-app is on average 1.61 MB in size (11x smaller when compared to native apps) and has hundreds of API invocations, most of the mini-apps are obfuscated, the largest set of mini-apps is in the education category, and the category with the highest average ratings is the Hair Salon.

2 BACKGROUND

2.1 The Mini-app Architecture

Under the mini-app paradigm, the mobile app that mini-apps run atop is called the *host app* (*e.g.*, WeChat in this paper) or *host* for short. As shown in Figure 1, to provide a native-like user experience, a mini-app relies on a (*i*) front-end, which runs atop the *host* for user interaction and handling of system resources including peripheral devices, and an optional (*ii*) back-end, which provides cloud services such as storage and other server side computations.

Front-end. The front-end of a mini-app executes atop the *host* and interacts with the end users. As shown in Figure 1, it is a layered architecture in which the *host* provides in-app APIs to the mini-apps. These APIs can be grouped into 42 categories [22], and they can be used to access the system resources (*e.g.*, Bluetooth, GPS, Microphone, and Camera) managed by the OS (*e.g.*, Android or iOS), operate with the data provided by WeChat (*e.g.*, obtaining the user's nickname and billing address from WeChat), communicate with other mini-apps and APIs, render the UIs, so on and so forth. To prevent a mini-app from abusing system resources, there is an in-app permission check by the *host* to determine whether the mini-app has the corresponding permissions (*e.g.*, granted by the end-users) when accessing a resource.

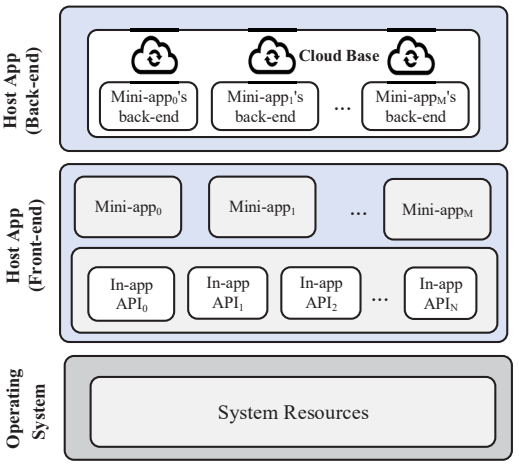

Fig. 1. A typical mini-app architecture

Similar to an Android APK, a mini-app binary is also packed in a compressed file in the format of wxapkg [23]. As illustrated in Figure 2, a wxapgk typically contains: (*i*) a configuration JSON file named app.json, which describes the general features of the mini-app (*e.g.*, the permissions the mini-app requires); (*ii*) one or multiple folders that contain the resource files (*e.g.*, images and audios); (*iii*) one or multiple folders that contain the files that describe the UIs (called pages by *Tencent*) of the mini-app. Particularly, each page is composed of four files including (1) a WXML file [24], which is written by a markup language WXML that defined by *Tencent* for UI design such as buttons or input boxes; (2) a JavaScript file,

specifying the logic to be executed when a user interacts with the corresponding UIs; (3) a cascading style sheet file encode with WXSS format, specifying how to display the UI elements; and (4) finally a style configuration file defined by *Tencent* [25], specifying the window behaviours of the UI (*e.g.*, UI orientation).

Back-end. A mini-app can offer various services to end-users, and there could be a mini-app specific server running in a back-end cloud for such services. To this end, WeChat provides back-end cloud services to mini-app developers, who just need to customize and configure the back-end and then directly use APIs in the mini-app to interact with the cloud back-end (*e.g.*, accessing a database). To prevent any misuse of the WeChat user's data, WeChat will vet all of the outgoing traffic of a mini-app and only allow them to communicate with the WeChat's back-end first, from which it can further connect to any other 3rd-party back-ends [26].

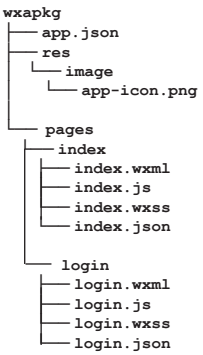

Fig. 2. The structure of a typical wxapkg.

2.2 The Mini-app Market

Unlike traditional mobile app market store such as Google Play, which usually has a web portal to show various apps in different categories for end-users to select the apps of interests based on the descriptions, reviews, and ratings, WeChat does not provide such a web portal. Instead, an end-user can only retrieve the list of the mini-apps by entering the keywords in a built-in searching interface within the WeChat app. At a high level, this WeChat interface will return a list of the mini-apps that match the keywords, and the user cannot see information other than names and the brief description of the retrieved mini-app. To further view other information about a specific mini-app, the user has to launch the app and check its property page, which contains more information about the mini-app such as its ratings (if enough users have rated this mini-app), the description, last updated time, and the developer account ID.

In addition to searching through the keywords, an end-user can also know the specific mini-apps by (1) QR code scanning in which the QR code can be placed online (*e.g.*, in web pages) or offline (*e.g.*, printed and displayed on shop's windows or desks), (2) URL sharing in which the URL of the mini-app can be shared with friends, and (3) Mini-app redirection with which a mini-app can promote other mini-apps by creating a button and allowing users to click and fetch.

3 CHALLENGES AND INSIGHTS

The goal of this work is to have an aggregated view of the mini-apps in the WeChat platform. To this end, we must design a crawler to obtain and index the mini-apps at a large scale. Unfortunately, this is non-trivial, particularly because WeChat is a closed ecosystem and there are many roadblocks for mini-app crawling. In this section, we describe the challenges encountered when developing a scalable crawler (§3.1) and our insights of how to address them (§3.2).

3.1 Challenges

There are multiple ways to obtain mini-apps. For instance, one could manually enter the keywords in the mini-app searching interface inside WeChat to download them, or scan the QR code, or use the shared URLs of the mini-app to obtain them, as mentioned in §2.2. However, none of these approaches is able to scale. Also, unlike Google Play store, there is no web portal of the mini-apps. As

such, state-of-the-art web-crawler will not work for mini-apps. For example, one cannot sniff the traffic between WeChat and its server with networking traffic analysis tools such as Burp Suite [27] to build the crawler, since WeChat has protected the communication between its client and the server using private and encrypted protocols. Therefore, we must reverse engineer the WeChat apps, identify the mini-app searching interface from the WeChat binary, automate the keywords generation, and obtain the mini-apps at scale. More specifically, we have to address the following challenges.

C1: Identifying the mini-app searching and downloading interface inside WeChat code. Since the only interface available for searching the mini-apps is inside the WeChat app, we must reverse engineer the WeChat code to identify it. However, WeChat is a giant software with over one hundred MB, and is composed of both native code and Java Bytecode. For instance, its recent Android version has over 82,256 files including 93 shared object (so) files, and 56,492 Java classes (after the decompilation by using tool Jadx [28]). Meanwhile, it heavily uses obfuscations (such as symbol renaming, anti-debugging, and anti-emulation) to thwart the reverse engineering attempts. How to pinpoint the code of our interest, invoke them with the right input, and parse the output returned from the server is the first challenge we have to address.

C2: Creating the keywords list with broader coverage. Even if we can identify the searching interface inside the WeChat app to collect the mini-app information based on the keyword, we have to define the list of the words, since it directly impacts how many mini-apps we can reach. An intuitive approach is to try to use all possible single words (*e.g.*, Chinese characters, English letters and numbers) for the searching. However, a single word search can only return at most 500 mini-apps based on their popularity and the closeness to the keyword, and only popular mini-apps will be collected. As such, we have to not only use single words, but also use phrases and their combinations for the search in order to collect as many mini-apps as possible. However, there are too many of phrases in Chinese, and it is not practical to try all of them.

C3. Circumventing the restrictions imposed by WeChat. While at client side WeChat already introduced various roadblocks to thwart the reverse engineering (*e.g.*, we cannot run WeChat in emulators), it also builds a risk-model for each user at the server side based on the collected user behaviors. For instance, if a user is aggressively using certain dynamic analysis framework such as hooking [29], her account could be banned from temporally restricted login to permanent account blocking, according to the proprietary risk-model for each user maintained by the WeChat server [30]. Once the account is blocked, it costs $5–$30 to obtain another one, since WeChat requires a unique cellphone number to be associated with the account. The efforts of crawling mini-apps can be daunting and costly when accounts are banned.

3.2 Insights

S1: Using both static and dynamic binary analysis. To address C1, we have to reverse engineer the WeChat app. There are two basic approaches in the reverse engineering: dynamic analysis and static analysis, where dynamic analysis can pinpoint the code of interests, particularly for large software such as WeChat, and static analysis can expand the code coverage missed by dynamic analysis. These two approaches are often used together.

In particular, while using WeChat to search the mini-apps, we notice that there is a search button. Then, there must exist a function to handle the click event, and this event handler (if we can locate it) will send the input request to the WeChat server to return the list of the available mini-apps matching the keywords searched. That is, there must be a server API and we just have to prepare for the right input to trigger this API. This preparation is performed by the event handler. Therefore, we first used dynamic analysis to locate this event handler. In Android, the event handler

is attached to the GUI element (also called `View` in Android) as a class field. However, based on our experiment, we found that the `View` is a `WebView` and the button is a HTML button which makes function call to Java (native code) through Javascript (web code). To locate the implementation of the searching logic in Java, we hooked and traced the `JavascriptInterface`, which is the only function that the Javascript code can use to call Java code.

Having identified function `JavascriptInterface`, we then started our static analysis by first decompiling the WeChat code with tool `Jadx` [28]. However, the code is heavily obfuscated. Fortunately, we found WeChat contains rich `logging` code, and the strings (*e.g.*, "`Constructors: keyword=%s`") used in the logging routine provide huge clues on the functionality of the code. With these logging strings, we quickly located the code that generate the HTTP(S) request to the WeChat server. Through further reverse engineering of the corresponding parameters of the HTTP(S) request including the URLs, HTTP headers, and tokens (by dynamically intercepting the outgoing networking APIs), we were then able to develop an independent python program to generate the searching request without running the WeChat app any more.

