Memory Forensics of Android Backdooring Based on App Virtualization

Enrique Anthony Galea

Supervisor: Dr. Mark Joseph Vella

June 2023

Submitted in partial fulfilment of the requirements for the degree of Computing Science.

L-Università ta' Malta
Faculty of Information & Communication Technology

Abstract

Smartphones have become ubiquitous in our daily lives, offering convenient access to our data, and making them an attractive target for cybercriminals. In fact, many different versions of Android backdoors have been developed and used to gain unauthorized access to users' smartphones and their data. Although capable of detecting and defending against malware, mobile devices are limited in performing more advanced detection techniques due to their power constraints. As malware authors continue to use advanced evasion techniques, mobile devices have become increasingly vulnerable to sophisticated attacks.

App virtualization is a technique that allows applications to run inside virtual environments created by other applications. In doing so, their visibility is hidden from other applications installed on the device. Such a technique can be potentially used by backdoors to evade detection and further enhance their stealth capabilities. By evading initial detection mechanisms, backdoors can more easily achieve objectives such as data exfiltration.

In this paper, we propose VirtuSleuth, a tool in the form of an Android application that can detect virtualized applications and recover their code for analysis. Our tool analyses the running processes on the device, identifies those belonging to virtualized applications, and extracts their code from volatile memory. The proposed solution offers an effective approach for analysing virtualized applications as it targets the live memory where the virtualized application's code must be loaded before it is executed. In doing so, we overcome app virtualization stealth and improve upon existing anti-malware solutions.

We conduct experiments to compare the stealth level of Android backdoors when they are not virtualized versus when they are virtualized, using indicators of compromise as the measure. We also evaluate our tool to determine its level of accuracy in detecting virtualized applications and practicality in the time taken to detect virtualized applications and extract their code. Finally, we discuss the limitations of the tool and future work.

Acknowledgements

First and foremost, I extend my deepest gratitude towards my supervisor, Dr. Mark Vella. Under his guidance, I have grown and developed in my academic pursuits. Committed to my success, he has shown remarkable patience and understanding throughout the entire project. Knowing how to approach complex subjects proved invaluable in helping me tackle the challenges I encountered. Ultimately, thanks to his support, I managed to overcome every challenge and complete this project. Moreover, I would also like to thank my friends and family for their support and encouragement.

Contents

List of Figures

List of Tables

List of Abbreviations

AOT	Ahead-of-Time
APK	Android Package
ART	Android Runtime
DEX	Dalvik Executable
ELF	Executable and Linkable Format
GUI	Graphical User Interface
JIT	Just-in-Time
JNI	Java Native Interface
NDK	Native Development Kit
UID	User ID
VM	Virtual Machine
XML	Extensible Markup Language

1 Introduction

Smartphones are an integral part of our daily lives, providing us with convenient access to information and services. App stores have evolved as a vital component of the mobile platform, providing users with a trusted and convenient place for downloading and updating apps. With millions of applications available on the Google Play Store [1], users have access to a wide range of software for their devices. However, the open nature of the Android platform also means that threat actors can easily distribute malicious applications. Despite Google's efforts to keep the platform safe and secure, malicious applications still manage to slip through the cracks [2]. With the growing number of threats targeting individuals and organizations [3], anti-malware applications have become important tools in keeping our devices and personal data protected.

There is no shortage of anti-malware applications available [4], many of which can be found on the Google Play Store. Some are even included with the operating system itself, such as Google Play Protect [5]. In response to the increasing sophistication of malware [6], anti-malware applications have evolved to become more effective at detecting and removing different types of malware. As a result, modern anti-malware applications incorporate a variety of advanced techniques such as dynamic analysis, machine learning, cloud protection, and more [7]. In addition, most anti-malware applications offer real-time protection, continuously monitoring the system for threats and blocking them before they can cause any harm. However, such advanced features require continuous processing and elevated privileges to be able to function.

1.1 Problem

Mobile devices are inherently limited in terms of their processing power due to their portable nature. Combined with the limitations imposed by the Android operating system on user installed applications, these anti-malware applications are forced to operate with more limited capabilities compared to their desktop counterparts, while trying to achieve the same goal. For example, techniques involving dynamic analysis cannot be performed because applications are sandboxed and cannot monitor the

behaviour of other applications. However, even if these limitations were not present, such techniques would still not be viable since the continuous monitoring required would significantly impact performance and battery life. As a result, mobile anti-malware applications primarily focus on providing basic protection against malicious applications in which malicious code is pre-embedded at the time of installation. However, these measures often fall short when dealing with more sophisticated instances of malware that dynamically load harmful code from external sources during runtime. Unfortunately, this means that users are left vulnerable to more sophisticated attacks.

App virtualization is a technique that allows an Android application to run within a virtualized Android environment created by another application [8]. While this approach has legitimate uses, such as for testing and sandboxing, it has also been exploited for malicious purposes [9]. When an application is virtualized, anti-malware applications might not be able to detect it, as it is hidden from other installed applications. The use of app virtualization by backdoors, such as to virtualize a malicious application obtained from an external source during runtime, means that on-device detection measures can be potentially evaded. Additionally, there is also the potential of circumventing the scanning conducted during submission to the Google Play Store, which is intended to compensate for weak on-device malware detection.

1.2 Proposed Approach

Figure 1.1 demonstrates a high-level overview of the problem. First, the user downloads a seemingly benign application with app virtualization capabilities. On installation, the application is analysed by any anti-malware applications on the device and not detected. Once launched by the user, the newly installed application fetches a malicious application from an external source. This malicious application is then virtualized, with its code being loaded inside volatile memory and not available to the anti-malware application.

