# iOS, Your OS, Everybody's OS:
# Vetting and Analyzing Network Services of iOS Applications

Zhushou Tang[1,6] Ke Tang[1] Minhui Xue[2] Yuan Tian[3]
Sen Chen[4] Muhammad Ikram[5] Tielei Wang[6] Haojin Zhu[1]

[1]Shanghai Jiao Tong University [2]The University of Adelaide [3]University of Virginia
[4]Nanyang Technological University [5]Macquarie University [6]PWNZEN InfoTech Co., LTD

## Abstract

Smartphone applications that listen for network connections introduce significant security and privacy threats for users. In this paper, we focus on vetting and analyzing the security of iOS apps' network services. To this end, we develop an efficient and scalable iOS app collection tool to download 168,951 iOS apps in the wild. We investigate a set of 1,300 apps to understand the characteristics of network service vulnerabilities, confirming 11 vulnerabilities in popular apps, such as `Waze`, `Now`, and `QQBrowser`. From these vulnerabilities, we create signatures for a large-scale analysis of 168,951 iOS apps, which shows that the use of certain third-party libraries listening for remote connections is a common source of vulnerable network services in 92 apps. These vulnerabilities open up the iOS device to a host of possible attacks, including data leakage, remote command execution, and denial-of-service attacks. We have disclosed identified vulnerabilities and received acknowledgments from vendors.

## 1 Introduction

A network service is built on an application programming interface (API) or a library that provides networked data storage, or other online functionality to applications. Many potential threats have spawned with the widespread use of smartphones with network service capabilities. Poor implementation practices expose users to denial-of-service (DoS) or remote code execution (RCE) attacks, and unauthorized access can occur due to the weak protection of network resources. Such threats have already been substantiated in the real world. One such example is the DoS or RCE attack against `WhatsApp` that can occur when a `WhatsApp` user accepts a call from a malicious peer [5, 17]. Another is the "wormhole" vulnerability, where open ports in Android apps allow an attacker to remotely access data or manipulate apps without sufficient authorization [51]. Recently, a proof-of-concept DoS attack that prevents communication between iOS devices has been demonstrated by utilizing the specific design flaw of the Apple Wireless Direct Link (AWDL) protocol [74].

Recent research evaluating the security of open port usage in Android apps has demonstrated new attack avenues that can exploit the vulnerability of network services and access unauthorized sensitive data previously unthought of [22, 32, 55, 80]. Some works have also proposed vetting methodologies to handle dynamic code loading [69], complex implicit control/data flows [31], or advanced code obfuscation [46, 79], techniques created to overcome the inherent limitations of Android app static analysis. Unfortunately, these sophisticated and ad hoc vetting approaches only target Android apps.

iOS's network architecture is built on top of `BSD sockets`. When acting as a resource provider, the app turns the iOS device into a server to provide services to a client once a connection is established. For example, the `Handoff` [23] feature of iPhone serves as a server to receive commands from a client in the same Wi-Fi network. Apple encourages network connections between different components through `Bonjour protocol` [28, 73], which broadcasts the network service to clients. Although Apple reviews third-party apps before releasing them on the iTunes App Store, The vetting process predominantly focuses on detecting *malicious* apps instead of network service vulnerabilities.

In this work, we propose the *first* vetting methodology of iOS apps' network services. There are three elements that make vetting and analyzing iOS apps more technically challenging than Android apps. *(i)* Android apps are easy to collect and analyze; however, a public repository of iOS apps is not readily available due to the closed nature of Apple's app ecosystem. *(ii)* Practical program analysis tools for automatically analyzing iOS apps (implemented in Objective-C or SWIFT) are not as well developed or diverse as tools for Android (written in Java) are [26, 45, 77]. *(iii)* The layout of code in Android apps is highly structural, but the boundaries of iOS code are obscure, causing previous methods for third-party library identification in Android apps [27, 48, 76] to function incorrectly on iOS apps.

To ensure the efficiency of our pipeline, we tailor our app collection (cf. § 3), vetting process (cf. § 4), and library identification (cf. § 5) techniques to overcome the unique challenges presented by iOS apps. *First*, to collect and analyze apps, we need to download, decrypt, and parse the executable, a process that leverages iTunes' unique download interface with a special decryption method to expedite app collection. Our collection methodology can download and decrypt over 5,000 apps per day using only two Apple accounts and two `jailbroken` iOS devices, providing better scaling up of tasks with lower latency than past works [62, 67]. After collection, we parse the iOS apps, obtain the metadata of apps, and feed it into a search engine for retrieval and subsequent analysis. *Second*, to improve the accuracy and efficiency of our vetting results, we write an "addon" which evaluates the network interface on the fly. To expedite the automated analysis, we leverage an on-demand inter-procedural [70] data-flow analysis tool to restore the implicit call introduced by the `message dispatch` property [24] of Objective-C or SWIFT runtime. *Third*, to deal with the obscure documentation of system and third-party network services, we propose a call stack based collection method that overcomes the limitations of the current class-clustering based third-party library identification [67]. In our method, we first identify system network service APIs by traveling the call stack of each app; then third-party network service libraries can be distinguished through similarity analysis on the runtime call stack.

We begin our analysis with a set of 1,300 applications, which we refer to as "seed apps". Seed apps are used to understand the characteristics of network service vulnerabilities and extract signatures for large-scale analysis of network services. To analyze the seed apps, we adopt the vetting methodology of "dynamic first, static later, and manual confirmation last". The dynamic analysis can check for misconfigured network interfaces on a large scale, which allows us to pinpoint a small portion of candidate network service apps. The comparably more time-consuming static analysis can then be used to perform a fine-grained check for potential vulnerabilities. Finally, manual confirmation is involved in verifying static analysis results. In addition, the precise call stack of `_bind` collected by dynamic analysis can be used for the identification of APIs and libraries. Knowledge gained from seed apps is then applied to the large-scale analysis, including measuring the distribution of network services of iOS apps, finding the association of network service libraries, and fine-grained analysis on three typical libraries. Vetting results show that vulnerabilities of the network service open up the iOS app to data leakage, remote command execution, or denial-of-service attacks (cf. § 7).

**Responsible disclosure.** We have reported these vulnerable apps to relevant stakeholders through the Security Response Center (SRC) of vendors. Three vulnerabilities have been acknowledged, including Google issue ID: 109708840 and Tencent issue IDs: 34162 and 23546 (see the list of major

**Table 1:** Major Vulnerabilities Found.

| App | Vendor | Vulnerability Impact | Severity (by vendor) | Status |
|---|---|---|---|---|
| **Waze** | Google | CE/RCE/DoS | N/A | Patched |
| **Scout GPS Link** | Telenav | CE | N/A | Pending |
| **QQBrowser** | Tencent | CE | High | Patched |
| **Taobao4iPhone** | Alibaba | CE | N/A | Pending |
| **Youku** | Alibaba | CE | N/A | Pending |
| **Handoff** | Apple | RCE/DoS | N/A | Patched |
| **Now** | Tencent | Privacy Leaks | High | Patched |
| **Amazon Prime Video** | Amazon | Privacy Leaks | N/A | Pending |
| **QQSports** | Tencent | Privacy Leaks | N/A | Pending |
| **KENWOOD** | WebLink | RCE/DoS | N/A | Patched |
| **JVC** | WebLink | RCE/DoS | N/A | Patched |
| **WebLink Host** | WebLink | CE/RCE/DoS | N/A | Patched |
| **Flipps TV** | Flipps Media | CE/RCE/DoS | N/A | Pending |
| **FITE TV** | Flipps Media | CE/RCE/DoS | N/A | Pending |
| **JDRead** | JD | Privacy Leaks | Medium | Patched |
| **QQMail** | Tencent | Privacy Leaks | N/A | Pending |

[1] CE: Command Execution.
RCE: Remote Code Execution.
DoS: Denial-of-Service.

vulnerabilities found in Table 1). We also helped the vendors patch these vulnerabilities and are currently discussing possibilities of vendor deployment of our vetting system. To foster further research, we release the dataset used in this paper and the code developed for analysis, and encourage readers to view short video demos of vulnerabilities we discovered at `https://sites.google.com/site/iosappnss/`.

The key contributions of this paper are as follows:

- **An efficient iOS app collection tool.** To facilitate our analysis, we introduce an iOS app collection tool thanks to the use of the headless-downloader and executable decryption. The headless-downloader enables us to download `.ipa` files from iTunes App Store fluently. The executable decryption we developed does not need to upload large `.ipa` files to iOS devices, install apps, or download entire decrypted `.ipa` files from iOS devices. The proposed downloading enables large-scale dataset collection with limited iOS devices, and can decrypt over 5,000 apps per day with only two iOS devices, improving the scalability of data collection by 17 times compared to the state-of-the-art collection method in [62]. The collection of such a large dataset of iOS apps is a significant resource and also serves as a useful benchmark for future research.

- **Systematic characterization of network services of iOS apps.** We apply dynamic analysis to collect a call stack from each app. Based on the call stack information, we extract system APIs by backward traveling the stack, identify the third-party network service libraries by comparing the tokens originated from the stack. By taking signatures of the network services, we systematically characterize network services in iOS ecosystem, including the prevalent usage of network services of iOS apps, the distribution of network services across app categories, and the association of these network services.

- **New vulnerabilities of iOS apps identified.** This is the first work for vetting the security of iOS apps' network services. We use dynamic analysis to assess the interface of the network service and then improve (and implement)

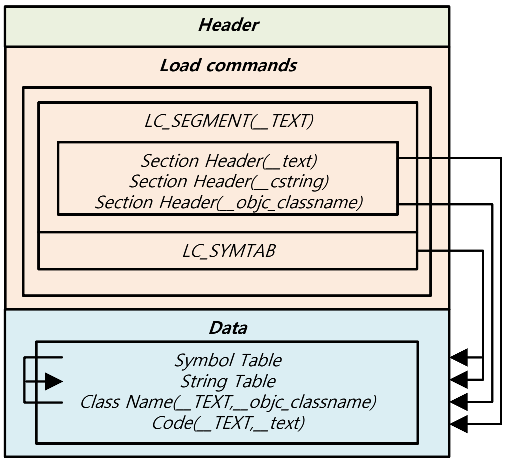


**Figure 1:** The simplified inner structure of a Mach-O file.

the state-of-the-art static data-flow analysis tool [49] to further scrutinize the apps at scale. The vetting process is performed on 1,300 seed apps, with 11 network service vulnerabilities confirmed manually, including some top popular apps, such as `Waze`, `QQBrowser`, and `Now`. By taking into account three typical third-party network service libraries integrated by 2,116 apps and case-by-case analysis, an additional 92 vulnerable apps are discovered. We cross check the vulnerabilities identified and find none of these vulnerabilities exist in Android apps.

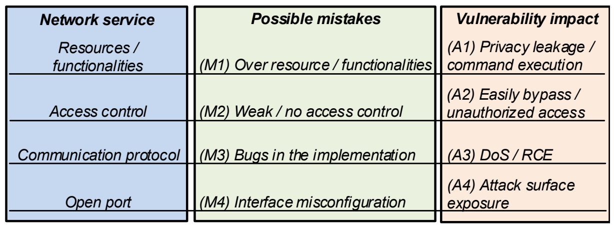
To the best of our knowledge, this is the first paper to systematically examine the security of network services within iOS apps on a large scale. The entire vetting methodology proposed in this paper can serve as a starting point for further study of this important area.

## 2 Background and Threat Model

We begin by introducing the structure of iOS apps, defining the network services of the iOS apps, and presenting the threat model in this study.

### 2.1 The Structure of iOS Apps

The iOS app is an archive file (i.e., `.ipa`) which stores an `Application Bundle` including `Info.plist` file, executable, resource files, and other support files. For the sake of digital rights management (DRM), Apple uses a `.supp` file containing the keys within the `.ipa` file to decrypt the executable [78]. The executable in the `Application Bundle` is encoded in Mach-O format [68] consisting of three parts: `Header`, `Load commands`, and `Data`. The `Load commands` region of a Mach-O file contains multiple segments and each segment specifies a group of sections. Each section within is parallel, such as the instructions in the `__text` section, C string in the `__cstring` section, and Objective-C class



Network service
Resources / functionalities
Access control
Communication protocol
Open port
Possible mistakes
(M1) Over resource / functionalities
(M2) Weak / no access control
(M3) Bugs in the implementation
(M4) Interface misconfiguration
Vulnerability impact
(A1) Privacy leakage / command execution
(A2) Easily bypass / unauthorized access
(A3) DoS / RCE
(A4) Attack surface exposure

**Figure 2:** Architecture of network service, use mistakes, and resulting vulnerabilities. Each row represents a possible mistake, which, according to the network service layer, could lead to serious security and privacy issues.

