JPush Away Your Privacy: A Case Study of Jiguang's Android SDK

Joel Reardon[1;3], Nathan Good[2;3], Robert Richter[3]
Narseo Vallina-Rodriguez[3;4;5], Serge Egelman[3;4;6], Quentin Palfrey[7;8]

TR-20-001

August 2020

Abstract

Our investigations into Android apps found that Chinese company Jiguang invasively monitors the activity of consumers who install apps that include their SDK. Jiguang's SDK can collect consumers' GPS locations, immutable device persistent identifiers, and even the names of all the apps they have installed—including when new ones are added or old ones removed. It does this collection even if the app that contains their code is not used. They send data over UDP sockets with misused cryptography, resulting in consumers' personal data being trivially vulnerable to eavesdroppers. We observed their SDK communicating with Jiguang in 31 apps.

[1]University of Calgary, [2]Good Research LLC, [3]AppCensus Inc., [4]International Computer Science Institute, [5]IMDEA Networks, [6]UC Berkeley, [7]Future of Privacy Forum

JPush Away Your Privacy: A Case Study of Jiguang's Android SDK

Joel Reardon[1,3], Nathan Good[2,3]. Robert Richter[3]
Narseo Vallina-Rodriguez[3,4,5], Serge Egelman[3,4,6], Quentin Palfrey[7]

[1]University of Calgary, [2]Good Research LLC, [3]AppCensus Inc.,
[4]International Computer Science Institute, [5]IMDEA Networks, [6]UC Berkeley,
[7]International Digital Accountability Council, [8]Berkman Klein Center for Internet & Society, Harvard

***Abstract*—Our investigations into Android apps found that Chinese company Jiguang invasively monitors the activity of consumers who install apps that include their SDK. Jiguang's SDK can collect consumers' GPS locations, immutable device persistent identifiers, and even the names of all the apps they have installed—including *when* new ones are added or old ones removed. It does this collection even if the app that contains their code is not used. They send data over UDP sockets with misused cryptography, resulting in consumers' personal data being trivially vulnerable to eavesdroppers. We observed their SDK communicating with Jiguang in 31 apps.**

I. Introduction

As is common in software development, some of the functionality within mobile apps is provided by third-party libraries and SDKs, many of which have business models based on personal data collection (e.g., ad networks and analytics). Recent work has emphasized the risks that such SDKs pose [18], [19], [4], [17], [15], [25]. In particular, third-party SDKs have access to the same set of permissions as the app in which they are embedded; an app that may have legitimate use of the location permission may have that permission abused by a third-party SDK embedded in it. Consumers have little transparency into—or control over—the various third-party SDKs, including tracking tools, that are embedded in the apps they use. Worse, they have even less insight into the behaviors of these SDKs and the personal data that they collect: it is not reasonable to expect consumers to reverse-engineer their apps or perform deep packet inspection, which are essentially the only tools available to them, given that app developers are under no obligation to disclose their presence in privacy policies.[1]

Not all third-party components embedded in mobile apps are primarily related to advertising and tracking services. For example, Unity's SDK provides a game development framework, and many companies provide SDKs for push notifications. Push notifications are small server-to-client messages that can reach audiences anywhere and anytime. They are at the core of *customer engagement* services, whose purpose is to create a direct communication channel between an external stakeholder (consumer) and an organization, such as a company, developer, advertiser, or brand. Push notification services offered by SDKs might be associated with companies that also offer analytics and advertisement services, but have not received the same degree of scrutiny as those *primarily* concerned with advertising and tracking.

In this paper, we provide an in-depth case study and analysis of one Android SDK offering a wide range of services, including analytics and push notifications, among others: Jiguang—also known as Aurora Mobile[2]. We study its prevalence, behavior, and its associated privacy risks. They claim to be present in more than a million apps and, incredulously,[3] more than 26 billion mobile devices [13]. We only examined free apps in the Google Play store and found their presence in approximately 400. Despite that, some of the apps have millions to hundreds of millions of installs, and thus their behaviors pose a serious risk to consumers. Similar techniques may also be used by other SDKs, impacting even more consumers. We observed the following:

- **Invasive personal data collection and monitoring:** We observed the transmissions of precise GPS location, full scan details of nearby wireless networks, immutable persistent identifiers (e.g., IMEI), and even the names of *other* apps consumers install and uninstall, that is, even those that do *not* contain this SDK. We note that the set of apps that a consumer curates on their mobile device can reveal a great deal about them, such as their interests, hobbies, health, political leanings, and sexual orientation [20].
- **Continuous background monitoring:** This information is collected in the background, even when the consumer never uses the app that contains Jiguang's code. Android facilitates this with two permissions: (i) the RECEIVE_BOOT_NOTIFICATION permission, which allows apps to start automatically when the the device is started; and (ii) the RECEIVE_USER_PRESENT permission, which can start apps and informs them whenever a consumer unlocks their phone screen (i.e., is present). Not all of the

[1]For example, both the GDPR and CCPA only require that companies disclose *broad categories* of data recipients, and do not require that individual companies be named, or that apps link to third parties' privacy policies.