While we have reverse engineered the mini-app searching API, we still were not able to download the mini-apps and we had to locate the downloading API from the WeChat code. We followed similar practice of dynamic analysis first and then static analysis next to locate the code. However, unlike the mini-app searching API which we can use an independent program to directly query the WeChat server by providing the corresponding parameters, the downloading API is much more complicated. More specifically, it involves many asynchronous calls, and even worse we cannot confirm whether it uses HTTP(S) or not. However, we can still download the mini-apps by running the real WeChat app, as long as we are able to find out the interface that takes mini-app identifier as input for the downloading. As such, we built a call graph from the handler function of download event and dynamically hooked all of the callees. By printing out all of their parameters, we successfully identified the function that takes a mini-app ID as input. Then, at run-time, we dynamically update this mini-app ID in real WeChat app, and we can thus download the corresponding mini-apps.

S2: Using NLP to collect and generate high quality keywords. To address C2, we need to optimize the keywords to maximize the coverage of the mini-apps and meanwhile minimize the amount of requests to the WeChat server. To this end, we use both the breadth-first search (BFS) and depth-first search (DFS) algorithms. In our BFS, we use the top 1,000 most commonly used Chinese characters as the keywords to search for the seed mini-apps, which will be further used for our DFS search. In our DFS, we expand the keywords from the collected names and descriptions, based on the insight that mini-apps in the same category tend to have similar descriptions.

While it is quite straightforward to perform BFS with the provided most commonly used words, we need NLP analysis for our DFS search. More specifically, in the DFS search, we have to perform word splitting (using NLP) on the seed mini-apps' name and description, and then use the split words as keywords. We continue such a process for all newly searched mini-apps. The key observation for this approach is that mini-apps' descriptions tend to contain high quality words, since WeChat limits the length of description to be no more than 120 words and it requires the developers to summarize the usage of the mini-app concisely and accurately. Therefore, the mini-apps providing similar functionalities are very likely to have common words in their description, and searching those keywords can allow us to reach more apps.

We believe that the mini-apps collected by our algorithm are representative for two reasons: (*i*) Our algorithm has used the top 1,000 most common Chinese characters as the initial inputs, and the searched results are likely to be the commonly used mini-apps. As discussed earlier, the only interface for an end-user to fetch a mini-app is to use the built-in interface, which requires the user to enter a Chinese character or word into the search box. A mini-app that uses a rarely-used

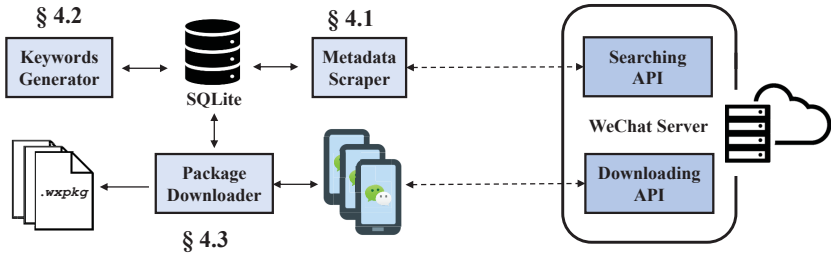

Fig. 3. The architecture of MiniCrawler.

Chinese character or word will make the mini-app less likely to be exposed to the user. (*ii*) Our algorithm achieves a large coverage: based on the current report [6], WeChat now have more than one million mini-apps available and our algorithm have collected 1,395,456 meta-data.

S3: Using premium accounts to avoid being blocked. WeChat has a sophisticated risk model to assess each user's account and block it when necessary, in order to fight for the "Internet Water Army" [31] or the fake account attack. This is because a WeChat account can be used to abuse coupon or click fraud [30]. Therefore, there is a risk score for each user's account. If an account is just registered, has few or no friends, barely talks, and has no activities other than just downloading mini-apps, it will very likely be banned, as confirmed in our study.

Therefore, to circumvent Tencent's account blocking, we notice that we can use premium accounts (*i.e.*, accounts with low risk), and they never get blocked during our experiment. To be more specific, the premium accounts are referred to as the accounts that (*i*) were registered months or even years ago, (*ii*) have many friends (not just one or two friends), (*iii*) have active activities such as posting WeChat moments and communicating with friends, and (*iv*) have bound payment card (debit or credit card) with the account to use WeChat Pay. By knowing the features of these premium accounts, we can intentionally use these type of accounts, and additionally performing more specific real-user type of activities to make them more real-user alike. For example, for a specific account, although we cannot manipulate the registration date of the account, we can add more friends and use this account more often to make the account look like a normal account.

4 DETAILED DESIGN OF MINICRAWLER

In this section, we present the detailed design of our MiniCrawler. At a high level, MiniCrawler consists of three key components as shown in Figure 3: (1) Metadata Scraper (§4.1), which takes a list of keywords as input and produces the metadata of the mini-apps by interacting with the mini-app searching API, (2) Keyword Generator (§4.2), which performs NLP analysis on mini-app name and description to generate high quality keywords, and (3) Package Downloader (§4.3), which takes a list of mini-app identifiers (IDs) as input, and downloads them by executing the corresponding WeChat code. In the following, we present the detailed design of these components.

4.1 Metadata Scraper

Recall in S1 (§3.2), we have discovered the mini-app searching API with the concrete values of the required information (*e.g.*, URL, HTML headers, and body), and therefore we implement a standalone Python script to interact with this API. Our Python script will (1) read the keywords from a SQLite database, (2) search the mini-apps from the server, and (3) store the returned result (*i.e.*, the mini-app metadata) to the database.

- **Reading the keywords.** We use a SQLite database to store the keywords and also the mini-app meta-data. While we can randomly choose a keyword from the database for the search, we decide to start from the most popular words (*i.e.*, words with higher frequency in our collected metadata) first since with those words our script is likely able to find more mini-apps. As such, every time, it will query the top 10 most popular keywords which has not been selected yet.
- **Searching the mini-apps.** With a provided keyword, the script will interact with the mini-app searching API to retrieve the matched mini-apps' metadata. More specifically, it will construct a HTTP(S) request message based on the keyword, and we use Python library `requests` to send it to the WeChat server.
- **Storing the results.** The server will respond in JSON format which contains a list of mini-app metadata. An example of such metadata is shown in Figure 4. We can see that, there is way more information about the mini-app in the response message than a user can see in the WeChat GUI, such as `appid`, the identifier of a mini-app; `nickName`, the nick name of the mini-app; `labels`, the category of the app; `evaluate`, which means the evaluation score (*i.e.*, the rating) of the the mini-app; and so on.

Note that, in the HTTPS request message, there is field `cookie`, which is a vital field in the request message, and the server will refuse to return data if it is invalid. We obtained an instance through our dynamic analysis as described in §3.2. Surprisingly, we found that the expiration time for this cookie is very long (more than 4 weeks).


```
{
  "appid": "wx5054764a3fdfb3b5",
  "appuin": 3508294916,
  "description": "微骰子，喝酒摇骰子！",
  "docID": "Aa558edb42192aef2cc57faee54089bf23eec8c30",
  "extra_json": {
    "title": "微骰子A",
    "labels": [ "休闲娱乐" ],
    "evaluate": "4.5分",
    ...
  },
  "iconUrl": "https://wx.qlogo.cn/mmhead/...",
  "jump_path": "/Search/specific/index.html?...",
  "nickName": "微骰子APP",
  "path": "",
  ...
}
```


Fig. 4. An example of the metadata in our response.

4.2 Keywords Generator

This component is responsible for generating high quality keywords for the searching, and it directly impacts how many mini-apps we can reach. As described in S2 (§3.2), we will use both BFS and DFS searching algorithms to search for the apps (BFS first and then DFS), and correspondingly we have to generate the keywords for both of them.

- **Generating the keywords for BFS.** Initially, there is no mini-app metadata. To start the searching, we need to provide a set of predefined keywords as the seeds. We decide to use the most frequently used 1,000 Chinese characters that we manually downloaded from website `thn21` [32] as the keywords. With these single character keywords, our BFS algorithm can cover a large set of mini-apps with different categories.
- **Generating the keywords for DFS.** After finishing BFS, we have obtained rich metadata about the mini-apps such as their names and descriptions, from which we further generate more keywords and use them to download more apps. More specifically, our keyword generator will load the names and descriptions of newly searched mini-apps from the database, and apply word splitting [20] (segmentation) on them. Since most of the mini-apps are in Chinese, we use the NLP engine `jieba` [33], which is specifically designed to split Chinese words. Similar to our BFS where we select the most frequently used words for the searching, we also count the word frequencies of the split keyword, and maintain a field to track its frequency, whenever a new app is added.

4.3 Package Downloader

As mentioned in S1 (§3.2), we have to reuse and dynamically trigger the code inside WeChat to download the min-apps by providing the corresponding mini-app IDs obtained in the meta-data. At a high level, our Package Downloader consists of (1) another Python script that runs on our own server to manage the downloading tasks by controlling the smartphones through the Android Debug Bridge (ADB) and (2) an Xposed [34] plugin that runs on the phones to download mini-apps, and the phones are connected to our server with USB cables.

- **The downloading Python script.** There are three procedures managed by this script. First, it will load a mini-app ID from the meta-data for a to-be-downloaded mini-app from the SQLite database once it detects an idle phone. Second, it will push the list of the mini-app IDs to a specific location `/data/data/com.tencent.mm/MicroMsg/<account_ID>/appbrand/pkg/` (the `<account_ID>` is a hex string which can be easily observed in the folder) in the idle phone to trigger the Xposed plugin for the downloading. Third, once the downloading process has finished, it will pull the mini-app packages from the phone to the computer storage through ADB. Note that, the mini-app packages are in the private folder of WeChat, and therefore we need the root privilege of the phone to pull them.
- **The Xposed plugin.** The plugin is responsible for downloading the mini-apps based on the provided IDs. As an Xposed plugin targeting WeChat, it will be loaded to WeChat process once WeChat starts. Since then, the plugin starts to operate. In particular, to make a function call to the downloading function (*i.e.*, com.tencent.mm.plugin.appbrand.jsapi.l.invokeHandler), the plugin needs to get the reference of the class instance to which the downloading function belongs. Therefore, our plugin hooks the class constructor functions which will be called when initializing a class, and then acquires the instance reference once a constructor is called. Furthermore, by making a function call to the downloading function with a mini-app ID, the plugin will trigger the code inside WeChat to download the corresponding package to a specific folder on the phone. The plugin will download the mini-apps one by one with the provided IDs stored in the downloading queue. The phone becomes idle if its downloading queue is empty.