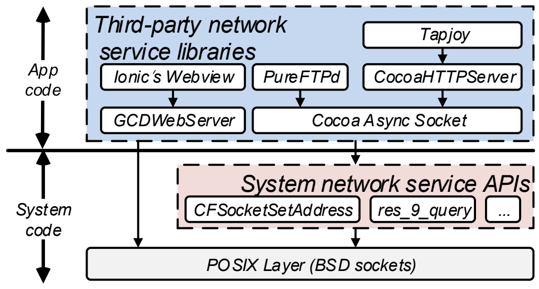


**Figure 3:** Overview of system network service APIs and third-party network service libraries. The top sub-figure shows the relation among different third-party libraries leveraging BSD socket either directly or via system network service APIs.

object name in the `__objc_classname` section. In particular, instructions in the `__text` section are encoded with the `ARM/THUMB` instruction set. The simplified Mach-O format file is depicted in Figure 1.

For security purposes, an iOS app's interactions with the file system are limited to the directories inside the app's sandbox directory [42, 43]. During the installation of a new app, the installer creates a `bundle container` directory that holds the `Application Bundle`, whereas the `data container` directory holds runtime generated data of the app. The `bundle container` directory and the `data container` directory reside in two randomly generated directories. For such design, if the root folder of a vulnerable network service is set to a `bundle container` directory, files within `Application Bundle` will be exposed. The randomly generated directories alleviate the path traversal threat due to the difficulty for the adversary to predict the `data container` path.

### 2.2 Network Services of iOS Apps

A network service is built on an API or a library that provides networked data storage, or other online functionality to applications. A bottom-up network service is defined as having "open port," "communication protocol," "access control," and "resources/functionalities" layers (see Figure 2). In the example of a GPS navigation app, termed `Waze` [15], the app generally projects the app's UI to the vehicle's screen via USB connection. In particular, the app integrates the `WebLink` [16]

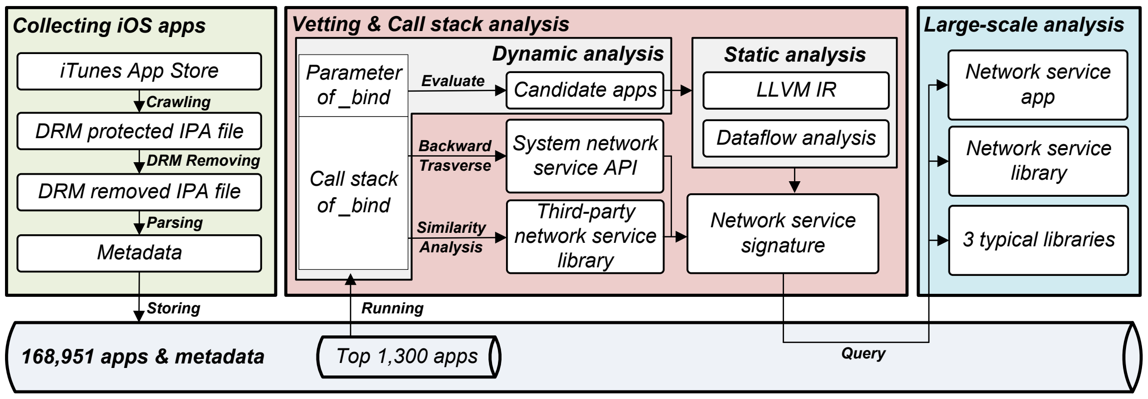


**Figure 4:** Overview of our system pipeline: (1) the green box shows the iOS app collection methodology (cf. § 3); (2) the red box shows the methodology for vetting the first 1,300 apps by using dynamic and static analysis (cf. § 4) and the call stack analysis for building signatures of system and third-party network services (cf. § 5); (3) the blue box shows the large-scale analysis on network service APIs/libraries over 168,951 iOS apps (cf. § 6) and a fine-grained analysis of 3 typical libraries; (4) the bottom gray bar includes two datasets of iOS apps for analysis.

library to stream a user's iPhone screen to the virtual app screen of the in-vehicle infotainment (IVI) system. Meanwhile, the app receives touch events on the in-vehicle device to respond to end-user's actions. In doing so, the `WebLink` library in the `Waze` app turns the app into a server to accept the connection from the IVI system.

As for the architecture of the network service of iOS apps, both system and third-party network service libraries are directly or indirectly built on top of `BSD sockets` (see Figure 3). As shown in the dashed, pink box of Figure 3, iOS wrapped the `BSD sockets` for developers to facilitate the development of network services. For example, the system API `_CFSocketSetAddress` [25] in `Core Foundation` framework bridges access to `BSD sockets`. Based on this API, developers can compose various applications on top of the TCP layer of the network protocol stack to provide network services. In addition, many third-party network service libraries are available for developers to use, as shown in the blue box of Figure 3. In general, network services provided by the third-party libraries operate on the application layer of the network protocol stack.

## 2.3 Threat Model

Previous works [55, 80] classified Android network service adversaries to local, remote, and web adversaries. However, we do not consider attacks by a hostile app installed locally on the device (i.e., local adversary) or by enticing the victim to browse a JavaScript-enabled web page (i.e., web adversary) in our study. For example, the `Libby`'s web service demonstrated in Figure 12(b) falls outside of our scope. This paper focuses on more practical remote adversaries for vulnerability analysis because these potential vulnerabilities are of high risk.

To find a potential victim, a remote adversary can scan and examine the network (i.e., the Wi-Fi network or cellular network) by designating specific port numbers [51]. Such an adversary subsequently compares the `banner`[1] returned from the connected server (i.e., a network service of the iOS app). If the `banner` is expected, the adversary then confirms the real victim and can mount a remote 0-click attack, such as stealing personal information for profit. A real-world attack targets Android device to be exposed in a cellular network to thwart end-user privacy for extortion [2].

To further break down the role of a remote adversary, Figure 2 shows that each layer allows for different remote attacks: *(i)* The interface would be exposed if the network service is activated and the "open port" is misconfigured. *(ii)* A poor implementation of "communication protocol," usually written in a universal language `C/C++`, may lead to DoS or RCE of apps [5, 17, 74]. *(iii)* Insufficient "access control" incurs unauthorized access to network resources/functionalities.

## 3 Methodology of iOS App Collection

Collecting apps and meta-information on Apple iTunes is not a trivial task. iTunes implements various restrictions for app collection, such as capping the number of requests to limit automated crawling methods and encrypting the executable for DRM consideration. Because of these challenges, previous collection methods are limited in scalability and efficiency. Current iOS app downloading methods are UI manipulation [67] and in-device app crawler [62]. They decrypt executable by using either `Clutch` [6], `dumpdecrypted` [10], or the `Frida` [8] extension `frida-ios-dump` [20]. We realize that recent research [62] expended three months to collect 28,625 iOS apps, lending evidence to the scalability issue when extending to large-scale analysis.

[1]Banner is a specific message to uniquely identify a network service. For instance, after connected to the network service of the `Waze` app, a client will receive the message "`WL`" from the server.

## 3.1 iOS App Collection

In this section, we describe our method for collecting iOS apps IDs, downloading the .ipa file from iTunes, removing DRM protection to get decrypted executable, and parsing executable. Our method consists of the following three modules (see green box of Figure 4):

**Collecting IDs and downloading apps from iTunes.** Each iOS app on iTunes has a unique identifier (i.e., ID). For example, Instagram is identified by the unique ID: 389801252, and can be accessed from iTunes by using this ID. Based on the iTunes Search API [13], we collect the ID list recursively. For example, the following request returns meta-information of the top 20 apps in the "Productivity" category, such as ID and the app name.

https://itunes.apple.com/search?term=productivity&country=us&media=software&limit=20.

Afterwards, we use a breadth-first-search approach that obtains "similar apps" using iTunes Search API. Queries are relayed by different proxies to bypass the crawler blocking of iTunes.

To purchase and download a DRM protected .ipa file from iTunes, we implement a headless-downloader. In essence, we implement the requests for purchasing and downloading of iTunes, sign method for the requests, and modify the requests header to bypass device identification authentication. Our headless-downloader leverages the Windows' version of iTunes' .dll files and invokes the interface of the .dll files. The headless-downloader accepts ID and Apple accounts as arguments to download the .ipa file.

**Decrypting the executable**. To investigate the code, we need to decrypt the executable of the downloaded apps. Since the state-of-the-art techniques require physical iOS devices to be involved in decrypting process [6, 10, 20], to avoid using many devices, we use an agent app which is pre-installed on a jailbroken iOS device. After the agent app is loaded into memory, the iOS system is set to decrypt the executable. We then suspend the decrypting process and inject the encrypted executable into the agent app to utilize the inherent decrypting process of the iOS system. After the iOS system decrypts the executable, we dump the executable on the jailbroken device, retrieve it through the USB connection, and merge the decrypted executable into the original .ipa file in a local desktop computer. In such a way, we obtain the decrypted executable without installation and uninstallation and only need to transfer the executable (not Application Bundle) between the desktop computer and the iOS device.

**Parsing the executable.** In order to facilitate subsequent analysis and share our dataset for further research, we parsed the executable by using JTOOL [14] and extracted relevant metadata such as the class name and string within an executable. Data in Info.plist is also withdrawn, such as bundle ID in "CFBundleIdentifier" field or the app name in "CFBundleName" field. These metadata and meta-information of an app, including category and popularity, are stored in a search engine, namely ELASTICSEARCH [50] for later queries.

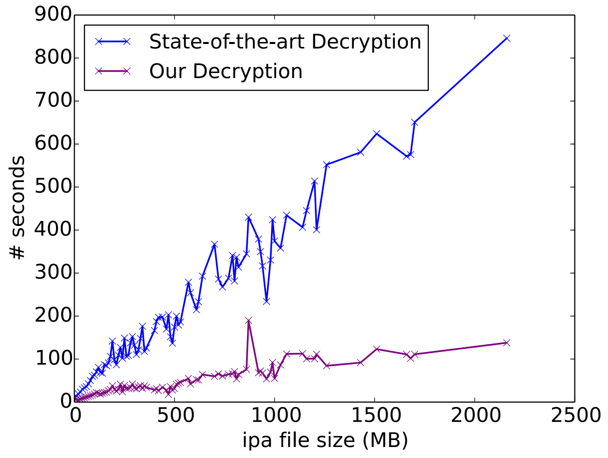


**Figure 5:** Performance of .ipa file decryption process. The time consumption is almost constant regardless the size of the .ipa file when only delivering the executable.

**Selecting seed apps.** Seed apps are used to understand the characteristics of network service vulnerabilities and extract signatures for large-scale analysis of network services. Seed apps are the iTune's apps downloaded from both the United States and China app stores. To choose seed apps, we take the top 20 free apps from each category on iTunes, composing 1,300 apps in total. Since the list of apps on iTunes App Store leaderboards is constantly updated, we use a snapshot of the lists collected on May 8, 2018. Among these 1,300 apps, we have 24 categories (480 apps in total) from China region and 41 categories (820 apps in total) from the United States region. Apple classifies the "Game" apps in the United States region into more fine-grained categories, such as "Games-Card" and "Games-Action". These 1,300 apps provide a huge diversity across all app categories. There is almost no overlap between the top popular apps in China and the United States, and the taxonomy of apps in both countries are almost the same. We only found two apps (i.e., Rules of Survival [19] and Dancing Line [18]) that were ranked in the top 20 in both the United States and China.

## 3.2 Evaluation of iOS App Collection

Collecting iOS apps effectively is a challenging and critical problem. To evaluate the efficiency of our app collection scheme, we experiment with two procedures: app download and app decryption. Our unique design of these two procedures is the key to the performance improvement for app collection. For the comparison of executable decryption speed, we attempt to automate the state-of-the-practice tools ideviceinstaller [12] and frida-ios-dump [20] adopted by research [33, 41, 67]. The decryption speed of these tools is largely concurrent with the download speed using our headless-downloader, which expends approximately 29 hours to decrypt the 1,300 seed apps with an iPhone 6s device, averaging out roughly 80 seconds per app. By contrast, our decrypting process, without manual handling `.ipa` files, takes approximately 21 seconds on average per app, almost four times faster than the tools. Nevertheless, we acknowledge that the speed-up of the app decryption is positively correlated to the existence of many "Game" apps in question (35.0% of the whole dataset), where their resource files are unnecessary to be delivered between a desktop computer and an iOS device (see Figure 5). Comparing the speed of downloads is not as trivial as comparing the speed of decryption. We acknowledge that a rigorous comparison of app download between ours and other de facto research-standard tools is difficult because of the unknown arguments of the UI manipulation adopted by CRiOS [67] (e.g., time interval for UI manipulation), available network bandwidth (e.g., 50mbs or 500mbs), and the vague description of the implementation of the in-device crawler proposed by Yeonjoon et al. [62].

