Practical Mobile Forensics

Otto Heino
Linköping university
Linköping, Sweden

Adam Halim
Linköping university
Linköping, Sweden

***Abstract*—With the increasing number of software applications and systems being released, the need for developers to protect their source code from others is increasing as well. The get access to the source code or the direct functionality of the source code, the application has to be reversed engineered. With an exposed source code, vulnerabilities are more likely to be detected as well as code stealing. This can be countered by using anti-reverse engineering techniques when building the application. The most common anti-reverse engineering techniques are root detection and obfuscation. In this paper, the focus has been on using reverse engineering techniques to bypass security mechanisms. This was done through several challenges called unCrackable from OWASPs Mobile Security Testing Guide. The techniques used to bypass the anti-reverse engineering were both static and dynamic analysis. For static analysis, JADX and Ghidra were used to decompile the applications for an analysis of the source code. For one part of the dynamic analysis, the decompiled programs were tampered with to remove root detection, which made it possible to debug the application and reveal further information compared to just analyzing the source code. Another dynamic analysis method was to use Frida to insert code when the program was running to extract information from functions in the program code. Out of these, Frida was to most powerful tool for analysis, but JADX and Ghidra were the simplest and most user-friendly. The anti-reverse engineering technique that was most succesful in preventing our analysis was obfuscation. Concidering unobfuscated code can already be difficult enough to understand, trying to comprehend obfuscated code can be a nightmare.**

I. Introduction

When writing software, it can sometimes be necessary to hinder others from stealing your code (or at least make it more difficult to do so). Protecting source code can be seen as a crucial objective for companies that don't want competitors to reap the benefits of the hard work of their employees and money invested. To combat this issue, techniques that make it more difficult to extract meaningful source code (i.e. reverse-engineering) have been developed.

These techniques are called anti-reverse-engineering techniques and are measures meant to make it difficult to extrapolate what the original source code could have looked like. Unfortunately, these techniques can be used by malicious entities as well, such as creators of malware. By using anti-reverse-engineering techniques when writing malware, it can be difficult to differentiate the malware from a safe application. As such, by using anti-reverse-engineering techniques, malware can bypass anti-malware detection systems and make its way onto trusted platforms, such as Google Play.

The goal of this paper is to investigate some of the common anti-reverse engineering techniques are, and which methods and tools are used to bypass them. The investigation is based on OWASP Crackmes as a basis and will be used to test and demonstrate different techniques. [1]

II. Background

A. APK

Applications for Android come packaged in an *Android package* (APK), which is a file that contains everything that is needed to install and run the application. The package file has the file extension `.apk` [2].

B. Reverse Engineering

Reverse engineering from a software perspective is the concept of accessing the high-level program code behind a functional application. When a program is compiled, the compiled file for the program is unreadable by the human eye as it is just binary bits; ones and zeros. The environment where this program is created is (often) written in a high-level program language such as Javascript, Java, Python, or C++. To get a better understanding and to analyze all aspects of a program or system, it is required to have access to the decompiled program code, meaning an interpretation of how the source code could have originally looked. When a program is decompiled, the program source code is extracted from the compiled binary file.

C. Debugging

Debugging is a technique that is used to find bugs in a program. There are many debugging tools for lots of programming languages and platforms. Android Studio, which is used to develop Android applications, comes with a debugger that enables an analysis of the program while the program is running. The code can be stepped through and it gives detailed information about the current function and variables[1].

D. Anti Reverse Engineering

Anti-reverse engineering is the art of interfering with known reverse engineering techniques. It increases the difficulty to analyze applications and systems. There are a lot of different anti-reverse engineering techniques and some of the most common will be listed below.

