On the hunt for mobile malware traces

Al Hareth Salem Adel
Information Security and Communication Technology
Norwegian University of Science and Technology
Gjøvik, Norway
ahadel@stud.ntnu.no

Sander Hauknes
Information Security and Communication Technology
Norwegian University of Science and Technology
Gjøvik, Norway
sanderah@stud.ntnu.no

Marie Heng
Information Security and Communication Technology
Norwegian University of Science and Technology
Gjøvik, Norway
marieshe@stud.ntnu.no

Quentin Jury
Information Security and Communication Technology
Norwegian University of Science and Technology
Gjøvik, Norway
quentinj@stud.ntnu.no

Niklas Menzel
Information Security and Communication Technology
Norwegian University of Science and Technology
Gjøvik, Norway
niklasme@stud.ntnu.no

Torben Vier
Information Security and Communication Technology
Norwegian University of Science and Technology
Gjøvik, Norway
torbenvi@stud.ntnu.no

***Abstract*—Malware attacks on mobile devices have increased significantly in recent years. In 2017 Android overtook Windows as the most popular operating system. As mobile computing devices become more prevalent and are used for work, online banking, smart home control, and in general taking over the tasks performed on regular computers, they are becoming increasingly more lucrative as a target for malware developers. Thus, the focus of attackers is increasingly shifting away from computers towards mobile devices such as cell phones and tablets.**

In this project, we expect to focus on Android devices as they have a higher market share and their more open nature makes them more vulnerable to malware than the "wall gardened" iOS devices. This also makes them easier to investigate as we have greater access to the system.

Our expectations is to observe the differences between malicious and non malicious software and how they behave on the system. Our focus will be more on the side of looking for traces of the malware on the system rather than reverse engineering the sample. An example of something we would like to investigate is the difference in behavior between a Trojan and its non-malicious counterpart and if there is a set of behavior one can look for to quickly discover an attack. To be able to run the malware without infecting our own machines we are planning on using a sandbox environment.

I. Introduction

The number of Android mobile users is expected to increase from 3.2 billion users in 2019 to 3.8 billion users in 2021 according to Statista. [1] Making Android the most used mobile operating system worldwide with over 70 percent market share. [2] The high share of Android users makes it attractive for mobile attackers to spread their malware in Android apps. Due to the broad use of mobile devices in various areas such as work, education, shopping or banking the temptation to plant malware got unstoppable. [1]
In Nokia's Threat Intelligence Report 2020 the Covid-19 pandemic was also pointed out as a trigger to people's recent fear, which is a key weakness for cyber criminals. That is how new malware types specifically related to Covid-19 are created. One of them is the "COVIDLock Android Ransomware", which is a Trojan Android app disguised as a tracking app about the worldwide coronavirus spread, but then would lock the device demanding ransom-pay for unlocking. [3]

Concerning the maintaining topicality of this issue, five Information Security Master students decided to get first insights into mobile malware analysis for their group project in IMT4114 Introduction to Digital Forensics course. The project started off with the group getting familiarized with well-known or current Android malware. Then literature research was done to gain an idea and better understanding of the mobile malware analysis techniques, process and tools. The whole group project process from start to completion of the project will be documented in this report.

Therefore, the rest of the report will be continued with a Literature Review in chapter 2 for describing the mobile malware analysis. Then in chapter 3 the selected malware types Cerberus, CovidLock and FluBot (or TeaBot) will be introduced. From chapter 4 on will be the practical part of the project beginning with analysing the obtained API calls from the malware samples for the static analysis part. Then chapter 5 will contain the dynamic analysis part. Since the dynamic analysis will be more extensive than the static analysis that chapter will be divided into two sub chapters, in which one describes the setup of the hands-on analysis experiment and the other for elaborating the execution of the experiment. Chapter 6 contains the results of the experiment, followed by chapter 7 with the conclusion to the experiment and the project.

II. Related work

In this section, we will describe the methodology of malware analysis and what are the methods of analysis. For this purpose, we will take example of the previous tests and research that have been done. [1] [4] [5] [6]

Because of the increase in the number of android devices sold, criminals have changed their target and now focus on stealing sensitive information from people's mobile. Thereby, in the last years, researches on the analysis and detection of malware began to emerge. To improve the android security system, researchers compared the behaviour of multiples applications and try to detect which one are malicious. All malware work differently from each other and thus it is important to understand how they work. It is therefore necessary to analyse malware. To extract features from devices and thus detect malware, three methods of analysis exist.

The first one is the static analysis. The static analysis examines the binary code of the potential malware without actually running the code. In this way the system is prevented of being infected by the malware. In general, this method implies the determination of the signature of the binary file. This signature is unique for each file and gives them their identity. It is possible to find the signature of the file by calculating the cryptographic hash of the file and understanding the components. Then, by using reverse-engineering, the file is disassembled in assembly language understandable by analysts. Finally, looking at the disassembled code makes it easier for analysts to understand how the malware works and how dangerous it can be to the system. This method can be done by other means than reverse-engineering such as file fingerprinting, virus scanning or analysing memory for example. However, this method is not effective against strong and sophisticate malware.

The second method of analysis is the dynamic method. Unlike the static analysis, the code must be executed in the dynamic analysis. By running and studying the code in a safe environment that will not harm the system, it is possible to observe the behaviour of the malware. By discerning the comportment of the malware, analysts can remove the infection of the malware or at least stop the spreading in other systems. In some advance dynamic analysis, a debugger can be used in order to determine the functionality of the malware. This method is behaviour-based and therefore analysts cannot miss important behaviour of the malware. Hence, this method is effective against all malware. However, this method is cost and time consuming. In addition to that, malware that have an anti-virtual machine or an anti-emulator will not be detected.

