The application of reverse engineering tools and analysis of common anti-analysis techniques

Maximilian Vorbrodt
Linköping university
Linköping, Sweden
maxvo113@student.liu.se

Felicia Posluk
Linköping university
Linköping, Sweden
felpo253@student.liu.se

***Abstract*—The purpose of this study was to investigate anti-analysis techniques used in Android applications and how reverse engineering may be used to combat these. The ultimate goal of this was to determine common anti-analysis techniques, tools that can circumvent them, as well as determining user friendliness of the tools. A mixed methodology approach was used, combining a literature study with practical experiments where OWASP crackme applications were solved to get a hands-on experience of how anti-analysis techniques are actually applied in Android applications. Contributions of this paper included what are the most common anti-analysis techniques and what makes them popular, tools that can be used to solve common anti-analysis techniques as well as an analysis of how easy they are to use and apply.**

I. Introduction

Malicious programs are everywhere and may infringe on user's privacy, generate financial losses, restrict access to resources and much more. To hinder malware from executing, malware analysis is used to provide defensive and protective procedures, it is also used for the development of other counter measurements of malware.

To combat malware analysis, attackers use anti-analysis techniques in order to prevent or increase the work it takes to detect, reverse engineer and analyze their malware. There exists a lot of different anti-analysis techniques, but the main idea of all of them is to make the program harder to read or make it appear as something it is not, so it will pass the necessary malware analysis checks and reach the end user.

To understand how these anti-analysis techniques are used and how to counter them, it is important to have a practical understanding of how they work and what tools can be used to reverse engineer applications containing malicious programs.

This is why the OWASP (The Open Web Application Security Project) has released a number of crackmes for the Android operating system, which are basically hacking challenges to help developers and programmers gain a better understanding of how anti-analysis techniques may be used and countered. These crackmes will be the focus of this paper.

Furthermore, several tools will be tested on the crackmes such as decompiler, debugger, static and dynamic tools for analyzing. This will create a solid foundation of what are effective ways different tools can contribute to hinder anti-analysis techniques.

A. Questions

In this paper the following questions are will be addressed:

- What are the most common anti-analysis techniques used today and why are they used?
- What tools are commonly used to circumvent anti-analysis techniques?
- What reverse engineering tools are considered user friendly?

B. Methodology

To answer the set questions that this paper aims to investigate, a mixed approach was used. The mixed approach consisted of reviewing peer reviewed papers and other trusted sources, as well as a more hands-on approach, where we solved numerous OWASP crackmes to gain a better a better understanding of how anti-analysis techniques are utilized and how common reverse engineering tools works.

II. Background

This section provides a wider understanding of Android, OWASP foundation, reverse engineering tools and anti-analysis techniques.

A. Android Architecture

Android is a mobile operating system with an open-source platform, based on Linux and developed by Google. The architecture of the system is composed of different layers, where each layer defines specific services and interfaces. [25]

As the operating system is based on Linux, the lowest level of the system is a variation of Linux Kernel, which deals with memory, processes, power management, allows key security features and more. On top of the Linux Kernel is the Hardware Abstraction Layer (HAL), which provides an interface between the higher-level Java API framework and the hardware capabilities. Above HAL we got Android Runtime (ART). Each application runs in its own process and has an instance of ART by itself. ART includes for example just-in-time (JIT) and Ahead-of-time (AOT) compilation. Above HAL we also got native C/C++ libraries, which are required by ART, HAL and other components/services. The higher-level Java API framework includes the building blocks needed to create Android apps. At the highest level we got system apps, which includes apps for users and key capabilities which developers can access from within the apps themselves. [4]

B. Application Signing

Android requires that all applications are digitally signed with a certificate before they can be installed on a device or updated. [15]

This is the first step in creating a sandbox for each and every application in the Android system. A signed application certificate helps determine what application is associated with what user ID, where different applications will run with different user IDs. This ensures that the application cannot access other applications, except through well-defined IPC (inter-process communication). [14]

C. Android Manifest

Every Android app has a AndroidManifest.xml file, which contains code in binary XML format and is located in the root directory of the app's Android Package Kit (APK) file. The manifest file contains information about the app's components, general app metadata, request permissions and more. [25]

In this case request permissions means both the permissions the app needs to access protected apps or sensitive parts of the system as well as any permissions other apps must have if they want to access data from the app itself. Each permission is identified by a unique label. [19]

D. OWASP crackmes

The OWASP nonprofit foundation is an abbreviation for Open Web Application Security Project. Several open-source software projects are developed by OWASP. These projects are used as educational tools and techniques to secure the web and mobile applications.[26]