5 THE PERFORMANCE OF MINICRAWLER

We have implemented MiniCrawler. In this section, we report the performance overhead of our MiniCrawler by running with one server and 5 Android phones. In particular, our server runs Ubuntu 18.04 Linux operating systems, and has an Intel i7-7700 CPU, 32 GB of memory, and 42 TB storage. The Android phones are Google Pixel XL with Android 7.1 and having WeChat version 7.0.3 installed. We launched our Metadata Scraper in August 2020 and it ran for about a month to collect the metadata, and our Package Downloader executed about four months for the downloading (note that we did not run the downloader all the time without interruption due to the fact that the phones were sometimes used in some other non-related experiments).

Metadata Scraper. As a standalone Python script, it directly communicates with the WeChat server using HTTPS protocol. Since HTTPS is a stateless protocol, we can use multi-threading to accelerate our searching process, in a way similar to PlayDrone [11] (which sent 100 requests per second). However, we do not want to impose too much load on WeChat servers, and therefore, we decide to just send at most 2 requests per second. For each request, it took 1,232 *ms* on average to send the request and receive the response. Note that our server is located in North America, and the WeChat server is located in Hong Kong according to the geolocation of its IP address we connected. To gain an understanding of how it performs daily, we present a real-time request

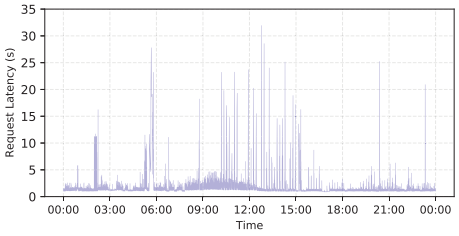

Fig. 5. Network latency for a single day.

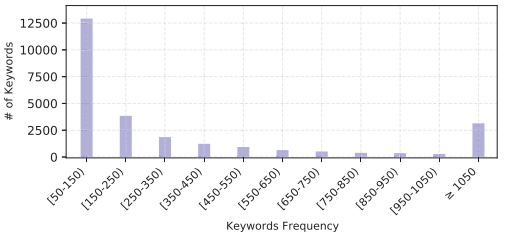

Fig. 6. Distribution of the keywords and their frequency.

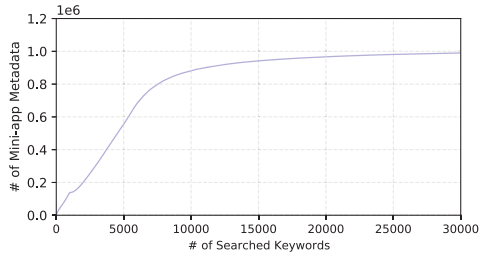

Fig. 7. Trend of # of discovered mini-apps with the increase of # of searched keywords.

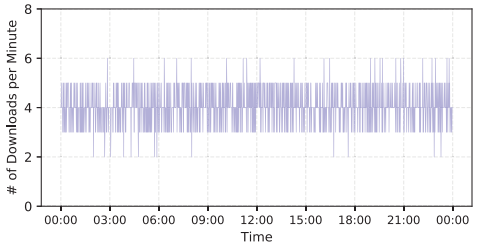

Fig. 8. Real-time package downloading speed (# of packages downloaded) for a single day.

and response latency diagram in a 24-hour time window in Figure 5. We can see that the latency increased between 8AM and 12PM (our time) which is the evening of Hong Kong, when the network is overloaded. In total, our Metadata Scraper searched 218,182 keywords and discovered 1,395,456 mini-app metadata in a one-month period. To store the collected metadata, it took the SQLite database 2.2 GB storage space.

Keywords Generator. As described in §4.2, we use both BFS and DFS algorithms to search the mini-app metadata. We found 138,168 mini-app metadata by just using the 1,000 most commonly used words in our BFS. For the DFS, our generator produced 217,182 new keywords from the mini-app metadata. More specifically, a mini-app name can be split into 4.37 words on average, which takes our NLP engine 21.56 microseconds to process. And a mini-app description can be split into 29.34 words on average, which takes our NLP engine 65.47 microseconds to process. Based on the frequency of the keywords, we separate them into two types: words with low frequency (we set its threshold to be 50) and words with high frequency. We found that, 193,692 keywords (over 88%) are with low frequency. With our manual analysis, we found that those words are often related to mini-app names which may contain their brand (*e.g.*, restaurant name, supermarket name). For the high frequency words, their distribution frequencies are presented in Figure 6, and we can see that there are around 3,000 very popular keywords, which have more than 1,000 occurrences. To understand the contribution of the keywords, we present the relationship between the number of searched keywords and the number of searched mini-apps for the top 30,000 keywords in Figure 7. We can see that, with more keywords searched, the increase of the total number of mini-apps becomes slow, because many mini-apps have been already discovered by other keywords.

Package Downloader. Unlike our standalone Metadata Scraper, our Package Downloader has to be executed within the WeChat app. To speed up our downloading process, we run 5 Android

phones, and each of them can download around 6,000 mini-app packages daily. For a single mini-app, it took a phone 14.79 seconds on average to get a mini-app downloaded. Similar to our Metadata Scraper, we present a daily downloading behavior for a single phone in Figure 8. We can see that, our downloader can even download up to 6 mini apps per minute depending on the sizes of the mini-apps.

While our Metadata Scraper has crawled 1,395,456 mini-app metadata, we only successfully downloaded 1,333,308 mini-app packages, which cost 2.14 TB storage (each package occupied 1.61 MB on average). There are three reasons why we did not successully download all of these mini-apps based on their meta-data, according to the error messages thrown by WeChat. (*i*) The providers of the mini-apps have violated the rules specified by WeChat, and Tencent has banned their mini-apps. For example, WeChat will ban a mini-app if it has lewd content, gory violence, terrorism, and so on [35]. (*ii*) Some apps heavily rely on their back-ends to provide services and store necessary resources. However, their back-ends may get out of services due to the insufficient balance. When insufficient balance occurs, the services will become unavailable. (*iii*) The providers have withdrawn their apps or closed their developer accounts.

6 MEASUREMENT BASED ON CRAWLED MINI-APP PACKAGES

Having collected over one million mini-app packages and their meta-data, we are able to perform various large scale analysis. In this section, we present our measurement result based on the mini-app packages, and we leave the result of our meta-data measurement in next section (§7). While we are able to answer many measurement questions at an aggregate level given the rich information available in the mini-app packages, we particularly focus on the following questions:

(1) **Storage consumption** (§6.1), which characterizes *how minimal* a mini-app really is;
(2) **Package complexity** (§6.2), which measures the amount of UI interfaces (*i.e.*, pages) and resource files, lines of code, number of called functions (*i.e.*, callees), and code complexity of a mini-app;
(3) **Mini-app API usage** (§6.3), which measures how often WeChat APIs are invoked and the most popular used APIs;
(4) **Code obfuscation analysis** (§6.4), which measures how mini-apps protect their source code;
(5) **Library usage** (§6.5), which measures how often a library is used and the popular libraries among the mini-apps.

6.1 Storage Consumption

Although the mini-app paradigm is believed to save more resources compared with traditional native apps, it remains unknown how much resource can really be saved (or "how minimal a `wxapkg` really is"). To answer this question, we group the mini-apps based on their package size (using round to hundredth MB precision), and then count the number of mini-apps for each size category. Figure 9 shows the cumulative distribution function (CDF) of the sizes of mini-apps. We can observe that the sizes of the mini-apps are ranging from 0.04 MB to 11.73 MB (we have not observed any mini-apps with size larger than 11.73 MB). The mean size of a mini-app is 1.61 MB MB, and most of the mini-apps are less than 4.0 MB (in fact mini-apps with less than 4 MB account for 99.74% of all our collected mini-apps).

In comparison, we also randomly selected 1,333,308 native apps from the dataset that contained more than two million mobile apps we have collected from Google Play in 2019 in our prior work BleScope [36]. We compared the size of these native apps with the mini-apps. This result is presented in Figure 10. We can see that the average size of a native app is 18.01 MB, which is 11X larger that of a mini-app (*i.e.*, 1.61 MB). Meanwhile, the measured mini-apps have a low

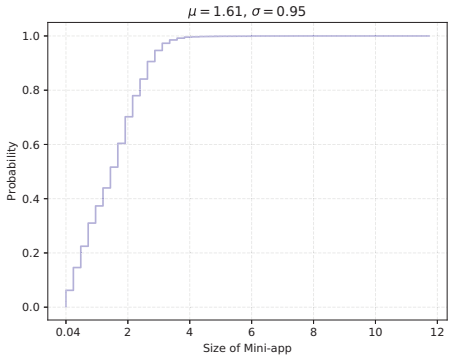

Fig. 9. Size distribution of the mini-apps.

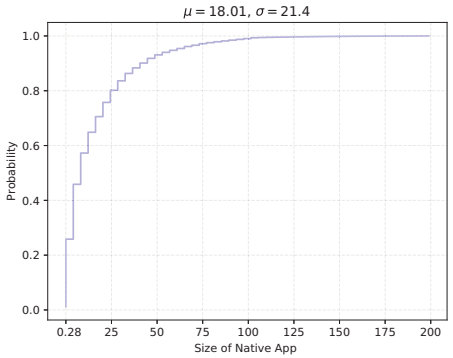

Fig. 10. Size distribution of native apps.

standard deviation ($\sigma = 0.95$), indicating that the sizes of mini-apps tend to be close to the mean size, whereas the native apps have a higher standard deviation ($\sigma = 21.4$).