Based on the speed we tested for downloading the 1,300 seed apps, downloading 168,951 iOS apps in the wild with a single download task and an iOS `jailbroken` device is estimated to complete in 160 (assuming 24/7 activity) days. To achieve this efficiently in practice, we combine six downloading tasks with two `jailbroken` devices for app collection. To evade iTunes' detection of our automated downloader, two Apple accounts are iteratively used to download the `.ipa` files. This scheme enables us to collect 168,951 apps within just 30 days. Overall, our app collection can significantly improve the collection rate by 17 times faster in comparison to the methodology used by Yeonjoon et al. [62], which took three months to collect only 28,625 iOS apps. We highlight that not only the decrypting process can positively contribute to the speed-up of the app collection, but our headless-downloader can also fully utilize bandwidth for parallel apps download. In summary, the scalable app collection tool, developed in this paper, enables us to complete the collection of 168,951 iOS apps.

**Ethical considerations**: We emphasize that routinely collecting and decrypting iOS apps using `jailbroken` iPhones is for the purpose of improving their service quality and security. The dataset and the research *per se* is to serve not only the research community but also to benefit the stakeholders, such as Apple.

## 4 Vetting Methodology

In this section, we introduce the vetting methodology (see the red box of Figure 4), which consists of dynamic analysis (cf. § 4.1) to select candidate apps, obtain a call stack from each app, static analysis and manual confirmation (cf. § 4.2) to scrutinize the network services of the candidate apps. The rationale behind the vetting methodology of "dynamic first, static later, and manual confirmation last" is that dynamic analysis can rapidly check for misconfigured network interfaces on a large scale, allowing us to pinpoint a small portion of candidate network service apps. The more time-consuming static analysis can then be used to perform a fine-grained analysis and check for potential vulnerabilities. Finally, we verify the identified vulnerabilities manually in order to ensure vulnerabilities are not misidentified.

### 4.1 Dynamic Analysis

Dynamic analysis is used to check for remote accessible network interfaces in the wild. Specifically, we use dynamic analysis to check which app utilizes a network service and analyze the interface of the network service while preserving the call stack of the app.

**Vetting if apps provide network services.** We leverage our dynamic analysis to detect whether apps provide network services. To provide network services, the standard process [25] in light of `POSIX Layer` (see Figure 3) is to *(i)* create a socket, *(ii)* bind it to a port, and *(iii)* begin listening for incoming connections on that port. During the second step of the process, namely invoking _`bind` API, developers can pass rich parameters, indicating the property of the network service, to the _`bind` API to limit the access scope of the network service by designating the network interface as loopback for local host access or LAN for remote access from Wi-Fi/cellular networks.

To study the interface of a network service, we implement an "addon" for `jailbroken` iOS devices by using `Cydia Substrate` [72]. The "addon" redirects the _`bind` API calls initiated by each analyzed app to the vetting code. As discussed in Section 2.3, we only consider remote adversaries because they are more practical threats to the apps. Therefore, by parsing parameters of _`bind` API, if the app uses the loopback interface (e.g., *127.0.0.1*), the vetting code considers the app as safe and terminates the analysis. For the apps that use the LAN interface, for example, a developer passes a parameter *192.168.1.3* to _`bind` API, the vetting code in "addon" reports the app is accessible (i.e., a candidate app). We later run static analysis on these apps to vet the security of the network service.

**Call stack extraction.** We carry out call stack extraction for generating unique signatures so we can identify system APIs and third-party libraries relevant to network services. For any active app, iOS maintains the runtime return address of a routine in a data structure known as the *call stack*. The call stack, filled with pointers, is depicted in the left-top box of Figure 8, where pointers indicate the site to which the routine should return when its execution is completed. Since the API _`bind` is a prerequisite for setting up a network service, to analyze the call trace reaching the _`bind` API, the call stack is preserved by our "addon" when analyzing the interface of the network service. The pointer in the call stack varies due

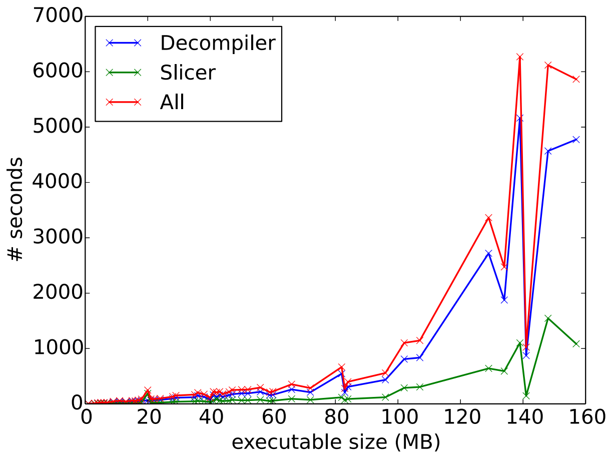


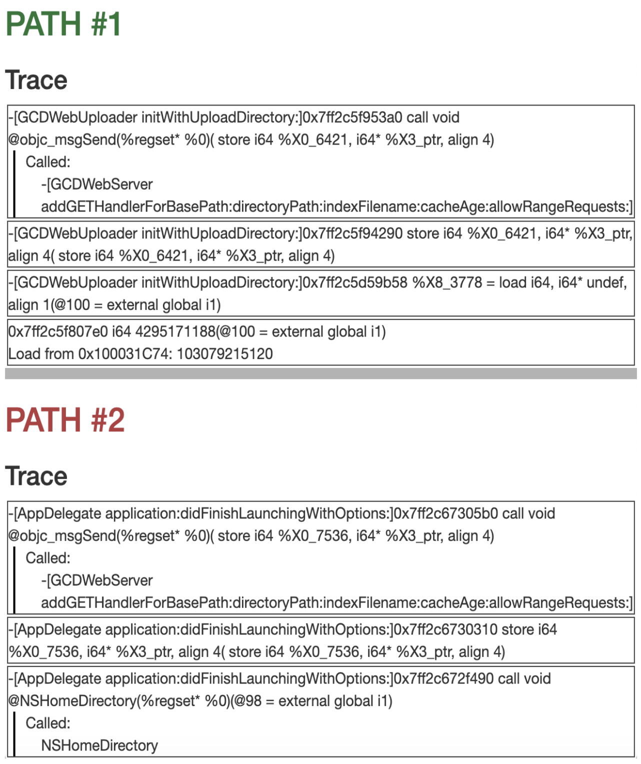
**Figure 6:** The performance of our static analyzer. After the 113 apps pass our dynamic analysis, the static analysis (including decompilation, optimization, and slicing) takes 54 minutes per app on average. The overhead of decompiler should be in line with the instructions within an executable; however, for the executable larger than 120MB, memory compression and swapping time is involved as per the exhausted memory (16G), leading to a sharp increase of the time consumption of the decompiler and overall performance. The dramatic drop at 140MB is an exception that the instructions of the app are not in line with the executable size. The overall performance benefits from the slicer (on-demand inter-procedural), with comparison to the overhead of original inter-procedural analysis [49] for program slicing, which takes in the order of days and is omitted herein.

to the Address Space Layout Randomization (ASLR) security mechanism of the iOS system. In order to map the runtime floating pointers in the call stack to the concrete offset of the static executable, the ASLR value for the executable is preserved.

**Limitations.** Region lock checks (nine apps) from either iTunes or the developer may occasionally impede the dynamic analysis. In addition, social security numbers required (29 apps) for registration process or jailbreak detection (four apps) by developers will also prevent the apps from running. These apps account for 3.2% of our 1,300 seed apps. Subject to the accuracy of UI automation [41], the dynamic analysis would involve human interaction if necessary (e.g., app registration).

## 4.2 Static Analysis and Manual Confirmation

We note that only network services behind the LAN interface can reach the static code analysis. Dynamic analysis selects candidate apps that provide network services and excludes apps that use the loopback network interface. Next, by using static analysis, candidate apps are further narrowed down by using `rules`. Static analysis results are then manually confirmed.



PATH #1

Trace

```
-[GCDWebUploader initWithUploadDirectory:]0x7ff2c5f953a0 call void
@objc_msgSend(%regset* %0)( store i64 %X0_6421, i64* %X3_ptr, align 4)
  Called:
    -[GCDWebServer
    addGETHandlerForBasePath:directoryPath:indexFilename:cacheAge:allowRangeRequests:]
-[GCDWebUploader initWithUploadDirectory:]0x7ff2c5f94290 store i64 %X0_6421, i64* %X3_ptr,
align 4( store i64 %X0_6421, i64* %X3_ptr, align 4)
-[GCDWebUploader initWithUploadDirectory:]0x7ff2c5d59b58 %X8_3778 = load i64, i64* undef,
align 1(@100 = external global i1)
0x7ff2c5f807e0 i64 4295171188(@100 = external global i1)
Load from 0x100031C74: 103079215120
```

PATH #2

Trace

```
-[AppDelegate application:didFinishLaunchingWithOptions:]0x7ff2c67305b0 call void
@objc_msgSend(%regset* %0)( store i64 %X0_7536, i64* %X3_ptr, align 4)
  Called:
    -[GCDWebServer
    addGETHandlerForBasePath:directoryPath:indexFilename:cacheAge:allowRangeRequests:]
-[AppDelegate application:didFinishLaunchingWithOptions:]0x7ff2c6730310 store i64
%X0_7536, i64* %X3_ptr, align 4( store i64 %X0_7536, i64* %X3_ptr, align 4)
-[AppDelegate application:didFinishLaunchingWithOptions:]0x7ff2c672f490 call void
@NSHomeDirectory(%regset* %0)(@98 = external global i1)
  Called:
    NSHomeDirectory
```

**Figure 7:** The static analysis result of the misuse of `GCDWebServer` in the `Now` app. The green item indicates a harmless usage of this library. The brown item reports another misuse of this library.

**Static analysis.** During the iOS app development, developers use a mixture of Objective-C and C or SWIFT to compose an app. To automatically analyze Objective-C and SWIFT binary, we opt to further optimize the open-source framework [49], which is a static slicer for inter-procedural data-flow analysis on LLVM IR of 64-bit ARM binary. Specifically, three phases are involved in analyzing an iOS app, i.e., decompiling machine code to LLVM IR by using DAGGER [9], optimizing the IR, and slicing on the IR. To adapt this framework to our analysis task, we attempted to enhance the framework from the following aspects.

*(i)* We supplement semantics of more ARM instructions to the decompiler. Additionally, since the IR of a moderate app always consumes gigabytes of memory, some instructions are simplified to shrink the memory usage, such as removing floating point instruction. The simplification has little effect on the analysis results.

*(ii)* We convert inter-procedural data-flow analysis to on-demand inter-procedural [70]. The complexity of point-to analysis in slicing is $O(n^3)$ [21], where $n$ is over ten million for a moderate app when performing inter-procedural analysis. This makes original analysis take several days to analyze an app. To speed up the performance, the on-demand inter-procedural analysis starts analyzing the function enclosing the reference to the expected class object name or method name of a network service API. After slicing on the function and the

callees (functions) are solved, it takes in all identified callees to start another slicing iteration. This strategy significantly reduces the *n* of point-to analysis. The overall performance of the static analyzer is depicted in Figure 6. We show that the overhead of the decompiler and slicer is almost linear in terms of the executable file size, and the slicing phase is bounded within a constant-time overhead.

*(iii)* We formulate and specify `rules` for the misuse of network services. For example, the static analysis result of the misuse of the `GCDWebServer` library is depicted in Figure 7. In comparison to dynamic analysis which investigates the network interface of an app, static analysis can check if the root folder of the web server is a `data container` directory, or a `bundle container` directory by using rule. The code of our static analyzer is publicly available at `https://github.com/pwnzen-mobile`.

**Manual confirmation.** To date, as the automated analysis is unable to verify iOS network service vulnerabilities end-to-end, we resort to six expert researchers (three co-authors and three external experts) to identify private (e.g., cookies) or non-private (e.g., video clips) information, privileged functionality (e.g., install apps) exposure, and to study how to build a request to bypass the weak protection (e.g., hard-coded passwords) with the help of static analysis. The six expert researchers are separated into three groups and each group reports if the apps are considered vulnerable. Specifically, we focus on remote vulnerabilities for exploits. For example, although `Waze` provides a network service on port 12345 for the LAN interface and 55432 for the loopback interface simultaneously on startup, we only check the network service on port 12345. If private information or privileged functionality is exposed to cellular networks via a network service, we rank the vulnerable network service as high risk. If it is exposed to Wi-Fi networks, we rank the network service as medium risk. For non-private or non-privileged functionality, we rank the network service as low risk. For example, obtaining video snippets from the `Prime Video` app without authorization is ranked as low risk, since the video snippets are considered non-private. After generating all reports, the researchers discuss and finalize ranking the vulnerabilities.

**Limitations.** The static analysis is efficient to identify security risks. Two types of divergence may occur in the static analysis: *(i)* Our on-demand inter-procedural analysis may result in loss of precision, subject to the failure of parsing 8.7% apps, leading to a false positive rate of 20.5%; (ii) 29.4% libraries implemented in C fail to be parsed through our static analyzer.