[1]https://developer.android.com/studio/debug/

1) Obfuscation: The goal of obfuscation is to prevent static analysis, which is done by making the program code unreadable by human eyes, as well as software developed by humans. The obfuscation process can be done in a number of different ways and the most basic obfuscation is done manually. Examples of manual obfuscation are purposefully renaming classes, variable names, and functions to make them look like they are doing something else than they are designed to do. In those cases, the program is designed in a way to makes it difficult to understand, both by humans and by other software. With this approach, it is harder to continue developing a program as regular source code should be as simple as possible to understand. A more common approach is to have an obfuscation algorithm that scrambles and "rewrites" the code before compiling. In this way, continuous development is easier and it is a more stable approach to development. [3]

2) Anti-root detection: Having root access to a device gives the user more freedom to use their device however they want. For example, one could allow apps that require root privileges to run, which is not possible without a rooted device. While this may be a valid reason to enable root access on a device, this comes with some downsides. One such downside is that it opens up for malware to be run in a privileged mode as well, which can have catastrophic results. Therefore, some applications today come shipped with root detection mechanisms, which are supposed to detect if the application has root privileges. The point of this is to combat the security risks involved with rooting a device. A common way of rooting a device is simply installing an `su` binary which any application on the device can execute to grant root privileges. [4]

A common root detection mechanism is checking if there is an `su` binary file somewhere on the device. Looking in common places, such as `/system/xbin/su` or `/system/app/Superuser.apk`, or by running the command `which su` to find where the file is, in a way of detecting if the device is rooted. Another mechanism is by checking the BUILD tag of the application. By default, Android applications downloaded from Google Play are built with a *"release-keys"* tag. If the tag *"test-keys"* is present, it means that the application is not an official build and can be indicative of malicious software. [4] Furthermore, by checking the directory permissions of certain folders, one can look if a folder that isn't supposed to be readable is readable. This is because some root installs have to make certain folders readable.

Detecting these root detection mechanisms is commonly done utilizing static analysis. If the app is heavily obfuscated, or if the root detection is done in a native library instead of in Java/Kotlin code, dynamic analysis can be employed instead.

In a study conducted by Sun, Cuadros, and Beznosov where 182 Android applications were studied for root detection and their effectiveness, they found that all root detection methods they encountered were discovered and could be evaded. [4] They also found that the most effective way of detecting if a device is rooted is by checking for directory permissions. This worked for 69% of applications tested. [4]

E. UnCrackable

The UnCrackable challenges are a set of mobile security challenges which are developed and maintained by the Open Web Application Security Project (OWASP). They have made a Mobile Security Testing Guide which the UnCrackable mobile challenges are a part. The challenges are developed specifically to further enhance the understanding of mobile security. [1]

III. Methodology

The methodology section will cover how information was gathered regarding common reverse-engineering and anti-reverse engineering techniques.

A. Jadx

The first thing we looked into was how to decompile the APK file into Java code. Looking at papers relating to Android reverse engineering and Android decompiling, several tools were mentioned as being popular for decompiling. Some of the tools mentioned are CFR[2], Jadx[3], JD Project[4], dex2jar[5], Soot[6] [2], [5]–[8]. A tool named *dare* (formerly *ded*) was also mentioned frequently, but the website for it is no longer available.

Using popular search engines with queries like "best APK decompiler" and "popular APK decompilers", we found several blogs listing Jadx as number one[7][8][9]. It was also easy to find tutorials for Jadx, and it seemed simple enough to use without much effort. It was therefore decided that Jadx would be the tool used to decompile the APK.

Jadx is a free open source program whose primary focus is to decompile Android DEX and APK files into Java source code. The code generated from the decompiler is not 100% correct so a bit of manual patching is needed. This is especially true if the code is obfuscated. [9]

B. Frida

Frida is an open-source dynamic instrumentation toolkit that is commonly used for reverse engineering. It allows injecting and hooking onto functions in a program, which allows you to inspect, or override a function when it is called. Frida scripts are written in Javascript and are supported by most major platforms and in our case, running it with Android is supported. [10] The project is very popular and has over 9000 stars on GitHub[10].