[2]https://www.jiguang.cn

[3]The GSM Association estimated that by 2025, there will be approximately 4.5B smartphone users worldwide [8].

Jiguang-communicating apps have the boot permission, but 100% of them have the user present one. Jiguang's Android SDK integration documentation states that this user present permission is essential to correctly integrate the SDK in Android apps [3].

- **Unusual development practices:** We also found three methods that Jiguang uses to obfuscate and hide their behavior and network activity, which we detail in depth. This includes encrypting network traffic using an insecure non-standard key exchange, the curious use of (non-QUIC) UDP datagrams to send JSON data, and the static Vernam-cipher encryption of all string constants in the compiled program—a behavior more appropriate in malware than legitimate software [16].

The organization of the paper is as follows. Section II describes our methods. Section III presents our results. Finally, Section IV draws conclusions.

II. App Analysis Methods

In this paper, we analyse Android apps using a combination of static and dynamic analysis provided by AppCensus.[4] We developed a heavily instrumented version of Android 7 (Nougat), which we deploy on actual Android Nexus 5X smartphones. Our instrumentation monitors access to sensitive data at runtime and reports all network traffic payloads, including traffic that is secured by TLS. We are also able to accurately attribute all network flows to the specific app that is responsible for it.

In order to scale up our analysis, the phones are connected to a computer that automatically installs apps through the Google Play Store if they are available, and then runs them by interacting with them using a UI fuzzer for a period of ten minutes. After this, the app is uninstalled along with any other app that may have been installed during the UI fuzzing. We collect the network traffic and filter for traffic related to the app being tested. We further installed and removed a different app—using `adb` commands—during this experiment to see if this event was reported by the app being tested. Our instrumentation further evades the detection of "rooting" or "jailbreaking" on the device, so apps execute without expecting possible scrutiny.

The next step is to process the captured network traffic. We apply a suite of decoders to reveal encoded network traffic, such as `base64` and `gzip`. We also manually analyze network traffic and statically analyze particular code libraries that use obfuscation to reveal how the data is being transmitted. This method is used for popular domains that receive data from consumers, but for which we identified no private information being sent during our initial automated testing. This technique frequently reveals that private information is sent in an obfuscated manner. For example, we scrutinize transmissions to domains associated with an SDK that has accessed private information, such as the consumer's location, but for which no transmission was observed using existing instrumentation. This signals that data may have been transmitted in ways that we were not readily able to detect, and the app is flagged for manual analysis. Whenever we find a new obfuscation method during manual analysis, we add it as a new decoder into our suite and reanalyze all the network traffic for new findings. Indeed, it was precisely this iterative analysis of obfuscation that lead us to discover the peculiar behavior of Jiguang's Android SDK.

More technical details about AppCensus' instrumentation and capabilities can be found in our previous research studies [19], [18], [10], [11].

[4]https://www.appcensus.io

III. Results

In this section we provide the results of our analysis of Jiguang's Android SDK. We first provide the list of apps we found communicating with Jiguang along with the private information that they send. We then describe their technique to obfuscate string constants that we saw in some instances of their code, the method that they use to obfuscate data over UDP, and a further method that they use to obfuscate data that appears in more-recent versions of their SDK. In Appendix D, we provide example network transmissions that comprise the variety of private data exfiltrations that we observed.

A. Apps and Data Transmissions

We began with a corpus of 118,926 apps downloaded from the Google Play Store; this includes historical versions for a total of 368,128 APK files. We winnowed it to 422 unique apps that may contain Jiguang's code by searching for a number of string constants that appear in their SDK. We then downloaded the newest version of these apps on January 14th, 2020, of which 415 were still available in Canada, which is where we ran our experiments. Using our dynamic analysis testbed, we ran each of these apps and recorded its network traffic, which we then used to study data transmission and obfuscation techniques.

Of the 415 apps, we found that the newest versions of 24 apps communicated with Jiguang during a 10-minute test performed between January 14th and the 22nd, 2020. Further independent testing of apps found seven additional apps that communicate with Jiguang, which we added to our data to bring the total to 31. Prior to publishing we redid our experimentation late August 5th, 2020, with the newest versions of all 31 apps. The results we provide are from the most recent analysis.

We define an app as having communicated with Jiguang as opening a socket and sending data to any of the domains that we attribute to them. This includes those ending in `jpush.io`, `jpush.cn`, or `jiguang.cn` (cf. Appendix A), as well as `easytomessage.com`, which is referenced by the Jiguang SDK and uses the same packet format and obfuscation technique (as described in Section III-C). Finally, we include specific IP addresses that are provided by one of these Jiguang domains to the running app as apparent endpoints for data harvesting.

