Empirical Security and Privacy Analysis of Mobile Symptom Checking Apps on Google Play

I. Wayan Budi Sentana[1][a], Muhammad Ikram[1][b], Mohamed Ali Kaafar[1][c] and Shlomo Berkovsky[2][d]

[1]*Department of Computing, Macquarie University, 4 Research Park Drive, Macquarie Park, NSW, Australia*

[2]*Centre for Health Informatics, Australian Institute of Health Innovation, Macquarie University, 75 Talavera Rd, North Ryde, NSW, Australia*

{

Keywords: Android Apps, Privacy, Security, Static Analysis, Dynamic Fingerprinting.

Abstract: Smartphone technology has drastically improved over the past decade. These improvements have seen the creation of specialized health applications, which offer consumers a range of health-related activities such as tracking and checking symptoms of health conditions or diseases through their smartphones. We term these applications as *Symptom Checking* apps or simply *SymptomCheckers*. Due to the sensitive nature of the private data they collect, store and manage, leakage of user information could result in significant consequences. In this paper, we use a combination of techniques from both static and dynamic analysis to detect, trace and categorize security and privacy issues in 36 popular SymptomCheckers on Google Play. Our analyses reveal that SymptomCheckers request a significantly higher number of sensitive permissions and embed a higher number of third-party tracking libraries for targeted advertisements and analytics exploiting the privileged access of the SymptomCheckers in which they exist, as a mean of collecting and sharing critically sensitive data about the user and their device. We find that these are sharing the data that they collect through unencrypted plain text to the third-party advertisers and, in some cases, to malicious domains. The results reveal that the exploitation of SymptomCheckers is present in popular apps, still readily available on Google Play.

1 INTRODUCTION

Smartphone users are increasingly using their smartphones for health-related activities. The majority of smartphones now have the ability to passively collect health data as users progress through their day (Trifan et al., 2019). The result of this passive logging has been the creation of specialized health-related mobile applications (SymptomCheckers) which track, manage and store the health data of users. The logging capabilities of SymptomCheckers go beyond passive tracking, as users can self-monitor their activities and manually record their personal data. A wide variety of categories are offered including; exercise and fitness tracker, sleep patterns and quality, cardiology and vascular health, mental and emotional health, and blood sugar levels for diabetes (Smahel et al., 2019).

Leveraging upon previous work (Ikram et al., 2016), this paper presents the first characterization study of SymptomCheckers with a focus on security and privacy offered by these apps. In particular, we perform static and dynamic analysis to analyze the Android permissions, the presence of malicious code as well as third-party tracking libraries, and investigate the (un)encrypted traffic for content and endpoints of the exfiltrated sensitive data.

We collect and extract from a corpus of more than 1.7 million Android apps, 36 SymptomChecker for which the name or the description suggest they enable to either track or check health-related activities of users. We then manually check that the apps actually fall into the category of SymptomChecker (§ 2.1). We use a set of tools to decompile the SymptomChekers and analyze the source code of each of the mobile SymptomChekers. We then inspect the apps to reveal the presence of third-party tracking libraries and sensitive permissions for critical resources on users' mobile devices. We summarize our analysis as follow:

- 13.8% (5) of SymptomCheckers' use a vulnerable encryption scheme (i.e., SHA1+RSA) for signing

[a] https://orcid.org/0000-0003-3559-5123
[b] https://orcid.org/0000-0003-0336-0602
[c] https://orcid.org/0000-0003-2714-0276
[d] https://orcid.org/0000-0003-2638-4121

Sentana, I., Ikram, M., Kaafar, M. and Berkovsky, S.
Empirical Security and Privacy Analysis of Mobile Symptom Checking Apps on Google Play.
DOI: 10.5220/0010520106650673
In *Proceedings of the 18th International Conference on Security and Cryptography (SECRYPT 2021)*, pages 665-673
ISBN: 978-989-758-524-1

certificates and security purposes.