[2]https://www.benf.org/other/cfr/
[3]https://github.com/skylot/jadx
[4]https://java-decompiler.github.io/
[5]https://github.com/pxb1988/dex2jar
[6]https://github.com/soot-oss/soot
[7]https://www.slant.co/topics/3101/~best-apk-decompilation-tools
[8]https://ssiddique.info/apk-decompilers.html
[9]https://www.edopedia.com/blog/best-apk-decompilers/
[10]https://github.com/frida/frida

C. Ghidra

Ghidra is a software for reverse engineering which is developed and maintained by the National Security Agency (NSA) in the United States of America (USA). Ghidra is mainly used for static analysis of malicious code and to find possible weaknesses in networks and systems. [11]

Other popular alternatives we found for reverse engieering and decompiling are IDA Pro[11] and Radare2[12]. Since IDA Pro requires a paid license, the choice was between Radare2 and Ghidra. Both of them are quite popular on GitHub, having 16000 and 32000 stars respectively. Furthermore, searches on Google Scholar gave a similar amount of results (~ 500 vs ~ 600). It was therefore decided that Ghidra would be used, simply due to its increased popularity.

D. Android Debug Bridge

Android Debug Bridge (adb) is a tool that is used to communicate with an Android device. It allows a client to send several commands to another Android device, such as `adb push` to transfer files, and `adb shell` to get access to the device's shell. [12]

E. CrackMe challenges

In the first two levels of challenges, there is a string is hidden somewhere inside the application. To beat the challenges, one has to find this secret string and type it into an input box. If the correct string is entered, a success prompt is displayed.

1) First challenge: In the first challenge, the string is saved in the code but is encoded and encrypted, making it difficult to use the hard-coded value directly. To make out what the string is, we have to figure out how it is encoded and how to decrypt it.

2) Second challenge: The second challenge is much like the first one except that the string comparison, as well as the hidden string, are inside a native library which is imported into the application. The level of obfuscation has also drastically increased compared to the first challenge.

F. Procedure

The first challenge presented was solved by using two approaches. The first approach was to use static analysis combined with using a debugger. The second approach was using dynamic analysis with Frida. The second challenge was solved by using means of static analysis with the help of Ghidra, which was used to reverse engineer a shared library.

In the GitHub repository for the crackme challenges, several solutions are listed. Among these solutions, the tools and approaches used are also listed. Some of these tools, beyond the ones already mentioned earlier in this paper, are Xposed[13], RMS[14], QBDI[15] and SCAMarvels (Jean Grey)[16]. Xposed seems to be used in a fashion similar to how Frida works; by hooking onto functions to modify the flow of execution, and to read directly from the memory. RMS is a web interface that uses Frida in the background and is essentially a frontend for Frida. QBDI is also, similarly to Frida, a dynamic binary instrumentation tool. In the context of the crackme challenges, it is used to trace function calls which then is taken advantage of using Frida. SCAMarvels (Jean Grey) is a tool that can perform differential fault analysis attacks and is used to break encryption. In the challenges, it is used to retrieve an encryption key.

When deciding which tools to use, the popularity and perceived ease of use were of utmost importance. This led to the approaches using static analysis with Ghidra, and the dynamic approach with Frida.

G. Limitations

Due to a limited time frame, the focus of the project was on testing the most common analysis techniques. This was done by focusing on the first two crackme challenges and applying different techniques to them. The difficult technical nature of Frida and our inability to fully understand the tool, combined with the restrained time frame of the project resulted in limited results for the dynamic analysis of the second challenge. If more time or previous knowledge of the analysis method was present, the second challenge would most likely be solved with a dynamic approach as well as the finished static analysis presented above in the paper.

IV. Results

A. First challenge

The string hidden within the first application was extracted both with a combined static and dynamic approach as well as a fully dynamic approach.

1) First approach: First the APK was downloaded from the OWASP Crackmes GitHub page[17]. The APK was loaded into Android Studio to be run on an Android virtual machine. This was done in order to observe how the application behaved when run in a native environment. When the application ran, it stated that it was running as root which was prohibited, which can be seen in fig. 1. The application then shut down itself.