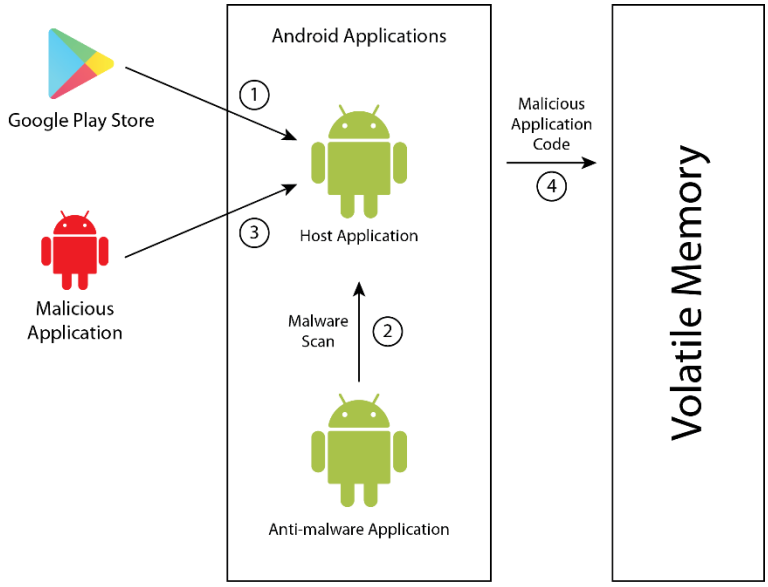

Figure 1.1: Problem Overview

To address the threat posed by app virtualization, we propose VirtuSleuth, a tool designed to detect and analyse virtualized Android applications. VirtuSleuth is an Android application that identifies virtualized application processes by examining the running processes on the device. Once identified, the bytecode of the virtualized application is extracted from memory. This effectively brings back the concealed components of virtualized applications, allowing them to be analysed for malware.

1.3 Aims and Objectives

The aim of this study is to determine how volatile memory forensics can be used to make stealthy, virtualized Android backdoors available again to malware detection engines. Our main objectives are to:

1. Develop an Android application that virtualizes a known backdoor, demonstrating the potential for malicious exploitation.
2. Develop a tool that can effectively detect virtualized Android backdoors and applications, showcasing the ability to counteract app virtualization stealth.
3. Demonstrate the increased stealth of app virtualization by comparing the indicators of compromise (IOCs) generated by a backdoor when executed as a standard Android application versus when virtualized through the application developed in the second objective.

4. Evaluate the tool's feasibility and practicality by measuring the detection accuracy and the time taken for detection and extraction of bytecode.

1.4 Organisation

Chapter 2 lays the groundwork necessary for understanding our project. It explores the details of Android, app virtualization, backdoors, and more. The section also concludes with a review of past research that is related to our work. Chapter 3 delves into the intricacies of VirtuSleuth. Here, we detail its design, implementation, and the reasoning behind our design choices. By doing so, we provide insight into VirtuSleuth’s approach for detecting virtualized applications. In Chapter 4, we outline the evaluation strategies used to validate VirtuSleuth's effectiveness. This includes our experimental setup, results, and a discussion of VirtuSleuth's limitations. Finally, Chapter 5 provides a summary of our work and achieved objectives. The section also discusses potential enhancements to VirtuSleuth, identifying the areas where future research and development can improve the tool's compatibility, functionality, and more.

2 Background and Related Work

In this background section, we will explore the foundations necessary for understanding the context and challenges surrounding Android and virtualized applications. We will begin with an overview of Android's Linux foundation, discussing its role in providing key features such as sandboxing and memory mapping. Next, we will delve into the structure and components of Android applications, highlighting how they are built and distributed. We will then discuss the Android runtime, including the roles of DEX and OAT formats in application execution. After, we discuss app virtualization itself, detailing how applications are emulated by the technique. Finally, we will provide an overview of Android backdoors, mentioning the ways in which malware authors exploit the platform and how app virtualization can be leveraged as a novel attack vector.

2.1 Android’s Linux Foundation

Android is an open-source mobile operating system based on the Linux kernel [10]. As the foundation of the operating system, the Linux kernel enables Android to share many features and technologies with other Linux-based systems, such as multitasking capabilities and support for diverse hardware configurations.

Leveraging the Linux kernel, Android provides strong isolation between applications through the use of a sandboxing mechanism, which prevents applications from accessing each other's data without explicit permission [11]. To achieve this, each application is assigned its own unique user during installation, and only this user is allowed to access the private memory and storage space of the application. This is combined with process isolation and restrictions on inter-process communication to effectively separate applications from each other. This design is shown in Figure 2.1.