One of these projects is OWASP crackmes, a mobile testing guide developed to inform and educate about mobile application security. The project contains different levels of reverse engineering challenges. Developers are encouraged to solve these challenges with different tools and techniques in order to create a deeper understanding of different utilities that are accessible when reverse engineering malicious applications. [5]

E. UnCrackable App for Android Level 1

The objective of the first level in the OWASP crackmes project is to find a secret string that is hidden within the application. To complete the level the string should be extracted by the developer. [5]

F. UnCrackable App for Android Level 2

The objective of the second level in the OWASP crackmes project is to find a secret string that is hidden within the application. However, there is an added difficulty when compared to the first level. In this challenge there are added traces of native code. The level is completed when the string has been successfully extracted.[5]

G. Analysis techniques

Malware behave differently depending on their programmed objective. Therefore, many tools and techniques have been developed to identify their malicious nature. Consequently, revers engineering is a necessary foundation for malware analysis. Furthermore, the process of selecting analysis tools with relevant functionalities is additionally important.

Knowledge about tools with static and dynamic analysis properties, are one of the necessary requirements for successful malware analysis. The main difference between the properties is that static analysis examines the malicious sample without executing the code, whereas dynamic analysis executes the malicious code in an controlled environment. Thorough descriptions about their individual advantages and disadvantages will be stated below. [1]

1) Static analysis: Static analysis is beneficial for determining the signature of binary files. This refers to identifying the origin of the malicious sample. The origin often leads to insight about the behaviour of the malicious code, which is the objective of malware analysis. Static analysis can be a straight forward process of acquiring information as no further analysis of the malware is necessary. However, this renders the analysis technique ineffective against sophisticated and advanced attacks. [1]

2) Dynamic analysis: Dynamic analysis techniques analyze the behaviour of the malicious code in a safe environment in comparison with the signature based analysis mentioned earlier. This involves a more thorough analysis where behaviours of the malware are executed in the sandbox environment. The analysis system must be a closed and isolated for the technique to work. [1]

Compared to static analysis there is a single tool used in the technique and that is the sandbox environment. There are different commercial tools that could be used, however, they all have the same basic function of creating a safe place to execute the malware. [1]

H. Anti-analysis techniques

There are plenty of anti-analysis techniques. Below are some of the most common, which this paper focused on identifying when conducting the study.

1) Obfuscation: Obfuscation is the process of writing or modifying the code and data in a way which makes it more difficult to comprehend, and is fundamental part of most software protection schemes. An important key concept when talking about obfuscation is that it is not something that can be turned off and on, but instead its an integral part of the program or system it is involved in.[24]

When analyzing an obfuscated app, a common approach is to first decompile the bytecode (if possible), disassemble included libraries and try to dissect and understand them. During this process there are some common factors that should be kept in mind, such as: meaningful identifiers (method names, class names, variable names) may have been discarded and string resources might have been encrypted. Also, code may be concealed through a combination of encryption and

packing (compressing), that is the code of the original file is unreadable until it has been unpacked and decrypted. [24]

2) Root Detection: Root detection in the context of anti-analysis techniques, is the process of making it more difficult to run an application on a rooted device. Thus, it hinders some of the techniques and tools used for reverse engineering, making it harder to analyze the code. On it's own however, root detection is not very useful, but when combined with multiple other root checks located at different places in the application it may increase the effectiveness of the technique. [24]

The following criteria can be used to identify applications containing root detection:

- root detection mechanisms operate on multiple API layers (e.g. Java APIs, Assembler calls, native library functions)
- Detection methods are implemented throughout the app at different locations and not grouped together at one spot
- Mechanisms are somewhat unique and not simply copy pasted from the internet).

Root detection may also be implemented through the use of libraries, RootBeer is an example of such a library. [24]

3) Anti-Debugging Detection: Debugging is a tool used to analyze application behaviour during runtime. This process allows the code to be step through, stopped and inspected and a lot more. The behaviour of how the application modifies the memory and how the variable states are altered can also be inspected with a debugging tool. [24]

Anti-debuggers are preventive or reactive tools used to hinder the processes of debugging. A preventative anti-debugger would identify the a debugging process is being initialized while the reactive part would shut down the application as a way to hinder the debugging process. There are many different ways that the anti-debugger can prevent or react, another example would be to trigger hidden behaviours to mislead the debugging. [24]

The process of debugging on Android application occurs on two different layers, on the Java API framework but also on the Native layer containing C and C++ libraries. A sophisticated anti-debugger would therefore operate on both levels. [24]

There is no general way of bypassing an anti-debugger and is highly dependable on which defences have been implemented by the anti-debugger. There are however, three common approaches that can be applied during these circumstances. [24]

- The first approach is to patch the anti-debugging functionality with the help of NOP instructions. This is a machine control instruction that stands for "No Operation". The instruction insures that the application does nothing during execution.
- The second approach consists of using dynamic tools that can hook onto API's on the Java and native layers. This technique will alter the values of functions used to detect the debugging process. Allowing the debugging to commence.
- The third approach is to alter the Android environment. This is possible because Android is an open environment.