- 44.4% (16), 58.3% (21), and 63.86% (23) of SymptomCheckers provide Exported Activities, Exported Services and Exported Broadcast Receiver, respectively, which can be exploited by a malicious app. We found that 10.81% (4) of SymptomCheckers contain malware code embedded in their source codes.
- To avoid source code analysis 89% (32) of SymptomChekers rely upon different types of anti-analytics or obfuscation techniques.
- 22.2% (8) of SymptomChekers embed at least five different third-party tracking and advertisement libraries sharing sensitive user information such as location with third-party analytics and advertisers.
- 5% of the traffic generated by SymptomCheckers' use insecure HTTP protocol for transmitting users' sensitive data in plaintext which can be intercepted and modified by malicious in-path proxies.

2 DATA COLLECTION AND ANALYSIS METHODOLOGY

In this Section, we present our data collection and analysis methodology.

2.1 Data Collection Methodology

Given that the Google Play store does not contain "Symptom Checking" apps' category, we devise a search methodology to find SymptomCheckers on Google Play. First, we use several keywords including "symptom", "SymptomChecker", and "health checker" in the description of 1.7 million Android applications collected in (Ikram and Kaafar, 2017). We obtain 353 apps that match those keywords. We manually check the descriptions of 353 apps to ensure our SymptomCheckers apps are real *Symptom-Checker*. We removed apps containing symptom trackers, health dictionaries or apps that are used to store *only* user-health history. We also discard the apps used for the learning process by medical students or the apps used to assist health practitioners. Overall we found 36 apps that met our search criteria and manual inspection.

We use `gplaycli` (matlink, 2018) to download 36 apps and collect textual information including apps unique identifier, category and price, regional availability, description, number of installs, developer information, user reviews, and apps rating. 32 (88.8%) out of 36 SymptomCheckers have a single APK and 4(11.1%) apps have multiple APK to support different models and versions of the Android operating system (OS).

Table 1: Top 5 Free SymptomCheckers sort by number of installs. The lower part of the table summarises the average statics of SymptomCheckers.

#	Apps ID	# of Installs	Rating
1	com.webmd.android	10,000,000+	4.44
2	com.ada.app	5,000,000+	4.74
3	md.your	1,000,000+	4.1
4	com.mayoclinic.patient	1,000,000+	3.9
5	com.programming. progressive.diagnoseapp	500,000+	4.52
	Average Statistics:		
	Avg. # of Install	50,000+	
	Avg. # of Ratings	3.27	

Table 1 shows the top 5 SymptomCheckers apps sort by number of install and average ratings. Among the 36 apps, 83.3% (30) apps have at least 1,000 devices. We found that WebMD (`com.webmd.android`) is the most popular app with at least 10 Million installs. The average rating, number of raters, and number of reviews are respectively 3.27, 10646.81, and 4341.35, showing the popularity of SymptomCheckers among users. We also found that ADA Symptom Checker (`com.ada.app`) is highly rated by 290,484 of users with average rating of 4.74.

2.2 Analysis Methodology

To have a wider perspective of the existing SymptomCheckers security, we perform comprehensive static (or source code) and dynamic (or runtime network traffic) analysis to inspect apps' source code and investigate the apps' behavior during the runtime, respectively. We also conduct user review analysis to determine the user perception of the analyzed SymptomCheckers.

Static Analysis. An APK is a mobile *app package* file format supported by the Android OS for distribution and installation. APK encloses all of the program's codes and it supports resources including `.dex` files, resources, assets, certificates, and manifest file, which are considered as the important objects in this SymptomCheckers static analysis. Since the APK distributed in byte-code format, we conduct preprocessing by leveraging `APKTool` (Apktool, 2020) to decompile the APK into `Smali` format and get all those files that will be useful in the following further analysis:

1. **Certificate Signing Mechanism.** Android OS requires all APKs to be digitally signed with a cer-

tificate before it is uploaded to Google Play Store or installed on a device. Application signing simplifies developers to identify the app's author and to update their application without administering complicated permissions and interface. This process also becomes an insurance policy for developers in terms of apps' integrity and the accountability of their apps' behavior thus preventing adversaries from inserting malware into legitimate apps by modifying and repackaging apps on the apps market (SSL, 2020).

To evaluate the certificate signing mechanism adopted by the SymptomCheckers, we extract the `CERT.RSA` file among all the files generated during the apps de-compilation via `APKTool`. We customize a script leveraging Keytool (Oracle, 2020) to obtain encryption and hashing mechanisms as well as the length of the public key of certificates.