The last method of analysis is the hybrid analysis. As its name suggests, this method uses both static and dynamic analysis. Thereby, it is possible to overcome both limitations of the two methods. In this method, first comes the analysis method. Analysts will inspect the signature of the malware. Then, they will continue by analysing the behaviour of the malware. Thus, this method has the advantages of both methods while avoiding their disadvantages. However, this method is time and resource consuming because it needs both methods.

To be able to analyse correctly and without danger the malware during a dynamic analysis, it is important to run the code in a safe and isolated environment. That is why analysts run the code in a sandbox. A sandbox is a safe environment where any applications (here malware) can be run without any consequences or danger for the system. In this way, analysts will run the malware without worrying of the integrity of the system and device. A sandbox environment has most of the time only the necessary requirements to analyse the malware to be able to protect the system correctly and efficiently.

The increase in the number of malware can be explained by the fact that it's easier for an attacker to infect the device. In addition to that, android has some weaknesses that the attacker can take advantage of. In fact, people are less likely to be vigilant with their phones (or tablets but mainly phones) than they would be with their computers. Furthermore it is also less likely that people install antivirus or any malware scanners on their phone. The rapid increase in the number of android devices in recent years is a threat because there are more targets for the criminals. The evolution of the technology is also a threat because nowadays criminals cannot only find banking credentials but also a lot of personal and sensitive information. Because Android is that popular, the field of applications has also grown. We can find a lot of applications in the Google Play Store. Because of this high number of applications, it is impossible to review in details every single application. That is why some of these applications are malicious. Even more, it is possible to find applications from third parties (external to the Google Play Store) and these applications are even more likely to be malicious.

The Android permissions system is the main key of the security of Android application system. To be able to work correctly, applications need a set of permissions. There are a lot of different permissions with different level of danger. A permission can control for example the use of contacts, alarm, sending/receiving SMS, or access the storage. This system is a defence against malicious applications because users need to consent to the permissions before using the application. However, this is also the main breach for attackers and malicious applications. Indeed, most of users will not read or understand the permissions that the application needs whereby they will accept it immediately.

Most of the time, the methodology for the research of malicious applications is the following: first comes the data collection, then the data examination and finally the risk assessment. To collect the data, analysts will take both malicious and benign applications from different sources. They will then validate their samples using a malware scanner or other software. Then comes the data examination step which includes sometimes machine learning. The APK file will be disassembled in its originally resources. Subsequently the permissions required by an app can be extracted from the

AndroidManifest.xml file. Then for each sample, according to the permissions the application needs, they will either be considered as benign or dangerous with different level of danger: low, medium, high (or low/high/critical). For our project we have followed this procedure.

III. Malware samples

After our literature research we got a profound overview of methods and procedures of malware analysis, and we started with the first practical step of our project. This was the selection of suitable Android malware samples to be examined in our forensic analysis. Our goal during the selection of malware examples was to take recent examples that are widespread. In order to search for suitable samples, we used different websites including Malpedia [7], MalwareBazaar [8], Androzoo [9], VirusTotal [10], and MalShare [11]. VirusTotal, for example, gave us a good overview and initial indication of individual samples, which was very helpful especially in the selection process. In Figure 1 an example report from VirusTotal is shown. Besides a summary of analyses from various known antivirus softwares and malware detection sites, the VirusTotal report also includes meta-data, behavioural attributes, and relationships to other samples included in the VirusTotal database.
During our search on these online sources, we noticed that the majority of Android malware samples submitted there were banking Trojans. These kind of malware infiltrate the victim's smartphone and capture the login credentials of the victim's online banking accounts. Two very popular banking Trojan malware are Cerberus [12] and FluBot. In addition to these two malware families we picked CovidLock as a third malware. Unlike the first two, this is ransomware. The aim of this malware is to encrypt all files on a phone and then extort money from the victim in order for the victim to get the decryption keys.
In total, we have selected three different malware sample families and we will present the three selected malware samples in the following in more detail.

A. Cerberus

Our first malware family is the banking Trojan Cerberus. It is a completely new developed banking Trojan without relations to other predecessor like Anubis or Exobot [13]. The main attack goal of the Trojan is to steal the victim's online banking credentials and intercept two-factor authentication codes sent via SMS [12]. The Trojan is hidden in various supposedly harmless apps such as Vodafone5G.apk, Chrome.apk, or currently in the context of the global pandemic Corona-Apps.apk. However, not only bank account credentials are in focus, but also credentials for other popular services outside the banking sector. Among others, Threatfabrik has identified the following service as a target of Cerberus: U.S. Bank, ING, Chase Mobile, Capital One® Mobile, GMail, Outlook, Instagram, and Uber [13].
In order for Cerberus to obtain the victims' credentials, the malware offers various functionalities. The most important ones are dynamic display of overlays, keylogging, receiving, reading, and sending SMS messages, and extracting a list of all installed apps. After the malicious app is installed and started, Cerberus establishes a connection with the Command & Control Server (C2 server). Using a list of commands, the C2 server can request information about the infected smartphone and launch targeted attacks [12]. An example procedure would be to first request the list of available apps and then use a customized overlay to trick the user into entering the credentials for the selected app. For our analysis we used the sample with the following SHA256 hash: *c5888a824cba71e4c6ebca35aad174120f8b681e48af867dfa95ef6a4ba0dc25*