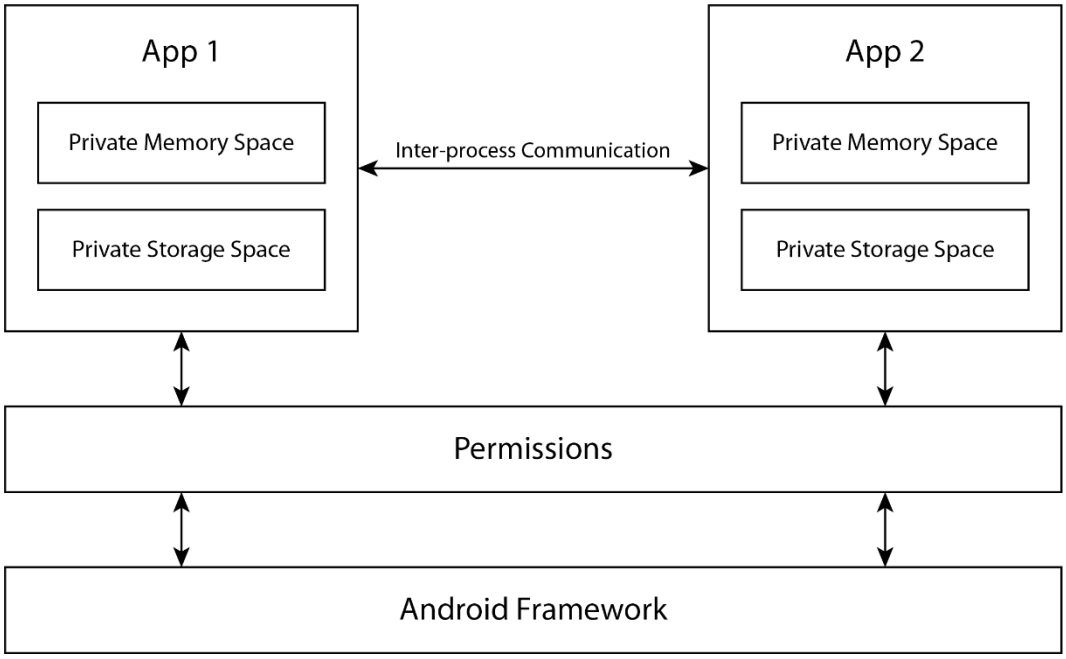

Figure 2.1: Application Sandbox

Another feature facilitated by the Linux kernel is the forking process used by Zygote. Forking is the creation of a new process by duplicating an existing one, which allows the child process to inherit properties, resources, and the environment of the parent process [12]. Zygote is an Android process that starts on boot and utilises forking to efficiently launch new applications by duplicating itself [13]. In this way, Zygote serves as a template process for all Android applications, ensuring that they inherit the necessary runtime environment and resources.

Processes in Linux, including Android, have their own virtual address space that is divided into regions, each of which is associated with different permissions and content. Memory mapping allows these regions to be associated with specific files, allowing reading or writing to the file as if it were a part of its own memory. However, when a file is memory mapped, it isn't necessarily mapped entirely or contiguously. It can be fragmented across memory, with each fragment starting at a different offset. To view all the mapped regions of a specific process, one can examine the contents of /proc/[PID]/maps [15]. However, to inspect the memory mapping of a process, one must have the appropriate permissions. Typically, only the user that owns a process or the root user can examine the memory map of that process.

Before executing an application, the associated code must be loaded into memory. This process is facilitated by specific file formats that are designed to be

straightforwardly mapped into different regions of the address space. On Linux-based systems, including Android, the Executable and Linkable Format (ELF) is used. ELF is a common standard file format for executables, object code, shared libraries, and core dumps. An ELF file is structured to enable easy memory mapping, consisting of a header, followed by program headers or section headers, or both. These headers provide necessary information to create a process's memory image, allowing a seamless transition from an on-disk file to an in-memory executable.

2.2 Android Applications

At the heart of Android's architecture lies the application framework, which exists alongside the native layer. This dual-layer structure allows developers to create applications using the Java and Kotlin programming languages, leveraging high-level APIs in the framework layer, as well as utilise low-level native code when needed [16].

Applications are built using a combination of four different components [17]. The most common component, activities, represent user interfaces associated with a single screen for user interaction. Services are background components that execute long-running operations or tasks without user interaction. Broadcast receivers allow an application to perform an action in response to a system-wide event or message. Lastly, content providers allow application to access shared data by other applications.

Android applications are distributed as APK files. An APK file is simply an archive that contains all the necessary components for the installation and execution of an Android application [18]. The most important components of the APK file include the AndroidManifest.xml file and the classes.dex file. The AndroidManifest.xml file contains important metadata about the application, such as its package name, required permissions, and declared components. The classes.dex file holds the compiled bytecode of the application in the Dalvik Executable format. During installation, this file is copied to a directory that stores the compiled bytecode for every application on the device. When the application is started, the file is loaded into memory from this location and executed. The contents of an APK file are shown in Figure 2.2.

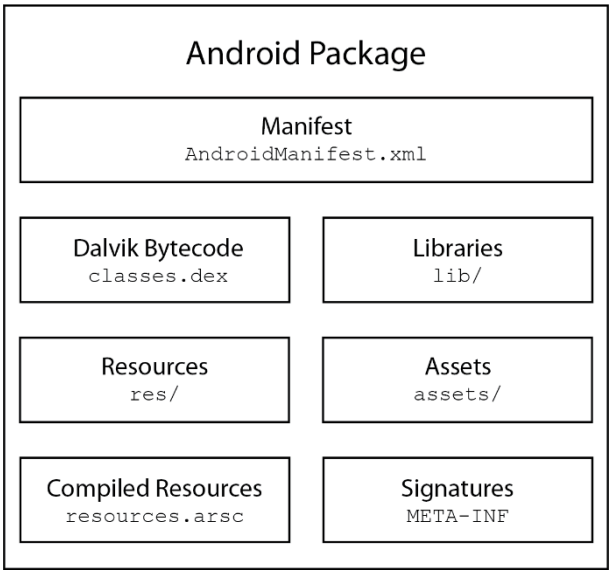

Figure 2.2: Android Package

Beyond the Dalvik bytecode contained in the classes.dex file, Android applications can also include natively-compiled code to optimize performance or interface with system-level components. This code is stored in the 'lib/' directory within the APK file and is organized by the instruction set architecture (ARM, ARM64, x86, etc.) it's compiled for. This natively-compiled code, usually in the form of shared libraries, is written in languages like C or C++ and compiled directly to machine code.

2.3 Android Runtime

To run Java or Kotlin code inside Android applications, it must first be compiled into CLASS files, a format for bytecode that is executable by the Java VM, which interprets the bytecode and translates it into native machine code at runtime [19]. While this approach allows the same code to be compiled once to run on different architectures, it results in less efficient performance compared to native code. Due to the limited processing power and memory capacity available on mobile hardware during the time of Android's development, a customized runtime for the Android platform was developed, known as Dalvik [20]. Dalvik executes DEX files, which are compiled from CLASS files using the dx tool [21].

Although the Dalvik runtime addressed key limitations with using the Java VM at the time, it still came with other limitations, such as slower garbage collection algorithms. To address these limitations, Google introduced ART as the successor to

Dalvik [22]. The biggest difference of ART compared to Dalvik is that it executes applications AOT instead of JIT. ART compiles DEX files into OAT files that contain native instructions. This process is performed by the dex2oat tool during app installation and allows applications to execute faster while also consuming less power. OAT files are wrapped inside an ELF file. This allows the operating system to utilize existing mechanisms for loading and executing native code. Thus, each OAT file is essentially an ELF file that contains sections for the OAT-specific data as well as the original DEX bytecode. The original DEX files are still maintained due to JIT still being required in certain scenarios, such as when debugging an application.