## 4.3 Results of Vetting

In this subsection, we present the results of our dynamic and static analysis and our six expert researchers' verification. This process is performed on seed apps. Even with manual confirmation (done by six expert researchers), the entire vetting process for the 1,300 apps can be completed within 15 days. Dynamic analysis takes 2 days with one `jailbroken` iOS device (may need interaction) and static analysis, including the manual confirmation, takes 13 days.

**Table 2:** The results of our dynamic analysis of the apps obtained in the China and United States.

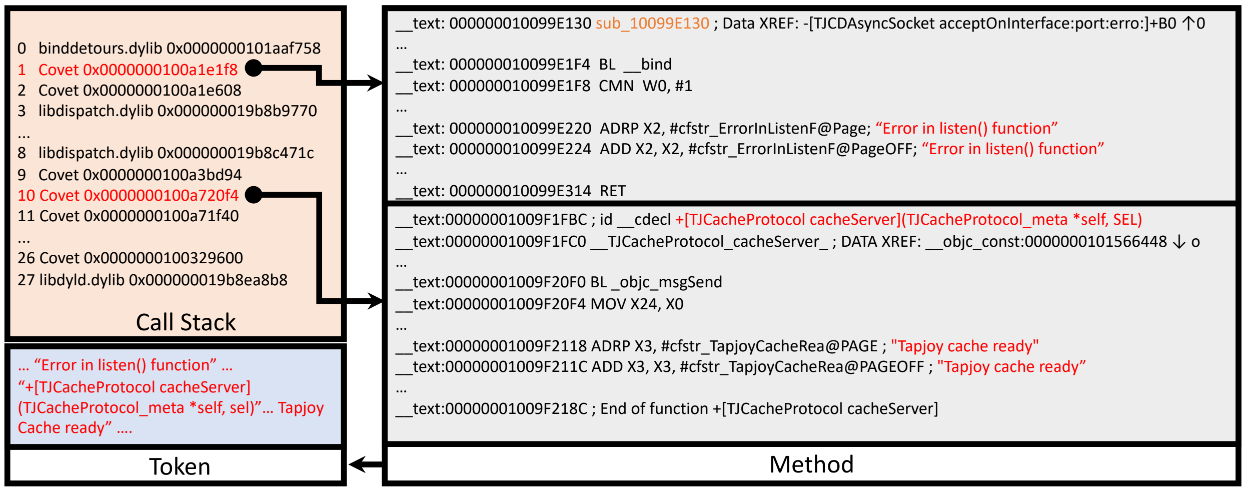
| | Dynamic Port (0) | Loopback Interface (e.g., 127.0.0.1) | LAN Interface |
|---|---|---|---|
| **China (480)** | 16 (3.33%) | 14 (2.91%) | **51 (11.04%)** |
| **United States (820)** | 42 (5.12%) | 43 (5.24%) | **62 (7.01%)** |
| **Total (1,300)** | 58 (4.46%) | 57 (4.38%) | **113 (8.69%)** |

**Results of dynamic analysis.** For the dynamic analysis, we install, launch, and uninstall each iOS app automatically by using `ideviceinstaller` [12]. When the app reaches the main view, we end the dynamic vetting process, and collect the call stack of each analyzed app. Overall, 172 unique apps, 13.2% of our collected total, provide network services for either local or remote clients. Table 2 shows the details of our dynamic analysis. Our observations are as follows: *(i)* The dynamic port (the second column of Table 2) to which a socket binds is usually used for in-app communication, and the network service on a dynamic port is immune to attacks; *(ii)* the apps that provide network services on multiple interfaces will be represented in each column; therefore, a unique app can be counted multiple times in this table. We found 65 unique apps from China and 107 from the United States that provide network services; and *(iii)* the analysis process was always performed in a Wi-Fi network. As shown in the last column of Table 2, a huge number (113) of iOS apps provide network services to other hosts in the same Wi-Fi networks, accounting for 8.69% of the 1,300 seed apps. Since developers can adjust their network services for different networks (i.e., Wi-Fi networks and cellular networks), the network services exposed to cellular networks are less than 8.69%. Compared to the apps in the United States, the apps in China are more inclined to provide network services on the LAN interface. That is, 11.04% vs. 7.01%.

**Results of static analysis and manual confirmation.** Based on dynamic analysis, we select candidate apps to examine in depth by static analysis and verify exploitable network services by six exports' confirmation. Ultimately, we confirmed that 11 (9.7%) of the 113 candidate apps have vulnerabilities, such as `Waze`, `QQBrowser`, `Now`, `Scout GPS Link`, and `Youku`. These vulnerable apps are described in Section 7.

## 5 Building Signatures for Network Services

Two types of interfaces are available for developers to start up network services: by invoking system network service APIs or by using third-party libraries (see "app code" of Figure 3). For large-scale analysis of apps across categories, we build



```
0   binddetours.dylib 0x0000000101aaf758
1   Covet 0x0000000100a1e1f8
2   Covet 0x0000000100a1e608
3   libdispatch.dylib 0x000000019b8b9770
...
8   libdispatch.dylib 0x000000019b8c471c
9   Covet 0x0000000100a3bd94
10 Covet 0x0000000100a720f4
11 Covet 0x0000000100a71f40
...
26 Covet 0x0000000100329600
27 libdyld.dylib 0x000000019b8ea8b8
```

```
__text: 000000010099E130 sub_10099E130 ; Data XREF: -[TJCDAsyncSocket acceptOnInterface:port:erro:]+B0 ↑0
...
__text: 000000010099E1F4  BL  __bind
__text: 000000010099E1F8  CMN  W0, #1
...
__text: 000000010099E220  ADRP X2, #cfstr_ErrorInListenF@Page; "Error in listen() function"
__text: 000000010099E224  ADD X2, X2, #cfstr_ErrorInListenF@PageOFF; "Error in listen() function"
...
__text: 000000010099E314  RET
```

```
__text:00000001009F1FBC ; id __cdecl +[TJCacheProtocol cacheServer](TJCacheProtocol_meta *self, SEL)
__text:00000001009F1FC0 __TJCacheProtocol_cacheServer_ ; DATA XREF: __objc_const:0000000101566448 ↓ o
...
__text:00000001009F20F0 BL _objc_msgSend
__text:00000001009F20F4 MOV X24, X0
...
__text:00000001009F2118 ADRP X3, #cfstr_TapjoyCacheRea@PAGE ; "Tapjoy cache ready"
__text:00000001009F211C ADD X3, X3, #cfstr_TapjoyCacheRea@PAGEOFF ; "Tapjoy cache ready"
...
__text:00000001009F218C ; End of function +[TJCacheProtocol cacheServer]
```

... "Error in listen() function" ... "+[TJCacheProtocol cacheServer] (TJCacheProtocol_meta *self, sel)"... Tapjoy Cache ready" ....

**Figure 8:** Overview of call stack analysis of `Covet Fashion` app. The subfigures on the left show the call stack and the extracted token for analyzing, the arrows indicate the returned address of a routine (right subfigure).

signatures for system network service APIs and third-party libraries (see the red box of Figure 4).

## 5.1 Signatures of System APIs

System network service APIs and corresponding signatures are built on the call stack information recorded by our "addon" in our dynamic analysis phase. Specifically, we navigate the call stack to locate the system APIs and build hybrid signatures for the APIs.

**Identifying system APIs.** The challenge for identifying system network services is that there is no clear documentation that details the effects of API calls. For example, the API `registerListener:` of class object `GKLocalPlayer` spawns a port to provide the network service, but the official documentation does not mention the network service behind the API. Therefore, we identify the system network service APIs by leveraging the call stack information of the dynamically analyzed apps. Specifically, we travel the pointer in the call stack from top to bottom until we find the API the app code invoked. As shown in the top-left box of Figure 8, we travel the call stack from item 0 to 27, and stop traveling at item 1 as this pointer points to app code. By checking the target API of the app code invoked (top-right box of Figure 8), we get the system API (i.e., `_bind`).

**Building signatures for system APIs.** The identified system APIs, presented as signatures for network services, can be used to determine whether the app is a potential network service app. There are two strategies for representing these APIs: *(i)* For network services provided by utilizing `POSIX` and `Core Foundation` [25], the API (e.g., `_bind` in Table 3) is directly called by app code. In this case, the code for invoking APIs is directly assembled in the executable. By querying for this code in metadata preserved in our database, we know there is a network service in app or not. *(ii)* For the Objective-C APIs provided by other system frameworks, developers have to pass a message to an object through message dispatch interface (e.g., `_objc_msgSend`) to invoke the API. In this circumstance, the first and the second arguments of the message dispatch interface represent the instance of a class (e.g., `_OBJC_CLASS_$_GKLocalPlayer` in Table 3) and a method (e.g., `registerListener:` in Table 3), respectively. This class and method combination designates the real API being invoked. Hence, for the APIs of Objective-C, class object name in "Symbol Table" and the method name in "String Table" are used as signatures (see column 2 of Table 3).

## 5.2 Signatures of Third-Party Libraries

Developers often use off-the-shelf third-party libraries to provide network services rather than building a server from scratch [27, 54]. There are many third-party network service libraries that reside on GITHUB or other repositories to help developers perform quick network service integration for their apps. For example, iOS app developers may opt for `CocoaHTTPServer` [7] to provide web services. In order to figure out the real distributions of third-party libraries in iOS apps and extend our findings of the vulnerable libraries to the whole dataset, we firstly identify third-party network service libraries and extract signatures for these libraries. Previous work on Android third-party library identification [27, 76] is based on structurally organized code, (e.g., package), which does not scale well to iOS third-party library identification. Because there is no structure information preserved in the iOS executable, the developer's code and the statically linked third-party libraries are assembled into an executable binary file with no clear boundary. To find third-party libraries of iOS apps, the proposed class name cluster method [67] expends enormous effort in building every library. But among these libraries, there are storage libraries for processing data,

**Table 3:** Signatures for system network service APIs and the network service distributions in iOS apps.

| Library (a.k.a., Framework) | Signatures | Location | China (480) | United States (820) | 1,300 apps | 168,951 apps |
|---|---|---|---|---|---|---|
| **libSystem** | _bind | Symbol Table | **353 (73.54%)** | **331 (40.37%)** | **684 (52.62%)** | **69,238 (40.98%)** |
| **libresolv** | _res_9_nquery | Symbol Table | 56 (11.67%) | 1 (0%) | 57 (4.38%) | 1,481(0.88%) |
| **CoreFoundation** | _CFSocketSetAddress | Symbol Table | 112 (23.33%) | 57 (6.95%) | 169 (13%) | 11,965 (7.08%) |
| **GameKit (1)** | _OBJC_CLASS_$_GKLocalPlayer | Symbol Table | 0 (0%) | 10 (1.22%) | 10 (0.77%) | 2,673 (1.58%) |
| | localPlayer | String Table | | | | |
| | registerListener: | String Table | | | | |
| **GameKit (2)** | _OBJC_CLASS_$_GKMatchmaker | Symbol Table | 1 (0%) | 12 (1.46%) | 13 (1%) | 5,580 (3.3%) |
| | sharedMatchmaker | String Table | | | | |
| | setInviteHandler: | String Table | | | | |
| **MultipeerConnectivity** | _OBJC_CLASS_$_MCSession | Symbol Table | 10 (2.08%) | 3 (0.37%) | 13 (1%) | 604 (0.36%) |

UI libraries for prettified views, etc. The third-party network service library is a subset of the whole library repository.

To identify the third-party network service libraries, we propose *call stack similarity analysis*, which is mainly used for hunting similar bugs [39, 65], to identify these libraries. Our call stack analysis is based on the runtime properties of a program. After the third-party network service libraries are identified, we use *Information Gain* [59] to select the most prominent signatures for these libraries.

**Identifying third-party libraries.** The top-left box of Figure 8 shows that there are no rich information in the call stack $C$. Consequently, we map the call stack to the executable with the help of the ASLR value preserved in our dynamic analysis phase. We collect the strings $s_{ii}$ (e.g., "Error in listen() function" in Figure 8) in each method (e.g., "sub_10099E130" in Figure 8) that the pointers in the call stack point to in order to build a token $t_i$. All $t_i$ acquired are concatenated to generate a longer token $T$ (bottom-left box of Figure 8). Considering that the app code the pointers point to is always a mixture of developer's code and third-party library's code, so the token $T$ collected is a mixture of $t_i$ from developer's code and third-party library's code. For example, the pointers in the call stack of the `Covet Fashion` app in Figure 8 point to libraries `Cocoa Async Socket` (1, 2, 5), `CocoaHTTPServer` (6, 9), `Tapjoy` (10, 11, 16, 17), and developer's code (26) respectively; the token $t_i$ in developer's code (26) will affect similarity analysis since developer's code varies in different apps. To reduce noise induced by developer's code in similarity analysis, we propose a *weighted edit distance* algorithm to focus on the third-party library's code.