When it was deducted that the application forbade to be run as root, static analysis of the code began. In the bytecode of the APK, two hard-coded strings were identified, seen in fig. 2. As it was known from the challenge description that it was a string that was sought after, it was likely that these two strings were significant. The string in their raw format however did not provide any answer on their own, as they seemed to be encoded and/or encrypted.

In the next step of the analysis, the APK was loaded into the APK decompiler Jadx. Jadx provided Java source code written for the application. As it was known that the correct string had to be provided to complete the challenge, the correct string had to be hidden somewhere in the code

[11] https://www.hex-rays.com/ida-pro/
[12] https://github.com/radareorg/radare2
[13] https://github.com/rovo89/XposedBridge
[14] https://github.com/m0bilesecurity/RMS-Runtime-Mobile-Security
[15] https://github.com/QBDI/QBDI
[16] \url{https://github.com/SideChannelMarvels/JeanGrey}
[17] https://github.com/OWASP/owasp-mstg/tree/master/Crackmes

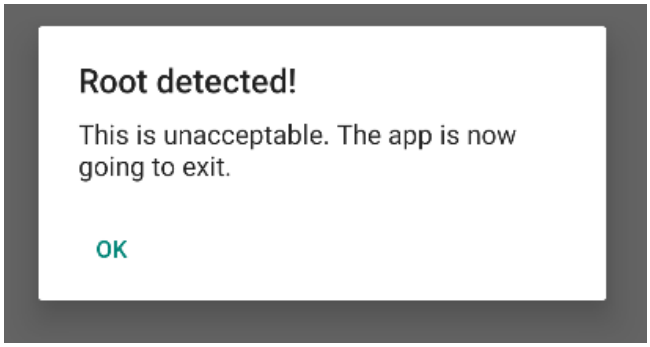

Fig. 1. Root detection mechanism

```
const-string v0, "8d127684cbc37c17616d806cf50473cc"

const-string v1, "5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2G0c="
```

Fig. 2. Encoded secret string in bytecode.

where a comparison between the user input and the string was conducted. Since the application was obfuscated, it was not intuitive what variables represented or what the goal of a specific function was. To identify where the comparison was made, we looked in the class `MainActivity` and found a method called `verify()`. It seemed like this was the method where the comparison was initiated. The method reads the input from the user, and then calls a method `a.a()` and passes the user's input. If the input is correct, the string *"This is the correct secret."* is displayed. Otherwise, *"That's not it. Try again."* is displayed. The method can be seen in fig. 3.

```
public void verify(View view) {
    String str;
    String obj = ((EditText) findViewById(R.id.edit_text)).getText().toString();
    AlertDialog create = new AlertDialog.Builder(this).create();
    if (a.a(obj)) {
        create.setTitle("Success!");
        str = "This is the correct secret.";
    } else {
        create.setTitle("Nope...");
        str = "That's not it. Try again.";
    }
```

Fig. 3. String comparison method `a.a(obj)`.

Since the actual comparison takes place in the method `a.a()`, we took a look at the aforementioned method. The contents of the method can be seen in fig. 4. It was in this method that the actual hard-coded string was present.

```
public static boolean a(String str) {
    byte[] bArr = new byte[0];
    try {
        bArr = sg.vantagepoint.a.a.a(b("8d127684cbc37c17616d806cf50473cc"), Base64.decode("5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2G0c=", 0));
    } catch (Exception e) {
        Log.d("CodeCheck", "AES error:" + e.getMessage());
    }
    return str.equals(new String(bArr));
}
```

Fig. 4. Encoded secret string inside the app.

The obfuscation made it difficult to identify what the correct string was as it was declared encoded and encrypted which made it look like random gibberish.