B. CovidLock

The second malware we have selected is CovidLock [14]. As the name suggests, the malware aims to blackmail people by abusing their fears about the corona virus during the global pandemic. The supposedly harmless app promises to provide real-time statistics and information about the Corona virus at the victim's current location. Furthermore, the app offers a notification function if there is an infected person in the surrounding area. Of course, this functionality is not really offered by the app.
Unlike Cerberus, this malware is not a Trojan horse with the goal of obtaining bank credentials, but a ransomware. The attackers' goal is to infiltrate the victim's smartphone, encrypt the data, lock the screen, and then demand a ransom in the form of Bitcoins. The ransomware attack consists of two phases:

1) Spread of the application (CoronaVirus Tracker)
2) Infiltrate and lock victims smartphone

The app is distributed via the website "*coronavirusapp.site*". This website provides a dashboard with the current number of coronavirus cases worldwide and a note that this service is also available as an Android app.
After downloading the CoronaVirus Tracker, the user is asked for various permissions. We will go into more detail about this in sections IV and V as part of our static and dynamic analysis. For our analysis we used the sample with the following SHA256 hash: *c844992d3f4eecb5369533ff96d7de6a05b19fe5f5809ceb1546a3f801654890*

C. FluBot/TeaBot

Our third and last malware family is the FluBot malware also known as Anatsa or TeaBot. Just like Cerberus, FluBot has a similar attack vector and functionalities. The malware can be rated as a banking Trojan but there are also other possible attack targets. FluBot malware is spread via SMS phishing, also known as smishing [15].
A classic example is an SMS notification that a package will be delivered soon and an attached link. On the linked

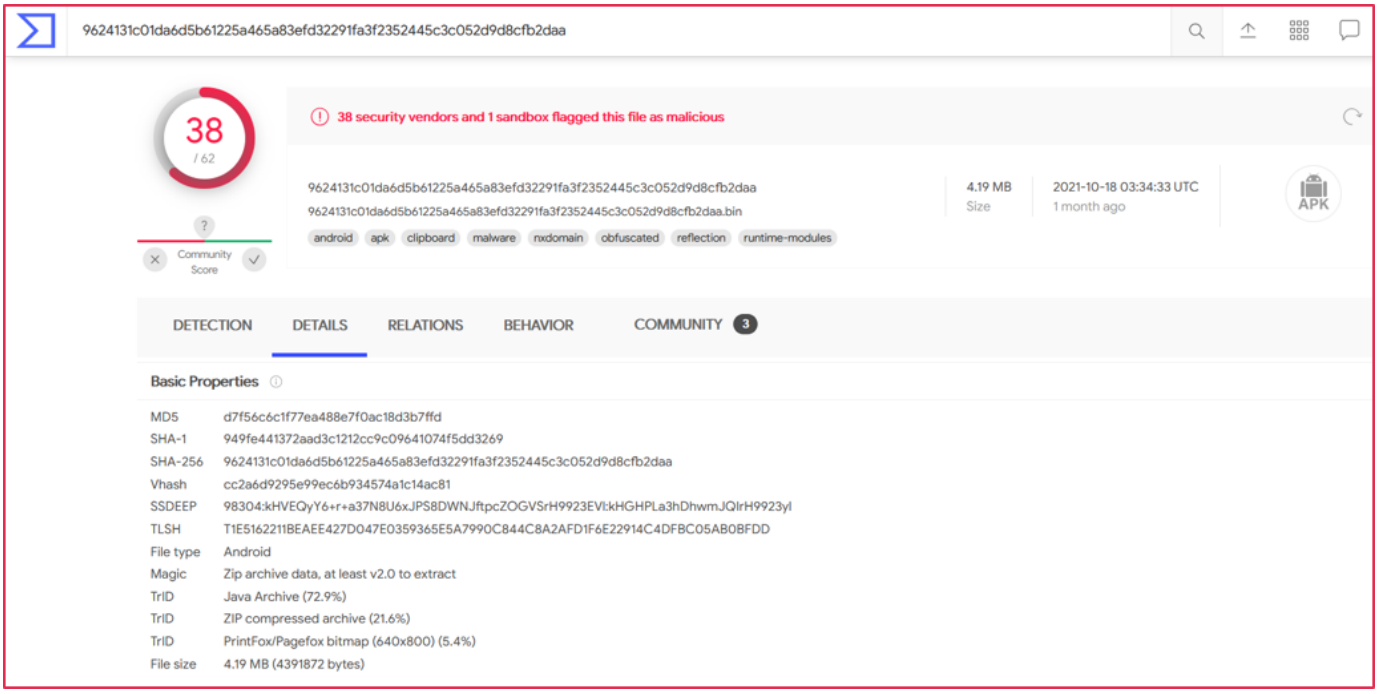

Fig. 1: Example of a VirusTotal report for a malware sample

website, a malicious app version of the parking company is offered for download. Subsequently after downloading and launching the app the victims smartphone gets infected by the FluBot. FluBot then notifies the C2 server that it has infected another device. An essential functionality of the malware is extracting contacts from the address book and forwarding them to the C2 server. The numbers contained there are used for further smishing attacks. The other functionalities are similar to those of Cerberus (Section III-A) and are therefore not mentioned again here [16]. For our analysis we used the FluBot sample with the SHA256 hash: *9624131c01da6d5b61225a465a83efd32291fa3f2352445c3c052d9d8cfb2daa*

IV. Static analysis

A. Tools

A number of different tools were used to conduct the static analysis. One thing we did first was look up the malware on virustotal. The thing we used virustotal for was looking up how often the malware had been flagged by security vendors. To dissasemble the malware we used APKTools. [17] Using APKTools we could get access to the AndroidManifest.xml file which we used to get information about the persmissions the malware had on the system. APKTools also gave us access to the strings.xml file which give us information about strings used in the malware. For CovidLock we also used Dex2jar to convert the APK file into a .jar file and jd-gui to read the jar file. [18]