Since the third-party library's code is pointed by pointers at the top of the call stack, the token $t_i$ related to the top of the call stack is assigned a larger weight $w_i$, and vice-versa. To factor in the weight, we duplicate $t_i$ multiple times according to the $w_i$ assigned to the token and then get a new longer token $T'$. After that, we measure the similarity ratio $R$ of call stacks by using different $T'$. In practice, we adopt a *Levenshtein edit distance ratio* [63] algorithm, that is

$$DistanceRatio(a,b) = 1 - \frac{EditDistance(a,b)}{|a|+|b|} \tag{1}$$

where $a$ and $b$ denote two tokens $T'$, respectively. The whole process is described in Algorithm 1.

**Algorithm 1** Weighted edit distance for identifying third-party network service libraries

```
Input: Call stack: $C_1$, $C_2$; Token for call stack: $T_1$, $T_2$;
Output: Weighted edit distance of the two call stacks: $R$;
 1: $W \leftarrow Max(Len(C_1),\ Len(C_2))$
 2: $T_1' \leftarrow$ GET_WEIGHTED_TOKEN$(W, C_1, T_1)$
 3: $T_2' \leftarrow$ GET_WEIGHTED_TOKEN$(W, C_2, T_2)$
 4: $R \leftarrow Levenshtein.ratio(T_1', T_2')$
 5: function GET_WEIGHTED_TOKEN$(W, C, T)$
 6:     for each $i \in [0, W-1]$ do
 7:         $w_i \leftarrow W - i$
 8:         $t_i \leftarrow T[i]$
 9:         $t_i' \leftarrow Duplicate(t_i, w_i)$
10:         $T' \leftarrow Concatenate(T', t_i')$
11:     return $T'$
```

The weighted edit distance can increase the edit distance ratio $R$ of the call stacks for the same third-party network service library in different apps, but has slightly less influence for different libraries (see Table 4). Empirically, we tune the parameter and finally consider as a third-party network service library if the ratio $R \geq 0.6$. Note that, the threshold is tuned to optimize the library identification. The results obtained are not overly-sensitive to the different thresholds chosen.

**Building signatures for third-party libraries.** In practice, if the similarity of two stacks reaches the threshold, the code pointed by the stack is considered as third-party libraries. Then we inspect the corresponding apps and tag the identified third-party network service libraries by searching GITHUB or GOOGLE. The most straightforward way to find the in-app network service is to identify the developer's code that exactly invokes the third-party network service API. However, this approach could be very time-consuming to scale up because it needs an extensive analysis of each app to build the API invocation due to the Objective-C runtime property, `message dispatch` [47, 67]. To address the challenge, we propose to use the string $s_{ii}$ relevant to the third-party library to generate a signature instead.

By leveraging the TF/IDF algorithm in GENSIM [71], we evaluate each $s_{ii}$ (bottom-left box of Figure 8) and obtain the prominent $s_{ii}$, which is used for identifying third-party network service libraries. Finally, we obtain a $< signature, tag >$ tuple for each library. For example, the `GCDWebServer` library is presented as <"%@ started on port %i and reachable at %@", "GCDWebServer">.

**Table 4:** Edit distance/weighted edit distance ratio *R* of call stack for third-party network service libraries.

| **Edit distance/ Weighted edit distance** | QQBrowser (CocoaHTTPServer) | Taobao4iPhone (wangxin.taobao) | Libby (GCDWebServer) | QQSports (TencentVideoHttpProxy) |
|---|---|---|---|---|
| bbtime (CocoaHTTPServer) | **0.74/0.82** | 0.16/0.18 | 0.36/0.37 | 0.28/0.28 |
| Tmall4iPhone (wangxin.taobao) | 0.16/0.18 | **1.00/1.00** | 0.19/0.22 | 0.12/0.18 |
| NOW (GCDWebServer) | 0.37/0.37 | 0.17/0.19 | **0.89/0.91** | 0.30/0.29 |
| KuaiBao (TencentVideoHttpProxy) | 0.31/0.31 | 0.15/0.20 | 0.30/0.30 | **0.54/0.66** |

By using signatures of third-party network service libraries, we can execute a large-scale analysis of iOS apps and push forward the analysis boundary from the system APIs to third-party network service libraries (e.g., Section 7.2). Furthermore, the extracted signatures enable us to apply association analysis to figure out the relation between these third-party network service libraries.

In summary, the proposed library identification approach is specifically designed for a call trace which reaches the `_bind` API. The third-party library to which the call stack points is a network service library. This approach outperforms the cluster-based method [67] by utilizing lower complexity (unnecessary to build the third-party library corpus before extracting network service libraries) and high precise (e.g., identifies the library `Unreal Engine 4` which provides the network service but is commonly known as a game library) analysis.

## 5.3 Results of Building Network Service Signatures

Using the proposed methodology, we identify six system APIs and 34 third-party libraries by analyzing the call stacks of seed apps. System network service APIs are collected by traveling the call stack. The results are shown in the first two columns of Table 3. Third-party network service libraries are collected by analyzing the similarity of the call stack. The results are shown in x-axis of Figure 10 and Table 7 in the Appendix.

Given that there is no ground truth for the identification of network services, each app must be inspected to confirm the existence of network service usage. Unfortunately, inspecting more than one thousand apps is tedious and time consuming, so we instead chose to randomly sample 130 apps (10%) from the seed dataset. Each of the six expert researchers separately inspected each app and identified the use of system APIs and third-party libraries. Our analysis of the randomly sampled dataset suggests 100% accuracy, with 0% disagreement among the expert researchers, showing the effectiveness of our proposed system. Although the perfect accuracy would probably not be supported through verification of every app that we collected, with more time and effort, manual verification of a sample size greater than 400 apps ($>$ 30%) would give a more pronounced success rate. Furthermore, experimental results show that among the 1,300 apps, none of the apps is obfuscated, suggesting that obfuscation is not wildly applied to iOS apps to affect the analysis result (the detail is available on our website). *We highlight that currently there is no benchmark dataset publicly available for any accuracy comparison of other iOS library identification approaches.*

## 6 Large-Scale Analysis of Network Services

We begin by analyzing the prevalence of the network service use in iOS apps. By taking signatures of APIs and libraries, we query the metadata of the collected apps stored in our database to find the percentage of apps that may use network services. We further analyze the association or interdependencies among these third-party network service libraries, in assistance with the extraction of apps for subsequent analysis. We highlight our main results in the remarks.

***(i)* System network service APIs.** To reveal the portion of iOS apps that make use of network services, we use the API signatures collected from the seed apps to query our database (see query result breakdowns in Table 6 in the Appendix). Apps assembling these APIs are potentially ready to start network services. As shown in Table 3 (columns 4 and 5), most of the apps follow the guidance of [25]; specifically, using the API `_CFSocketSetAddress` of `Core Foundation socket` and the API `_bind` of `BSD sockets` can compose a network service. `_res_9_nquery` is an undocumented API used by iOS apps. Although Apple has documented the remaining three APIs, it does not clarify whether these APIs provide the network services.

Compared to the dynamic analysis results shown in Table 2, our query found several-fold more apps capable of invoking system APIs for network services. We believe the reasons for the discrepancy are as follows: *(i)* The code snippet for invoking a system API for network services may be dead (i.e., unused or dummy) code; *(ii)* UI interaction may hinder execution of the code snippet that invokes these APIs, so dynamic analysis fails to pick it up.

The percentage of apps using network services decreases from 52.62% when querying the 1,300 seed apps to 40.98% when querying the 168,951 iOS apps (see the last two columns of Table 3), since general apps are not as fully-featured as many of the top rate apps. Results grouped by category reveal that different categories of apps exhibit markedly different trends in their use of network services. Most iOS apps in the "Game" category are inclined to provide network services for multi-peer connection. These apps account for over 60% of

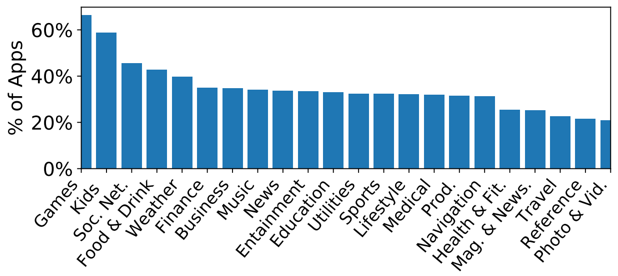


**Figure 9:** Network services across app categories.

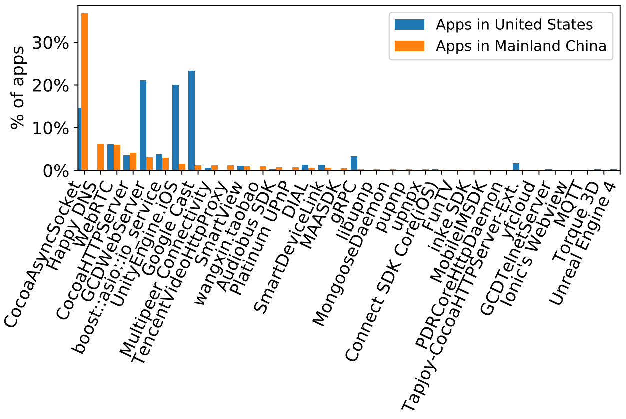


**Figure 10:** Third-party network service libraries detected in the seed apps.

the designated categories. The categories “Reference” and “Photo & Video” are comparably less likely to provide network services. Other libraries are distributed uniformly in different categories. The query results are depicted in Figure 9.

**Remark 1.** Network services are prevalent in iOS apps. 40.98% apps potentially invoke system APIs to provide network services. The results show that almost every top popular app in China (73.54%) contains code to start a network service. Queries further reveal that China apps are almost twice as likely to invoke network service APIs than their US counterparts (over 73.54% vs. over 40.37%).

***(ii)* Third-party network service libraries.** iOS apps commonly integrate third-party libraries to provide their network services. In order to characterize the distribution of third-party network service libraries in iOS apps, we query the third-party libraries in top popular apps by using the collected signatures. As shown in Figure 10, we note that *(i)* as a basic support for establishing network services, `CocoaAsyncSocket` is a prevalent used third-party library in both the United States and China. *(ii)* Apps from the United States are more willing to integrate the `GCDWebServer`, `Google Cast`, and `UnityEngine.iOS` third-party libraries. *(iii)* Due to poor accessibility of network resources in China, the `Google Cast` library is rarely used in apps from China. Libraries in China are largely more scattered in all categories than those in the United States. We further

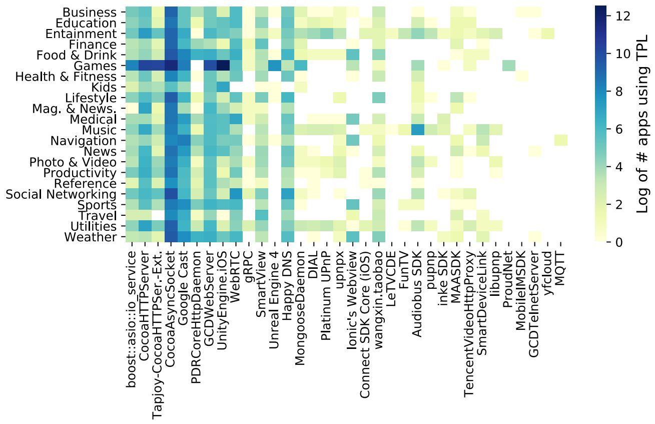


**Figure 11:** Third-party network service libraries across app categories. The color encodes the logarithm of the number of apps ($\log_2$(# apps)) using third-party libraries.

extend our analysis to the 168,951 iOS apps, and the results are grouped by the category of iOS apps (see breakdowns in Table 7 in the Appendix). As shown in the corresponding heatmap of Figure 11, we have the following observation.

**Remark 2.** Apps in the “Game” category are most likely to use third-party libraries. Besides the libraries of `CocoaAsyncSocket` and `UnityEngine.iOS`, the “Game” category mainly uses `CocoaHTTPServer` and `Tapjoy-CocoaHTTPServer-Extension` libraries. Among the top five used network service libraries, there are third-party libraries `CocoaHTTPServer` and `GCDWebServer`, providing various interfaces for developers to customize (e.g., designate the access interface, specify resources/functionalities) the network services. This may potentially lead to the library misuse.