To analyze the string in the comparison moment, it would be beneficial to debug the application so that the text could be read at runtime. To be able to do this, the original Java code had to be tampered with, since without tampering, the program simply exits instantly. From Jadx, the source code was exported as a Gradle project, which could then be loaded into Android Studio. The code which checks for root detection could then be commented out, which would disable the root detection. To be able to debug the program, the code that checks if the program is being debugged was also commented out, as seen in fig. 5.

```
@Override // android.app.Activity
protected void onCreate(Bundle bundle) {
/*
    if (c.a() || c.b() || c.c()) {
        a("Root detected!");
    }
    if (b.a(getApplicationContext())) {
        a("App is debuggable!");
    }
*/
```

Fig. 5. Bypassing root detection mechanism.

A breakpoint was then set in the method `a.a()` and the program was then run in debug mode. At the moment when comparing the strings, the string was read as a sequence of bits. These were presumably ASCII encoded characters. Using an online ASCII decoder, it was revealed that the secret string was "I want to believe".

2) Second approach: The first step was to get Frida running. This required the Android VM to run `frida-server`, and the client to run `frida`.

To upload `frida-server` to the Android VM, the `frida-server` binary was first downloaded from GitHub[18]. Then, `adb push` was used to upload the file to the VM. With the sever on the VM, the server was run as root by first running `adb root`, and then by running `adb shell` to get a shell into the VM, and then running the `frida-server` binary.

With the server running on the VM, we could now inject our Javascript code to run when the uncrackable app was run. This was done by running the command `frida -U -f owasp.mstg.uncrackable1 -l level1Script.js`.

The first challenge was to bypass the root and debug detection. Since the application closed by running `System.exit(0)`, we could simply hook onto the method and override its function. Instead of exiting, we could simply return nothing. The second challenge was to read the contents of `bArr` (as seen in fig. 4). Since Frida does not allow direct access to variables of a method, we had to imitate what was done in the method and print the contents of the string before

[18]https://github.com/frida/frida/releases/tag/15.1.17

returning. By doing this, we could read out the string as ”I want to believe”. The code used can be seen in fig. 6.

```
setImmediate(() => {
    Java.perform(() => {
        let system = Java.use('java.lang.System');
        system.exit.overload('int').implementation = () => {
            console.log('[+] Evaded system exit\n');
        }

        let a = Java.use('sg.vantagepoint.uncrackable1.a')
        let aa = Java.use('sg.vantagepoint.a.a');
        let Base64 = Java.use('android.util.Base64')

        a.a.implementation = () => {
            const bArr = aa.a(a.b("8d127684cbc37c17616d806cf50473cc"),
                Base64.decode("5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2GOc=", 0))
            for (let i = 0; i < bArr.length; i++) {
                console.log(String.fromCharCode(parseInt(bArr[i])))
            }
            return true
        }
    })
})
```

Fig. 6. `level1Script.js`, used to solve the first challenge.

B. Second challenge

The second challenge was solved purely with static analysis which was done with decompiling and an analysis of the program code. Jadx was used to decompile the APK into readable Java code and Ghidra was used to decompile a native library into readable C code. From the decompiled code it was manageable to extract the hidden string.

There was an attempt to solve the challenge using Frida, like the second approach in the first challenge but unfortunately, we did not succeed.

1) First approach: The second challenge initially began in the same premise as the first one, and the APK was downloaded from the OWASP uncrackable website. It behaved similarly when run through Android Studio which led to that initially, the same procedure for defeating root detection was used for this challenge. However when the APK was decompiled, the application was significantly larger than the first one, and the level of obfuscation had also increased. When the decompiled code was put into Android Studio, a significant number of errors were reported. This made it impossible to compile and run the Java code. The number of errors where so many that it would be inefficient to find and fix them all

Therefore a more straightforward static analysis approach was adopted, as just like in the first challenge some kind of comparison had to be made in order to validate the string provided by the user. From looking in the code, this comparison method was located in the class `CodeCheck` and was called `a()`. This method in turn returned a native method called `bar()`, which was imported from a library called libfoo. Neither Android Studio nor Jadx was able to provide any information that was readable for a human eye from libfoo.

To decompile the native library, the decompiler Ghidra was used. This provided several exported C functions. Browsing the symbol tree in Ghidra, we could see a function called

`Java_sg_vantagepoint_uncrackable2_CodeCheck_bar`

as seen in fig. 7.