TABLE I
APPS COMMUNICATING WITH JIGUANG ALONG WITH THE TYPES OF PERSONAL INFORMATION WE OBSERVED BEING TRANSMITTED.

Package Name	Version	Installs	Identifiers				Apps		Location	
			IMEI	Android ID	MAC Addr	Serial #	Install	Listing	Router MAC	GPS
com.e2link.tracker	64	5K+	×		×		×	×	×	×
com.eyugame.ow.globalard	16	1M+					×	×	×	
com.ipc300	44	1K+							×	×
com.juanvision.jcloud	1196	10K+	×		×	×	×	×	×	×
com.lovestruck1	497	100K+							×	×
com.maaii.maaii	290006	1M+	×		×		×	×	×	×
com.macrovideo.v380	57	5M+							×	
com.macrovideo.v380pro	25	1M+							×	×
com.mfw.roadbook	869	100K+							×	×
com.nexttrucking.trucker	146	50K+							×	×
com.qcplay.slimegogogo	85	1M+			×		×		×	
com.specialyg.ippro	1591	1M+	×		×	×	×	×	×	×
com.starvebird.client	164	1K+	×		×		×	×	×	×
com.suncitygroup.apps.suncity	396	5K+							×	×
com.thinker.swift	10	1K+	×		×		×	×	×	×
com.vbikes.us.alpha	33	10K+							×	×
premom.eh.com.ehpremomapp	118	500K+							×	×

The apps integrating Jiguang's SDK range from selfie editors to dating apps to sales and CRM apps. According to the Google Play Store, some have been installed on as few as 10,000 phones [21], and some more than 100,000,000 [9]. There is diversity in the maturity levels of the apps using these SDKs: some have a PEGI 3 / Everyone rating [5] and others are rated PEGI 18 / Mature [14]. This suggests that some of these apps can potentially track vulnerable populations like minors, who are protected by strict regulations like COPPA in the USA [19], [23] and GDPR-K in the EU [22].

Table I shows the app name and versions that sent private information to a Jiguang domain during our testing in August 2020. This does not include the apps that communciated with Jiguang domains but without sending personal information. We searched for particular types of personal information in these transmissions; when they were observed we denote with a × in the tables corresponding to the app (row) and data type (column). We group personal data collection into three categories:

1) *Unique identifiers*. This category includes immutable identifiers, such as the IMEI, WiFi MAC address, and serial number, as well as "semi-resettable" ones like the Android ID. Persistent identifiers, typically bound to hardware components, cannot be changed except with extraordinary steps, such as rooting or jailbreaking the phone and installing a custom operating system.[5] The Android ID can be reset, but only through a *factory reset* of a phone, something that most consumers do not do. Android 7, on which we ran our experiments, does not allow access to the WiFi MAC address, but instead returns a "blank" value.[6] Jiguang SDKs can access the true value using a side channel enabled by the `getNetworkInterfaces()` method in the `java.net.NetworkInterface` class [12]. Incredibly, this side channel, known for at least five years, is still exploitable and exploited on Android 10, despite Google's awareness.[7] As of Android 10, access to the serial number and IMEI is limited to "privileged" apps (e.g., pre-installed ones) [2]. This means that Android does not consider apps' access to these identifiers to be legitimate under any other circumstances.
2) *Location data*. This category has two data types that correspond to the device's geolocation. The first is the MAC address of the WiFi router to which the phone is connected, also known as the "BSSID." This information is known to be a surrogate for the device's location [1]. This is because WiFi routers typically remain immobile once installed in a location, and therefore commercial APIs exist that map router MAC addresses to their physical locations [6]. The United States Federal Trade commission (FTC) reached a $4M settlement with analytics firm InMobi for its alleged deceptive collection of location data in a similar manner [24].
   Since Android 7, apps must request a location permission to access the MAC address of the connected WiFi router.[8] The second is precise GPS location; we require three decimal points of precision (i.e., accuracy of approximately 100m)[9] and the transmission of both latitude and longitude in the same network flow to be considered a transmission of location data.
3) *User actions and system events*. The final data type

[5]The IMEI is the most persistent of these identifiers, and in some places (e.g., the United Kingdom) it is illegal to change.

[6]https://developer.android.com/about/versions/marshmallow/android-6.0-changes.html

[7]Our bug report number 159780579 was marked as a duplicate of bug 32554324, which suggests it has been known for quite some time.

[8]https://developer.android.com/reference/android/net/wifi/WifiInfo.html#getBSSID()

[9]https://en.wikipedia.org/wiki/Decimal_degrees

reports whether the app is monitoring user actions like the installation and removal of apps. We tested this by installing and removing an app that was not on our device and was not part of the set of apps we were testing. We did this installation and removal once for each app we tested, and looked to see if the app we installed appeared in the network transmissions for the app we were testing.