***(iii)* The dependency relationship of network service libraries.** The error-prone use of third-party libraries (e.g., `GCDWebServer`, `CocoaHTTPServer`) are widely used in iOS apps. It is likely that these third-party network service libraries are supporting infrastructure for other libraries. We use the FP-GROWTH algorithm [52] to mine the association of third-party libraries. The rules discovered by FP-GROWTH is listed in Table 5. From Table 5, we find dependencies between different third-party network service libraries. For example, the dependency of `Tapjoy-CocoaHTTPServer-Extension` can be depicted as `Tapjoy-CocoaHTTPServer-Extension` $\longrightarrow$ `CocoaHTTPServer` $\longrightarrow$ `CocoaAsynSocket` $\longrightarrow$ `_bind` (lines 1, 5, and 11 in Table 5). This is verified by checking source code of this library. Even for closed source libraries, we know the dependency of the libraries from the table. For example, analysis result reveals that the closed source library `TencentVideoHttpProxy` is built on top of the open source `CocoaAsyncSocket` library (line 14). In the real world, the relations of third-party network service libraries are shown in the blue box of Figure 3. We also find the prevalent usage of `Happy`

**Table 5:** Association of third-party network service libraries and system network service APIs.

| # | Library/API | Library/API |
|---|---|---|
| 1 | Tapjoy-CocoaHTTPServer-Extension | CocoaHTTPServer |
| 2 | Tapjoy-CocoaHTTPServer-Extension | CocoaAsyncSocket |
| 3 | PDRCoreHttpDaemon | _CFSocketSetAddress |
| 4 | Ionics_Webview | GCDWebServer |
| 5 | CocoaHTTPServer | CocoaAsyncSocket |
| 6 | Happy_DNS | _res_9_nquery |
| 7 | MAASDK | CocoaAsyncSocket |
| 8 | Ionics_Webview | _bind |
| 9 | wangxin.taobao | _CFSocketSetAddress |
| 10 | MongooseDaemon | _bind |
| 11 | CocoaAsyncSocket | _bind |
| 12 | Tapjoy-CocoaHTTPServer-Extension | _bind |
| 13 | CocoaHTTPServer | _bind |
| 14 | TencentVideoHttpProxy | CocoaAsyncSocket |
| 15 | Platinum_UPnP | _bind |
| 16 | GCDWebServer | _bind |
| 17 | upnpx | _bind |
| 18 | DIAL_UPnP | _bind |
| 19 | WebRTC | _bind |
| 20 | SmartDeviceLink | _bind |
| 21 | Connect_SDK_Core_(iOS) | DIAL |
| 22 | FunTV | CocoaAsyncSocket |
| 23 | Unreal_Engine_4 | Game_Kit_(2) |
| 24 | TencentVideoHttpProxy | CocoaHTTPServer |
| 25 | wangxin.taobao | _bind |
| 26 | UnityEngine.iOS | _bind |

DNS library (demonstrated in Figure 10) in China leads to the prevalent usage of undocumented API usage _res_9_nquery in iOS apps (line 6). Based on the relation of network service libraries, we find that the widely used Ionic's Webview is built on top of the GCDWebServer. As the most recent version of Ionic's Webview has been adjusted to use loopback interface when integrating GCDWebServer, we can skip checking the apps using both GCDWebServer and Ionic's Webview libraries.

## 7 Determining iOS App Vulnerabilities

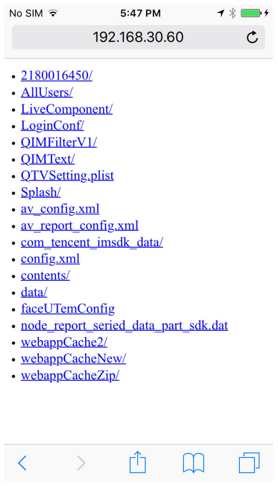
In this section, we first closely examine the network service vulnerabilities discovered after vetting the 1,300 seed apps. We summarize four categories of vulnerabilities, and explain the details of two real-world vulnerable apps acknowledged by vendors, which are Waze, Now, and QQBrowser. Note that the acknowledgment of some vulnerabilities is pending. We finally scrutinize the vulnerabilities of 3 typical network service libraries in 168,951 iOS apps and discuss the underlying reason.

### 7.1 Vulnerabilities in Seed Apps

Previous 11 vulnerabilities identified among the 1,300 seed apps fall into four categories: *(i)* Connected with an IoT device with no/weak access control (Waze and SCOUT GPS LINK). *(ii)* Served as a command server to execute command per the client's request (QQBrowser, Taobao4iPhone, and Youku). *(iii)* Served as a file server to share files between a desktop computer and an iOS device (Now). *(iv)* Served as a content distribution networks (CDN) node to share videos with other peer devices. We regard these vulnerable apps (Amazon Prime Video, QQSports, etc.) of this category as low risks since the video clips shared are usually non-private.

*(i)* **Remote Command Execution and Denial-of-Service: A case of an iOS app connected with an IoT device with no/weak access control.** To connect with an IoT device, vulnerable apps always turn the iOS device to be a server. Two vulnerable apps, Waze and SCOUT GPS LINK, provide network services in the LAN interface for the IVI system, but these apps provide little to no access control. For example, Waze is a popular community-based traffic and navigation app in the United States. Dynamic analysis reveals that the app starts network service on port 12345 through the LAN interface. We also find that the network service on port 12345 accepts any connection attempts, and processes remote command messages in which a valid command message starts with "WL". The potential threats with the Waze network service are as follows. *(i)* For any incoming message (see $M4$ listed in Figure 2) starting with "WL", Waze will cache the message until the memory resource is exhausted (see $M3$ listed in Figure 2). Attackers can then drain the network traffic to crash the app remotely ($A3$). *(ii)* A message with format "WL|msgID|msgSize|msg" can be accepted by Waze and a malformed overlong message will lead to remote memory corruption, including OOB (out-of-boundary) access or UAF (use-after-free) ($A3$). *(iii)* The message "msgID" set to 48 can be used to send touch events to manipulate the app. The message can further reset the destination to maliciously navigate an end-user to alternative places ($A1$). The network service is pervasive, such that an attacker can even probe and attack iOS device with Waze running in cellular networks ($A4$). We have to mention that this vulnerability only exists in the iOS version of Waze since the Android version does not provide these network services. We reported the vulnerability to Waze company acquired by Google, Waze fixed this security issue three days after we reported, and Google finally acknowledged the vulnerability.

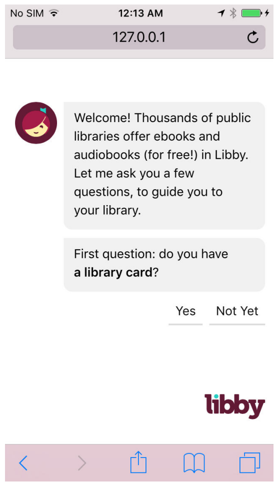
*(ii)* **Data Leakage: A case of sharing files between a desktop computer and an iOS device.** The current inconvenient file sharing of the iTunes client provides a chance for developers to ease the sharing process for users. Some apps turn an iOS device to be a web server for file sharing. The privacy-preserving sharing should be considered for access control but Now breaks the rule for file sharing. Now is a live broadcast and a popular social networking app in China. Dynamic analysis discovers that the app provides the network service on port 8080. Static analysis later reveals that the app sets the root folder of the network service to data container directory when using the third-party library GCDWebServer


No SIM 5:47 PM

192.168.30.60

- 2180016450/
- AllUsers/
- LiveComponent/
- LoginConf/
- QIMFilterV1/
- QIMText/
- QTVSetting.plist
- Splash/
- av_config.xml
- av_report_config.xml
- com_tencent_imsdk_data/
- config.xml
- contents/
- data/
- faceUTemConfig
- node_report_seried_data_part_sdk.dat
- webappCache2/
- webappCacheNew/
- webappCacheZip/

(a) `Now` app exposes content in its `data container`


No SIM 12:13 AM

127.0.0.1

Welcome! Thousands of public libraries offer ebooks and audiobooks (for free!) in Libby. Let me ask you a few questions, to guide you to your library.

First question: do you have a library card?

Yes Not Yet

libby

(b) `Libby` app starts a web server on the loopback interface

**Figure 12:** Safari web-browser used to access the network service in the same host and Wi-Fi network.

(see Figure 7). With this service, the app allows an unauthorized attacker ($M2$) to access the credentials within the `data container` directory of the app ($M1$). Data exposed by the app is depicted in Figure 12(a). By downloading credentials from the victim and uploading the acquired credentials to the attacker's device, the attacker can sign in the app using the victim's identity to perform in-app purchases with the pre-deposit money ($A1$). Similar to the `Waze` vulnerability, the attacker can scan the cellular network to identify the victims ($A4$). The vendor of this app, "Tencent Technology (Shenzhen) Company Limited," ranked this security issue as a high risk. This vulnerability was patched by switching off the relevant functionality remotely after we reported.

*(iii)* **Remote Command Execution: A case of an iOS app executing command per the client's request.** iOS apps may provide network services for end-users to manage the apps. However, the weak authorization may expose the services to any host in the same network with the victim. For example, an attacker can remotely compromise an iOS device by exploiting the flaw, e.g., the "exit" command disables the network service in `Youku` or "set" command controls UDID of `Taobao4iPhone`. Besides these two apps, a high risk vulnerability is discovered in the `QQBrowser` app. `QQBrowser` is a popular app, especially in China. Previous work showed the `QQBrowser` network service vulnerability in the Android app [32]. By exploiting the vulnerability discovered in the Android app, an attacker can remotely perform unauthorized sensitive data access (e.g., obtain the app list or app setup) on the Android device. However, the same vulnerability has not been patched for the iOS counterpart. The port 8786 is used for connecting for Android, whereas the port 13145 is for iOS. Android implements the network service on `NanoHTTPD` library, whereas the iOS network service is established through an open source repository `CocoaHTTPServer`.

On top of the HTTP server, the iOS app provides the network service for two commands (i.e., "url" and "installurl"). Apart from the "installurl" command that drives the app to navigate to the items in iTunes, there are 9 additional sub commands behind "url" that provide more functionality, such as "tel" for dialing a specific number and "sms" for sending sms message ($M1$). These sub commands are enclosed in the body of a post request. In order to ensure the validity of each post request for the network service, the app enforces a Triple DES encryption to each post request and body data. However, the key (`kM7hYp8lE69UjidhlPbD98Pm`) for decryption is hard-coded in the app code ($M2$). This weak authorization can be bypassed by building a valid request for an attacker ($A2$). We demonstrate an example that the valid request would trigger the app to dial "10086": `requests.post(http://+ip+:13145/+encrypt_3des(data=send?uuid=a8f349666b833151a861e8beb611f21a&type=url, key = key), data = encrypt_3des(data='tel:10086', key = key), headers=headers).` We have reported this vulnerability to "Tencent Security Response Center," which has ranked this security issue as a high risk. This vulnerability has already been patched in the most recent version.

## 7.2 Extensible Vulnerabilities of Affected Network Service Libraries

Through the lightweight large-scale analysis, we identify apps that use system APIs or integrate third-party libraries for network services. To have a better understanding of the network service vulnerabilities in the wild, we carry out static analysis of 2,116 apps, filtered out from the whole dataset, by using only the signatures of `WebLink` (3 apps), `libupnp` (16 apps), and `GCDWebServer` (2,097 apps). In the C library (e.g., `WebLink` and `libupnp`) vetting process, we manually verify the vulnerability; for Objective-C library, `GCDWebServer` for instance, we perform static analysis by using our static analysis tool. Dynamic analysis acts as an auxiliary for manual confirmation. We further identify an additional 92 vulnerabilities, and summarize them into three categories: *(i)* using vulnerable libraries, *(ii)* the abuse out-of-date vulnerable libraries, and *(iii)* the misuse of libraries.

*(i)* **Using the vulnerable WebLink library.** `WebLink` library, which renders the `Waze` app vulnerable, is used by another 3 apps: `WebLink for KENWOOD`, `WebLink for JVC`, and `WebLink Host`. In order to project a smart phone's screen to in-vehicle infotainment (IVI) systems, the `WebLink` library creates a virtual app screen on the IVI systems. To receive the commands from the virtual screen, it turns the iOS device into a server. By using this library, the app can capture user interactions on IVI systems. After studying this service, we find developers of the `WebLink` library make two mistakes in the design of the library. *(i)* Vendors mistakenly take the virtual screen and the smartphone as two logically separate devices as they use the LAN interface for connection ($M4$).

In fact, the virtual screen is a projector of the smartphone. *(ii)* There is no authorization required for executing the restricted functionality ($M2$), such that adversaries can remotely connect the smartphone via these apps and send touch events to manipulate ($A1$) or crash the app ($A3$).

*(ii)* **Abusing the out-of-date vulnerable portable UPnP library.** UPnP is a protocol that enables discovery, event notification, and control of devices over a network, independent of the operating system, programming language, or physical network connection. Portable UPnP SDK as known as libupnp [1] implements UPnP. Many projects, such as HD Network DVD Media Player, aMule CVS tarballs, are built on top of libupnp, which sets up a UDP network service on port 1900. Per CVE [3], several exploitable vulnerabilities exist in libupnp's old versions. These vulnerabilities would affect routers, media servers, etc. [64]. To patch these vulnerabilities, Google requires that the apps submitted to Google Play Store should adopt a new version (higher than version 1.6.18) of libupnp [4]; however, there is no warning for iOS apps. To quantify the impact of the Portable UPnP library vulnerability (i.e., # apps affected), we search this library among our collected dataset by using a signature. The result shows that 16 apps integrate the libupnp library, among 13 apps using out-of-date libupnp, 6 apps are seriously impacted by this library, these vulnerable apps have been installed millions of times. Interestingly, we find that the vendor "Flipps Media Inc." has upgraded the library in the product iMediaShare, whereas other products, Flipps TV (version 6.3.8) and FITE TV (version 2.1), are still using the vulnerable version of the library (e.g., "1.6.13."). The impacted apps are verified vulnerable by using module "multi/upnp/libupnp_ssdp_overflow" of Metasploit [58], which can crash the app remotely ($A3$) .