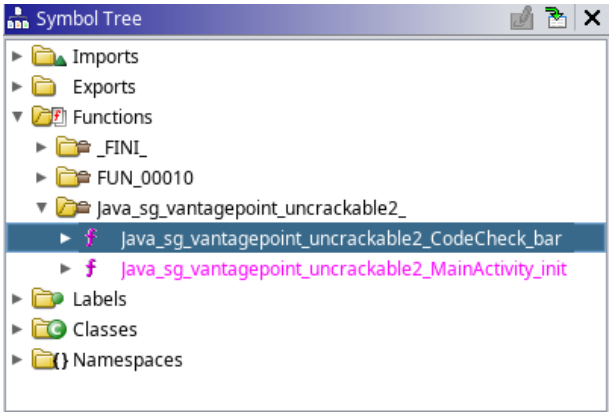

Fig. 7. Ghidra Symbol Tree, listing exported functions.

In this file, a C string comparison function is called, (`strncmp`), taking three arguments:

```
strncmp(const char *str1,
  const char *str2, size_t n).
```

The function compares two strings, by at most `n` bytes, and returns a non-zero value if the strings are not equal. The first argument, `str1`, is the user's input in the application text field, and the second argument, `str2`, is the hardcoded secret string. Luckily, this hardcoded string was also part of the exported function and had to be the string that was sought after.

The string values were stored as hexadecimal numbers, which here ASCII encoded characters. This can be seen in fig. 8. The characters were stored in reverse order, meaning the least significant bits were at the beginning of the string. Taking this into account when translating the numbers using an ASCII translator, we got the following string: *”Thanks for all the fish”*.

```
local_30 = 0x6e616854;
local_2c = 0x6620736b;
local_28 = 0x6120726f;
local_24 = 0x74206c6c;
local_20 = 0x6568;
local_1e = 0x73696620;
local_1a = 0x68;
__s1 = (char *)(**(code **)(*param_1 + 0x2e0))(param_1,param_3,0);
iVar1 = (**(code **)(*param_1 + 0x2ac))(param_1,param_3);
if (iVar1 == 0x17) {
  iVar1 = strncmp(__s1,(char *)&local_30,0x17);
  if (iVar1 == 0) {
    uVar2 = 1;
    goto LAB_00011009;
  }
}
```

Fig. 8. Contents of the exported function `bar()`, including the secret string.

2) Second approach: There were difficulties solving this challenge with Frida. The approach was to hook onto the native function `bar()` and try to read the contents of the string by reading the correct location in memory, or by simply doing a hexdump and trying to find the string from the output.

It was identified that the function `strncmp` was loaded, and its memory address was also recorded. By attaching to the function and reading the function, the goal was to read the first argument (recall that the first argument to `strncmp` is the hard-coded string, as seen in fig. 8) and read at that memory address and 23 bytes forward, giving us the entire string. Unfortunately, this did not work, and the output was pure gibberish. The code can be seen in fig. 9

```
setImmediate(() => {
  console.log('[+] Starting our hook');
  Java.perform(() => {
    let system = Java.use('java.lang.System');
    system.exit.overload('int').implementation = () => {
        console.log('[+] Evaded system exit\n');
    }
    var strncmp = undefined;
    let imports = Module.enumerateImportsSync("libfoo.so");
    for(let i = 0; i < imports.length; i++) {
    if(imports[i].name == "strncmp") {
            strncmp = imports[i].address;
            break;
        }
    }
    console.log('strncmp address: ', strncmp)
    let memAddr = ptr(0xecb35960)
    let strncmpcall = new NativeFunction(memAddr, 'pointer', ['int'])
    Interceptor.attach(strncmpcall, {
      onEnter: function (args) {
        console.log('args[0]: ',args[0],'args[1]:',args[1],'args[2]: ',args[2])
      },
      onLeave: function (retval) {
        return retval
      }
    })
  })
})
```

Fig. 9. Attempt at using Frida to solve level 2.