All of the apps in Table I held the RECEIVE_USER_PRESENT permission, which allows code in the app to run whenever the consumer unlocks their phone (i.e., is present at the device). We observed this behaviour in practice, where apps that we had left installed on devices while doing our analysis would continue to transmit our location and other data as soon as we unlocked the phone after rebooting it.

B. String Constant Obfuscation

Android apps are typically compiled into "byte code", an assembly-like language that is processed by a virtual machine. It is a common practice for developers to shrink and obfuscate their code when deploying a production release. Tools designed for this purpose remove the inherent human-readability from code to reduce its footprint [7].

This works by replacing all semantically meaningful class names (like `HttpRequest`), method names (like `dispatch`), and variable names (like `result`) with semantically meaningless replacements. This results in lines of code looking like an alphabet soup of `A.a.c(b, c)`.

While investigating one sample of Jiguang's SDK, we found an obfuscation technique that we had not yet seen in any of our previous research into the Android SDK ecosystem. They obfuscated the string constants themselves, replacing them with pseudo-encrypted values that are decoded at runtime [16]—a technique more in line with malware development practices than legitimate software. This technique has the effect of actually bloating the program because each class provides its own code to do the decoding and the size of the "encrypted" string is the same as the original. Note that this technique was not used in all versions of Jiguang's SDK; where we did see it, however, it was not being done to the entire application code, but only to Jiguang's SDK code. Thus, this may be the actions of a developer to elect to obfuscate Jiguang's presence, rather than Jiguang providing the SDK in that state.

The exact procedure worked as follows. Consider the "app_version", which is needed within Jiguang's SDK. We saw this string actually stored in byte code as "IbgT\u0015M‘db\u000cF" and it is decoded *at runtime* into the value required for the program to work. In this example, the parts "\u0015" and "\u000c" are how the hexadecimal value `0x15` and `0x0c` are represented (respectively), because they are not printable characters that can be typed and stored in a string.

This string obfuscation is done with a Vernam cipher using a repeating code word—an XOR-based variant of the classic Viginère cipher that can easily be broken using standard cryptanalysis. The Viginère cipher takes a short "word" as a secret key, and "adds" the word to the message letter by letter. For example:

	h	a	p	p	y	*(input)*
+	k	o	a	l	a	*(key)*
	r	o	p	a	y	*(result)*

So, 'h' (7) and 'k' (10) becomes 'r' (17). When the secret word is shorter than the message to "encrypt", then the secret word is repeated as often as necessary, so the input `happytrail` would be encrypted with `koalakoala`. Jiguang's string obfuscation is similar, except that it uses a bit-wise XOR operation instead of an alphabetic shift.

Every source code file has its own key to "encrypt" the strings contained in that file. Since the source code is written in Java, each file is its own class and the class's static constructor is used to run the deobfuscation code. In effect, this builds a dictionary that maps an identifier to the actual meaningful string that is stored in memory whenever the program is running. Then, each string access is replaced with an access to the relevant data structure at runtime. In Appendix B we provide the decompiled byte code (in smali syntax) for the string obfuscation code with our own explanatory comments.

Jiguang's technique complicated our investigations because the use of string constants is so crucial to any deep analysis of the behaviour of a compiled artifact and for attribution purposes. String constants act as a kind of anchor that allows us to take tens of thousands of lines of code and rapidly find the area that needs investigation. As such, we wrote a tool that takes a smali byte code file as input and writes an equivalent file with each string juxtaposed with a comment that reveals its decoding, i.e., its true value. Appendix C provides this tool.

C. UDP Network Traffic

Another interesting behaviour we observed was the use of UDP (on port 19000) to transmit JSON data using a broken "roll-your-own" cryptographic protocol. This behaviour was conspicuous as JSON data is seldom transmitted over non-QUIC UDP flows.

Jiguang uses the following method to encrypt their UDP traffic. First they generate a random 32-bit number and remove the high-order 8 bits. They then permute it as detailed by the algorithm in Figure 1. The result is converted to a string based on its decimal representation, which is prefixed with "JCKP". That string is then hashed with MD5 to create a 128-bit binary string. That binary string is then converted to a 256-bit string by taking the hexadecimal representation of the string (e.g., `01ef`), converting *that* to ASCII, and taking the bits of the ASCII as the binary string (e.g., 00110000001100010110010100110110).