6.2 Package Complexity

Next, we zoom in each `wxapkg` and understand its complexity. Note that a package complexity can be characterized by many dimensions. In this paper, we particularly focus on the dimensions from the amount of (1) UI interfaces (*i.e.*, number of pages), (2) resource files, (3) lines of codes (reflecting the amount of code to be executed), and (4) code complexity (indicating the complexity of the mini-app logic). Finally, we also made a (5) comparison to compare these dimensions between the corresponding native apps. We believe these metrics can provide an approximation of how a mini-app looks like.

(1) Number of Pages. A mini-app often has its user interface (UI) in a graphic window to interact with end-users, and each window is called a page in mini-apps. We measured the numbers of pages contained in a mini-app by first grouping the mini-apps based on their total number of pages, and then counting the number of mini-apps for each grouped category. Figure 11 shows the cumulative distribution function (CDF) of the total number of pages contained in the mini-apps. On average, each mini-app contains 16.1 of pages (there is one app that has the maximum number of pages 433), and 99.1% of the mini-apps have less than 80 pages and 79.8% mini-apps have less than 20 pages.

(2) Number of Resource Files. A mini-app usually has various resource files such as images, audios, and videos. Similar to how we measure the number of pages, we also measure the total number of resource files a mini-app can have, and we show this result in Figure 12. We can see that on average a mini-app contains 145.24 resource files. We also find that 99.7% of them have less than 500 files, and 78.6% have less than 250 files. There are only 0.12% mini-apps that contain more than 1,000 files, and most of them are games. An example of such a game is the "`red and blue`", which is a Pokemon game containing 1,226 resource files (1,168 of them are images of the Pokemon).

(3) Number of Lines of Code (LoC). The LoC could reflect the amount of code that will (likely) be executed by a mini-app. Similar to how we measure the number of resource files, we next measure the LoC of the JavaScript files, which are fundamentally the program code executed by mini-apps. We report this result in Figure 13. We can see that on average a mini-app contains 15,845 LoC,

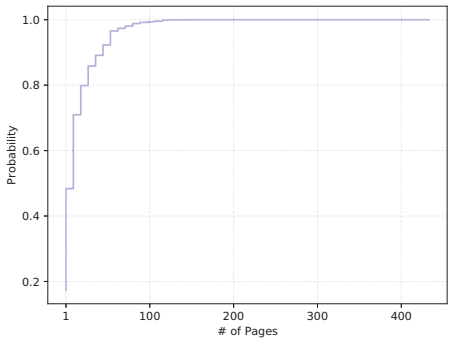

Fig. 11. Distributions of the mini-apps based on the number of pages.

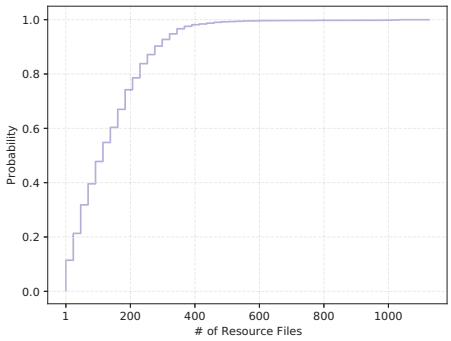

Fig. 12. Distributions of the mini-apps based on the number of resource files.

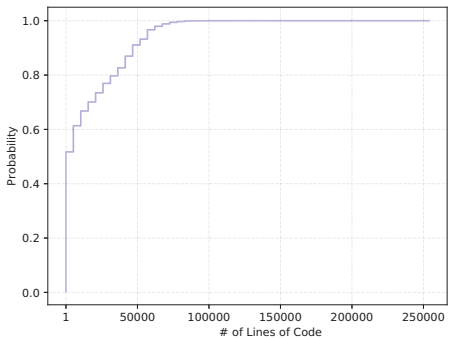

Fig. 13. Distributions of the mini-apps based on the number of lines of code.

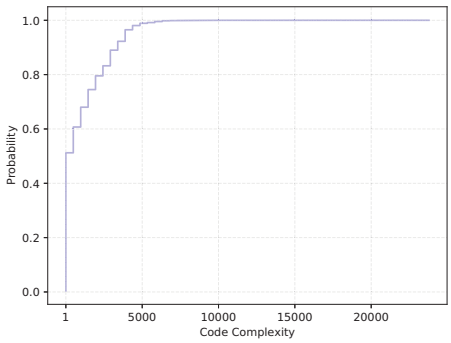

Fig. 14. Distributions of the mini-apps based on the number of code complexity.

which is less than that of a regular iPhone app which has 50,000 LoC [37]. We also find that 51.8% of them have less than 5,000 LoC, and 31.4% has less than 1,000 LoC.

(4) Code Complexity. Next, we also measure the complexity of the code to reflect how complicated a mini-app could be. There are a number of algorithms to measure the code complexity, and we particularly use the McCabe's cyclomatic complexity [38], which counts the number of linearly independent paths a program can have, to measure the code complexity of the JavaScript code for each mini-app, and then we report the aggregated result as other experiments in Figure 14. We can see that 99.1% of the mini-apps with the code complexity of less than 5,000 and 83.2% with code complexity less than 2,000, and the average code complexity of a mini-app is 1,191.25.

(5) Comparison to Native Apps. While we have provided an aggregated view on the storage consumption between mini-apps and native apps in §6.1, this comparison is not one-to-one between them (e.g., Amazon mini-app vs. Amazon mobile app). Therefore, to provide an in-depth and one-to-one comparison between mini-apps and native apps, we selected 20 popular mini-apps and their corresponding native apps to compare them using the aforementioned dimensions. Among our selected apps, 9 of them are also selected by Lu *et al.* [6] in their mini-app study, and the rest

App Name	Mini-app						Native App					
	AppID	Size	# Pages	# LoC	CC	# Files	Package Name	Size	# Activity	# LoC	CC	# Files
Airbnb	wxc2ada47bce176219	3.28	31	20,241	1,252	124	cm.aptoide.pt	45.50	37	1,476,663	150,697	21,903
Amazon	wx3265fbb010daacc5	4.00	13	54,551	3,807	428	com.amazon.mShop.android.shopping	54.60	145	3,186,264	293,736	29,906
BMW	wxe22ecdc7380717d8	3.04	355	49,289	2,976	387	de.bmw.connected.na	51.72	206	1,516,377	137,923	17,995
Burger King	wx599036e8b7735aea	2.91	15	25,587	1,477	169	com.emn8.mobilem8.nativeapp.bk	12.63	6	519,291	56,739	5,461
Calvin Klein	wx9c72ac56248ced62	3.90	45	41,337	2,968	469	com.calvinKlein.android	28.71	67	1,272,343	135,988	13,919
Channel	wxaa2a91a9b866d2f7	0.98	23	9,089	922	207	com.chanel.fashion.public	109.32	15	1,362,957	141,921	12,671
DHL	wxef742bdd4c132e3e	2.33	34	36,318	5,394	382	com.dhl.exp.dhlmobile	24.51	7	300,858	35,805	3,914
FedEx	wx3542aa451046fb76	2.71	1	24,702	1,713	118	com.fedex.ida.android	39.91	88	1,272,637	139,014	15,857
Gucci	wxd701b7c0f864a732	2.19	19	5,857	563	270	com.gucci.gucciapp	98.71	15	1,038,491	122,975	10,888
Grammarly	wx27ed163b52b83974	0.43	9	6,706	1,274	113	cm.aptoide.pt	99.31	35	439,269	55,118	4,836
HP	wx6fde234a26a0b79c	2.07	116	24,493	3,804	850	com.hp.android.printservice	34.42	37	834,933	99,030	8,091
H&M	wxe538b3c2a404630b	3.68	71	39,052	2,596	346	com.hm.oneteam	78.80	45	1,308,095	135,554	14,997
Lacoste	wx34cc116dec74e971	2.77	95	68,420	12,020	952	com.hp.wearable.lacoste	54.70	35	329,927	39,504	6,461
McDonald's	wxe7985a3d339996c5	2.43	24	33,688	2,051	253	my.com.mcdonalds.delivery	17.70	86	425,672	48,701	5,973
Nike	wx096c43d1829a7788	3.18	23	66,865	4,460	680	com.nike.omega	84.01	139	3,168,929	295,441	41,550
PizzaHut	wxd5f7974681bcdbce	2.74	20	13,516	1,414	225	com.yum.pizzahut	63.11	12	795,480	81,816	9,073
SEPHORA	wx0afe7aac882e6563	4.16	17	48,742	3,367	399	it.sephora.sephoraitaly	82.84	108	2,044,849	217,508	25,097
SHEIN	wxb2ea8bc777732d52	2.20	69	44,936	2,517	386	com.zzkko	62.53	206	2,520,067	254,065	25,646
ToryBurch	wx9b8b9a92c338e1b3	0.95	35	29,473	3,500	160	com.toryburch.connected	22.10	105	710,291	75,625	11,390
Walmart	wx09d1aae1bd6787f8	2.70	43	24,480	1,694	165	com.walmart.android	103.26	297	4,020,255	329,205	42,414
ZARA	wxd95a72c5f595b6a3	2.47	10	44,590	7,904	237	com.inditex.zara	47.52	132	1,735,822	188,049	21,247

Table 1. Comparison between mini-apps and native apps. Note that CC stands for code complexity; also native apps do not use "Page" to refer their UIs, and therefore, we count the number of activities (an activity can be associated to a graphic window of a native app).

of them are randomly selected. To avoid having any potential biases, we installed and manually checked both the mini-apps and the native apps to make sure they provide the same or similar functionalities. For example, Amazon have multiple native apps (e.g., Amazon Prime Video is used to watch movies while Amazon shopping is used to shop online), and we selected the shopping mini-app from Amazon for the comparison.

The detailed result of our one-to-one comparison is presented in Table 1. Note that we obtained Javascript code after unpacking a mini-app, and decompiled Java byte code after unpacking a native app. We then use these code to count the number of lines and measure their code complexity. Also, note that the McCabe's cyclomatic complexity [38] is programming language agnostic. This is because this algorithm counts the number of linearly independent paths that a program can have regardless of the programming languages. In a nutshell, we can observe from Table 1 that mini-apps not only save the storage resources but also tend to have fewer number of pages, fewer number of LoC, fewer number of files, and less complex code.