2. **Apps's Requested Permissions.** The permission system is a core security architecture in the Android OS. All applications request permissions to access sensitive data, system features, components, or other sensitive resources in the operating system are managed by these systems. Once granted, apps may collaborate with a potentially malicious application to perform various attacks such as permission escalation (Melamed, 2020). In this study, we first parse `Manifest.xml` of SymptomCheckers to determine the requested permissions and to analyze any potentially dangerous permissions. We then observe whether all the available list permissions are used by the functions in the respective SymptomCheckers. Therefore, we map the API calls or methods of each app with the permission requests in the Manifest using AXPLORER (Backes et al., 2016). As a result, all permission lists in the SymptomCheckers are requested at least once in the API calls or methods.

3. **Exported Component Analysis.** Android apps consist of several components: Activity, Service, Content Provider, and Broadcast Receiver. These components collaborate with each other to implement and provide apps' functionalities. Commonly, a function in an app will be triggered by the user via the activity. The Android platform allows these components to be accessed and triggered from other applications by setting the exported status equal to `True`. However, this exported component is also a surface attack for malware to exploit an app. Melamed et al., (Melamed, 2020) demonstrates how these exported components can manipulate apps' components to compromise apps for malicious activities (CWE-926, 2020). To identify the presence of an exported component in the SymptomCheckers, we use the Android Drozer(F-Secure-Labs, 2020) to analyze the `Manifest` file for each app. Drozer–a commonly used penetration testing tool–uses several checks to exploit vulnerabilities in mobile apps.

4. **Malware Detection.** To detect the presence of malicious codes in the SymptomCheckers, we scan APKs using `VirusTotal` (VirusTotal, 2020). VirusTotal is a multitude of malware scanning tools that provide a comprehensive result by aggregating more than 70 anti-virus engine and URL/domain blacklisting services. The tools have been widely used to identify the emergence of malicious apps, executable files, application software as well as domains. To automate the scanning process, we take advantage of the API provided by VirusTotal, and create a script to upload all samples to the VirusTotal repository.

5. **Obfuscations Detection.** Obfuscation technique refers to any means of evading, obscuring, or disrupting the analysis process by parties other than application developers. These techniques have both positive and negative sides. On the one hand, this technique is useful to hardening the apps and protecting the source code against analyzing and reproducing. On the other hand, this technique can be used by malware developers to evade basic analysis layers of application distribution services such as Google Play (Chau and Jung, 2019). Research in (He et al., 2020) found that 52% of its malware samples leverage this technique to evade the analysis tools.

   To detect such behavior in the SymptomCheckers, we use APKID(RedNaga, 2016) to analyze the `.dex` files obtained in decompiled APK. APKID returns at least one compiler name for each APK. If the apps leveraging any anti-analysis technique, the APKID will return several labels that we grouped as a manipulator, anti-virtual machine (vm), anti-debug, anti-disassembly, and obfuscator.

6. **Trackers Analysis.** The existence of third-party libraries and trackers on android apps has raised privacy and security concernsThese third-party libraries can exchange information and infer user personal information based on demographic data and user behavior harvested during user interaction with the apps. To reveal the existence of these libraries, we analyze the decompiled APK and comprehensively search sub-directories in decompiled APKs. These unique directories names correspond to the libraries embedded by apps' developers in the source codes. We rely on our list

of libraries to the previous research conducted in (Ikram and Kaafar, 2017) to filter and obtain the third-parties in SymptomChecking apps.

Dynamic Analysis. We conduct runtime network behavior (also called dynamic analysis) to measure the security and privacy of SymptomCheckers apps during the runtime by capturing the traffic transmitted by Apps to the Internet. To avoid the obfuscation (i.e., Anti-Virtual Machine) techniques, often used by certain apps, we use a dedicated Android device and channeling the connection via MITMProxy (mitmproxy, 2020) to the WiFi access point. Since all the SymptomCheckers developed in SDK version 25 or above, we create a script to add self-signed security certificate exemption to read the traffic transmitted in HTTPS protocol.