*(iii)* **Misuse of the GCDWebServer library.** The misuse of the GCDWebServer library exposes privacy or functionality to adversaries. To locate the misuse of this library, we look into the interface of the library and analyze how apps use this library.[2] We highlight that multiple factors lead to the misuse of this library. In the case of a vulnerable "file listing service" when using this library, the following three factors constitute a rule for locating the misuse issue. *(i)* Arguments are passed to the library, indicating the use of the LAN interface. *(ii)* The root folder is set to the data container directory. *(iii)* The built-in file listing functionality is used by this app. The query result reveals that 2,097 apps integrate the GCDWebServer library. By using the association rule shown in Table 5, the app integrating GCDWebServer which is a support for other libraries, is excluded. Finally, 517 apps are screened out, meaning they use this library exclusively. After checking these 517 apps using both static data-flow analysis, dynamic analysis and manual confirmation, 83 apps that misuse the GCDWebServer library are verified vulnerable. Note that, static analysis helps us to find more vulnerabilities behind user interaction. For instance, with the vulnerability in GCDWebServer library (CVE-2019-14924 [11]), the vulnerability in the JDRead app arises when a user is turning on the file sharing functionality of the app, and the QQMail exposes attachment to the adversary in the same Wi-Fi network when a user is reviewing the attachment in an email.

[2]The automated static analysis process and results are available at https://sites.google.com/site/iosappnss/home.

## 8 Related Work

**Vetting the security of network services**. There has been a plethora of work dedicated to vetting the security of network services [22, 32, 55, 80] as well as hunting security bugs [35, 38] and malicious behaviors [34, 36, 37] of Android apps. However, the analyzer for Android apps cannot be squarely applied to iOS due to the different programming language (e.g., Java and Objective-C). In addition, much work focuses on other security aspects of iOS apps, such as the secure usage of TLS/SSL certificates of iOS apps [67] and the cryptographic misuse of iOS apps [49]. Kobold [44] examines access control flaws on iOS. However, security vetting for network services of iOS apps has not been extensively explored.

**Third-party library identification**. The current third-party library identification methodology falls into four categories: text-based [29], token-based [57], tree-based [30], as well as semantics-based [60]. Android researchers have contributed widely to the third-party library identification [27, 48, 76]. CRiOS [67] is the only work focusing on third-party library identification in iOS. They clustered and studied the dependencies of classes in iOS apps in order to identify third-party libraries. CRiOS requires building all the third-party library repositories, whereas ours only builds a small portion of third-party network service libraries.

**Software testing techniques on iOS.** *(i) Dynamic analysis of iOS apps*. Szydlowski et al. [75] proposed an approach to tracking sensitive API calls by using debugger breakpoints and tried to automate the process by simulating the interaction with the identified UI views. ICRAWLER [56] explored the UI states of iOS apps by hooking techniques to inspect the UI elements. DIOS [61] utilized UI automation to retrieve the UI hierarchy and interact with UI elements to cover more code paths of iOS apps. IRIS [41] transported the instrumentation framework, termed VALGRIND [66], to iOS to vet private API abuse. *(ii) Static analysis of iOS apps*. PIOS [47] performed data-flow analysis to build the CFG and static taint analysis on top of the IDA [53] to track the privacy transferred. MoCFI [40] extracted the CFG of an app on top of the PiOS [47] and checked whether the instructions that change an execution flow are valid at runtime. Chen et al. [33] studied libraries in iOS and Android apps by considering invariant features between the two. We cross check the vulnerabilities identified and find none of these vulnerabilities exist in Android apps. Feichtner et al. [49] proposed static analysis by using LLVM IR for iOS apps; however, the methodology needs to be adopted at scale.

To the best of our knowledge, this is the first paper to systematically examine the security of iOS apps' network services on a large scale. We believe the vetting methodology and the results in this paper can inform security researchers as they closely inspect the iOS security in the future, and in particular, inform app developers and network operators on whether the policy of using the LAN network should be rectified.

## 9 Concluding Remarks

Thanks to its open source framework, much work has already tested the security of Android apps. Unfortunately, Apple's closed ecosystem makes vetting iOS systems much more difficult. This paper proposes the first methodology for conducting a large-scale security analysis of iOS apps' network services. When applied to the top 1,300 iOS apps, our proposed approach found 11 apps with vulnerabilities, three of which were acknowledged by their vendors. Extending our analysis to 168,951 apps found an additional 92 vulnerabilities and showed that the most popular provenance for an iOS device remote attack involves turning the device into a server.

With hindsight that the inconsistent functionalities between Apple and Google will potentially trigger vulnerabilities,[3] for mitigation, we therefore recommend app developers to use the loopback interface as much as possible to avoid unnecessary use of the LAN interface, and to enforce the deliberately designed access control when using the LAN interface. Furthermore, to mitigate the attack via public Wi-Fi or cellular networks, we recommend network operators to implement stricter firewall strategies and block unknown connection attempts originating from the same LAN network. System vendors such as Apple should also apply a host-based firewall, such as the one adopted by the OS X system, to the iOS system. We hope that our findings can motivate iOS app developers to focus more on the security of their network services and that our methodology for determining faulty libraries can be used by stakeholders to vet the apps they choose to use or make.

## Acknowledgments

We thank William Enck and the anonymous reviewers for their valuable feedback. We thank Cameron Ballard and Benjamin Zhao for proofreading the early version of this paper. We thank Xiaobo Chen, Tao Huang, and Jian Zhang, affiliated to PWNZEN InfoTech Co., LTD, for their valuable assistance of our manual analysis. This work was supported, in part, by the National Key Research and Development Program of China (2018YFE0126000, 2017YFB0802901) as well as the National Science Foundation of China (61972453, 61672350). Haojin Zhu (zhu-hj@cs.sjtu.edu.cn) is the corresponding author of this paper.

[3]One example is that Android provides cast functionality to project smartphone's screens to third-party screens while iOS developers must adopt an error-prone TCP-relay to implement such functionality.

## References

[1] Linux, sdk. for UPnP Devices (libupnp).

[2] Wormhole. http://xlab.baidu.com/wp-content/uploads/2016/01/wormhole_external_final.pdf.

[3] libupnp vulnerability. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=libupnp.

[4] How to fix apps with the portable SDK for UPnP library vulnerabilities. https://support.google.com/faqs/answer/6346109?hl=en-GB.

[5] CVE-2018-6344. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-6344.

[6] Clutch. https://github.com/KJCracks/Clutch.

[7] Cocoahttpserver. https://github.com/robbiehanson/CocoaHTTPServer.

[8] Frida. https://www.frida.re/.

[9] Dagger. http://dagger.repzret.org/.

[10] dumpdecrypted. https://github.com/stefanesser/dumpdecrypted.

[11] CVE-2019-14924. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-14924.

[12] libimobiledevice. https://github.com/libimobiledevice/ideviceinstaller.

[13] iTunes search API. https://affiliate.itunes.apple.com/resources/documentation/itunes-store-web-service-search-api/.

[14] jtool. http://www.newosxbook.com/tools/jtool.html.

[15] Waze. https://www.waze.com/.

[16] Weblink. https://www.abaltatech.com/press/weblink-from-abalta-technologies-brings-popular-waze-smartphone-app-into-the-connected-car.

[17] CVE-2019-3568. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-3568.

[18] Dancing line. https://apps.apple.com/us/app/dancing-line-music-game/id1177953618.

[19] Rules of survival. https://apps.apple.com/us/app/rules-of-survival/id130796175.

[20] frida-ios-dump. https://github.com/AloneMonkey/frida-ios-dump.

[21] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[22] D. Antonioli, N. O. Tippenhauer, and K. Rasmussen. Nearby threats: Reversing, analyzing, and attacking Google's' 'nearby connections' on Android. In *NDSS*, 2019.

[23] Make and receive calls on your Mac, iPad, or iPod touch. https://support.apple.com/en-hk/HT209456.

[24] Objective-c runtime. https://developer.apple.com/documentation/objectivec/objective-c_runtime?language=objc.

[25] Writing a TCP-based server. https://developer.apple.com/library/archive/documentation/NetworkingInternet/Conceptual/NetworkingTopics/Articles/UsingSocketsandSocketStreams.html#//apple_ref/doc/uid/CH73-SW8.

[26] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM Sigplan Notices*, 2014.

[27] M. Backes, S. Bugiel, and E. Derr. Reliable third-party library detection in android and its security applications. In *ACM CCS*, 2016.

[28] X. Bai, L. Xing, N. Zhang, X. Wang, X. Liao, T. Li, and S.-M. Hu. Discovering and exploiting novel security vulnerabilities in Apple zeroconf. In *Black Hat USA*, 2016.

[29] B. S. Baker. On finding duplication and near-duplication in large software systems. In *IEEE Working Conference on Reverse Engineering*, 1995.

[30] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *IEEE ICSM*, 1998.

[31] R. Bonett, K. Kafle, K. Moran, A. Nadkarni, and D. Poshyvanyk. Discovering flaws in security-focused static analysis tools for Android using systematic mutation. In *USENIX Security Symposium*, 2018.

[32] W. Bu, M. Xue, L. Xu, Y. Zhou, Z. Tang, and T. Xie. When program analysis meets mobile security: An industrial study of misusing Android Internet sockets. In *ACM FSE*, 2017.

[33] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang, B. Ma, A. Wang, Y. Zhang, and W. Zou. Following devil's footprints: Cross-platform analysis of potentially harmful libraries on Android and iOS. In *IEEE S&P*, 2016.

[34] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu. Stormdroid: A streaminglized machine learning-based system for detecting Android malware. In *ACM ASIACCS*, 2016.

[35] S. Chen, T. Su, L. Fan, G. Meng, M. Xue, Y. Liu, and L. Xu. Are mobile banking apps secure? What can be improved? In *ACM ESEC/FSE*, 2018.

[36] S. Chen, M. Xue, L. Fan, S. Hao, L. Xu, H. Zhu, and B. Li. Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach. In *Elsevier Computers & Security*, 2018.

[37] S. Chen, L. Fan, C. Chen, M. Xue, Y. Liu, and L. Xu. GUI-Squatting Attack: Automated generation of Android phishing apps. In *IEEE TDSC*, 2019.

[38] S. Chen, L. Fan, G. Meng, T. Su, M. Xue, Y. Xue, Y. Liu, and L. Xu. An empirical assessment of security risks of global Android banking apps. In *ACM/IEEE ICSE*, 2020.

[39] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. ReBucket: A method for clustering duplicate crash reports based on call stack similarity. In *IEEE ICSE*, 2012.

[40] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *NDSS*, 2012.

[41] Z. Deng, B. Saltaformaggio, X. Zhang, and D. Xu. iris: Vetting private API abuse in iOS applications. In *ACM CCS*, 2015.

[42] L. Deshotels, R. Deaconescu, M. Chiroiu, L. Davi, W. Enck, and A.-R. Sadeghi. SandScout: Automatic detection of flaws in iOS sandbox profiles. In *ACM CCS*, 2016.

[43] L. Deshotels, R. Deaconescu, C. Carabas, I. Manda, W. Enck, M. Chiroiu, N. Li, and A.-R. Sadeghi. iOracle: Automated evaluation of access control policies in iOS. In *ACM AsiaCCS*, 2018.

[44] L. Deshotels, C. Carabaș, J. Beichler, R. Deaconescu, and W. Enck. Kobold: Evaluating decentralized access control for remote NSXPC methods on iOS. In *IEEE S&P*, 2020.

[45] Androguard. `code.google.com/p/androguard`.

[46] Y. Duan, M. Zhang, A. V. Bhaskar, H. Yin, X. Pan, T. Li, X. Wang, and X. Wang. Things you may not know about Android (un) packers: A systematic study based on whole-system emulation. In *NDSS*, 2018.

[47] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in iOS applications. In *NDSS*, 2011.

[48] J. Feichtner and C. Rabensteiner. Obfuscation-resilient code recognition in Android apps. In *IEEE ARES*, 2019.

[49] J. Feichtner, D. Missmann, and R. Spreitzer. Automated binary analysis on iOS-a case study on cryptographic misuse in iOS applications. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2018.

[50] C. Gormley and Z. Tong. *Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine*. " O'Reilly Media, Inc.", 2015.

[51] B. Guangdong and Q. Zhang. 3G/4G Intranet scanning and its application on the wormhole vulnerability. 2017.