Different Android versions compile applications into different versions of the OAT file format. For example, Android 7.0 and 7.1 use version 79 of the OAT file format. This version is organized into various sections, including the OAT Header, DEX Files, OAT Classes, Garbage Collector, and various lookup tables [23]. These sections can be parsed to extract the native code of the application or the DEX files embedded within. Tools that parse OAT files, such as LIEF [24], are available for this purpose. Android Oreo and later also make use of VDEX files. VDEX files contain the uncompressed DEX code of the APK, with some additional metadata to speed up verification [57]. This allows for speedier app startup times at the cost of an additional layer of complexity to the process. An overview of the transformation process from Java to Dalvik bytecode is shown in Figure 2.3.

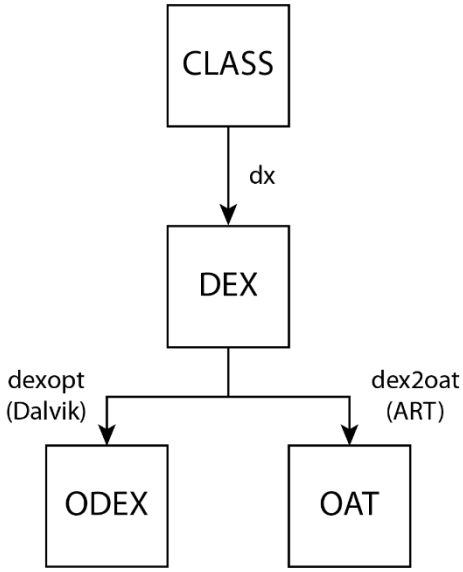

Figure 2.3: Code Compilation Stages

2.4 App Virtualization

App virtualization allows an application to emulate another application by setting up a virtual Android environment. In this context, the application setting up the virtual environment is referred to as the host application while the applications being emulated are referred to as plugin applications. The technique is easily maintainable and works across a variety of devices and configurations [25]. Integration into an existing project involves importing a library supporting app virtualization and calling a function to install and execute the plugin application inside the virtual environment. Multiple libraries supporting app virtualization exist, such as VirtualApp [25] and DroidPlugin [26].

The virtual Android environment includes emulated framework and native layers. The framework layer provides a virtualized Android framework that intercepts calls from the plugin applications and translates them to the corresponding methods, while the native layer provides a virtualized native layer that allows plugin applications to load native libraries and execute native code [25]. After the host application establishes the virtual environment, it forks its process into multiple processes, equivalent to the number of plugin applications it intends to run. Each plugin application shares the same UID as the host application but operates in its own distinct process. The host application can then compile the bytecode from the plugin application APK file into its private data directory and execute the plugin application.

At this point, any requests sent by the plugin application are received from the host application by the Android OS. However, the host application does not know the names of the plugin application components it will emulate at compile time, so it is not able to include them inside its manifest. To address this, the host application declares several stub components inside its AndroidManifest.xml and uses dynamic code hooking to intercept each request and reply going towards to or coming from Android. Once intercepted, the names of plugin application components are dynamically changed to correspond to the stub components declared in the host application manifest. This is illustrated in Figure 2.4. For permissions, the host application simply declares all permissions so that any permissions declared by the plugin application will be provided.

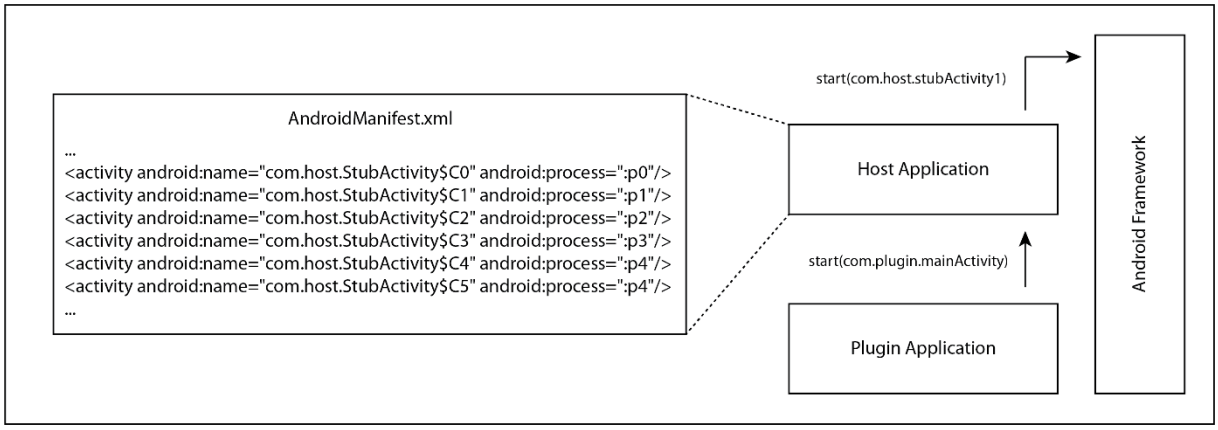

Figure 2.4: App Virtualization Proxying

VirtualApp (VA) provides an easy way to load an APK plugin through its API. To utilize VA in a project, one simply needs to include its library and invoke some methods in just a few lines of code. The 'installPackage' method is first called to install the APK in the virtual environment. This method returns an 'InstallResult' object, from which the package name can be extracted. To launch the installed APK, an intent is created using the 'getLaunchIntent' method, specifying the package name and a user identifier as arguments. Finally, the 'startActivity' method of 'VActivityManager' is called to initiate the plugin's main activity

2.5 Android Backdoors

Android backdoors are malicious applications that gain unauthorized access to a device and its data [27]. An attacker making use of a backdoor can gain unauthorized access to the device remotely, without the need for physical access. The backdoor could be an application itself or it might be included inside another application. Backdoor functions include data exfiltration, remote command execution, and system modifications [56]. While backdoors try to operate covertly, they must still interact with the Android operating system and its various APIs to carry out their activities. This interaction may leave traces in the form of IOCs. An IOC is simply any piece of information that can be used to identify potentially malicious activities within a system or network, such as unusual network traffic. Today's backdoors strive to minimize their IOCs and evade detection mechanisms through many techniques.