B. Cerberus

According to Virustotal only 10 out of 59 security vendors falgged Cerberus as malicious. [19]

We disassembled the apk file with the help of Apktool. As described in several similar scientific works, the AndroidManifest.xml provides insightful information about the permissions required by an app. In total there were 21 permissions requested by the app. You can also get information about the permissions using online analysis websites such as Pithus. Pithus flagged 10 of the permissions as ”high risk”. [20] As stated earlier the malware has functionality relating to reading, writing, sending and recieving sms. The malware contains requests to the permissions READ_SMS, WRITE_SMS, SEND_SMS and RECEIVE_SMS, which does as their names imply give the malware the ability to read, write, send and recieve SMS. The permission request REQUEST_INSTALL_PACKAGES allows the malware to request installing packages. The permission GET_TASKS allows the malware to retrieve information about current and previously running tasks. The malware asks for permission to use internet as evidents by the the call to android.permission.INTERNET. For Cerberus there was no strings.xml file created by APKTools that we could use to find information about the strings used for the program.

In addition we looked up the the malware on MITRE ATTCK as Cerberus had a page on attack.mitre.org. MITRE ATTCK confirmed a lot that we had already discovered such that the malware can get the contact list, collect and send sms messages. There was also things which we weren’t able to find in the static analysis such that the malware having the ability to track the device. According to MITRE ATTCK Cerberus is masquerading itself as an adobe flash player installer.

C. CovidLock

Our starting point for the static analysis of the CovidLock malware sample was also the report from Virustotal [21]. Compared to the Cerberus sample, significantly more sites tagged the CovidLock apk as malicious or dangerous, 29/62 sites in total. Next, we disassembled the apk with the help of Apktool [17]. As described in several similar scientific works, the AndroidManifest.xml provides insightful information about the permissions required by an app. Since Apktool has already provided us with the AndroidManifest.xml, we only need to search for the string ”permission” with grep in order to get an overview of all permissions. Our result is documented in the following list:

- FOREGROUND_SERVICE
- RECEIVE_BOOT_COMPLETED
- REQUEST_IGNORE_BATTERY_OPTIMIZATIONS
- BIND_DEVICE_ADMIN
- BIND_ACCESSIBILITY_SERVICE

To further analyze the app’s source code, we used dex2jar to convert the apk into a jar file and then used jd-gui [18] to search for interesting code snippets. Since we know that CovidLock is ransomware, our focus lay on finding CovidLock’s encryption/blocking feature. In the BlockedAppActivity.class, we came across the *verifyPin()* function. The function is responsible for unlocking the infected device after paying the ransom. For this purpose, it checks whether the key entered matches the one defined by the ransomware. It is noticeable that the developers of the ransomware have hardcoded the pin for unlocking the device in the source code. As a result, every victim receives the same key upon payment, and secondly, it is not necessary to pay the ransom because the key can already be found online. If someone becomes a victim of CovidLock ransomware, all he has to do is enter ”4865083501”.

Another place we looked for malware traces is the strings.xml file in the ”*res/values*”-folder. According to the Android developers guide, the file ”...provides text strings for your application with optional text styling and formatting.” [22]. We were able to extract the entire ransomware message from the strings.xml file. The ransom note is shown in Figure 2. The images of the app from DomainTools’ [14] analysis confirm that this is the ransom text, which is displayed on the locked smartphone. As a conclusion, it can be said that the developers of CovidLock ransomware did not make much of an effort to cover their tracks and make static analysis difficult, as the hardcoded unlock key has already shown.

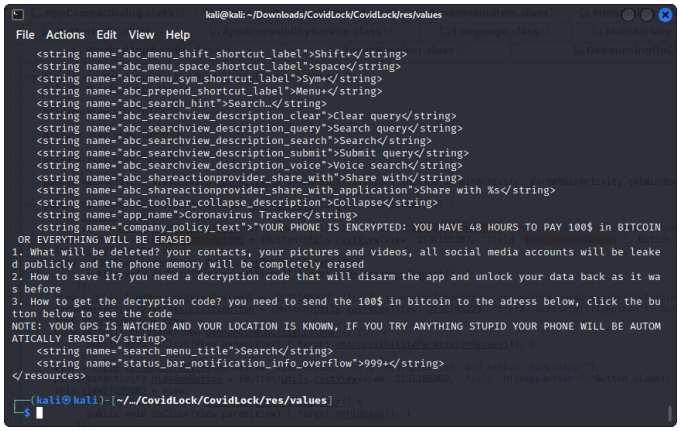

Fig. 2: Extracted ransom note string from the CovidLock.apk

D. FluBot/TeaBot

FluBot is flagged as malicious by 39 out of 64 security vendors according VirusTotal, this is far more than for the similar Cerberus malware. The malware was as with previous samples dissambled using APKtool. FluBot has a total of 19 android permissions. Of these about 7 was flagged as ”high-risk” by Pithus.