4) Reverse Engineering Tools Detection: Reverse engineering tools detection is a method used to detect a reverse engineering "attack". The application identifies the presence of a reverse engineering application. There are different identifiers, one is that reverse engineering applications usually run on root level. Another identifier is that the tool forces the application into debugging mode. When the reverse engineering attack is detected, the application may respond in different ways to hinder the process, for example the application may terminate itself. [24]

One method that identifies applications using this anti-analysis technique, is to look at what reverse engineering tools an application uses. Additionally, the formation of packages, files, processes, and other modifiable data. [24]

I. Tools for reverse engineering

These were the tools used in the study for dynamic and static analysis of the applications.

1) Decompiler: APK Decompilers are used to extract native data and code components from Android applications, enabling decoding of the applications code. It does this by taking in a class file as input and then produces the source code as output, where the decompilation is the exact reverse process of compilation. [20]

There are plenty of decompilers which are used for the purpose of reverse engineering Android applications. According to [6], some of the more well-known are:

- JADX [27]
- Bytecode Viewer [21]
- Apktool [10]

2) Debugger: Debuggers are commonly used to test and locate bugs, errors or inconsistencies in programs. Android Studio for example, comes with a built-in easy to use debugger tool which allows developers to set breakpoints in the code, examine variables, evaluate expressions at runtime and more. [13]

3) Disassemblers: Disassemblers are a static analysis tool used when overviewing a malware's binary code. Disassembles convert the machine code into assembly language, it can therefore be used to analyze native-code components of applications. If the malware is primitive, then this tool can be an efficient alternative for reverse engineering. [16]

III. Experiments

This section will focus on the process of solving crackme level 1 and level 2. The levels objective are described i section II-E and II-F. Level 1 will be solved through two different methods and level 2 will be solved with one method.

A. Choice of tools

The OWASP foundations guides to, how to solve the different levels, was a starting point when it came to deciding what tools we should work with. We had no experience within the field of reverse engineering, therefore, it was difficult to know what tools were widely known and used within the field. We also wanted to use tools of both a static and dynamic nature.

There were 8 guides in the OWASP git repository [5] for solving level 1 crackme and 3 of them used Frida as a dynamic tool. Therefore, we chose the tool Frida as it seemed to have guides to support us during the reverse engineering process. In addition, the guides presented all the other tools they used during the process which introduced us to the tools adb, Ghidra, Apktook and jdb. The tool JADX was suggested by our supervisor and Android Studio was a familiar program from an earlier course about Android applications. Our motivation of choosing our tools have therefore been familiarity, accessibility to information, guidance and recommendations.

B. Solving UnCrackable App for Android Level 1

The goal of the crackme level 1 is to enter a secret string into the application provided by crackme. When the application has been installed on an Android emulator and is first started the user is greeted with a screen displaying a text saying "Root Detected!", see figure 1. If the user presses "OK" the application will close, and it is not possible to press outside of the popup to close the it.

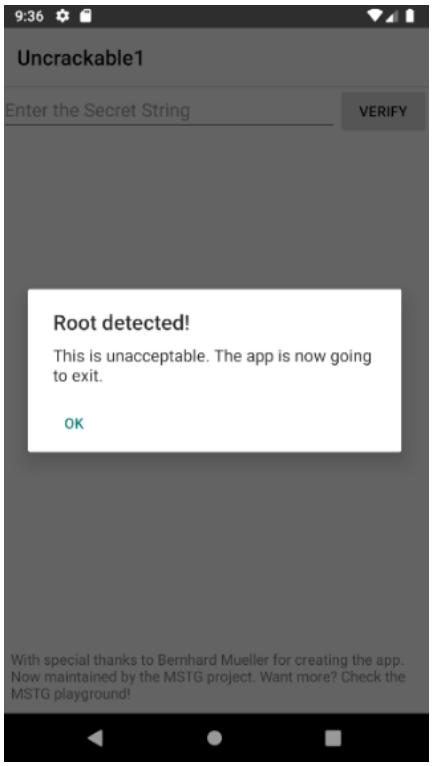

Fig. 1. UnCrackable App for Android Level 1 starting screen

There are several ways that one can continue from this to solve the application. In this experiment two approaches were used, both utilizing dynamic analysis to solve the crackme.