V. Discussion

The approach that successfully extracted the hidden string in both challenges was conveniently also the least complicated method of the three which were tested. This was the pure static analysis approach and it was the least complicated as both Jadx and Ghidra had great documentation and were user-friendly. The opportunities which presented themselves when access to the program code was presented were nearly limitless. Being able to freely search the source code for a string comparison made the task less complicated as the hidden string had to pass through comparison and once this was found, it was often a matter of understanding the comparison function to extract the string. In experience of working with the crackme challenges, obfuscation was the biggest obstacle in the way of analysis when it comes to decompiling. Program code is already hard to read and truly understand with documentation and well-documented code and as neither of these exists when decompiling, adding a layer of obfuscation on top of that makes it frustratingly hard to understand what's going on in the program code. A great obfuscation can also make an application uncompileable when decompiled as the decompiler program does not understand the compiled program fully, which in turn makes it difficult to turn the static analysis into a semi-dynamic analysis which also further complicates the analysis process.

When the initial APK is downloaded and run through Android Studio, it shutdown thanks to root detection. The root detection made debugging as an analysis tool worthless in that particular aspect. But as mentioned above, this anti-analysis technique only works if the APK is obfuscated to a degree that a decompiled version could not compile again. This is because it prohibits the analyst to remove the root detection from the program code.

The root detection can also be bypassed dynamically without decompiling the code by hijacking a function in the code to run self-written code. This code can analyze what is going on in the application without triggering the root detection. This can be done using Frida. If an application is obfuscated to a degree that makes static analysis difficult, and a decompiled APK is uncompilabe, a dynamic approach with Frida might be one of the few remaining alternatives for analysis.

If there was not a limited time window for the research, a deeper understanding of Frida would have made it possible to crack the second challenge with a fully dynamic method. A deeper understanding of Frida would be the main focus if the project would start over or if the next challenges would be attempted.

VI. Conclusion

In this paper, a study was conducted where we looked at common anti-analysis methods, and tools used to bypass them. We found that bypassing most anti-analysis methods is possible by using a combination of static analysis and dynamic instrumentation tools, allowing for reverse-engineering of applications.

References

[1] c. c. muellerberndt, sushi2k, "Uncrackable mobile apps," https://github.com/owasp-mstg/Crackmes, 2022.

[2] "Application fundamentals," Nov 2021. [Online]. Available: https://developer.android.com/guide/components/fundamentals

[3] A. Balakrishnan and C. Schulze, "Code obfuscation literature survey," *CS701 Construction of compilers*, vol. 19, 2005.

[4] S.-T. Sun, A. Cuadros, and K. Beznosov, "Android rooting: Methods, detection, and evasion," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 3–14. [Online]. Available: https://doi.org/10.1145/2808117.2808126

[5] H. Jang, B. Jin, S. Hyun, and H. Kim, "Kerberoid: A practical android app decompilation system with multiple decompilers," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2557–2559. [Online]. Available: https://doi.org/10.1145/3319535.3363255

[6] A. Desnos and G. Gueguen, "Android: From reversing to decompilation," *Proc. of Black Hat Abu Dhabi*, vol. 1, p. 1, 2011.

[7] N. Mauthe, U. Kargén, and N. Shahmehri, "A large-scale empirical study of android app decompilation," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 400–410.

[8] W. Enck, D. Octeau, P. D. McDaniel, and S. Chaudhuri, "A study of android application security." in *USENIX security symposium*, vol. 2, no. 2, 2011.

[9] skylot, "jadx," https://github.com/skylot/jadx, 2022.

[10] [Online]. Available: https://frida.re/

[11] c. n. dragonmacher, ghidra1, "ghidra," https://github.com/NationalSecurityAgency/ghidra, 2022.

[12] "Android debug bridge (adb) nbsp;— nbsp; android developers," Mar 2022. [Online]. Available: https://developer.android.com/studio/command-line/adb