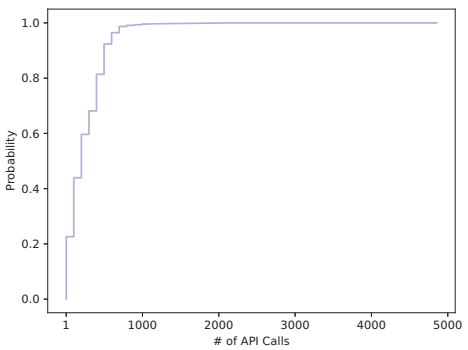

Fig. 15. Distributions of the mini-apps based on number of called APIs.

We then reverse engineered these apps and found three reasons of why the mini-apps are smaller and have less complex code. (*i*) To offer an instant and installation-free experience for end-users, mini-apps are supposed to only contain the core business logic of a full version the native app. Therefore, some functions are not available at the mini-apps. (*ii*) *Tencent* recommends that resources files such as images or videos should not be included as one part of the mini-app.

Instead, these resource files should be located at the remote server, and mini-apps could download them whenever needed. This practice could have reduced the size of the mini-app and also its number of files. (*iii*) Moreover, WeChat has many off-the-shelf built-in models for mini-apps to achieve sophisticated functionalities. For example, the mini-apps can easily implement registration and login interface using the APIs provided by WeChat with just one line of JavaScript code.

6.3 Mini-App API Usage

Next, we characterize the semantics (*i.e.*, the meanings) of the mini-apps. However, to truly capture a program's semantics without execution is challenging, we can only approximate its meaning through static analysis. One dimension that can be used to reflect the meaning of an app is to look at the invoked APIs. Therefore, in this measurement, we seek to analyze (1) how often an API (regardless of its name and its category) is called by a mini-app (*i.e.*, the frequency of an API call), (2) what are the most popular APIs (based on the names) and how often they are used by a mini-app, (3) what are the most popular API categories and similarly how often they are used, and (4) finally whether there are any privacy sensitive APIs and if so how they are being used.

(1) How Often a Mini-app Calls an API. Similar to how we measured the package complexity by using the number of lines of code of a mini-app, we also measured the number of API invocations for a mini-app. Figure 15 shows the CDF of the number of API invocations. We can see that mini-apps tend to invoke a lot of APIs, and on average, the API invocation is appeared 328.9 times based on the static analysis. We also find that 99.1% of the mini-apps has less than 1,000 API invocations.

(2) How Often a Mini-app Calls a Specific API. Having measured the API invocations in general regardless of their names and categories, next we measure the usage for each specific API. To this end, we first measure the number of apps that have invoked the specific APIs, and then measure how often this specific API is invoked by a particular app.

- **Number of mini-apps that invoke the specific APIs.** We first group the specific APIs based on their names (there are in total 580 APIs provided by WeChat SDK [22] at the time of this writing), and then count the number of mini-apps that have invoked each specific API based on its name. Since there are in total 580 APIs, we cannot show the result for all of them. Instead, we only present the top 20 APIs that have been invoked by the mini-apps, and this result is presented in Figure 16. We can see that the three top used APIs are all debugging related, and they are API console.log (1,326,992 out of 1,333,308, *i.e.* 99.98%, mini-apps invoked it), console.warn (used by 1,326,565 apps with 99.49%), and console.error (used by 1,326,305 apps with 99.47%). Also, 1,216,326 (91.26%) mini-apps have invoked API wx.request to send HTTPS requests to their back-ends, indicating that most mini-apps have their own back-ends. Other than the top 20 APIs that have been invoked by the mini-apps, we also find interestingly that `wx.requestPayment`, which has been used by 825,496 (61.91%) mini-apps to allow WeChat users to make mobile payments for online transactions.
- **Number of times for a specific API invoked by a mini-app.** One mini-app may call a specific API multiple times. For example, API `console.log` can be invoked whenever the developers want to log the status of their mini-apps. Therefore, we measure how often a specific API is invoked by a mini-app. In particular, we measure the average number of specific APIs called by a mini-app statically (without running the mini-apps). Figure 17 shows this result. Among all the APIs, we can notice that console.log is the most invoked API (on average, each mini-app calls it 33.15 times). The second most API is wx.nativegateTo (13.90 times). It is surprising to know that on average, each mini-app calls API wx.makePhoneCall

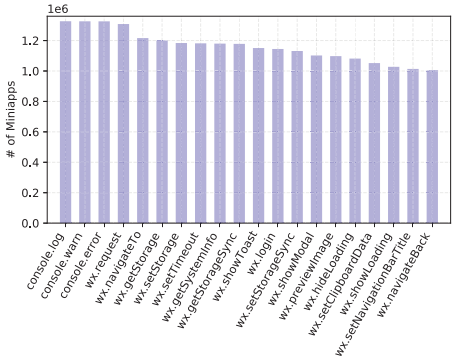

Fig. 16. Top 20 APIs that have been invoked by the mini-apps.

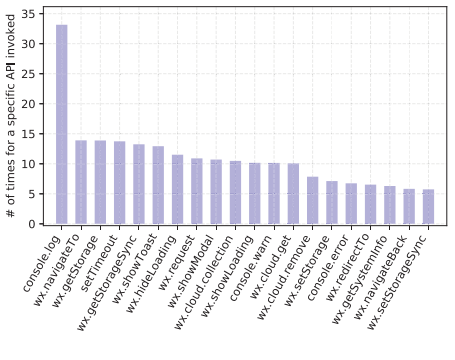

Fig. 17. Top 20 APIs based on the number of statically invoked times by a mini-app.

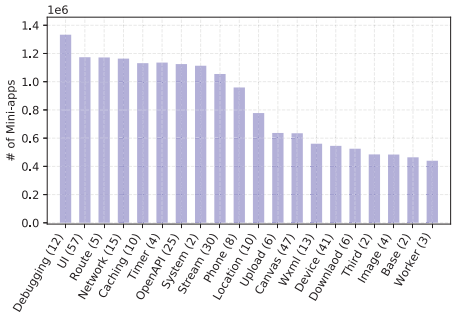

Fig. 18. The number of mini-apps that invoke APIs in specific categories.

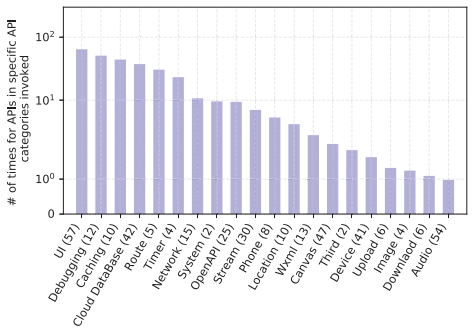

Fig. 19. The number of times for APIs in specific categories invoked by a mini-app.

3.40 times. We manually investigated some of these mini-apps, and found that mini-apps often use it for customer services (and each service could have different numbers).

(3) Specific API Category Measurement. Recall that there are 42 API categories as described in §2, next we measure the usage of specific API categories similar to how we measure the usage of specific APIs.

- **Number of mini-apps that invoke APIs in specific API categories.** Since we have already counted the number of mini-apps that invoke each specific API, we just need to group these APIs based on their categories. Figure 18 shows this measurement result. We can observe that the APIs in the debugging category is the most popular one (1,332,992 out of 1,333,308, *i.e.* 99.98% mini-apps invoked the APIs in this category). The second mostly used APIs belong to the UI category, and there are 1,173,328 (88.00%) mini-apps that call the API in this category to control the UI elements, including showing Toasts (*i.e.*, `wx.showToast`)), displaying animations (`wx.createAnimation`), and so on.

APIs	Type	Permission Name	Sensitive Data or Resource Accessed
wx.requestPayment	App specific data	N/A	Payment
wx.getUserInfo	App specific data	scope.userInfo	User profile
wx.getWeRunData	App specific data	scope.werun	Gait & Sports
wx.chooseAddress	App specific data	scope.address	Billing Address
wx.chooseInvoice	App specific data	scope.invoice	Invoice
wx.chooseInvoiceTitle	App specific data	scope.invoiceTitle	Invoice
wx.getLocation	System resources	scope.userLocation	Location
wx.chooseLocation	System resources	scope.userLocation	Location
wx.openLocation	System resources	scope.userLocation	Location
wx.startLocationUpdateBackground	System resources	scope.userLocationBackground	Location
wx.createCameraContext	System resources	scope.camera	New Photo & Video taken by Camera
CameraContext.takePhoto	System resources	scope.camera	New Photo & Video taken by Camera
wx.scanCode	System resources	scope.camera	QR Code Scanned by Camera
wx.startRecord	System resources	scope.record	Audio Record
wx.saveImageToPhotosAlbum	System resources	scope.writePhotosAlbum	Photo
wx.saveVideoToPhotosAlbum	System resources	scope.writePhotosAlbum	Video
wx.getPhoneNumber	System resources	N/A	Phone number
wx.chooseImage	System resources	N/A	Photo in Album
wx.chooseVideo	System resources	N/A	Video in Album
wx.openBluetoothAdapter	System resources	N/A	Bluetooth
wx.getBLEDeviceCharacteristics	System resources	N/A	Bluetooth
wx.getConnectedBluetoothDevices	System resources	N/A	Bluetooth
wx.getBluetoothDevices	System resources	N/A	Bluetooth
wx.getBLEDeviceServices	System resources	N/A	Bluetooth
wx.readBLECharacteristicValue	System resources	N/A	Bluetooth
wx.getWifiList	System resources	N/A	WiFi list
wx.openDocument	System resources	N/A	Document Files
wx.getClipboardData	System resources	N/A	Clipboard Data
wx.addPhoneContact	System resources	N/A	Contact
FileSystemManager.readFile	System resources	N/A	Files
FileSystemManager.readFileSync	System resources	N/A	Files

Table 2. Privacy sensitive APIs provided by WeChat.