Given that the number of SymptomCheckers is small, once the apps installed on the device, we navigate the apps manually and observe all activities on the apps while MITMProxy intercepts the transmission between each app to the internet. We then convert the intercepted traffic to HTTP/S Archive (`.har`) file to simplify traffic analysis of each SymptomCheckers. We extract the secure communication line and privacy factors by observing the potential of privacy leaks. From the security factor, we analyzed the percentage of the encrypted HTTPS protocol adoption on the SymptomCheckers' communication lines. For this purpose, we extract intercepted unique URLs and identify the type of transmission protocol used.

User-review Analysis. A user review analysis was performed to capture the users' perceptions of SymptomCheckers. In this analysis, we have fetched Google Play store reviews (N = 76,817) of 30 apps. App reviews were categorized as positive (with 3, 4, or 5 stars ratings) and negative (with 1 or 2 stars ratings). We focus on the 1 or 2 stars with average ratings ≤ 2 to investigate the concerns around the app functionality, security, and privacy conduct. We obtained the complete list of reviews from the app's home page on the Google Play store. Leveraging on previous work (Ikram and Kaafar, 2017), we used an automated keywords-based search method and employed manual classification of the users' text to classify them into various types of complaints. First, we created dictionaries of keywords[1], as listed in Table 5, belonging to different complaints' categories to filter reviews and then performed manual validation of the resultant complaints categories. The co-authors were involved in manual validation. Based on this manual observation, each author re-classified each users' comment into only one of the 12 classes of complaint categories.

[1]For instance, keyword '`force close`' mapped to '`bugs`' under app's '`usability`' category and '`personal data`' mapped to '`privacy`' for the '`Privacy`' category.

Table 2: Summary of Certificate Signing Analysis; Top : Certificate signing mechanism ; Bottom: Public Key length present in Digital certificate.

Signature Algo.	**# of Apps, N=36 (%)**
SHA256 + RSA	31 (86.1%)
SHA1 + RSA (weak)	5 (13.8%)
Key Length (bits)	**# of Apps, N=36 (%))**
4096	15 (41.6%)
2048	21 (58.3%)

3 ANALYSIS

We explore the vulnerability of SymptomCheckers by identifying security gaps, suspicious behavior, level of communication security and user perspective. The results of the analysis present several aspects including; The strength of the encryption algorithm, exploitation of attack surface, intrusive permission, suspicious malware and anti-analysis appearance, adoption of secure communication protocol, and user-related security aspects.

3.1 Certificate Signing Mechanisms

We found 31 (86.1%) of the analyzed SymptomCheckers use vulnerable and weak encryption schemes such as SHA256 and RSA for signing and security purposes. While the remaining 5 (13.8%) apps were signed using a combination of SHA1 and RSA encryption mechanism thus potentially exposing legitimate SymtomCheckers to modification and repackaging by malicious developers (SSL, 2020).

We also include the length of the public key as part of the security measurement in the SymptomCheckers certificate signing scheme. We are referring to the minimum standard public key length of 2048 bits published by (CSRC, 2016). As shown in Table 2, all SymptomCheckers have met minimum security standards of public key length, where 21 (58.3%) apps use a public key with a length of 2048 bits and 15 (41.6%) apps leverage 4096 bits.

3.2 Permission Analysis

Figure 1 depicts the number and types of different permissions requested by SymptomCheckers. We found that 55% of SymptomCheckers request at most 10 permissions while 39% of the SymptomCheckers acquire resources protected by 'Dangerous' permission.

Out of a total of 427 permission requests, 73 % (312) permissions are categorized as dangerous, 20 % (84) were categorized as normal, and 8 % (33) permissions were categorized as signatures. The number of dangerous permissions is requested by 33 of the 36 apps on the SymptomCheckers list.

From a total of 79 unique type permissions, `INTERNET` is the most requested permission by 32 different apps while `WRITE_EXTERNAL_STORAGE` and `WAKE_LOCK` are requested by 22 and 19 SymptomCheckers, respectively.

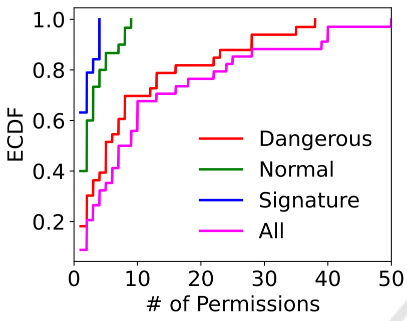

Figure 1: Empirical cumulative distribution function (ECDF) of permissions requested by SymptomCheckers.