The resulting key is then used in the AES cipher in cipher-block chaining (CBC) mode. CBC mode requires a random 128-bit initialization vector (IV), for which Jiguang uses the first half of the 256-bit key. For CBC mode to work correctly, IVs are supposed to be random for each message, and so this does not follow best practices.

```
int32_t jiguang_permute(int32_t seed) {
        static int32_t mul[10] = {
                0x8, 0x5, 0x17, 0x3, 0xd,
                0x11, 0x7, 0x1f, 0x1d, 0x25};
        static int32_t quo[10] = {
                0x4a, 0x58, 0xf, 0x49, 0x60,
                0x31, 0x44, 0x27, 0x29, 0x5b};
        int32_t offset = seed % 10;
if (offset < 0) offset += 10;
        int32_t val = seed * mul[offset];
        int32_t remain = seed % quo[offset];
if (remain < 0) remain += quo[offset];
        val += remain;
        return val;
}
```

Fig. 1. C code snippet that performs Jiguang's permutation.

Standard practices to secure UDP sockets is the use of DTLS. Instead Jiguang uses their own implementation based on a 24-bit random number. This number is further included in the UDP packet payload, presumably so that Jiguang is able to decrypt the resulting transmission. This makes it trivial for anyone to decrypt and observe the traffic, meaning that they are sending consumer's precise GPS coordinates over the Internet to servers in China, using effectively no security precautions to prevent eavesdropping. We were able to decrypt all the traffic we had already collected simply by finding the 24-bit number in the header, running their algorithm, and decrypting the payload.

D. TCP Network Traffic

A further communication format was found using similar techniques as the UDP transmission. These were done over TCP, but used neither HTTP nor TLS in their transmissions. They used high port numbers—over 7000—on the server side. They also did not use hostnames, but rather received the IP and port from a command and control server that was first communicated with using the UDP format.

They also provided a 24-bit number stored as a little-endian 32-bit number within a 37-byte header. It occurs in different places in the header and we did not determine how to predict that so we instead attempt each 24-bit number that follows a zero byte.

This method takes this 24-bit number and performs the same sequence to derive the encryption key, i.e., permuting, prefixing, hashing, and ASCII-fying to generate a 256-bit AES encryption key used in CBC mode. Instead of the first half of the key being used as the IV, they used the fixed value `iop203040506aPk!` for these transmissions. The decrypted data may be further compressed with `gzip`, so if the magic number is present, we then decompress.

E. Jiguang V3 Reports

The final obfuscation technique we found is when Jiguang issues HTTP POST requests to `alidata.jpush.cn/v3/report`. This V3 protocol used HTTP over TLS, i.e., best practices for securing communications. Nevertheless we found it being used *in addition* to both the UDP and TCP communications we found earlier, all of which were used to transmit sensitive user information.

We analyzed a sample of the code to study their new transmission method. It consisted of a randomly generated encryption key used to AES-encrypt the data in CBC mode. Key generation also uses the same "permute and hash a 24-bit random number" technique described in Section III-C to create a 256-bit binary string for use as an encryption key. The initialization vector is the same for all values and equal to the bytes of ASCII string `iop203040506aPk!`.

Similarly to how the random number is transmitted in the UDP-version of Jiguang's packets, this new technique also sends the encryption key alongside the data. In this case, it leverages the HTTP `Authorization` header to send the encryption key to Jiguang so that it can decrypt the traffic. For example, we saw the transmission of the following HTTP header (newlines added):

```
Authorization: Basic MzQ0MTE3NTc0ODc6MjFlN
DJhNjYyYWExOTAzZjkwNTZmMTJmMDg1NTk2OGU5MjE
0OTZkMzo2M2VmMTY5ZmI3YTQ3N2MxYmU1MjU1OTQ0N
GNmNmJkYw==
```

The base64 decoding of that value is the following (newlines added):

```
34411757487:21e42a662aa1903f9056f12f085596
8e921496d3:63ef169fb7a477c1be52559444cf6bdc
```

The encryption key is the last component:

```
63ef169fb7a477c1be52559444cf6bdc.
```

Again, because the encryption key is transmitted in the packet, we were able to decrypt all the traffic after the fact by adding a new decoder into our analysis suite. In this case, consumers are protected by the fact that Jiguang *also* uses TLS to secure the network transmission; in this case the additional encryption step simply serves to obfuscate the transmissions and has no security benefit to consumers.

IV. Conclusion

In this paper, we provide a detailed analysis of Jiguang's Android SDK, which we found present in a number of Android apps and which was performing invasive monitoring of consumers. In particular, it was collecting immutable persistent identifiers, precise GPS locations, and monitored different user actions like the installation and removal of other apps that are not connected to the app containing Jiguang's code.

Jiguang's SDK is particularly concerning because this code can run silently in the background without the consumer ever using the app in which it is embedded. Moreover, they send sensitive information insecurely over the Internet allowing any eavesdropper to monitor the traffic. We have provided examples of the network traffic generated by Jiguang's SDKs, along with the use of obfuscation techniques which impede the analysis of software using traditional methods. While the

majority of the previous research efforts focused on SDKs specialized in analytics and advertising services, the results of our analysis call for the need of analyzing and regulating the behavior of the whole third-party SDK ecosystem due to their privacy and consumer protection implications.