- **Number of times for APIs in specific API categories invoked by a mini-app.** We measure the popularity of specific API categories based on how often the APIs in the corresponding category are invoked. This result is shown in Figure 19. We can notice that the APIs in the UI category get used most often (they are called 64.30 times), and the second most is for Debugging (50.88 times), followed by Caching (44.15 times). Surprisingly, through the analysis of the number of API calls in specific API categories, we were able to obtain some interesting results. In particular, we found some developers may mistakenly control the smartphone sensors. For instance, we observed there are 23,442 mini-apps called API wx.startAccelerometer to start the accelerometer but only 18,513 of them called wx.stopAccelerometer) to stop it. Similarly, 453 mini-apps invoke wx.startGyroscope to start the Gyroscope but only 278 of them invoke wx.stopGyroscope.

(4) Privacy Sensitive API Measurement. Next, we measure whether there are any privacy sensitive APIs, and if so, how they are used in practice.

- **Identification of Privacy Sensitive APIs.** *Tecent* provides 580 APIs in total for mini-app programming currently. We manually went through all of them to identify the ones that are deemed privacy sensitive. An API is privacy sensitive if it can fetch the sensitive information from the host app or the Operating System (though sometimes it requires permissions authorized by the users to do so). In total, we identified 31 privacy sensitive APIs, as shown in Table 2. In particular, we identified 6 APIs (row 1 – 6) that can access app-specific data such as user profile or billing address (5 of these accesses require permission authorization from end-users, as illustrated in the Scope column in Table 2); 25 APIs (row 7 – 31) that can access system resources such as Bluetooth, camera, and user location. Interestingly, we can notice that the app-specific data tends to be more sensitive when compared with the

system resources. For example, WeChat allows the mini-apps to access the user profile, which contains the name (or nickname), gender, and residential location.

- **Number of mini-apps that invoke the specific privacy sensitive APIs.** Since we have already counted the number of mini-apps that invoke each specific API in §6.3(3), we just need to select these privacy sensitive APIs from the previous results, as shown in Figure 20. We can see that the most popular privacy sensitive API is `wx.requestPayment` (used by 825,496 apps with 61.91%), followed by `wx.getLocation` (used by 704,683 apps with 52.8%) and `wx.openLocation` (used by 674,982 apps with 50.62%).

6.4 Code Obfuscation Analysis

Since mini-apps use Javascript code, which is human-readable by default, they have to rely on code obfuscation to thwart the reverse engineering attempts. To this end, WeChat provides a built-in JavaScript obfuscator called `Uglify` for mini-app developers to obfuscate their JavaScript code. By using this obfuscator, the names of JavaScript functions and variables are all replaced with meaningless words. While the use of obfuscation option is enabled by default, there might be mini-app developers who do not attempt to protect their code or may have mistakenly turned off this option. In this measurement, we seek to identify the mini-apps that do not use obfuscation. Since fundamentally `Uglify` changes the names of functions and variables by shortening them to just few (often less than 3) alphabetic letters (*e.g.*, "a", "ab" or "abc"), the lengths of the function and variable names of a mini-app can reflect the usage of `Uglify`. As such, we can just simply search and count their length to determine whether a mini-app has used obfuscation. As shown in Figure 21, we found that the average length of all variable and function names is 2.37, indicating that most of the mini-apps have enabled the obfuscation option. We also observed that about 4.6% mini-apps have their average length of the variable names longer than 4, indicating very likely these mini-apps have not used `Uglify`. We have randomly sampled 100 such mini-apps, and confirmed this observation—indeed they are not obfuscated.

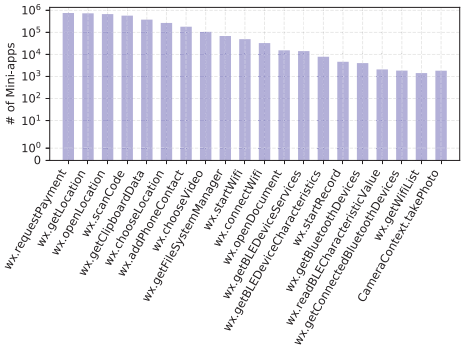

Fig. 20. Sensitive APIs that have been invoked by the mini-apps

6.5 Library Usage

Next, we analyze the library usage in mini-apps. Note that WeChat allows a mini-app to include 3rd-party libraries by enabling the package management using the NPM feature[1] during the development. With NPM, developers are able to include various libraries with different built-in functionalities through an API named `require`. For example, mini-apps can use `md5` function from library `md5` to generate the message digest. By inspecting the library name based on `require`, we can easily identify the included libraries of a mini-app. As such, in this measurement, we focus on (1) how often a library (regardless of its name and its category) is used by a mini-app, and (2) what are the most popular libraries (based on the names).

[1]NPM is a package manager for the JavaScript programming language, and it allows developers to easily share and reuse JavaScript code [39].

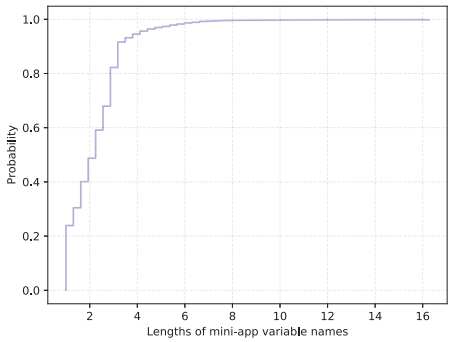

Fig. 21. Distributions of the mini-apps based on the length of variables.

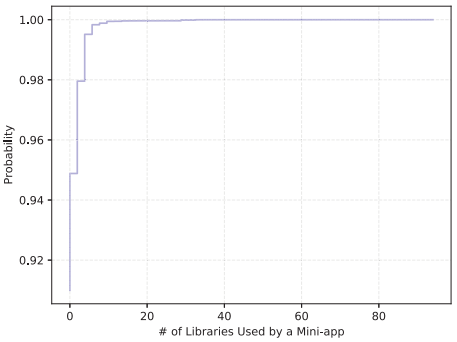

Fig. 22. Distributions of the mini-apps based on the usage of libraries.

(1) How often a library is used by a mini-app. Among the total 1,333,308 mini-apps, we have identified that most mini-apps actually do not use library at all, and only 120,031 (9%) of them have used libraries. The total number of unique libraries used by these mini-apps is 2,866 (based on the library name). We group the number of libraries used by mini-apps, and plot the cumulative distribution function in Figure 22. We can observe that on average each min-app may require around 0.21 libraries. However, 3.89% of the mini-apps require one library and 0.62% mini-apps require more than 3 libraries. There is one very unusual mini-app that has used 94 libraries.

(2) What are the most popular libraries. We then measure the most popular libraries used by mini-apps. While in total there are 2,866 used libraries, we only measure the usage for the top 20 most commonly used libraries. We report this result in Figure 23. We can see that these top used libraries vary from reactive development Toolkit (*e.g.*, `vertx`, which is used by 63,968 mini-apps, accounting for 53.29% of all mini-apps that have used libraries) to utility function libraries including `base64` (13,266 with 11.05%) and `md5` (12,162 with 10.13%). We further examined how mini-apps use these libraries, and found most of the time the libraries just provide additional encapsulation. For example, we found that 695 and 155 mini-apps have used the library `wxApi` and library `wxPage`, respectively, and these two libraries do not have any add-on functionalities but just encapsulation of existing APIs, allowing developers to simplify their mini-app programming.

7 MEASUREMENT BASED ON CRAWLED MINI-APP META-DATA

We have measured the mini-apps based on their packages in §6, and in this section we present our measurement result based on their meta-data. While the meta-data contains a variety of information about the apps, as shown in Figure 4, including such as the `appid`, `description`, `labels` (i.e., the app category), `nickName`, and `evaluate` score (i.e., ratings), we decide to only provide an aggregated view of the apps according to their categories (§7.1) and ratings (§7.2).

7.1 Categories of Mini-apps

According to the meta-data we have crawled, there are in total 274 categories defined by WeChat. We count the number of mini-apps in each category and sort them in descending order based on the mini-apps contained in each category. Figure 25 shows the cumulative distribution of categories, where each integer at the x-axis represents a specific category. We observed that the top 20 mini-app categories accounts for 98.4% of our dataset. To further investigate the what these

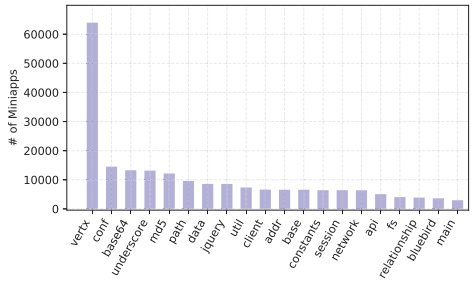

Fig. 23. Top 20 libraries used by mini-apps.

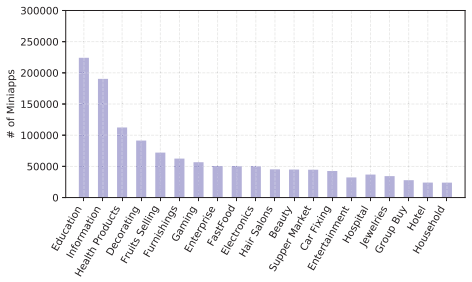

Fig. 24. Top 20 mini-app categories.

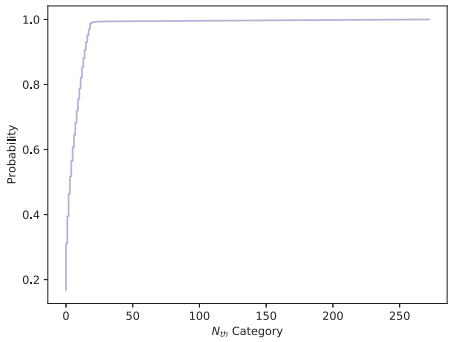

Fig. 25. Distribution of the usage of each specific category.