[52] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM Sigmod*, 2000.

[53] IDA Pro Disassembler and Debugger.

[54] M. Ikram and M. A. Kaafar. A first look at mobile ad-blocking apps. In *IEEE International Symposium on Network Computing and Applications*, 2017.

[55] Y. J. Jia, Q. A. Chen, Y. Lin, C. Kong, and Z. M. Mao. Open doors for Bob and Mallory: Open port usage in Android apps and security implications. In *IEEE EuroS&P*, 2017.

[56] M. E. Joorabchi and A. Mesbah. Reverse engineering iOS mobile applications. In *IEEE Working Conference on Reverse Engineering*, 2012.

[57] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. 2002.

[58] D. Kennedy, J. O'gorman, D. Kearns, and M. Aharoni. *Metasploit: The penetration tester's guide*. No Starch Press, 2011.

[59] J. T. Kent. Information gain and a general measure of correlation. 1983.

[60] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *International Static Analysis Symposium*. Springer, 2001.

[61] A. Kurtz, A. Weinlein, C. Settgast, and F. Freiling. Dios: Dynamic privacy analysis of iOS applications. 2014.

[62] Y. Lee, X. Wang, K. Lee, X. Liao, X. Wang, T. Li, and X. Mi. Understanding iOS-based crowdturfing through hidden UI analysis. In *USENIX Security Symposium*, 2019.

[63] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, 1966.

[64] H. Moore. Security flaws in universal plug and play: Unplug. don't play. 2013.

[65] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk. Automatically discovering, reporting and reproducing Android application crashes. In *IEEE ICST*, 2016.

[66] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan Notices*, 2007.

[67] D. Orikogbo, M. Büchler, and M. Egele. CRiOS: Toward large-scale iOS application analysis. In *ACM SPSM*, 2016.

[68] X. OS. Mach-O file format reference. 2009.

[69] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! Analyzing unsafe and malicious dynamic code loading in Android applications. In *NDSS*, 2014.

[70] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, and D. D. Yao. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized Java projects. In *ACM CCS*, 2019.

[71] R. Rehurek and P. Sojka. Gensim–Python framework for vector space modelling. 2011.

[72] L. SaurikIT. Cydia substrate, the powerful code modification platform behind Cydia. 2016.

[73] D. H. Steinberg and S. Cheshire. *Zero Configuration Networking: The Definitive Guide*. " O'Reilly Media, Inc.", 2005.

[74] M. Stute, S. Narain, A. Mariotto, A. Heinrich, D. Kreitschmann, G. Noubir, and M. Hollick. A billion open interfaces for Eve and Mallory: MitM, DoS, and tracking attacks on iOS and macOS through Apple wireless direct link. In *USENIX Security Symposium*, 2019.

[75] M. Szydlowski, M. Egele, C. Kruegel, and G. Vigna. Challenges for dynamic analysis of iOS applications. In *Open Problems in Network Security*. Springer, 2012.

[76] Z. Tang, M. Xue, G. Meng, C. Ying, Y. Liu, J. He, H. Zhu, and Y. Liu. Securing Android applications via edge assistant third-party library detection. 2018.

[77] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. IBM Corp., 2010.

[78] T. Wang, Y. Jang, Y. Chen, S. P. Chung, B. Lau, and W. Lee. On the feasibility of large-scale infections of iOS devices. In *USENIX Security Symposium*, 2014.

[79] M. Y. Wong and D. Lie. Tackling runtime-based obfuscation in Android with TIRO. In *USENIX Security Symposium*, 2018.

[80] D. Wu, D. Gao, R. K. Chang, E. He, E. K. Cheng, and R. H. Deng. Understanding open ports in Android applications: Discovery, diagnosis, and security assessment. In *NDSS*, 2019.

## Appendix

**Table 6:** Official network service APIs across app categories (see Section 6).

| **Categories** | _bind | Game Kit (2) | Game Kit (1) | _CFSocketSetAddress | _res_9_nquery | Multipeer Connectivity |
|---|---|---|---|---|---|---|
| **Business** | 1425 | 4 | 0 | 671 | 56 | 11 |
| **Education** | 1659 | 13 | 23 | 256 | 56 | 7 |
| **Entainment** | 1525 | 23 | 21 | 498 | 40 | 23 |
| **Finance** | 1311 | 0 | 0 | 652 | 39 | 9 |
| **Food & Drink** | 2022 | 2 | 2 | 781 | 171 | 11 |
| **Games** | 40375 | 5349 | 3017 | 2166 | 60 | 279 |
| **Health & Fitness** | 909 | 2 | 5 | 342 | 65 | 11 |
| **Kids** | 2811 | 140 | 112 | 41 | 0 | 17 |
| **Lifestyle** | 1415 | 5 | 5 | 608 | 126 | 23 |
| **Magazines & Newspapers** | 1081 | 1 | 2 | 322 | 23 | 6 |
| **Medical** | 1329 | 5 | 9 | 409 | 84 | 6 |
| **Music** | 1168 | 7 | 13 | 567 | 37 | 20 |
| **Navigation** | 1129 | 1 | 5 | 329 | 35 | 8 |
| **News** | 1286 | 4 | 3 | 398 | 89 | 5 |
| **Photo & Video** | 818 | 5 | 3 | 331 | 42 | 28 |
| **Productivity** | 1073 | 1 | 2 | 549 | 27 | 30 |
| **Reference** | 745 | 1 | 1 | 254 | 21 | 11 |
| **Social Networking** | 1838 | 2 | 3 | 721 | 238 | 18 |
| **Sports** | 1290 | 7 | 8 | 390 | 69 | 17 |
| **Travel** | 708 | 3 | 1 | 291 | 26 | 9 |
| **Utilities** | 1405 | 4 | 4 | 673 | 33 | 39 |
| **Weather** | 1915 | 1 | 1 | 716 | 144 | 16 |
| **Total** | 69237 | 5580 | 3240 | 11965 | 1481 | 604 |

**Table 7:** Third-party network service libraries across app categories (see Section 6).

| **Categories** | **Business** | **Education** | **Entertainment** | **Finance** | **Food** | **Games** | **Health** | **Kids** | **Lifestyle** | **Magazines** | **Medical** | **Music** | **Navigation** | **News** | **Video** | **Productivity** | **Reference** | **Social Networking** | **Sports** | **Travel** | **Utilities** | **Weather** | **Total** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **boost::asio::io_service (C)** | 30 | 21 | 33 | 11 | 12 | 284 | 11 | 5 | 21 | 1 | 14 | 20 | 12 | 14 | 11 | 28 | 4 | 33 | 12 | 6 | 20 | 9 | 612 |
| **CocoaHTTPServer (OC)** | 45 | 38 | 163 | 15 | 19 | 1315 | 35 | 16 | 57 | 128 | 15 | 112 | 18 | 73 | 83 | 95 | 53 | 68 | 49 | 6 | 139 | 11 | 2553 |
| **Tapjoy-CocoaHTTPServer-Extension (OC)** | 3 | 4 | 40 | 4 | 7 | 1220 | 6 | 13 | 24 | 3 | 4 | 15 | 7 | 4 | 17 | 21 | 5 | 43 | 13 | 0 | 27 | 4 | 1484 |
| **CocoaAsyncSocket (OC)** | 593 | 397 | 545 | 557 | 776 | 3203 | 290 | 48 | 674 | 391 | 384 | 330 | 272 | 425 | 343 | 412 | 220 | 956 | 358 | 247 | 631 | 709 | 12761 |
| **Google Cast (OC)** | 42 | 40 | 103 | 70 | 275 | 333 | 76 | 15 | 76 | 14 | 151 | 88 | 330 | 171 | 90 | 60 | 56 | 73 | 232 | 96 | 80 | 236 | 2707 |
| **PDRCoreHttpDaemon (OC)** | 12 | 3 | 1 | 17 | 84 | 0 | 1 | 0 | 3 | 1 | 14 | 3 | 22 | 12 | 2 | 6 | 4 | 6 | 28 | 2 | 8 | 83 | 312 |
| **GCDWebServer (OC)** | 30 | 33 | 64 | 13 | 99 | 999 | 38 | 37 | 28 | 73 | 56 | 58 | 59 | 69 | 45 | 46 | 48 | 39 | 80 | 18 | 72 | 93 | 2097 |
| **UnityEngine.iOS (OC)** | 5 | 41 | 38 | 3 | 36 | 5725 | 12 | 122 | 6 | 17 | 46 | 48 | 28 | 46 | 6 | 3 | 13 | 6 | 58 | 1 | 5 | 37 | 6302 |
| **WebRTC (C)** | 58 | 58 | 25 | 54 | 91 | 45 | 35 | 0 | 44 | 6 | 117 | 13 | 31 | 18 | 14 | 33 | 8 | 192 | 67 | 24 | 22 | 69 | 1024 |
| **gRPC (OC)** | 1 | 1 | 1 | 3 | 2 | 2 | 0 | 0 | 2 | 2 | 2 | 0 | 1 | 0 | 2 | 1 | 1 | 4 | 0 | 0 | 5 | 0 | 30 |
| **SmartView (OC)** | 10 | 21 | 9 | 44 | 12 | 11 | 15 | 1 | 28 | 2 | 7 | 11 | 9 | 17 | 16 | 12 | 2 | 25 | 11 | 53 | 18 | 10 | 344 |
| **Unreal Engine 4 (OC)** | 1 | 0 | 2 | 0 | 1 | 195 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 202 |
| **Happy DNS (OC)** | 34 | 41 | 30 | 23 | 74 | 10 | 37 | 0 | 69 | 12 | 39 | 29 | 12 | 38 | 32 | 12 | 11 | 142 | 39 | 15 | 24 | 54 | 777 |
| **MongooseDaemon (OC)** | 4 | 2 | 12 | 4 | 3 | 67 | 1 | 2 | 0 | 0 | 0 | 5 | 0 | 2 | 2 | 1 | 0 | 2 | 1 | 0 | 8 | 0 | 116 |
| **DIAL (C)** | 1 | 4 | 16 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 1 | 6 | 0 | 1 | 2 | 1 | 1 | 1 | 0 | 0 | 7 | 1 | 44 |
| **Platinum UPnP (C)** | 1 | 3 | 24 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 1 | 6 | 0 | 0 | 3 | 2 | 0 | 0 | 0 | 0 | 9 | 0 | 51 |
| **upnpx (C)** | 2 | 2 | 10 | 0 | 2 | 0 | 0 | 0 | 3 | 0 | 1 | 5 | 2 | 3 | 5 | 4 | 0 | 1 | 0 | 0 | 4 | 2 | 46 |
| **Ionic's Webview (OC)** | 1 | 0 | 0 | 1 | 45 | 11 | 1 | 0 | 0 | 0 | 31 | 0 | 34 | 1 | 0 | 0 | 0 | 0 | 41 | 3 | 1 | 47 | 217 |
| **Connect SDK Core (iOS) (OC)** | 0 | 1 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 4 | 0 | 13 |
| **wangxin.taobao (OC)** | 8 | 5 | 4 | 4 | 20 | 2 | 8 | 0 | 26 | 0 | 6 | 2 | 3 | 6 | 2 | 12 | 1 | 17 | 6 | 8 | 7 | 24 | 171 |
| **LeTVCDE (OC)** | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 |
| **FunTV (OC)** | 0 | 0 | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 1 | 0 | 15 |
| **Audiobus SDK (C)** | 1 | 6 | 19 | 0 | 0 | 15 | 6 | 3 | 3 | 2 | 8 | 158 | 0 | 3 | 10 | 6 | 1 | 11 | 2 | 0 | 10 | 0 | 264 |
| **pupnp (C)** | 0 | 1 | 10 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 7 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 23 |
| **inke SDK (OC)** | 0 | 0 | 2 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 5 | 4 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 2 | 15 |
| **MAASDK (OC)** | 2 | 3 | 3 | 4 | 1 | 0 | 0 | 0 | 10 | 3 | 0 | 0 | 3 | 7 | 1 | 1 | 0 | 2 | 1 | 5 | 2 | 3 | 51 |
| **TencentVideoHttpProxy (OC)** | 1 | 3 | 2 | 0 | 2 | 2 | 0 | 0 | 1 | 0 | 0 | 2 | 0 | 1 | 0 | 0 | 0 | 4 | 1 | 0 | 6 | 0 | 25 |
| **SmartDeviceLink (OC)** | 0 | 0 | 4 | 1 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 11 | 8 | 3 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 4 | 37 |
| **libupnp (C)** | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 3 | 1 | 0 | 0 | 0 | 0 | 2 | 0 | 16 |
| **ProudNet (C)** | 0 | 0 | 0 | 0 | 0 | 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 15 |
| **MobileIMSDK (OC)** | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| **GCDTelnetServer (OC)** | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 4 |
| **yfcloud (OC)** | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 |
| **MQTT (C)** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 |