REFERENCES

[1] J. P. Achara, M. Cunche, V. Roca, and A. Francillon, "Short paper: Wifileaks: underestimated privacy implications of the ACCESS_WIFI_STATE Android permission," in *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, 2014, pp. 231–236.

[2] Android Developers, "Privacy changes in Android 10," https://developer.android.com/about/versions/10/privacy/changes, 2019, accessed: January 23, 2020.

[3] Aurora Mobile, "Android SDK Integration Guide," https://docs.jiguang.cn/en/jpush/client/Android/android_guide/, 2018, accessed: January 23, 2020.

[4] A. Feal, J. Gamba, N. Vallina-Rodriguez, P. Wijesekera, J. Reardon, S. Egelman, and J. Tapiador, ""Don't accept candies from strangers": An analysis of third-party SDKs," in *CPDP*, 2020.

[5] Foscam, Inc., "Foscam," https://play.google.com/store/apps/details/?id=com.foscam.foscam, 2020, accessed: January 23, 2020.

[6] Google, Inc., "The Google Maps Geolocation API," https://developers.google.com/maps/documentation/geolocation/intro, accessed: September 29, 2017.

[7] Google, Inc, "Shrink, obfuscate, and optimize your app," https://developer.android.com/studio/build/shrink-code, 2019, accessed: January 23, 2020.

[8] GSM Association, "The mobile economy 2018," https://www.gsma.com/mobileeconomy/wp-content/uploads/2018/05/The-Mobile-Economy-2018.pdf, 2018, accessed: January 23, 2020.

[9] Hago Games, "HAGO - Play With New Friends," https://play.google.com/store/apps/details/?id=com.yy.hiyo, 2020, accessed: January 23, 2020.

[10] C. Han, I. Reyes, A. Elazari Bar On, J. Reardon, Á. Feal, S. Egelman, and N. Vallina-Rodriguez, "Do you get what you pay for? comparing the privacy behaviors of free vs. paid apps," in *IEEE ConPro Workshop*, 2019.

[11] C. Han, I. Reyes, Á. Feal, J. Reardon, P. Wijesekera, N. Vallina-Rodriguez, A. Elazari Bar On, K. Bamberger, S. Egelman *et al.*, "The price is (not) right: Comparing privacy in free and paid apps," *Proceedings on Privacy Enhancing Technologies*, vol. 2020, 2020.

[12] Informatic0re on StackOverflow, "How to get the missing Wifi MAC Address in Android Marshmallow and later?" https://stackoverflow.com/questions/31329733/how-to-get-the-missing-wifi-mac-address-in-android-marshmallow-and-later/32948723, 2015.

[13] Jiguang, "Aurora mobile," https://www.jiguang.cn/en/, 2020, accessed: January 23, 2020.

[14] Love Group Hong Kong Limited, "Lovestruck - Real Dating," https://play.google.com/store/apps/details?id=com.lovestruck1, 2020, accessed: January 23, 2020.

[15] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: fast and accurate detection of third-party libraries in android apps," in *Proceedings of the 38th international conference on software engineering companion*. ACM, 2016, pp. 653–656.

[16] Y. Moses and Y. Mordekhay, "Android app deobfuscation using static-dynamic cooperation," *virus bulletin*, 2018.

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

[18] J. Reardon, Á. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez, and S. Egelman, "50 ways to leak your data: An exploration of apps' circumvention of the android permissions system," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 603–620.

[19] I. Reyes, P. Wijesekera, J. Reardon, A. E. B. On, A. Razaghpanah, N. Vallina-Rodriguez, and S. Egelman, ""Won't Somebody Think of the Children?" Examining COPPA Compliance at Scale," *Proceedings on Privacy Enhancing Technologies*, vol. 2018, no. 3, pp. 63–83, 2018.

[20] S. Seneviratne, A. Seneviratne, P. Mohapatra, and A. Mahanti, "Your installed apps reveal your gender and more!" *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 18, no. 3, 2015.

[21] Suntronics Technologies Inc., "Swift Wifi Cam," https://play.google.com/store/apps/details/?id=com.thinker.swift, 2018, accessed: January 23, 2020.

[22] The European Union, "Article 8: Conditions applicable to child's consent in relation to information society services," https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:02016R0679-20160504&from=EN%23tocId12, April 27 2016.

[23] U.S. Federal Trade Commission, "How to comply with the children's online privacy protection rule," http://www.ftc.gov/bcp/conline/pubs/buspubs/coppa.htm.