1) Solving crackme level 1 using the java debugger: To solve the UnCrackable App for Android Level 1 the following tools were used: adb, Apktool, JADX, jdb (the Java debugger) and the Android emulator installed through Android Studio.

The basic idea to solve it is to first make the application debuggable, analyze the code, then use a debugger to find a way around the alert(s) and finally get to the point where the user input is compared to the secret string, so it can be extracted.

Apktool was used to first make the application debuggable, through adding *android:debuggable="true"* to the Manifest file of the application.

After the application was made debuggable, it was not signed and had to be signed before it could be installed on the emulator. To sign the application the user must use their keystore, or create a new one, and then sign the application with the use of it.

Once the application was installed and started, the user was greeted with a starting screen similar to the first one, but now with a message displaying the text "App is debuggable!". However, like before it is not possible to close the popup without closing the application as well.

The next step was to start looking at the source code. Here JADX gui was used to look at the source code. Through looking at the code it was possible to determine the spot, where the application uses a variable to check if the alert-popup can be cancelled by pressing outside of it. Through using jdb, we were able to set a breakpoint at this point and change the value of the variable to true during run-time, which allowed us to press outside of the alert-popups to cancel them.

The source code became legible with the help of the decompiler JADX. However, as a part of the obfuscation process, the names of variables and functions were difficult to interpert.

The last step left was then to figure out what the secret string was. To do this, the source code was investigated again where it could be determined where the user string and secret string are compared. Everything left at this point was to set an appropriate breakpoint using jdb and enter a dummy text to trigger the comparison event between the strings. Through investigating the *locals* variables in the debugger during the string comparison it could be determined that the secret string was "I want to believe", which then could be entered to trigger the success popup event.

2) Solving crackme level 1 using Frida: The following tools were used for obtaining the solution: Frida, Android studio, adb and JADX.

There are multiple ways of bypassing the root and debuggable detection, because there are several functions that could be hooked. Through the process of hooking, Frida is able to add additional functionality to the application and override processes in the source code. [17] The chosen solution was to hook the java.lang.System class and change the implementation of a function called *exit()*. [11] By hooking to the *exit()* function the application would print out a string to the console, instead of exiting the application. Frida hooked on to the *exit()* function and overwrote the function by running script code. Script code is self-created Java code, in this case it was a print function. When the script had been loaded instead of the original code the application became accessible. The search bar accepted different inputs, however, to find the correct one the original code needed to be inspected again.

There were two functions within the applications java code that were of interest. The function *verify()* and the function *a()*. *verify()* checked if the input was correct and this was achieved with the help of function *a()*. During further analysis it was shown that the function *a()* contains a hard coded encrypted string that was used to verify the correct input. Thereafter, the script was further refined to contain decrypting functionality.[11] With the new script implementation the decrypted string would be printed out on the console when the verify button submitted an input. Frida was used once again to hook onto the applications functionality. The incorrect string *test* was used as an input and the console returned the string *I want to belive*. The decrypted string was entered into the application and the level 1 crackme was thus solved.

C. UnCrackable App for Android Level 2

The following tools were used for solving crackme level 2: Frida, JADX, Android Studio, adb and Ghidra. Ghidra was the new addition to the previous used tools and had the purpose of analysing the native code.

The Java code of the application was obtained with the decompiler JADX. The application consisted of the same debugging and root detection code as level 1. Therefore, the same script as mentioned in the solution for level 1 was used to bypass the first challenge. [11] Thereafter, Ghidra was used to gain access and read the native code. The native code in a file called MainActivity_init consisted of a if-satement that returned true if the variable is a string of length 23. This piece of code seemed promising, therefore, Frida was used to hook on to it. The Script used during the hook would try to send in an incorrect string that had a length of 23 characters. When the application compared the incorrect string with the correct one, the script will print out the correct string into the Frida console. [8] The incorrect string used was "I want your secret asap" and the printed string was "Thanks for all the fish". The secret string had been found and level 2 was completed.

IV. Result

The following section provides information about results related to the research questions.

A. Anti-analysis techniques found in OWASP crackmes

As mentioned earlier we chose to focus mainly on four anti-analysis techniques. During the experiments of solving the crackmes level 1 and 2, several of these anti-analysis techniques were discovered.

When the application is opened there is an alert *a* that activates the pop-up "Root detected!", which is the anti-analysis technique called root detection. This anti-analysis technique hinders the use of reverse engineering tools.

After having made the application debuggable an alert is displayed showing the text "App is debuggable", which means the anti-analysis technique called anti-debugging detection is utilized in the application to prevent tampering and debugging.