3.3 Exported Component Analysis

We found 44.4 % (16), 58.3 % (21) and 63.86 % (23) of the analyzed SymptomCheckers apps contain *exported activities*, exported services and exported broadcast receivers, respectively. None of SymptomCheckers apps containing *exported content provider*. In total there are 74 Exported Activities, 42 Exported Services, 78 Exported Broadcast Receivers and none Exported Content Provider.

We group the apps based on the number of exported components in each app as shown in Table 3. We found that 11 (30.5 %) apps have exported components while 21 (58.3 %) apps have exported services, and 18 (50 %) apps have exported broadcast receivers, in the range of 1 to 5 respectively. In addition, there are 9 (25 %) apps that have exported components in the range of 6 to 10. We also notice massive exported activities by WebMD with 12 activities, indicating that there are 12 surface attacks that can be exploited by malicious activities in this app. Upon scanning SymptomCheckers using VirusTotal, we find 4 (10.81%) apps have malware codes in their source codes according to Virustotal. For instace, WebMD consists of 66 activities, 10 Services, 11 Broadcast Receivers, and 6 Content Providers.

Table 3: Summary of exported components; here Act = exported activities ; Ser = exported services; Rec = exported broadcast receivers; Pro = exported content Providers; A = number of apps ; C = number of exported components.

Range	**Act**		**Ser**		**Rec**		**Pro**	
	A	C	A	C	A	C	A	C
>10	1	12	0	0	0	0	0	0
6-10	4	36	0	0	5	30	0	0
1-5	11	26	21	42	18	46	0	0
0	20	0	15	0	13	2	36	0

3.4 Obfuscation Analysis

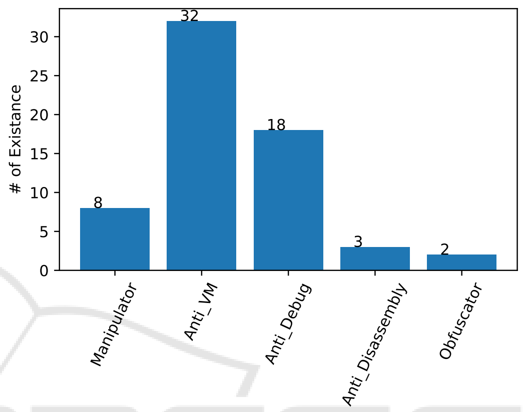

Figure 2: Anti-Analysis techniques adopted by SymptomCheckers. 89% SymptomCheckers apps leveraging anti-VM technique to detect emulated environment.

We found that 89% of the analyzed SymtomCheckers use at least one of the following obfuscation (or anti-analysis) techniques in their source codes.

- **Manipulator.** Each SymptomChecker's APK consists of *Dalvix Executable (.dex)* files converted from *Java.class* using (originally) *dx* compiler. We marked the apps containing the manipulator if the *.dex* files were created using other than *dx* compiler. As shown in figure 2, we found that 8 (22%) SymptomCheckers leverage *dexmerge* compiler to protect their source codes.
- **Anti Virtual Machine.** This technique is used to detect whether the apps are running on the emulator or real devices. The emulator detection aims to increase the difficulty level of apps running on emulators which impedes certain reverse-engineering tools and techniques.

  We found 32 SymptomCheckers use anti-virtual machine techniques to check the emulator presence based on various indicators in *build.prop* and *Telephony Manager*.
- **Anti Debug.** There are two levels of debugging as well as anti-debugging protocol in Android (OWASP, 2020). First, the debugging can

be conducted at the Java level using Java Debug Wire Protocol (JDWP) which is used as a communication protocol between debugger and Java Virtual Machine. Second, we can conduct debugging at the native layer level by using *ptrace* in Linux *system call*. This anti-debugging level check is also known as a traditional anti-debugging process. We found 18 (50%) SymptomCheckers tailor anti-debug techniques to avoid source code analysis tools. All of those apps detect debuggable state by activated *isDebuggerConnected()* routine in *android.os.Debug class*, which is part of JDWP anti-debugging level technique.