[24] ——, "Mobile Advertising Network InMobi Settles FTC Charges It Tracked Hundreds of Millions of Consumers' Locations Without Permission," 2016. [Online]. Available: https://www.ftc.gov/news-events/press-releases/2016/06/mobile-advertising-network-inmobi-settles-ftc-charges-it-tracked

[25] J. Zhang, A. R. Beresford, and S. A. Kollmann, "Libid: reliable identification of obfuscated third-party android libraries," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 55–65.

APPENDIX A: DOMAINS AND IP RANGES USED BY JIGUANG

```
ali-stats.jpush.cn
bjuser.jpush.cn
easytomessage.com
gd-stats.jpush.cn
sis.jpush.io
s.jpush.cn
stats.jpush.cn
tsis.jpush.cn
update.sdk.jiguang.cn
102.230.236.0/24
117.121.49.100
121.46.20.0/24
121.46.25.0/24
121.46.30.0/24
124.202.138.0/24
175.25.50.0/24
183.232.25.0/24
43.247.88.0/24
49.4.114.240
```

APPENDIX B: JIGUANG'S STRING OBFUSCATION CODE

The code we found in a Jiguang SDK that obfuscated all strings. We have added comments, which appear after an octothorpe (#), to provide context on what the code is doing.

```
.method static constructor <clinit>()V
    # init v0 to be an array of 65 strings
    const/16 v0, 0x41
    new-array v0, v0, [Ljava/lang/String;

    const/4 v1, 0x0
    const-string v2, "E{dxCIbg+\u0015M`db....."
    # store an obfuscated string in position 0
    # ... store the other 64 strings ...
    # then for each string in array, loop over
    # every letter. Here v7 is offset string
    # and v5 is the string
    :cond_0
    aget-char v2, v5, v7

    # compute v7 modulo 5 to find pos in key
    rem-int/lit8 v6, v7, 0x5
    packed-switch v6, :pswitch_data_0
    goto :goto_2

    # a switch statement to get secret byte
    # to xor. in this case the mask is the
    # 5-byte string 0x2812170b63
    :pswitch_40
    const/16 v6, 0x28
    goto :goto_3
    :pswitch_41
    const/16 v6, 0x12
    goto :goto_3
    :pswitch_42
    const/16 v6, 0x17
    goto :goto_3
    :pswitch_43
    const/16 v6, 0xb
    goto :goto_3
    :goto_2
    const/16 v6, 0x63

    # xor key to character and store result
    :goto_3
    xor-int/2addr v2, v6
    int-to-char v2, v2
    aput-char v2, v5, v7
    add-int/lit8 v7, v7, 0x1

    # exit the loop when the offset is equal
    # to string length
    :cond_1
    if-gt v4, v7, :cond_0
```

APPENDIX C: CODE TO DEOBFUSCATE STRINGS

We wrote a python program to deobfuscate the string constants in Jiguang SDKs that made use of string obfuscation (cf. Section III-B). This program takes as input a `smali` file as generated by `apktool`, and juxtaposes comments next to obfuscated string constants with their true value. It can be run in batch for all smali files in the current directory and subdirectories using UNIX's `find` command with an exec argument.

```
""" usage: python filename.py smali_file """

import codecs
import sys

d = open(sys.argv[1], 'r').read().split('\n')

if len(d) == 0: sys.exit()

init = False
stop = 0
strs = []
for n, i in enumerate(d):
if i == '': continue
if "<clinit>()" in i and ".method" in i:
init = True
continue
if ".method" in i:
init = False
continue
if init == True:
if 'xor-int/2addr' in i:
stop = n

if stop == 0: sys.exit()
i = stop
byte = []
while i > 0:
if "const/4 " in d[i] or "const/16 " in d[i]:
byte.append(int(d[i].split(' ')[-1], 16))
if "packed-switch" in d[i]:
break
i -= 1
byte.reverse()

if len(byte) != 5: sys.exit()

def demask(s, m):
i = 1
out = ''
```

```
s = codecs.escape_decode(s)[0]
while i < len(s) - 1:
if "\u00" == s[i:i+4]:
x = int("0x" + s[i+4:i+6], 16)
out += chr(x)
i += 6
else:
out += s[i]
i += 1
ret = ''
for i in range(0, len(out)):
c = ord(out[i])
c = c ^ m[i % len(m)]
ret += chr(c)
return ret

def getstr(s):
for i in range(0, len(s) - 2):
    if s[i:i+2] == ", ":
    return s[i+2:]
return ""


f = open(sys.argv[1], 'w')

for n, i in enumerate(d):
f.write(i + '\n')
if i == '':
continue
if "<clinit>()" in i and ".method" in i:
init = True
continue
if ".method" in i:
init = False
continue
if init == True:
if 'const-string' in i:
result = demask(getstr(i), byte)
f.write("\t#  const-string  \""
+ result + "\"\n")
f.close()
```