An evasion technique employed by Android backdoors is the use of DexClassLoader. The DexClassLoader is a part of the Android application framework

and enables applications to load and execute DEX files dynamically at runtime [28]. Attackers can leverage the DexClassLoader to conceal malicious code by loading it dynamically during runtime, rather than bundling it with the application package. This makes the malicious code more difficult to detect through static analysis methods employed by anti-malware solutions, as the code is not present in the application's original binary.

Another evasion technique is the use of Java Reflection. Reflection is a powerful feature of the Java programming language that allows developers to inspect, modify, and invoke methods and fields of objects during runtime [29]. Attackers exploit Java Reflection to obfuscate their malicious code by dynamically loading and invoking methods from other classes. This makes it more difficult for static analysis tools to identify the relationships between different components of the backdoor. Additionally, reflection can be used to modify the behaviour of existing code, enabling the attacker to inject malicious functionality into seemingly benign classes.

App virtualization can be used as an attack vector for Android backdoors, providing several advantages. Since plugin applications are emulated and not installed on the device, they will not appear in the list of installed packages. Additionally, plugin applications can be started without user interaction and do not send ACTION_PACKAGE_ADDED broadcast receivers when installed in the virtual environment, meaning anti-malware applications making use of this component will not receive it. While the APKs of installed applications on the device can be found in /data/app, APKs of virtualized applications are stored in the private data directory of the host application instead, making them inaccessible to other applications for analysis due to the application sandbox. Lastly, app virtualization also enables the possibility of stealing plugin application private data due to the shared UID between host and plugin applications [30]. A malicious host application can emulate an application already installed on the device to impersonate the original and perform dynamic hooking to intercept and steal data.

2.6 Related Work

2.6.1 Machine Learning Based Detection

Machine learning-based detection techniques have been tested as a way to improve malware detection and classification. Studies such as those by G. Canfora et al. [31] and D.-J. Wu et al. [32] have proposed machine learning algorithms to identify and classify Android malware effectively. These methods typically rely on features extracted from the application package, dynamic analysis, or a combination of both. Additionally, the vast majority of anti-virus vendors today leverage machine learning in their security solutions [55]. While machine learning-based approaches can be effective in detecting malware, their efficacy in detecting virtualized malicious applications remains an open question.

2.6.2 Anti-malware Evasion

Many studies have already explored the subject of evading Android anti-malware applications, such as Rastogi et al. [33], Meng et al. [34], and Lars Richter [35]. However, these works have concentrated on employing code obfuscation techniques in repackaged malware to circumvent malware detection engines. In contrast, app virtualization offers an alternative method for evading anti-malware applications without resorting to code obfuscation or repackaging. Moreover, another form of evasion involves the use of firmware backdoors, such as Triada [54], where malicious actors embed malware directly into the device's firmware. However, this falls outside the scope of our study.

2.6.3 App Repackaging

Khanmohammadi et al. conduct an empirical study on repackaged Android applicationsa to gain insights into the factors that drive the spread of repackaged apps [36]. Repackaging an Android application requires the attacker to obtain and decompile the application package, reverse engineer it, modify the required components to change their behaviour, rebuild the application, and finally resign it. Repackaging an Android application is another way of delivering malware, but is different from app virtualization in that the malicious payload is bundled with the application.

2.6.4 App Virtualization

Wu et al. [37] and Luo et al. [38] have developed frameworks that enable applications to detect whether they are being virtualized by utilizing timing measurements and runtime information respectively. Although our goal is to also detect virtualization, we must do so externally from the virtualized application in order to be able to scan for all applications that are being virtualized and extract the code from memory once they have been identified.

Shi et al. [39] and Alecci et al. [40] explore the potential of app virtualization to generate new attack vectors on the Android operating system, such as the extraction of sensitive user data by taking advantage of the shared user ID of apps running within virtual environments. These studies present important examples of the ways in which attackers can utilise app virtualization after successfully installing a trojan on the user's device.

Ruggia et al. use app virtualization to build a secure environment where apps can be checked at runtime to protect against repackaging [41]. Chen et al. [42] and Pizzi et al. [43] investigate how virtualization can be used to fix security vulnerabilities and deploy patches quickly, while Backes et al. [44] investigate how the technique can be used to safely sandbox Android applications to enable dynamic analysis and detect malicious behaviour. These studies provide additional insight into app virtualization but are not related to our work.

2.6.5 Memory Forensics

Mobile memory forensics tools and frameworks, such as LiME [45], have been developed to retrieve entire memory images of devices for analysis. However, utilizing LiME on an Android device requires that a kernel module used for memory extraction be compiled and loaded with the device kernel. Since kernels shipped with devices do not support module loading, the specific version of the kernel used on the device must be recompiled with module loading support and replaced. Additionally, dumping the entire memory contents of a device is unnecessary when only the partial memory contents used by a specific application is required in our case. While LiME could be used, it would be more complicated and less practical to do so compared to a more targeted approach that only extracts the required contents from memory.

2.6.6 Process Memory Analysis

Process memory analysis has been used by Sanggeun Song et al. [46] to prevent Android ransomware attacks using statistical methods based on processor, memory, and I/O usage. The aim of their system is to detect processes with abnormal behaviours and stop them, while storing information about the suspicious processes in a database. Although their research focuses on ransomware detection, the process monitoring approach they propose shares similarities with our work in monitoring processes to detect virtualized applications.

Saltaformaggio et al. [47] and Ali-Gombe et al. [48] employ process memory forensics techniques to reconstruct GUI components and runtime memory structures, enabling them to determine user activity from a device memory image. While this approach has the potential to identify the presence of virtualized applications, our focus is on detecting these applications while the process is running on the device to form part of a real-time detection setup.