Finally, when looking at the Java source code with a tool like JADX for example, we saw that the files and functions had nonsensical names like *a*, *b* and *c*, which is a type of obfuscation meant to make it harder for a developer to interpret what is going on.

However, one thing we did not find was reverse engineering tools detection. We used several tools for reverse engineering, for example Frida, but non of the applications reacted to the use of the tools in any way.

B. Popularity of anti-analysis techniques and their effectiveness

According to [2] Android the primary anti-analysis techniques can be divided into three categories: trivial APK techniques, code obfuscation and preventive techniques.

The survey by [2] shows that the yearly numbers of publications related to the keywords "Android + Obfuscation" is steadily increasing from close to zero year 2011 and around 50 year 2020. It is also mentioned by [2] that code obfuscations is considered an effective and popular technique to make reverse engineering applications harder.

However, it should also be noted that the overall publications related the keywords "Android + Security" has increase from about 75 year 2011 to 500 year 2020 implying that the overall interest in the field is increasing. [2]

Trivial APK techniques are for example manifest file modification, repackaging or anything that require significantly less technical skills and which do not require code-level modifications but more simple operations. Previous studies have shown that over 86% of 1200 malware families used repacking to introduce malicious code into applications. [2]

Due to Trivial APK techniques incurring low overhead and being less complex have made them popular amongst developers and malware authors, however machine learning approaches are likely on the rise and will yield these kinds of techniques less successful as they are not as complex to solve as for example obfuscation. [2]

Preventive techniques prevents the application from executing in a test environment, making it difficult to analyze the application. Some examples are: root detection, anti-debugging, anti-emulator. These kind of techniques may make it more difficult to for example solve obfuscation through dynamic analysis of the application and therefore they certainly have a place amongst anti-analysis techniques. [2]

In [2] it is shown that preventative techniques are moderately touched upon in a comparative analysis of modern Android malware anti-analysis approaches, but it is noteworthy to mention that obfuscation seems to be much more of a hot topic within the Android security literature.

The exact effectiveness of these anti-analysis techniques is still uncertain as there is a lack of standard methodology and benchmark to evaluate their effectiveness and efficiency. [2]

OWASP crackmes aims to highlight and educate developers on important aspects of the anti-analysis techniques. In their level 1 and level 2 crackme they partly made use of root-detection, anti-debugging and obfuscation which may be an indication of what OWASP considers to be the most important for the community to know about currently.

C. What tools are commonly used to circumvent anti-analysis techniques?

The OWASP foundation has a page dedicated to mobile security techniques and tools. There are a total of 36 different testing tools listed on the page that are used either for Android or all platforms. Within these 36 alternative, the tools Frida, Ghidra, Adb, Android Studio, Apktool and JADX were mentioned. However, none of these tools were described as common or popular. Nevertheless, the tool radare2 is described by OWASP as being popular. [23]

In addition, OWASP provided solutions to the crackme levels and Frida was one of the tools most commonly used in level 1 and level 2. Other tools that were often metioned in the solutions were Adb, Apktool and Ghidra. JADX and Android Studio were only mentioned twice out of the 14 provided solutions. Furthermore, certain solutions mentioned that the user could use any emulator or decompiler of choice. [5]

In a survey made on malware detection and analysis tools, a selection of tools were presented as helpful for researches and security engineers. Apktool was one out of the four tools mentioned for mobile malware analysis. However, nothing about the tools popularity was mentioned. The survey mentions that the tools listed are a guide that can assist researches in choosing appropriate tools for their malware analysis. [3]

Lastly, Apktool is referenced as the most prominent tool used for unpacking and decoding apk files in an article that overviews Android obfuscation techniques and relevant tools and techniques. [22]

D. What reverse engineering tools are considered user friendly?

The tool JADX is provided with installation and usage documentation as well as answers to typical questions that the user may have. The information can be found in the git repository of the tool. [28] The tool can be used in the computer terminal or through a graphical user interface. The OWASP foundation has JADX listed in their tool list with a link to the installation guide on the git repository. [23] Lastly, JADX is mentioned as an application that is easy to install according to an article by Albakri et al. [7]

Apktool has documentation for both installation and usage on its homepage. [10] According to the earlier mentioned article, Apktool is considered user-friendly because of its ability rebuild code into readable Java code. [7] The OWASP foundation has provided documentation of the different files that are provided after the installation process and how to manage them. However, the installation process is not provided or referenced to. [23]

The homepage for Frida contains an installation and a usage guide on its homepage. Frida is solely used inside the command prompt and requires the user to be familiar with JavaScript. The tool injects user written JavaScript code into the application during debugging. [17] The OWASP foundation has provided documentation and a separate installation guide on their homepage. [23]