- READ_SMS
- WRITE_SMS
- SYSTEM_ALERT_WINDOW
- SEND_SMS
- READ_PHONE_STATE
- RECEIVE_MMS
- RECEIVE_SMS

Just like Cerberus this malware has the ability to read, write, send and receive sms. Having requested the same permissions as Cerberus (READ_SMS, WRITE_SMS, SEND_SMS and RECEIVE_SMS) to do so. The SYSTEM_ALERT_WINDOW permission allows the malware to take over the entire screen of the phone. In the strings.xml file located in the ”res/values” folder we can find that the malware contains strings telling the user to sign-in with Google. This indicates the malware is trying to steal the Google login credentials from the infected user. TeaBot also has multiple messages asking the user to update Google Play. With the knowledge we have of the malware having the ability to show system alert windows, this could be a way the malware tricks the user into giving downloading malicious packets thinking it’s an update for the Google Play store. In the folder we could find multiple images of DHL logos. This relates to Teabot using phishing to trick the user into thinking it has a package delivered from DHL prompting them to click on a malicious link.

```
<string name="common_google_play_services_enable_button">Enable</string>
<string name="common_google_play_services_enable_text">%1$s won't work unless you enable Google Play services.</string>
<string name="common_google_play_services_enable_title">Enable Google Play services</string>
<string name="common_google_play_services_install_button">Install</string>
<string name="common_google_play_services_install_text">%1$s won't run without Google Play services, which are missing from your device.</string>
<string name="common_google_play_services_install_title">Get Google Play services</string>
<string name="common_google_play_services_notification_ticker">Google Play services error</string>
<string name="common_google_play_services_unknown_issue">%1$s is having trouble with Google Play services. Please try again.</string>
<string name="common_google_play_services_unsupported_text">%1$s won't run without Google Play services, which are not supported by your device.</string>
<string name="common_google_play_services_update_button">Update</string>
<string name="common_google_play_services_update_text">%1$s won't run unless you update Google Play services.</string>
<string name="common_google_play_services_update_title">Update Google Play services</string>
<string name="common_google_play_services_updating_text">%1$s won't run without Google Play services, which are currently updating.</string>
<string name="common_google_play_services_wear_update_text">New version of Google Play services needed. It will update itself shortly.</string>
<string name="common_open_on_phone">Open on phone</string>
<string name="common_signin_button_text">Sign in</string>
<string name="common_signin_button_text_long">Sign in with Google</string>
```

Fig. 3: Strings extracted from TeaBot

V. Dynamic analysis

After the static analyses we wanted to go a step further and dig deeper into Android malware patterns. In order to get a better insight into the malware behaviour we decided to apply dynamic analysis to the samples. When conducting a dynamic analysis on malware one of the most important aspects to consider is having a controlled test environment. Configuring the test environment can be tedious and challenging, especially if it is your first time. Moreover, the nature of these types of malware requires for instance a complex configuration of a relevant network topology in order to understand how they behave, which makes the dynamic analysis even more challenging. We want to be able to conduct a dynamic analysis in a scalable way that allows for repeatable experiments in order to confidently say something about how the malware sample behaves. Over the years we have been facing increasingly dangerous malware and as such it is imperative that we have the right tools and the capabilities of dissecting and analysing these types of malware in an effective way. Since we had no experience in manually debugging Android software we decided that running the samples in a safe environment such as a sandbox was the best approach. This also allows us to keep track of the programs state. Most notably this did not only allow us to log which APIs where actually used at run-time but also to keep track of the parameters and return values passed to or returned by the API being invoked. At first we tried to use an open source sandbox for Android malware, namely Android Malware Sandbox [23]. The sandbox uses the emulator and the Android Debug Bridge (ADB) - that are delivered with the Android SDK [24] - in order to run and connect to virtual machines that contain the target Android system. Furthermore it uses Frida [25] which is a JavaScript based toolkit that allows to inject code into the emulated system. By doing so the sandbox can hook important APIs and thus is able to report parameters and return values back to the client if one of the APIs is invoked. While the sample is running information are stored in a local database. After the analysis finishes a post processing step is launched that translates the logged data into a human readable format which is then saved in an HTML file. Getting the system up and running already included a lot of work since the Android Malware Sandbox repository is neither very well maintained nor recently updated. First we had to set up an emulator with a frequently used Android version in Android Studio. We decided to use Marshmallow 6.0 since it had a good market share [2] over last couple of years. Subsequently we had to revise the *requirements.txt* delivered in the repository in order to install the right Python dependencies. After that we could finally start using the sandbox. We first tested the sandbox for a few samples that were mentioned in the repository. For these samples the sandbox created useful outputs. Unfortunately this wasn’t the case for our modern malware samples. Each sample showed a different kind of behavior during analysis but all of the analyses failed. We suspect the three samples to use sophisticated anti emulation and obfuscation techniques that the Android Malware Sandbox

can neither detect nor prevent. For the Cerberus we were able to take a closer look into the anti emulation techniques deployed by the malware authors. We will now give a short overview on the execution of our experiments.

A. CovidLock

The analysis for the CovidLock sample started normally and the app was displayed in the emulator. After a short period of time the execution stopped and the user is informed that the app crashed. Figure 4 shows the app in the emulator. To our disappointment the report was very short and only contained a few entries. They were about the app accessing cache files. These information were not sufficient to infer anything from them.

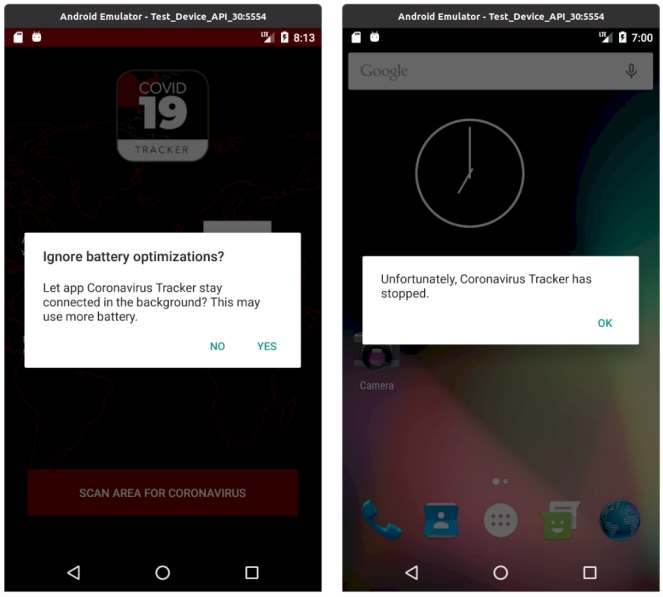

Fig. 4: CoronaLock crashing

B. FluBot/TeaBot

For the Flubot the results were even more disappointing since the app couldn't be loaded by the Android Malware Sandbox. The error message reported that no application with the given identifier could be found after installation as shown in Figure 5. We couldn't resolve this issue but we suspect it occurring due to an obfuscation technique that hides the actual entry point of the app.

```
DEBUG Frida:start()
ERROR unable to find application with identifier 'cake.ice.reform'
DEBUG Device:uninstall_application()
```