APPENDIX D: EXAMPLE TRAFFIC

Example of network traffic containing an app installation event. This has been deobfuscated and headers removed. The JSON has added whitespace for clarity. The app being installed is `com.duolingo`.


```
{
  "content": [
    {
      "action": "add",
      "appid": "com.duolingo",
      "itime": 1579092994,
      "type": "app_add_rmv",
      "account_id": "",
      "install_type": 0
    }
  ],
  "platform": "a",
  "uid": 34871748965,
  "app_key": "84887e3fdfe4ab64d0f38c99",
  "sdk_ver": "3.1.3",
  "core_sdk_ver": "1.2.3",
  "share_sdk_ver": "",
  "ssp_sdk_ver": "",
  "statistics_sdk_ver": "",
  "channel": "developer-default",
  "app_version": "5.4 Build 20191231"
}
```


Example of network traffic listing all installed apps. This has been deobfuscated and headers removed. The JSON has added whitespace for clarity.


```
{
  "content": [
    {
      "itime": 1579129190,
      "type": "app_list",
      "account_id": "",
      "data": [
        {
          "name": "com.android.cts.priv.ctsshim",
          "pkg": "com.android.cts.priv.ctsshim",
          "ver_name": "7.0-2996264",
          "ver_code": 24,
          "install_type": 1
        },
        {
          "name": "Phone and Messaging Storage",
          "pkg": "com.android.providers.telephony",
          "ver_name": "7.1.2",
          "ver_code": 25,
          "install_type": 1
        },
 /* close to 100 apps reports truncated */
        {
          "name": "Contacts Storage",
          "pkg": "com.android.providers.contacts",
          "ver_name": "7.1.2",
          "ver_code": 25,
          "install_type": 1
        },
        {
          "name": "CaptivePortalLogin",
          "pkg": "com.android.captiveportallogin",
          "ver_name": "7.1.2",
          "ver_code": 25,
          "install_type": 1
        }
      ],
      "slice_index": 1,
      "slice_count": 1
    }
  ],
  "platform": "a",
  "uid": 34072608256,
  "app_key": "7ce1b5f314f2990e7df02319",
  "sdk_ver": "3.2.0",
  "core_sdk_ver": "1.2.7",
  "share_sdk_ver": "",
  "ssp_sdk_ver": "",
  "statistics_sdk_ver": "",
  "verification_sdk_ver": "",
  "channel": "developer-default",
  "app_version": "3.3.10"
}
```


Example of deobfuscated UDP traffic. The JSON has added whitespace for clarity and the precise GPS coordinates were redacted.

```
{
   "type": 1,
   "appkey": "b959efb19b6e37fd4745660b",
   "sdkver": "2.1.2",
   "platform": 0,
   "uid":34002478781,
   "opera":"",
   "lat":XXX.XXXXXX,
   "lng":YYY.YYYYYY,
   "time":1579090476
}
```


Example of deobfuscated TCP traffic using the newest "v3" reporting. Whitespace was added, the full network scan truncated and private data redacted.


```
{
  "content": [
    {
      "itime": 1579092210,
      "type": "loc_info",
      "account_id": "",
      "network_type": "WIFI",
      "local_dns": "192.168.0.101",
      "wifi": [
        {
          "mac_address": "XX:XX:XX:XX:XX:XX",
          "signal_strength": -39,
          "ssid": "REDACTED",
          "age": 0,
          "itime": 1579092208,
          "tag": "connect"
        },
        {
          "mac_address": "XX:XX:XX:XX:XX:XX",
          "signal_strength": -42,
          "ssid": "REDACTED",
          "age": 0,
          "itime": 1579092208,
          "tag": "strongest"
        },
/* seven other wifi reports truncated */
      ],
      "cell": [
        {
          "mobile_network_code": -1,
          "cell_id": 28975913,
          "radio_type": "gsm",
          "signal_strength": -93,
          "mobile_country_code": -1,
          "carrier": "",
          "location_area_code": 11229,
          "generation": "wifi",
          "itime": 1579092208
        }
      ],
      "gps": [
        {
          "lat": XXX.XXXXXXX,
          "lng": YYY.YYYYYYY,
          "alt": ZZZ.ZZZZZZZ,
          "bear": 0,
          "acc": 21.773000717163086,
          "tag": "network",
          "itime": 1579092210
        }
      ]
    }
  ],
  "platform": "a",
  "uid": 34871748965,
  "app_key": "84887e3fdfe4ab64d0f38c99",
  "sdk_ver": "3.1.3",
  "core_sdk_ver": "1.2.3",
  "share_sdk_ver": "",
  "ssp_sdk_ver": "",
  "statistics_sdk_ver": "",
  "channel": "developer-default",
  "app_version": "5.4 Build 20191231"
}
```