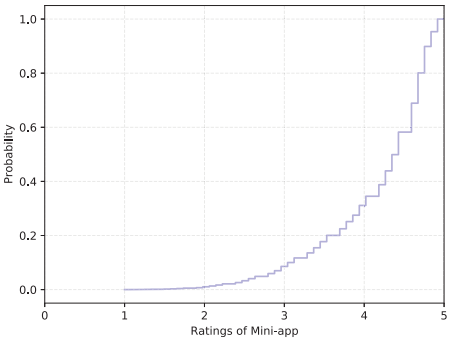

Fig. 26. Distribution of the mini-app ratings.

mini-apps are, we zoom in the top top 20 mini-app categories as shown in Figure 24. We can see that most mini-apps fall into the category of education (16.81%), information (14.28%), health products (8.43%), decoration (6.85%), fruit selling (5.41%), furnishing (4.68%), gaming (4.25%), and so on. It is interesting to notice that the education category has the largest number of mini-apps. We further manually checked the apps in this category and found there are a variety of education related mini-apps, such as Service portals for specific colleges and universities, Service portals for paid or free curriculum materials, and Emulators for training particular skills (*e.g.*, stock trading).

7.2 Ratings of Mini-apps

To create a better environment for the mini-app community, WeChat has provided a feedback mechanism for mini-apps users to evaluate their mini-apps. Evaluation criteria is designed on a five-point scale, and each mini-app can have a specific score to reflect its quality. However, not all the apps's rating is available, and only when an app has been rated by sufficient amount of users can we obtain its rating. Therefore, in our metadata, we only observed that 332,097 mini-apps have their ratings and our measurement is only for these apps. Although many of the mini-app ratings are not available, we believe that the current rating measurement is representative. This is because

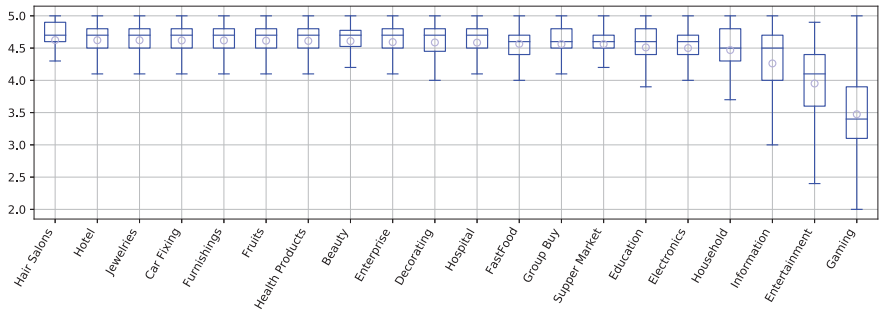

Fig. 27. The Top 20 best rated mini-app categories and their mean ratings.

that only when enough users rate the mini-app, the rating of the mini-app are visible to other users. The mini-apps without ratings are likely to be the less popular mini-apps, and our measurement is thus based on the popular mini-apps and they are reviewed by the majority of the users.

Similar to many of our other measurements, we first group the mini-apps based on their ratings, and then count the numbers of the mini-apps based on their specific rating categories. As shown in Figure 26, 99.2% of these mini-apps have a rating higher than 2.5, and 31.1% even with rating higher than 4.5. We believe this is likely due to the incentive mechanism provided by WeChat, motivating the developers to provide better services. For instance, WeChat will reward the operators or developers with privileged access to certain program or increased visibility when their app rating reaches a higher score, and penalize those with lower scores (*e.g.*, their mini-app may become inaccessible for a period of time) [40].

Recall in our previous measurement (§7.1), 98.4% of the mini-apps fall into the top 20 categories. Therefore, we would like to understand the quality of these mini-apps based on their available ratings. Specifically, as shown in Figure 27, we present a box-plot of the apps in the top 20 categories and report their mean values (the blue circles) and middle bars. We can find that the average score of the apps in the top 20 categories is 4.48. Particularly, mini-apps fall into the Hare Salons category has the highest mean rating score (4.62), whereas games have the lowest (3.47).

App ID	App Name	Category	Rating
wxb6bf7ad9d01606f3	Zhili	Beauty	5.0
wxfe919da247ca0ce7	Dimei	Healthy Product	5.0
wx45cfa7fdcc04ee59	Chunli	Juice & Milk Tea	5.0
wxc6fe31db7712ed76	Taoxian	Group Buy	5.0
wx4cab5b5364df446a	Baby's Dream	Healthy Product	5.0
wx2f51d7eba8e7cd14	Puti	Education	5.0
wx50b4105f2b81017f	Bianli	Education	5.0
wx14ebeb7a39ea80c9	Reading Space	Education	5.0
wx32aecf0ee625fc4e	Fixing	Info Query	5.0
wx3e3d96e8f3bc9853	E Home	Decorating	5.0
wxd0c34c24be39f757	LangDa Glasses	Beauty	5.0
wx6cd9d9a1ce3f65a9	BYN BaiYou	Baby Products	5.0
wx1df06a1c55247e95	WanWei Xin	Supper Market	5.0
wx5a794b6306d01f63	Western China # 1	Restaurant	5.0
wxd47c82dd05c5563e	Angel	Hospital	5.0
wx76306ec09dda16d6	CBS	Broadcasting	5.0
wxf07f8fc0fdb0e6c4	HuangPu	Education	5.0
wx2f70b4554420038d	DingDing Car	Ride-hailing	5.0
wx1a692c61755c9fe9	Zhongyinjie	Education	5.0
wx0b542aa15bb0f443	Eagle King	Fruit Selling	5.0

Table 3. Top 20 best rated mini-apps

App ID	App Name	Category	Rating
wx73b18fbf286b76c3	My 40-Meters Knife	Game	1.3
wxea2ccd29615c9333	My Life	Game	1.3
wxb26aa7a38df3882f	Twenty One	Game	1.3
wx1bed0245e56155ec	WiFi Password Inquiry	Info Query	1.3
wx771f8c822972dfbc	Protect QiuQiu	Game	1.2
wxcaf44cf5c938c4e8	War-Fire	Game	1.2
wx4bf77c65623b2b17	Follow Me	Game	1.2
wx22d25ec6a8ef4266	Cloud WiFi	Info Query	1.2
wx17e7bd3814f94db4	Renting Hose	Renting	1.2
wxf3981d7be84c6801	Supper Kan	Game	1.2
wxc0de602e8de8df8f	AnShun	Beauty	1.2
wxd83fb186bbd9f1d4	Car Bank	Bank	1.1
wxb51c2c10a1d7e3e3	Happy Glass	Game	1.1
wxd9a5c1d0140c0424	Cloud	Game	1.1
wxa8d7e5c4015105cd	Tuzhong	Office	1.1
wx05dc48f13a43f5c0	SanGuo Da	Game	1.0
wx7b03473f23f1d1ce	Yin	Game	1.0
wx6553e45faa9952b0	I'm Gang	Game	1.0
wx88facc1d5d585e99	Bao	Game	1.0
wx5439114b332007da	FGO	Info Query	1.0

Table 4. Top 20 worst rated mini-apps

The best and worst rated mini-apps. To understand why the mini-apps have higher (or lower) rating scores, we further manually checked 20 mini-apps with the highest scores and 20 mini-apps with lowest scores. Specifically, there are 5,186 mini-apps with 5.0 score, and we randomly selected 20 from them as the best mini-apps, and presented them in Table 3. Next, we found that 131 mini-apps with scores lower than 1.3 (including 1.3), and among them, 16 mini-apps have scored lower than or equal to 1.2. Therefore, we selected these 16 apps first, and then randomly selected 4 other apps with score 1.3 as the top 20 worst rated mini-apps as shown in Table 4. It can be observed from Table 3 that many best rated mini-apps are from the education category, which is reasonable since most mini-apps fall into this category. Meanwhile, these mini-apps usually have well designed UIs. However, from Table 4, we can observe that most of the worst mini-apps are from the Game and Information Query categories. We then tested some of these Games and observed they contain heavy advertisement. For the three Information Query mini-apps, we tested two of them (*i.e.*, WiFi Password Inquiry and Cloud WiFi) and found they do not always work when provided the SSID to query the corresponding password.

8 DISCUSSION

Limitation and Future Works. While we have obtained a number of aggregated results on the mini-apps such as their size distribution and API usage, our measurement is certainly not perfect and there are many avenues to improve it. For instance, we could have performed a periodical analysis on the meta-data to measure when an app is published in the market and when it disappears, which can provide a longitudinal view of the market-available apps. Second, currently we only crawled the meta-data for about 1.39 million mini-apps, and we can certainly keep improving our keyword lists to obtain other uncovered apps. Finally, there will be many other interesting studies such as detecting the bugs or vulnerabilities in the mini-apps. We leave these to future research efforts. Meanwhile, to have more community's interests (and also in support of the open science), we have released the source code of our MiniCrawler so that other researchers in the community can also investigate the research questions of their interests with mini-apps.

Ethics. We did take ethics into the highest consideration when conducting this measurement study. First, we have followed the community practice of data crawling (such as PlayDrone [11], and other web crawlers [41]). Second, we did not attack any WeChat accounts, and only used our own accounts to query the WeChat server and crawl the data. Third, we also did not attack the WeChat servers (no denial of service) and meanwhile we limited our number of requests per seconds to just two while issuing the meta-data query (note that PlayDrone used 200 requests per second). Fourth, we have only made our MiniCrawler public available, but we will not release the collected mini-apps to protect the privacy and also the intellectual property of the mini-apps if there are any. Finally, we have also been engaging with *Tencent* by actively disclosing the issues we have found such as the unexpected long live session tokens, and also the developers' improper practices on controlling the smartphone sensors (*e.g.*, only starting the sensors without stopping it).