Fig. 5: Error message displayed for the Flubot sample

C. Cerberus

The Cerberus sample also could be started in the Emulator but it hung up in an infinite loop. Luckily we found a publicly available analysis done by JoeSandbox [26]. It showed us a clear hint on why the samples wouldn't behave as usual in our Sandbox setup. The reason were anti emulation techniques deployed by the malware authors.

To keep this short, JoeSandbox is a commercial product that allows users to upload any Android sample which is then run in their highly developed sandbox. Since we couldn't get an account we had to rely on publicly available analyses. We will now take a detailed look at the behavior of the Cerberus sample which hung up the Android Malware Sandbox in an infinite loop. The analysis of Joe Sandbox shows that the malware utilizes multiple techniques to evade dynamic analysis systems. First of all it tries to detect the QEMU emulator which is used by the Android Emulator shipped with Android Studio. It does so by querying the system property *ro.kernel.qemu* which is present on QEMU emulated systems. It then also queries information about build information of the OS running. This includes information on the OS version, the device and the manufacturer which all are pretty generic on our system. Moreover it does access the systems CPU details which can be a good hint for the sample that it is running in an emulated environment.

The Android Malware Sandbox implements some basic mitigation for some of the anti emulation techniques used by the Cerberus sample. They are implemented in the *anti-emulator.js* script in the repository. As shown in Listing 1 the Android Malware Sandbox actually replaces the values of build constants such as the manufacturer and the brand. Besides of that it also fakes the return values of important functions like *getDeviceId* or *getLine1Number* which is responsible for returning the phone number of the device.

```
const Build = Java.use("android.os.Build")
...
replaceFinaleField(Build,
    "MODEL", "C1505")
replaceFinaleField(Build,
    "MANUFACTURER", "Sony")
replaceFinaleField(Build,
    "BRAND", "Xperia")
replaceFinaleField(Build,
    "SERIAL", "abcdef123")
```

Listing 1: Anti-Emulation mitigation in Android Malware Sandbox

However, there seem to be two flaws in the sandboxes mitigation techniques in regards to our sample malware. The first one is pretty obvious since the sandbox doesn't implement any mitigation for the case that the malware checks the systems CPU details. The other flaw is more on how the mitigation are implemented. The authors used static values as their fake return values which means that the emulator is still easy to detect for a malicious actor if they are aware of this open source project. A malware author could simply add values like the string *"abcdef123"* to their heuristics used when checking the string stored in *android.os.Build.SERIAL*. This would be very effective since it is very unlikely for an android build to have this exact serial but the Android Malware Sandbox

will always return this value. The same could obviously be done for the values returned by functions like *getDeviceId* or *getLine1Number* which are supposed to be unique for each device.

VI. Execution of the experiment

We went into this project with the mindset of adhering to the principles mentioned above, but the anti-emulation techniques in the malware made it nearly impossible to do so. This forced us to base the dynamic analysis on publicly available ones like Joe Sandbox. Although this sandbox provides extensive and detailed reports on the behaviour of our malware samples, it is still a closed-source service, which presents a challenge for us when elaborating on the execution of the experiment. They do however provide details as to how their sandbox works in practice. Joe Sandbox uses for instance Hypervisor Based Inspection which not only lets it detect API calls in both user-mode and kernel-mode, but also makes it substantially more difficult for the malware to detect that it is being run in an analysis environment [27].

With a plethora of available malware analysis sandboxes with different analysis techniques and frameworks it is difficult for a user to determine which one offers the best services. Besides, as we experienced with Android Malware Sandbox, some of the sandboxes are not regularly maintained. We needed a reliable analysis of high-quality fitting of an academic report. At first glance, Joe Sandbox looked like the right choice for us due to the very detailed analysis it provided for our samples. This notion was further substantiated by the fact that Joe Sandbox performed remarkably well in a study comparing several Android sandboxes in 2014 [28]. It is by no means the ultimate analysis tool, as different sandboxes will perform differently based on the sample you run due to the complex nature of the processes involved.

Our goals with the dynamic analysis were to discover more about the behaviour of the malware than what the static analysis provided. It is widely known that well developed modern malware uses code obfuscation or encryption to thwart attempts at static analyses [29]. Not only that, but over the years malware has began implementing detection heuristics aimed at detecting virtualized and/or sandboxed environments. As mentioned, we discovered this early on, which prompted us to research this subject more. We found an interesting study [30] that found more than 10,000 emulation/virtualization detection heuristics. These detection heuristics (red pills) can easily be implemented by malware developers to find out the environment the app is being ran in. Some of the easiest ones to implement are the Android API call *getDeviceId* as well as checking if the file ***/sys/qemu_trace*** exists. An emulator would return a generic string for the *getDeviceId* call, and if the file ***qemu_trace*** exists then it means the environment is a QEMU based emulator. These two red pills are only 2 of several thousand different ones that can be used by malware developers, which illustrates the need for a solution to combat this problem.