Ghidra's homepage provides an installation and usage guide in both written and video format. Ghidra is utilized solely through the graphical user interface. [18] The OWASP foundation has provided general documentation, a cheat sheet and a separate installation guide on their homepage.[23]

The installation guide, general documentation and usage guide for Android Studio is accessible through their popular webpage for Android developers. [12] Android Studio's graphical user interface is easy to interact with, understand and it also has a fairly large community online, which makes it easier to solve errors and problems. [9] The OWASP foundation gives no additional information about Android Studio besides a hyperlink that redirects the user to the official website. [23] The tool adb, in addition to others, are installed during the installation of Android Studio SDK. [9]

E. Limitations

The topic of Android security is enormous, as well are the number of possible anti-analysis techniques. Therefore, when solving the OWASP crackmes the scope of this paper was to mainly focus on four commonly used anti-analysis techniques and see if they could be identified. The outcome of what was learnt by solving the crackmes was partly limited by the number of applications that were tested, had more applications been tested it might have changed the final outcome.

Another limitation surrounding determining tools for reverse engineering Android applications and their user friendliness is that only a few tools were tested. To gain a better understanding of how user friendly different reverse engineering tools are it is recommended that more tools are tested and compared in the future.

There are little to no benchmarks, standard methodology and literature on how to determine what are the most effective and/or popular anti-analysis techniques, as well as Android reverse engineering tools, which means it is hard to make an exact prediction. Therefore, a qualified guess based on our own experiences combined with the little literature that exists on the topic, results in the best possible prediction that can be made at this point.

V. Discussion

In this section the connections between the practical experiment and trusted sources, will be made to answer the questions that this paper aims to investigate.

A. Common anti-analysis techniques and why are they used?

The most common anti-analysis techniques can roughly be divided into three categories: trivial APK techniques, code obfuscation and preventive techniques.

Trivial APK techniques, such as Mainfest file modification and repackaging, are very common as anti-analysis techniques, as they are easy to apply while still adding some complexity when it comes to reverse engineering. Trivial APK techniques requires significantly less complexity then other kinds of anti-analysis techniques making them easier to crack. This could potentially affect their effectiveness negatively. As machine

learning approaches are likely to increase, it could potentially lessen the popularity of trivial APK techniques in the future, as they tend to require less complexity to solve.

Code obfuscation is also very popular amongst applications containing malware, to make reverse engineering more difficult. Based on the result from of the survey conducted by [2] the yearly publications related to obfuscation of Android applications have increase from close to none 2011 to around 50 year 2022, meaning the topic of obfuscation has become significantly more popular. This implies that utilizing obfuscation as an anti-analysis technique in Android applications may also be on the rise. As stated in [2] code obfuscation techniques are already considered to be very effective and popular, and as the complexity can be made high it may result in that they will continue to be widely adopted in the feature.

Preventive techniques, such as Anti-debugging and Anti-rooting, have their place amongst anti-analysis techniques as they can help prevent cracking of other anti-analysis techniques through dynamic analysis for example. According to [2] preventive techniques are popular, but it doesn't seem to be as "trendy" as obfuscation. It is unknown if this is because they are already so widely adopted and effective, or because there simply is not much left to develop further within the field of preventive techniques. However, as preventive techniques have unique functionalities in hindering reverse engineering it is quite likely that they will continue to be applied in the future as well.

The fact that all these three categories are exemplified in the OWASP crackmes Android applications level 1 and 2, may contribute to the thesis that all of the three categories are in fact commonly used and effective as anti-analysis techniques.

B. What tools are commonly used to circumvent anti-analysis techniques?

There were no surveys or research articles that provided a straightforward or relevant answer to this question. At best the articles recommended certain tools as a starting point or the tools were mentioned as an example of how to combat certain anti-analysis techniques. In addition, there were no motivation provided on why the tools were selected to be in the articles. [7] [3] [22] In the research article [22] it is stated that Apktool is the most prominent tool for decoding and unpacking apk files. However, there are no references referring to this statement, therefore, it is unknown where the article obtained the information.

The lack of research can be an indicator for that there is more that can be done within the field of software security and anti-analysis techniques. However, the lack of research can also be an indicator that the tools used are less important and that it is the understanding of how to use the tool in the correct circumstances that is more relevant. The articles picked out certain tools as an example of what to use when combating anti-analysis techniques. This does not necessarily mean that there are no better alternatives on the market.