Bellizzi et al. explore the use of hooking applications to obtain timely captured memory dumps when specific calls or features are employed [49] [50], allowing for the analysis of memory images at a later stage in case evidence of a compromise is discovered. Although their research also utilises memory forensics to detect malicious activity, the scope of our study differs in that we concentrate on using memory forensics to extract virtualized application code.

2.7 Conclusion

This chapter laid the foundation for understanding the context around our work, including the structure of Android applications, the Android Runtime environment, app virtualization, and Android backdoors. An overview of related work in fields like anti-malware evasion, app virtualization, memory forensics, and process memory analysis was also discussed. This necessary groundwork sets the stage for the introduction of our tool, VirtuSleuth, which we will discuss in the next chapter.

3 VirtuSleuth Design

In this section, our first two objectives will be addressed. Deriving its name from the words "virtualization" and "sleuth," VirtuSleuth embodies the concept of investigating virtualized applications by detecting virtualized application processes running on a device and extracting their bytecode from volatile memory for analysis. In doing so, we address the challenges related to the increased stealth provided by app virtualization.

Figure 3.1 represents the stages involved in the virtualized application detection and extraction process. This section starts by detailing the development of an offensive solution, continues with a discussion of the design of every stage of the VirtuSleuth process, and ends with the design choices made during the development of VirtuSleuth.

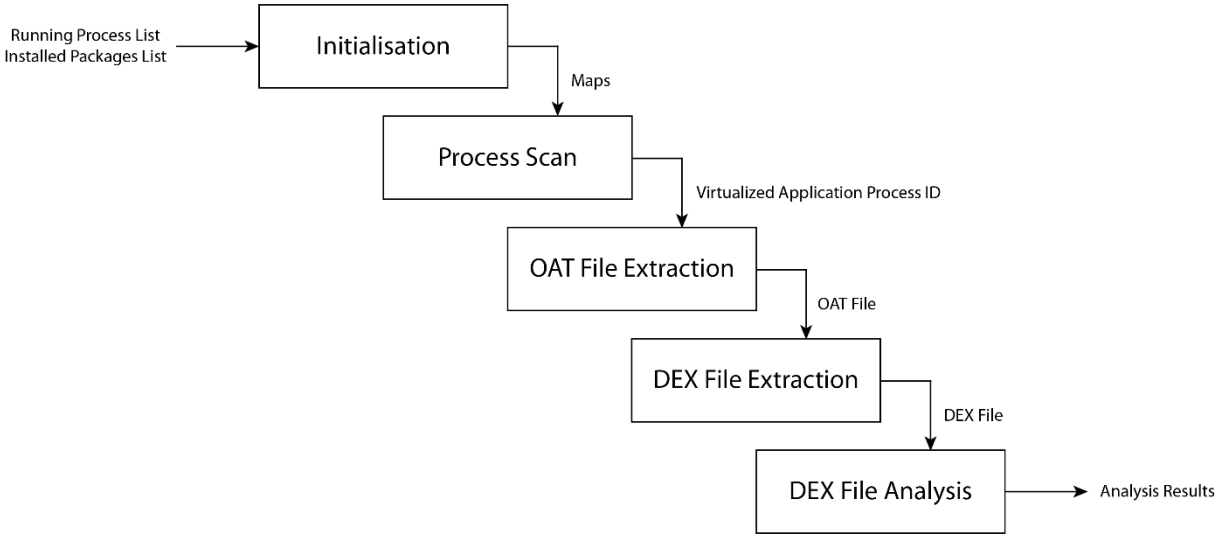

Figure 3.1: VirtuSleuth Process Flowchart

3.1 Backdoor Virtualization

An offensive solution was first created before starting development on the defensive solution. This involved creating an Android application that is capable of virtualizing a malicious Android backdoor using the VirtualApp (VA) library.

The process began with incorporating the VA library into the host application project. The VA library offers an API that makes it easy to virtualize applications by requiring only the path to the APK on the internal storage of the device. We opted to

hardcode a path to an APK file in the downloads folder, allowing us to virtualize different applications by simply replacing the APK without having to rebuild the host application every time.

With the Meterpreter APK in the specified path, the host application is programmed to install the APK using VA's installPackage() function. Following successful installation, the host application then utilized VA's getLaunchIntent() function to retrieve an Intent capable of launching the virtualized backdoor's main activity. This Intent was subsequently passed to VA's startActivity() function to initiate the execution of the backdoor in the virtual environment.

With Meterpreter virtualized, the backdoor could operate and interact with the Android system as if it were a standalone application, while being masked by the host application's identity. There was also no indication that Meterpreter was being executed as the main activity of the backdoor does not provide a user interface, but we could still connect to the reverse shell established by Meterpreter and execute commands.

The process of virtualizing Meterpreter was successful, meeting the project's objective of creating an offensive solution. This also served as a stepping stone for the development of the defensive counterpart, VirtuSleuth. By having a working model of a virtualized malicious application, we could accurately test VirtuSleuth's ability to detect and analyse such threats in a real-world scenario.

3.2 Architecture

Figure 3.2 provides a high-level depiction of VirtuSleuth's integration within a standard Android device setup. Installed as a typical Android application, VirtuSleuth operates in conjunction with any on-device anti-malware applications. Standard applications on the device are managed by the anti-malware applications, while VirtuSleuth is responsible for the detection of any stealthy virtualized applications. Upon detecting such applications, VirtuSleuth extracts their bytecode from volatile memory and analyses them for malware.