- **Anti Disassembly.** A common technique to protect the byte-code in Android is to create the important code segment in C or C++ using Native Development Kit (NDK)(SIMFORM, 2015). NDK provides platform libraries to manage native activities and access physical device components(Developer, 2020). NDK uses *CMake* as a native library compiler that creates a different *byte-code* structure compared to the code written in Java or Kotlin. Hence, it impedes common Android tools such as APKTool or Smali to disassemble the *byte-code*.

  We found 3 (8%) SymptomCheckers use anti-disassembly techniques, including AITibot , Appstronout and Mayo Clinic. Analysis of those apps returns the value of "illegal class name", indicating the decompiler result violating the standard structure of Java or Kotlin.

- **Obfuscator.** Code obfuscation is the process of modifying code into meaningless phrases by renaming or encrypting the file names, methods, or strings without reducing the code functionality. The aim is to protect the executable file from being analyzed or being reversed by an unauthorized party. Proguard and Dexguard are two most common tools utilized in Android programming obfuscation.

  We found 2 (6%) SymptomCheckers deploying obfuscators. Analysis results on Healthily return 'unreadable field names' and 'unreadable method names' indicating that a certain number of field names and method names in that app were renamed or encrypted. However, APKID failed to identify the tools used to conduct obfuscation. While analysis on Mayo Clinic identifies the apps adopting Dexguard to conduct the obfuscation.

Table 4: Top 5 Third Party Libraries in SymptomCheckers.

No	Third Party Libraries	Count
1	Google Firebase Analytics	16
2	Google AdMob	14
3	Google CrashLytics	11
4	Google Analytics	10
5	Facebook Login	9

Table 5: Summary of Traffic Analysis. 19% traffic directed to third-party domains and 1% of the traffic relying on unencrypted HTTP protocol.

	Unique URL	HTTP	HTTPS
First-party	1283 (81%)	69(4%)	1214 (77%)
Third-party	301 (19%)	5 (1%)	296 (19%)
Total	1584 (100%)	74 (5%)	1510 (95%)

3.5 Third-party Ads and Tracking Analysis

The results of the tracking library analysis show that 69% (25) SymptomCheckers adopt at least one tracker while 22% (8) of the analyzed apps leverage more than 5 tracking libraries in their code. For instance, `com.caidr.apk`, with 100,000+ installations and 4.1 average ratings, has the highest number (10) of tracking and advertisement libraries. The app consists of 26 activities where almost 50% of the activities are proposed to handle third-party libraries including trackers.

We also analyze the type of third-party libraries. We found that Google Analytics, Google Tag Manager, Google Admob, Google Firebase Analytic, and Google CraschLytic, and the Facebook group consisting of Facebook Ads, Facebook are the most integrated third-party libraries by SymptomCheckers. Table 4 shows the Top 5 third-party libraries used in SymptomCheckers. Although each library collects limited information, when all these libraries exchange information, it raises concerns about privacy violations because these third-party libraries can infer personal information based on the behavior and demographic data exchanges (Seneviratne et al., 2015).

3.6 Traffic Analysis

Overall, we obtained 1,584 unique URLs from the traffic of the analyzed SymptomCheckers. We found that 74 (5%) communications are done via HTTP protocol while 3 SymptomCheckers communicate 50% of their traffic un-encrypted using HTTP while 95% of the SymptomCheckers use HTTPS for secure communication.

To observe potential privacy leaks, we consider the existence of third-party libraries adopted by each

Table 6: Analysis of users' perception of SymptomCheckers analyzed through user reviews categories and keywords.