The OWASP foundation has a list of 36 tools with different user and installation guides. Many of these tools have an overlap in functionality and application. In addition, the list of tools is not ordered in any special way. [23] The foundation does state that radare2 is a popular tool, however, similarly to the articles, no additional information about how this conclusion was made is provided.[23] The OWASP foundation is, among other things, a source of security education for developers. [26] Therefore, the list of tools can be interpreted as a overview of possible resources for developers, especially when the tools have no particular priority. Furthermore, the understanding of anti-analysis techniques becomes more of significant than the choosing of the most conventional tool.

The OWASP foundations guides to, how to solve the different levels, was a starting point when it came to deciding what tools we should work with. We had no experience within the field of reverse engineering, therefore, it was difficult to know what tools were widely known and used within the field. We also wanted to use tools of both a static and dynamic nature. There were 8 guides in the OWASP git repository for solving level 1 crackme and 3 of them used Frida as a dynamic tool. Therefore, we chose the tool Frida as it seemed to have guides to support us during the reverse engineering process. In addition, the guides presented all the other tools they used during the process which introduced us to the tools adb, Apktool, Ghidra and jdb. The tool JADX was suggested by our supervisor and Android Studio was a familiar program from an earlier course about Android applications. Our motivation of choosing our tools have therefore been familiarity, recommendations, accessibility to information and guidance.

C. What reverse engineering tools are considered user friendly?

Considering that the previous question concluded that it is undetermined what tools are commonly used to circumvent anti-analysis techniques, the report will limit the discussion to the tools used in section III.

Similarly to the previous project question, there are no clear answers to how user friendly common reverse engineering tools are. This question is subjective and requires a surveys on large sample groups to be able to draw any useful conclusions. It was stated in [7] that JADX is easy to install and Apktool is considered user friendly, however no reference was supplied or further information about how this conclusion was made.

There are many aspects to consider when concluding if a tool is user friendly. All tools had aspects that made them more or less user friendly. The different tools had a mix of user interfaces, easy download and installation alternatives, guides and application. Additionally, the perception of what is user friendly is a personal preference. Due to the small sample size of studies within this topic, it is difficult to draw any kind of definite conclusion from only a literature study.

Nevertheless, there were different factors that shaped our opinion on the reverse engineering tools user friendly attribute. The first factor was the accessibility to different guides for installation and tool application. If there was minimal information online, the installation and application process became

significantly more difficult. The installation of Apktool was difficult as there were not many guides. In addition, many guides contained the same information about the installation and application process. This resulted in very low variety in information which was not beneficial if the guide did not work. This occurred when trying to connect with the Frida server. An additional factor was the accessibility to information about solving error occurrences. This issue prolonged the reverse engineering process significantly and could result in a demoralizing feeling. This was a more general issue with all of the applications. JADX was easy to install and through the built in graphical user interface, it was straight forward of how it could be used. Therefore, we felt that the user friendliness of JADX was high. The Java Debugger (jdb) was fairly difficult to deal with, specifically it was hard to connect to the application and set the correct breakpoints, but due to a large community and many tutorials and guides surrounding it, it was possible to solve most of the problems that we ran into. Due, to this it is our opinion that the user friendliness of jdb is somewhat low on its own, but with a large supportive community around it becomes more accessible and user friendly.

VI. Conclusion

The concluded answers obtained from section V is provided below. Each research question is answered individually.

A. What are the most common anti-analysis techniques used today and why are they used?

The three groups of anti-analysis techniques that can be considered the most common today is: Trivial APK techniques, obfuscation and preventive techniques. Trivial APK techniques, such as such as Mainfest file modification and repackaging, are commonly used due to the relatively little complexity that is necessary to implement them. However, their simplicity might become their downfall in the future, as more machine learning techniques may be implemented to identify malware according to [2].

Anti-analysis techniques implementing obfuscation are popular due to them being effective, as well as for allowing a variation of complexity. That is, they can add a lot or little complexity depending on how they are implemented.

Preventive techniques, such as anti-debugging and root-detection, have a unique places within anti-analysis techniques as they may work like a first line defense in hindering reverse engineering of the application.

B. What tools are commonly used to circumvent anti-analysis techniques?

There is no supporting research for any particular tool being more commonly used than their counterpart when used for circumventing anti-analysis techniques.

C. What reverse engineering tools are considered user friendly?

The answer is inconclusive as there is an inadequate amount of studies made on the topic.

References

[1] Michael Sikorski and Andrew Honig. *Practical Malware Analysis - The Hands-On Guide to Dissecting Malicious Software*. 2012. URL: http://dtors.net/Hacking/Practical % 20Malware % 20Analysis . pdf (visited on 04/25/2022).

[2] Pradeep Singh Vikas Sihag Manu Vardhan. *A survey of android application and malware hardening*. 2019. URL: https://www.sciencedirect.com/science/article/pii/S1574013721000058 (visited on 04/25/2022).