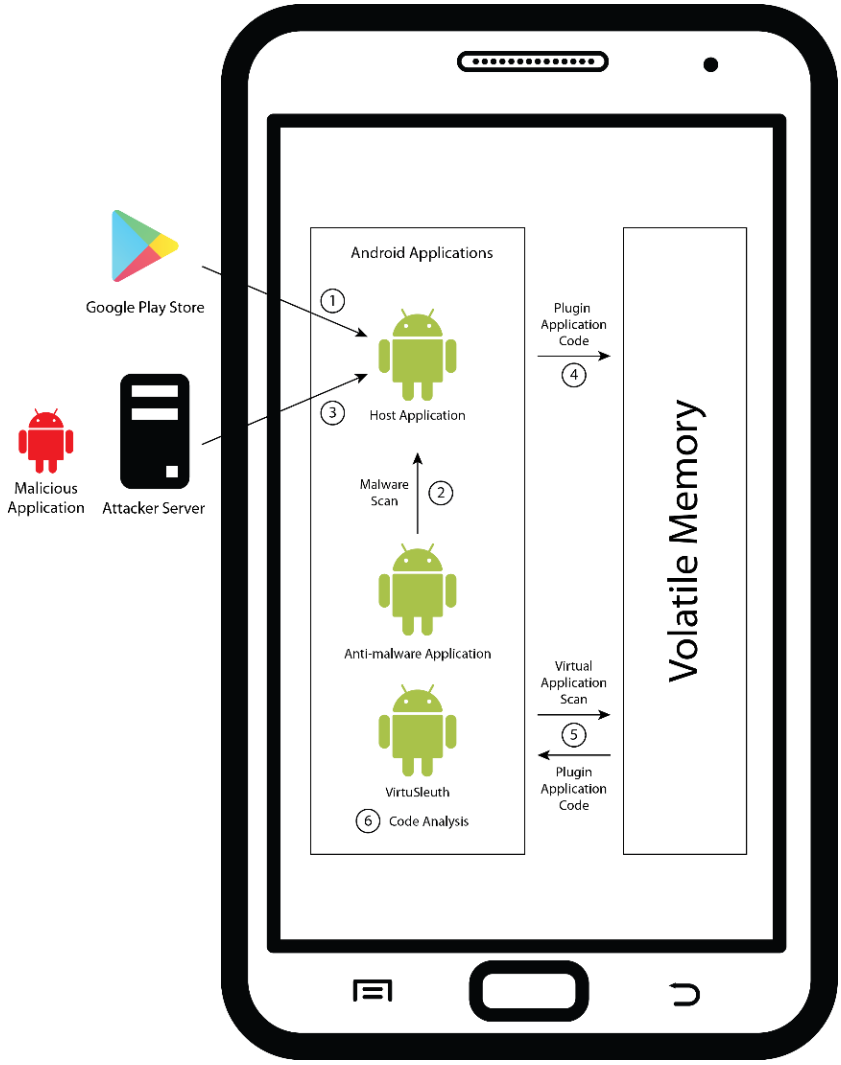

Figure 3.2: Solution Overview

3.3 Initialisation

VirtuSleuth exploits the fact that plugin applications run in a distinct process while sharing the same UID with the host application. By analysing running processes on the device and the corresponding users that started them, it is possible to identify virtualized processes. To analyse the processes on the device, two structures are required. The first is a map of users to processes, which allows easy grouping of users and their running processes. The second structure is a map of users to packages installed on the device, used to determine which packages share the same UID.

3.4 Process Scan

Before identifying the virtualized processes, it is necessary to filter out instances where multiple processes owned by the same user are unrelated to app virtualization. The following scenarios were identified in which users may have multiple processes apart from app virtualization:

- Packages with the same signature utilizing the sharedUserId manifest attribute.
- Simultaneous operation of multiple components of the same applications.
- Shell processes launched by the application.
- Opening of files such as shared libraries.

A copy of the map mapping users to processes is created, and these instances are identified and filtered out of this new copy. After doing so, any remaining users with more than one process indicate that these processes correspond to host and plugin applications. Subsequently, the host and plugin application processes can be discerned by verifying which process corresponds to a package already installed on the device. The pseudocode for this algorithm is listed in Algorithm 3.1.

Algorithm 3.1 Virtualized Application Detection

```
1:  input user_processes_map, user_packages_map
2:  filtered_user_processes_map = copy(user_processes_map)
3:  for user, processes in filtered_user_processes_map.items():
4:      processes = filter_out_unwanted_processes(processes)
5:  for user, processes in filtered_user_processes_map.items():
6:      if len(processes) > 1:
7:          plugin_process_pid = identify_plugin_process(processes)
```

3.5 OAT File Extraction

With the host and plugin application processes identified, the next step is to determine the location of the bytecode in memory so that it can be extracted. The memory-mapped files of the plugin application process are iterated through to determine mapped instances of the classes.dex file. Once all mapped instances are found, any duplicate instances are removed, and the remaining ones are sorted to be in sequential order. With the list of all mapped segments and their locations, these are dumped separately to internal storage and joined together. Algorithm 3.2 details the OAT file extraction process.

Algorithm 3.2 OAT File Extraction

```
1: input plugin_app_pid
2: mapped_files = executeCommand("cat /proc/" + plugin_app_pid + "/maps")
3: mapped_dex_files = mapped_files.findMappedSequences('classes.dex')
4: for each dex_file in mapped_dex_files:
5:     extractMemory(plugin_app_pid, dex_file.start, dex_file.length)
6: combine_dex_files()
```

3.6 DEX File Extraction

The classes.dex file extracted from memory is the compiled OAT for the virtualized application. Before the bytecode can be analysed, the DEX files need to be retrieved from the OAT file. There are several tools available that can parse OAT files and extract the embedded DEX files. However, these could not be used, as the last few hundred bytes of the OAT file are not mapped in memory, breaking the OAT file format and preventing the tools from recognizing the format. Since the DEX files are located at the beginning of the file, they are not missing and can be obtained by parsing the file manually instead. The parsing algorithm goes through the extracted OAT file and outputs the individual DEX files embedded within it to the same folder in internal storage. The sections of the OAT file relating to DEX files are illustrated in Figure 3.3.