9 RELATED WORK

There has been a large body of research measuring the native mobile apps from either mobile appstore or pre-installed in Android firmware. For instance, PlayDrone [11] conducted the first large-scale characterization with over one million Android apps available on Google Play including their evolution, library usages, and app clones. Complementary to PlayDrone, Wang et al. [14] measured the app similarities in 3rd-party appstores (other than Google play). Ali et al. [42] compared app ratings and prices on both Apple AppStore and Google play with 80,000 apps. Wang

et al. [43] investigated the removed 791,138 apps from Goolge play to explore the reasons of the removal. Additionally, Wang et al. [44] analyzed over 1.2 million apps and 320,000 developers to understand their app development practices. With respect to the pre-installed apps, DroidRay [45] scanned 24,009 pre-installed apps from 250 Android firmware, and discovered that 1,947 (8.1%) pre-installed apps have signature vulnerability and 7.6% of the firmware contain malware. Most recently, Elsabagh et al. [46] analyzed 331,342 pre-installed apps in 2,017 Android firmware images and identified 850 unique privilege-escalation vulnerabilities.

Unlike native mobile apps which have been intensively studied, there is only one work [6] that has looked into the issues with the mini-apps so far. In particular, Lu et al. [6] conducted the first systematic study of how WeChat manages the system resources (*i.e.*, the permission model of WeChat), and discovered the security flaws such as stealthily privilege escalation to access camera, photo gallery, or microphones without user's awareness. In addition, they also identified other possible attacks such as the phishing attacks with mini-apps due to the fact that mini-apps often run in the fullscreen window, which can trick the user into believing a mini-app is a native app.

10 CONCLUSION

We have presented MiniCrawler, a scalable mini-app crawler that is able to automatically download mini-apps from WeChat server. We describe how we have addressed various challenges encountered when building our crawler. We have used it to download more than one million mini-apps. With them, we have performed a large scale measurement study and discovered a number of interesting aggregated results including a mini-app is 11x smaller than a native app on average, it often has hundreds of API invocations, most of the mini-apps are obfuscated, the largest category of the mini-apps is in the education category, and the apps in this category also tend to have higher ratings.

ACKNOWLEDGEMENT

We are grateful to the anonymous reviewers and also our shepherd Dr. Zihui Ge for their very helpful comments. This work was partially supported by NSF awards 1834215 and 1834216.

REFERENCES

[1] C. Lee, "WeChat launches mini-app feature," https://www.zdnet.com/article/wechat-launches-mini-app-feature/, 01 2017, (Accessed on 04/21/2021).
[2] L. Eadicicco, "How facebook, Apple, Google copied china's WeChat messaging app - business insider," https://www.businessinsider.com/facebook-apple-google-copied-wechat-app-trump-executive-order-2020-8, 08 2020, (Accessed on 04/21/2021).
[3] K. Leswing, "Three ways to get iPhone software without using Apple's App Store," https://www.cnbc.com/2020/09/01/how-to-get-iphone-software-without-using-apples-app-store.html, 9 2020, (Accessed on 04/21/2021).
[4] A. Ha, "Daily Crunch: Snapchat is getting mini apps," https://techcrunch.com/2020/06/12/daily-crunch-snapchat-is-getting-mini-apps/, 06 2020, (Accessed on 04/21/2021).
[5] "How brands are using WeChat mini programs," https://mavsocial.com/wechat-mini-programs-for-brands/, 2018.
[6] H. Lu, L. Xing, Y. Xiao, Y. Zhang, X. Liao, X. Wang, and X. Wang, "Demystifying resource management risks in emerging mobile app-in-app ecosystems," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 569–585.
[7] "Number of monthly active WeChat users from 2nd quarter 2011 to 3rd quarter 2020," https://www.statista.com/statistics/255778/number-of-active-wechat-messenger-accounts/, 3 2020, (Accessed on 04/21/2021).
[8] "WeChat data, insights and statistics: user profile, behaviours, usages, market trends," https://wechatwiki.com/wechat-resources/wechat-data-insight-trend-statistics/, 03 2019, (Accessed on 04/21/2021).
[9] "The total size of all subpackages of a Mini Program cannot exceed 12 MB," https://developers.weixin.qq.com/miniprogram/en/dev/framework/subpackages.html, 06 2020, (Accessed on 04/21/2021).

[10] A. Rafi, "Android app size limit increased from 50 MB to 4GB," https://www.androidguys.com/news/android-app-size-limit-increased-from-50mb-to-4gb/, 5 2012, (Accessed on 04/21/2021).
[11] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," in *The 2014 ACM international conference on Measurement and modeling of computer systems*, 06 2014, pp. 221–233.
[12] S. Seneviratne, H. Kolamunna, and A. Seneviratne, "A measurement study of tracking in paid mobile applications," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 6 2015, pp. 1–6.
[13] H. Wang, H. Li, and Y. Guo, "Understanding the evolution of mobile app ecosystems: A longitudinal measurement study of google play," in *The World Wide Web Conference*, 09 2019, pp. 1988–1999.
[14] H. Wang, Z. Liu, J. Liang, N. Vallina-Rodriguez, Y. Guo, L. Li, J. Tapiador, J. Cao, and G. Xu, "Beyond google play: A large-scale comparative study of chinese android app markets," in *Proceedings of the Internet Measurement Conference 2018*, 2018, pp. 293–307.
[15] W. Liu, G. Zhang, J. Chen, Y. Zou, and W. Ding, "A measurement-based study on application popularity in android and ios app stores," in *Proceedings of the 2015 Workshop on Mobile Big Data*, 2015, pp. 13–18.
[16] C. A. Kardous and P. B. Shaw, "Evaluation of smartphone sound measurement applications (apps) using external microphones—a follow-up study," *The Journal of the acoustical society of America*, vol. 140, no. 4, pp. EL327–EL333, 2016.
[17] "WeChat mini program development guide," https://developers.weixin.qq.com/miniprogram/en/dev/framework/, 08 2017, (Accessed on 04/21/2021).
[18] "Reference documentation for mini program frameworks," https://developers.weixin.qq.com/miniprogram/en/dev/reference/, 08 2020, (Accessed on 04/21/2021).
[19] "WeChat account protection," https://help.wechat.com/cgi-bin/micromsg-bin/oshelpcenter?opcode=2&lang=en&plat=android&id=170417vMBnEB170417InAF36&Channel=helpcenter, 08 2020, (Accessed on 04/21/2021).
[20] H. Liu, P. Gao, and Y. Xiao, "New words discovery method based on word segmentation result," in *2018 IEEE/ACIS 17th International Conference on Computer and Information Science*. IEEE, 2018, pp. 645–648.
[21] "Introduction to android hook framework Xposed," https://programmer.ink/think/introduction-to-android-hook-framework-xposed.html, 06 2019, (Accessed on 04/21/2021).
[22] "WeChat API categories," https://developers.weixin.qq.com/miniprogram/en/dev/api/, 03 2020, (Accessed on 04/21/2021).
[23] "Directory structure (offical document)," https://developers.weixin.qq.com/miniprogram/en/dev/framework/structure.html, 03 2020, (Accessed on 04/21/2021).
[24] "WXML," https://developers.weixin.qq.com/miniprogram/en/dev/reference/wxml/, 03 2020, (Accessed on 04/21/2021).
[25] "WXSS," https://developers.weixin.qq.com/miniprogram/en/dev/framework/view/wxss.html, 03 2020, (Accessed on 04/21/2021).
[26] "Configuration of server domain name (WeChat official document)," https://developers.weixin.qq.com/miniprogram/en/dev/framework/ability/network.html, 2020.
[27] A. Mahajan, *Burp Suite Essentials*. Packt Publishing Ltd, 2014.
[28] "Dex to java decompiler," https://github.com/skylot/jadx, 06 2015, (Accessed on 04/21/2021).
[29] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi, "ASM: A programmable interface for extending android security," in *23rd USENIX Security Symposium*, 2014, pp. 1005–1019.
[30] "Account security," https://007.qq.com/account-guard.html?ADTAG=index.block, 01 2020, (Accessed on 04/21/2021).
[31] C. Chen, K. Wu, V. Srinivasan, and X. Zhang, "Battling the internet water army: Detection of hidden paid posters," in *2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. IEEE, 2013, pp. 116–120.
[32] L. Zhong, "Ranking of the most commonly used 1,000 chinese characters," https://www.thn21.com/base/zi/17300.html, (Accessed on 04/21/2021).
[33] ""Jieba" (Chinese for "to stutter") chinese text segmentation: built to be the best python chinese word segmentation module." https://github.com/fxsjy/jieba, (Accessed on 02/01/2021).
[34] "Xposed," https://repo.xposed.info/, (Accessed on 02/01/2021).
[35] "Weixin mini program platform operation rules," https://developers.weixin.qq.com/miniprogram/en/product/, 2020.
[36] C. Zuo, H. Wen, Z. Lin, and Y. Zhang, "Automatic fingerprinting of vulnerable ble iot devices with static uuids from mobile apps," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1469–1483.
[37] J. Desjardins, "How many millions of lines of code does it take?" https://www.visualcapitalist.com/millions-lines-of-code/, 02 2017, (Accessed on 04/21/2021).
[38] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[39] R. E. Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara, "Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages," in *2018 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2018, pp. 559–563.
[40] "WeChat miniapp evaluation," https://developers.weixin.qq.com/community/develop/article/doc/00028a270781c01547b81c2565b013, 2019, (Accessed on 04/21/2021).
[41] S. M. Mirtaheri, M. E. Dinçktürk, S. Hooshmand, G. V. Bochmann, G.-V. Jourdan, and I. V. Onut, "A brief history of web crawlers," *arXiv preprint arXiv:1405.0749*, 2014.
[42] M. Ali, M. E. Joorabchi, and A. Mesbah, "Same app, different app stores: A comparative study," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems*. IEEE, 2017, pp. 79–90.
[43] 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 *2018 IEEE/ACM 15th International Conference on Mining Software Repositories*. IEEE, 2018, pp. 231–242.
[44] 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*, 2017, pp. 163–172.
[45] M. Zheng, M. Sun, and J. C. Lui, "Droidray: a security evaluation system for customized android firmwares," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014, pp. 471–482.
[46] M. Elsabagh, R. Johnson, A. Stavrou, C. Zuo, Q. Zhao, and Z. Lin, "FIRMSCOPE: Automatic uncovering of privilege-escalation vulnerabilities in pre-installed apps in android firmware," in *29th USENIX Security Symposium*, Aug. 2020.

Received February 2021; revised April 2021; accepted April 2021