[3] Sajedul Talukder and Zahidur Talukder. "A Survey on Malware Detection and Analysis Tools". In: *International Journal of Network Security Its Applications* 12 (Mar. 2020). DOI: 10.5121/ijnsa.2020.12203.

[4] Google. *Platform Architecture*. 2021. URL: https : / / developer . android . com / guide / platform (visited on 03/06/2022).

[5] Bernhard Mueller. *UnCrackable Mobile Apps*. 2021. URL: https : / / github . com / OWASP / owasp - mstg / tree / master/Crackmes (visited on 03/07/2022).

[6] Abdul Raffay. *32 Best APK Decompilers*. 2021. URL: https://ssiddique.info/apk-decompilers.html (visited on 04/25/2022).

[7] Ashwag Albakri et al. "Survey on Reverse-Engineering Tools for Android Mobile Devices". In: *Mathematical Problems in Engineering* 2022 (Jan. 2022). DOI: 10 . 1155/2022/4908134.

[8] *Android Anti-Reversing Defenses*. 2022. URL: https : / / 1337 . dcodx . com / mobile - security / owasp - mstg - crackme-2-writeup-android (visited on 04/28/2022).

[9] *Android studio*. 2022. URL: https://developer.android.com/studio (visited on 04/27/2022).

[10] Apktool. *Apktool*. 2022. URL: https : / / ibotpeaches . github.io/Apktool/ (visited on 04/26/2022).

[11] Chandrapal Badshah. *Solving OWASP MSTG Android CrackMe using Frida (Level 01)*. 2022. URL: https://www.youtube.com/watch?v=OyoLM0zU1wY&t=620s (visited on 04/28/2022).

[12] Android Developers. *Android Studio*. 2022. URL: https://developer.android.com/studio (visited on 05/10/2022).

[13] Android Developers. *Debug your app*. 2022. URL: https: / / developer . android . com / studio / debug (visited on 04/25/2022).

[14] Android developers. *Application Signing*. 2022. URL: https://source.android.com/security/apksigning (visited on 05/10/2022).

[15] Android developers. *Sign you app*. 2022. URL: https: / / developer . android . com / studio / publish / app - signing (visited on 05/10/2022).

[16] *Disassembling and Decompiling*. 2022. URL: https : / / mobile - security . gitbook . io / mobile - security - testing - guide / android - testing - guide / 0x05c - reverse - engineering - and - tampering # disassembling - and - decompiling (visited on 04/28/2022).

[17] *Frida*. 2022. URL: https : / / frida . re/ (visited on 04/27/2022).
[18] *Ghidra*. 2022. URL: https://ghidra-sre.org/ (visited on 04/27/2022).
[19] Google. *App Manifest Overview*. 2022. URL: https://developer.android.com/guide/topics/manifest/manifest-intro (visited on 03/06/2022).
[20] Javatpoint. *Java Decompiler*. 2022. URL: https : / / www . javatpoint . com / java - decompiler (visited on 04/25/2022).
[21] Konloch. *Bytecode Viewer*. 2022. URL: https://github.com/Konloch/bytecode-viewer (visited on 05/10/2022).
[22] Sanjay Madan, Sanjeev Sofat, and Divya Bansal. "Tools and Techniques for Collection and Analysis of Internet-of-Things malware: A systematic state-of-art review". In: *Journal of King Saud University - Computer and Information Sciences* (2022). ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2021.12.016. URL: https://www.sciencedirect.com/science/article/pii/S1319157821003621.
[23] Bernhard Mueller. *UnCrackable Mobile Apps*. 2022. URL: https : / / mobile - security . gitbook . io / mobile - security - testing - guide / appendix / 0x08 - testing - tools (visited on 04/26/2022).
[24] OWASP. *Android Anti-Reversing Defenses*. 2022. URL: https://mobile-security.gitbook.io/mobile-security-testing-guide/android-testing-guide/0x05j-testing-resiliency-against-reverse-engineering#testing-anti-debugging-detection-mstg-resilience-2 (visited on 04/25/2022).
[25] OWASP. *Android Architecture*. 2022. URL: https : / / mobile - security . gitbook . io / mobile - security - testing - guide/android-testing-guide/0x05a-platform-overview (visited on 03/06/2022).
[26] OWASP. *Who is the OWASP Foundation?* 2022. URL: https://owasp.org/ (visited on 03/07/2022).
[27] skylot. *JADX*. 2022. URL: https://github.com/skylot/jadx (visited on 05/10/2022).
[28] skylot. *JADX*. 2022. URL: https://github.com/skylot/jadx (visited on 04/26/2022).