From a high-level perspective there are several ways to detect emulators. From a networking point of view, most emulators use the network interface ***eth0*** while most real devices either use ***wlan0*** or ***rmnet***. This can easily be checked with the API call *getTetherableIfaces*. From an audio point of view, we can look at the GPIO/I2C buses. These exist in most android phones but are not emulated in the emulators. One can also request the baseband of the radio of the device, which in an emulator simply returns "unknown". These are only some high-level examples. For more information regarding this subject, we recommend the study conducted by Jiming, et al., 2014 [30].

In regard to the malware samples we chose we discovered that several of these detection heuristics were present. In this section we will cover some of the detection heuristics that the analysis from Joe Sandbox found. Starting with Malware 1, our Cerberus sample. According to the report, more than 60 different heuristics were found. The majority of these were querying the build information of the device using obfuscated code. Joe Sandbox was able to deobfuscate the underlying code. One example of a line of code that attempts to find out the manufacturer of the device is the obfuscated code

```
com.open.stove.a.a
```

which when deobfuscated translates to

```
android.os.Build.MANUFACTURER
```

. An obvious giveaway is the "a.a" at the end, which is a common result of code obfuscation using ProGuard which Android mentions in their own documentation [31]. We cannot ascertain what methods the malware authors used for obfuscating their code, but one possibility could be that they used obfuscation services from the underground community in the dark web. A study from 2020 found that obfuscation-as-a-service in the dark web is an online service aimed at automating the process of code obfuscation for large-scale attackers like for instance Android malware developers [32].

Apart from attempting to find out the manufacturer of the device, the Cerberus malware is also interested in finding out about model, brand, and board of the device. It queries the Android API call

```
android.os.Build.BOARD
```

which will return the name of the underlying board, which can for instance be "goldfish", according to Google [33]. Goldfish is the virtual hardware platform that most QEMU-based emulators are based on. This malware sample also queries the network operator name with the obfuscated API call

```
com.open.stove.qvzmjbmjbmvjwbe.chjwqskfx
```

which translates to the Android API call

```
android.telephony.TelephonyManager.
getNetworkOperatorName
```

. For an emulator this is likely going to return a generic name like “android”. Another example is that the malware sample also queries the phone number of the device using the Android API call

```
android.telephony.TelephonyManager.
getLine1Number
```

, which for an emulator will return a generic phone number that for many emulators is hardcoded to start with “1555521” and end with 4 digits based on the port number the emulator application has been assigned.

Moving on to our second malware sample, CovidLock. For this sample we did not find any analysis from Joe Sandbox. After much investigation we could not find any other public dynamic analysis of this sample. For the third sample, the Flubot sample, it used most of the same detection heuristics found in Cerberus. One difference is that the Flubot sample queries the Android API call

```
android.os.Debug.isDebuggerConnected
```

. This is not exactly an anti-emulation heuristic but is used to determine if a debugger is attached, which is common among analysis tools.

VII. Result

This section gives an overview over the result of the project.

The static analysis was realizable with the help of available tools. The Apktool was used to get to the AndroidManifest.xml. To get an insight into the permission requests of all of our sample malware the permissions can be bluntly searched for by search string. The dex2jar tool was then used for apk-to-jar-file-conversion and the jd-gui for searchings in code snippets, which allows a systematic search for known or suspected features and functions of the malware. During this process we also learned that the strings.xml can also contain traces of malware.

On the other hand conducting the dynamic analysis was more challenging. The open source sandbox Android Malware Sandbox was difficult to set up. Due to the lack of maintenance the sandbox could not provide useful outputs of our samples as in the repository mentioned samples, that were tested by us before. Therefore, our finding in this process was that the Android Malware Sandbox is not taking modern malware as our samples into consideration, because they contain sophisticated anti emulation techniques, and the sandbox was not able to bypass them. Then the accessible dynamic based malware analysis tool Joe Sandbox was used for executing the dynamic analysis, which could provide us with detailed and useful analysis result. However, it was not possible to gain a better understanding in the dynamic analysis through it since Joe Sandbox is a closed-source service.

During the search for understanding the malware’s behavior we came to another finding that there are over 10,000 emulation or virtualization detection heuristics, which allows the malicious app to know in which kind of environment it is currently executed.

VIII. Conclusion

The significant increase in the number of smartphones in recent years, especially with an Android operating system, and the associated increase in malware attacks on smartphones pose a serious threat. For this reason, it is essential to reliably detect malicious apps in order to limit their spread and prevent them from causing damage. Therefore, we have decided to focus on the analysis of Android malware samples in our project as part of the IMT4114 Introduction to Digital Forensics course. The project includes besides an initial literature search both a static and dynamic analysis of three recent Android malware samples.

During the static analysis, we were able to identify several indications for the infection of the app for each of the selected apps. On the one hand, by evaluating the required permissions and, on the other hand, by reverse engineering the source code. But in the subsequent dynamic analysis, we were unsuccessful in confirming our findings from the static analysis. All of our selected malware samples use anti emulation techniques to prevent them from running in virtual environments such as a sandbox. We noticed during the project that our current knowledge in this area was not sufficient to modify the sandbox used to bypass the built-in anti emulation techniques. In summary, however, it can be said that we were able to deepen and expand our knowledge from the lecture, especially in the area of mobile forensics, through this very practical project.