Complaint Category	#Comp (%)	# Apps (%)	Case-insensitive, Searched Keywords
Usability			
Bugs	429(20.64)	12(41)	force close; crash; bug; freeze; glitch; froze; stuck; stick; error; disconnect; not work; not working
Battery	74(3.56)	9(31)	battery; cpu; processor; processing; ram ; memory
Mobile Data	87(4.19)	10(34)	mobile data; gb; mb; background data;
Mal-behaviour			
Scam	28(1.35)	6(21)	scam; credit card; bad business; bad app
Adult	23(1.11)	6(21)	porn; adult; adult ad
Offensive/Hate	2(0.10)	2(7)	hate; offensive; sexist; LGBT; trolling; racism; offensive; islamophobia; vile word; minorities; hate speech; shit storm
Privacy			
Privacy	53(2.55)	7(24)	privacy; private; personal details; personal info; personal data
Ads	1280(61.60)	24(83)	ads; ad; advertisement; advertising; intrusive; annoying ad; popup; inappropriate; video ads; in-app ads
Trackers	43(2.07)	10(34)	tracker; track; tracking
Security			
Security	12(0.58)	4(14)	security; tls; certificate; attack
Malware	5(0.24)	3(10)	malware; trojan; adware; phishing; suspicious; malicious; spyware
Intrusive Permissions	42(2.02)	6(21)	permission

SymptomCheckers. Hence, we refer to Section 3.5 to create a list of third-party libraries and compare it to the intercepted unique URLs. We then categorize the corresponding URLs as traffic that directed to third-party domains and the rest is traffic that directed to domains provided by the app developer (first-party) or domains that are used to support apps functionality. The percentage of traffic leading to a third-party domain is an indicator of a possible privacy leak.

We found 81% of the SymptomCheckers' traffic is destined towards the first-party domain or apps supporting domain. However, out of 1,584 total traffic intercepted, there are 301 (19 %) unique URLs that point to third-party domains. Moreover, 8 (22%) apps had traffic leading to a third-party with more than 50% of the total traffic. This indicates that the app uses more than 50% of its functionality for third-party libraries.

We provide a summary of traffic analysis in Table 5. Of 1,584 intercepted unique URLs, there are 1,510 (95%) traffic transmitted over the encrypted HTTPS protocol line and 74 (5%) traffic transmitted via the un-encrypted HTTP protocol. In addition, of the total Unique URLs, there are 1,283 (81%) URLs pointing to first-party or supporting domains and 301 (19%) URLs pointing to third-party domains. In more detail, 69 (4%) of traffic to the first-party domain is transmitted via the un-encrypted HTTP and 1,214 (77%) of traffic to the first-party domain is transmitted via the encrypted HTTP. While, 5 (1%) and 296 (19%) unique URLs were directed to the third-party domain via HTTP and HTTPS, respectively.

3.7 User Perception Analysis

We obtain N = 4,807 negative (1 or 2 stars) reviews to analyze the users' perception of app usability, malicious behavior (or mal-behavior), privacy, and security. Users often complained about apps' usability (see Table 6) where bugs, battery and mobile data issues caused frustration. We found that 429 (20.64%) complaints were made for 12 (41%) apps where users raised their concerns and requested to fix the usability issues to improve the user experience. While some might find the app useful but an apparent usability issue may hinder the useful experience. Some users stated that "great app is very useful; however I have gotten a lot of 'force close' messages please fix", "would not load pictures and kept freezing" or "had to uninstall as they kept crashing every time you open the app".

Apps' mal-behavior had captured minor users' attention where such apps were listed as scams and raised concerns about presenting adult or offensive content. The user stated as "No way to verify that this is a legitimate app. No clear connection to the NHS. Looks like a scam to steal personal information".

Notably, under the privacy category users mainly complained about in-app advertisements 1280 (61.60%) for 24 (83%) reviewed apps.Users elaborated "don't want seeing this app advertisement through my phone", "I need to answer your questions and all I see is advertisement..." or "... I cannot find anything not my blood type not my last appointment everything is advertisement...". Comparatively user reviews suggest little understanding of apps privacy and tracking behaviour. However, some users stated “privacy policy is bad”, “sharing personal data is not cool“ or “removed the app after finding out that it shared private health data with third-parties”.

Similar to privacy users’ perception of apps security was also limited. There were minor complaints made for app security, presence of malware and app requiring intrusive permissions as shown in Table 6. Users mentioned their concerns for specific apps as “security risk, asks for personal details that would facilitate identity theft. if in doubt be cautious” or “spyware, why does it need my device serial number; my location; read SD card; to record audio when I get up and go to sleep; all combined with full internet access?”.

Overall, only bugs (20.64%) in 12 apps and ads (61.60%) present in 24 apps captured the most users’ attention where other privacy and security issues were mainly neglected.