References

[1] J. Mohamad Arif, M. F. Ab Razak, S. R. Tuan Mat, S. Awang, N. S. N. Ismail, and A. Firdaus, "Android mobile malware detection using fuzzy ahp," *Journal of Information Security and Applications*, vol. 61, p. 102929, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214212621001484

[2] statista.com, "Mobile operating systems' market share worldwide from january 2012 to june 2021." [Online]. Available: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

[3] onestore.nokia.com, "Threat intelligence report 2020." [Online]. Available: https://onestore.nokia.com/asset/210088

[4] M. Alazab, M. Alazab, A. Shalaginov, A. Mesleh, and A. Awajan, "Intelligent mobile malware detection using permission requests and api calls," *Future Generation Computer Systems*, vol. 107, pp. 509–521, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X19321223

[5] M.-Y. Su, K.-T. Fung, Y.-H. Huang, M.-Z. Kang, and Y.-H. Chung, "Detection of android malware: Combined with static analysis and dynamic analysis," in *2016 International Conference on High Performance Computing Simulation (HPCS)*, 2016, pp. 1013–1018.

[6] S. N. Yus Kamalrul Bin Mohamed Yunus, "Review of hybrid analysis technique for malware detection," 2020. [Online]. Available: https://www.researchgate.net/publication/342050193_Review_of_Hybrid_Analysis_Tech

[7] F. FKIE, "malpedia - android related malware." [Online]. Available: https://malpedia.caad.fkie.fraunhofer.de/library?search=android

[8] abuse.ch, "Malware sample exchange." [Online]. Available: https://bazaar.abuse.ch/

[9] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 468–471. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508

[10] [Online]. Available: https://www.virustotal.com/

[11] [Online]. Available: https://malshare.com/

[12] Lumos, "Cerberus analysis - android banking trojan," Jun 2021. [Online]. Available: https://nur.pub/cerberus-analysis

[13] T. Fabric, "Cerberus - a new banking trojan from the underworld - threatfabric," 8 2019. [Online]. Available: https://www.threatfabric.com/blogs/cerberus-a-new-banking-trojan-from-the-underworld.html

[14] DomainTools, "Covidlock update: Deeper analysis of coronavirus android ransomware," 3 2020. [Online]. Available: https://www.domaintools.com/resources/blog/covidlock-update-coronavirus-ransomware

[15] A. Buescher and G. S, "How flubot targets android phone users and their money," 5 2021. [Online]. Available: https://www.nortonlifelock.com/blogs/norton-labs/flubot-targets-android-phone-users

[16] Cybleinc, "Flubot variant masquerading as the default android voicemail app," 9 2021. [Online]. Available: https://blog.cyble.com/2021/09/09/flubot-variant-masquerading-as-the-default-android-voicemail-app/

[17] "Apktool - a tool for reverse engineering android apk files." [Online]. Available: https://ibotpeaches.github.io/Apktool/

[18] "Java decompiler - yet another fast java decompiler." [Online]. Available: http://java-decompiler.github.io/

[19] "Cerberus analysis by virustotal." [Online]. Available: https://www.virustotal.com/gui/file/c5888a824cba71e4c6ebca35aad174120f8b681e48af8

[20] "Android.infostealer.cebruser." [Online]. Available: https://beta.pithus.org/report/c5888a824cba71e4c6ebca35aad174120f8b681e48af867dfa MpqH-ghXV234d7auxKDhW-bBN28cQSZeSMYcT$_c$

[21] "Covidlock analysis by virustotal." [Online]. Available: https://www.virustotal.com/gui/file/c844992d3f4eecb5369533ff96d7de6a05b19fe5f5809ceb15

[22] Google, "String resources: Android developers." [Online]. Available: https://developer.android.com/guide/topics/resources/string-resource#java

[23] Areizen, "Android-malware-sandbox: Android malware sandbox." [Online]. Available: https://github.com/Areizen/Android-Malware-Sandbox

[24] G. LLC, "Android studio." [Online]. Available: https://developer.android.com/studio

[25] O. A. V. Ravnås, "Frida - dynamic instrumentation toolkit." [Online]. Available: https://frida.re

[26] "Sandbox analysis of the cerberus sample." [Online]. Available: https://www.joesandbox.com/analysis/519295/0/html
[27] "Deep malware analysis," 11 2021. [Online]. Available: https://www.joesecurity.org/joe-sandbox-technology
[28] S. Neuner, V. van der Veen, M. Lindorfer, M. Huber, G. Merzdovnik, M. Mulazzani, and E. Weippl, "Enter sandbox: Android sandbox comparison," 2014.
[29] A. Bacci, A. Bartoli, F. Martinelli, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Impact of code obfuscation on android malware detection based on static and dynamic analysis." in *ICISSP*, 2018, pp. 379–385.
[30] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu, "Morpheus: Automatically generating heuristics to detect android emulators," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 216–225. [Online]. Available: https://doi.org/10.1145/2664243.2664250
[31] "Shrink, obfuscate, and optimize your app : Android developers." [Online]. Available: https://developer.android.com/studio/build/shrink-codeobfuscate
[32] V. Sembera, M. Paquet-Clouston, S. Garcia, and M. J. Erquiaga, "Uncovering automatic obfuscation-as-a-service for malicious android applications," Oct 2021. [Online]. Available: https://vblocalhost.com/uploads/VB2021-Sembera-etal-updated.pdf
[33] "Build : Android developers." [Online]. Available: https://developer.android.com/reference/android/os/BuildBOARD
[34] F. H. da Costa, I. Medeiros, T. Menezes, J. V. da Silva, I. L. da Silva, R. Bonifácio, K. Narasimhan, and M. Ribeiro, "Exploring the use of static and dynamic analysis to improve the performance of the mining sandbox approach for android malware identification," 2021.
[35] K. Jamrozik, P. von Styp-Rekowsky, and A. Zeller, "Mining sandboxes," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 37–48. [Online]. Available: https://doi.org/10.1145/2884781.2884782