4 CONCLUSION

The increasing number of mobile SymptomCheckers available on apps’ markets such as Google Play and the growing number of complaints raised by users indicate serious security and privacy as well as usability issues thus necessitate the urge to analyze this unexplored eco-system. The average mobile user rates SymptomCheckers positively even when 10% have malware presence. According to our study, 3% of negative reviews are related to (or concerned with) the security and privacy issues of SymptomCheckers. Our app review analysis suggests an alarming situation as users show minimal interest or awareness of SymptomCheckers privacy and security.

The results found in this paper reveal that the exploitation of SymptomCheckers is present in popular apps, still readily available on Google Play and we believe that our work could be extended to study the qualitative analysis such as users satisfaction and effectiveness of SymptomCheckers.

ACKNOWLEDGEMENTS

The first Author is a Scholarship Awardee of the Indonesian Endowment Fund for Education (LPDP) of the Republic of Indonesia, with the LPDP ID 20193221014024. This research is partially funded by the Optus Macquarie University Cyber Security Hub.

REFERENCES

Apktool (2020). A tool for reverse engineering 3rd party, closed, binary android apps. https://ibotpeaches.github.io/Apktool/.

Backes, M., Bugiel, S., Derr, E., McDaniel, P., Octeau, D., and Weisgerber, S. (2016). On demystifying the android application framework: Re-visiting android permission specification analysis. In *USENIX Sec*.

Chau, N. and Jung, S. (2019). An entropy-based solution for identifying android packers. *IEEE Access*.

CSRC (2016). Recommendation for key management, part 1: General (revision 3), computer security resource center. https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-3/archive/2012-07-10.

CWE-926 (2020). Cwe-926: Improper export of android application components. https://cwe.mitre.org/data/definitions/926.html. Accessed: 18/12/2020.

Developer, A. (2020). The android ndk: toolset that lets you implement parts of your app in native code, using languages such as c and c++. https://developer.android.com/ndk.

F-Secure-Labs (2020). Drozer: Comprehensive security and attack framework for android. https://labs.f-secure.com/tools/drozer/. Accessed: 18/12/2020.

He, R., Wang, H., Xia, P., Wang, L., Li, Y., Wu, L., Zhou, Y., Luo, X., Guo, Y., and Xu, G. (2020). Beyond the virus: A first look at coronavirus-themed mobile malware.

Ikram, M. and Kaafar, M. A. (2017). A first look at mobile ad-blocking apps. In *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, pages 1–8. IEEE.

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

matlink (2018). Google play downloader via command line v3.25. https://github.com/matlink/gplaycli. Accessed: 10/10/2020.

Melamed, T. (2020). Hacking android apps through exposed components. https://www.linkedin.com/pulse/hacking-android-apps-through-exposed-components-tal-melamed.

mitmproxy (2020). - an interactive https proxy. https://mitmproxy.org.

Oracle (2020). keytool. https://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html. Accessed: 18/12/2020.

OWASP (2020). Testing anti-debugging detection (mstg-resilience-2) - android anti-reversing defenses. https://mobile-security.gitbook.io/mobile-security-testing-guide/android-testing-guide/0x05j-testing-resiliency-against-reverse-engineering. OWASP Mobile Security Guide - Accessed: 18/01/2020.

RedNaga (2016). Apkid - anti-analyisis open source tools. https://github.com/rednaga/APKiD.

Seneviratne, S., Kolamunna, H., and Seneviratne, A. (2015). A measurement study of tracking in paid mobile applications. In *WiSeC*.

SIMFORM (2015). How to avoid reverse engineering of your android app? https://www.simform.com/how-to-avoid-reverse-engineering-of-your-android-app/.

Smahel, D., Elavsky, S., and Machackova, H. (2019). Functions of mhealth applications: A user's perspective. *Health informatics journal*, 25(3):1065–1075.

SSL (2020). Why you need code signing certificate for your android app? https://aboutssl.org/why-you-need-code-signing-certificate-for-android-app/.

Trifan, A., Oliveira, M., and Oliveira, J. L. (2019). Passive sensing of health outcomes through smartphones: Systematic review of current solutions and possible limitations. *JMIR mHealth and uHealth*.

VirusTotal (2020). Multitude anti-virus engines. https://www.virustotal.com/gui/home/upload.