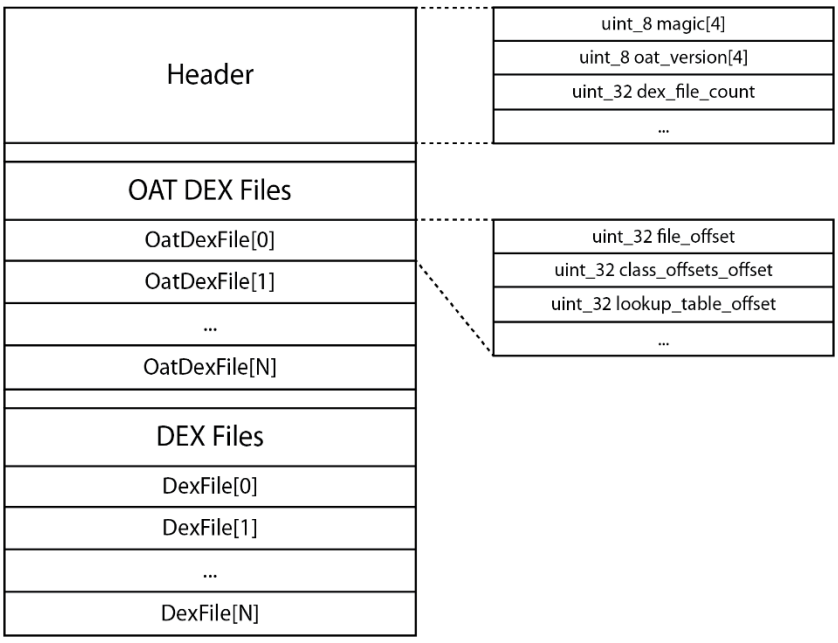

Figure 3.3: OAT File DEX Sections

The parsing algorithm must first determine the offsets of the embedded DEX files. This is done by reading the "dex_file_count" value from the OAT header and using this value to iterate over the next section of the file to determine the offset values of the DEX files. It then extracts these regions and saves them to internal storage. This process is described in Algorithm 3.3.

Algorithm 3.3 DEX File Extraction

```
1:  input plugin_application_pid
2:  mapped_files = get_mapped_files(plugin_application_pid)
3:  oat_file_name = find_classes_dex_file(mapped_files)
4:  oat_file_path = extract_oat_file_from_memory(oat_file_name)
5:  open(oat_file_path)
6:  skip_bytes(4096)
7:  verify_magic_value()
8:  verify_version_value()
9:  skip_bytes(12)
10: dex_file_count = read_bytes(4)
11: skip_bytes(44)
12: key_store_size = read_bytes(4)
13: skip_bytes(key_store_size)
14: location_size = read_bytes(4)
15: skip_bytes(location_size)
16: skip_bytes(4)
17: file_offset = read_bytes(4)
18: skip_bytes(4)
19: lookup_table_offset = read_bytes(4)
20: for i in range(dex_file_count):
21:     skip_bytes(4)
22:     file_offset = read_bytes(4)
23:     skip_bytes(4)
24:     lookup_table_offset = read_bytes(4)
25:     extract_bytes(file_offset, lookup_table_offset)
```

3.7 DEX File Analysis

Once the DEX files are obtained, the final step involves analysing them for malware. In our case, a classification is obtained by uploading the files to VirusTotal via their API, and the number of engines that classify the DEX file as malware is displayed on the screen after the analysis is completed. While we opted to use VirusTotal, any form of malware detection algorithm can be implemented here.

3.8 Design Choices

While developing VirtuSleuth, various design choices were made to ensure the efficiency and effectiveness of the tool:

- **OAT File Extraction (Memory vs. Storage)**: The OAT file may also be found in the private data directory of the host application. However, accessing this directory is only possible with root access and would violate the sandbox model. Furthermore, there is no guarantee that the OAT file will remain in this directory post-virtualization, as the host application can easily remove it. Thus, we opted to search memory, as code must be loaded there whenever it needs to be executed.
- **Memory Dumping (Full vs. Process Only)**: We could have obtained the full memory contents of the device using tools like LiME or dumped the contents of the virtualized process only. We chose the latter, as it is more straightforward and efficient. Using LiME involves recompiling the device kernel, which is a complicated and time-consuming process that introduces significant overhead.
- **Process Memory Extraction (dd vs. ptrace)**: To extract specific regions of memory of a process, we could use the dd tool to pull from /proc/[PID]/mem or ptrace to attach to the process and access its memory. In our testing, ptrace was found to be much faster than dd, as dd would only extract memory with a small buffer size.
- **DEX File Analysis (Local vs. Cloud)**: We opted to use the VirusTotal API for our tool, as it is simple to implement and this project is focused on the detection of virtualized applications. However, local scanning offers several advantages, such as independence from internet connectivity, faster results, and the capacity to customize or enhance the scanning process for this specific purpose.

3.9 Conclusion

In this chapter, we discussed the design details of VirtuSleuth. We began with the initialization process, moving on to the scanning of processes, and the extraction and analysis of OAT and DEX files. We provided descriptions of each stage, along with the rationale behind key design choices. Having understood the design of VirtuSleuth, the next chapter will provide the implementation specific details of certain parts of the tool.

4 VirtuSleuth Implementation

In this section, we will go through the implementation specific details of VirtuSleuth for the relevant stages in the detection and extraction process.

4.1 Initialisation

The map of users to processes is created by parsing the output of the "ps" shell command to obtain the UID, PID, and name of every running process on the device. The use of the "ps" shell command is necessary because, with normal privileges, the application can only see its own processes. Thus, the app is set up with root privileges, allowing the application to execute shell commands. The map of users to packages is created by querying the value of the UID of every installed package on the device using the PackageManager. While dumpsys commands can also be used to determine the UID of an installed package, using framework APIs is much faster.

4.2 Process Scan

Instances of packages with the same UID are checked by using the map of users to packages that was created in the initialization stage. The rest of the scenarios can be determined by examining the name of the process in question. Processes of files or libraries are identified because they start with a forward slash and list the file or library's path. Processes of application components are identified because they start with the package name of the application, followed by a semicolon and the name of the component belonging to the process. Shell processes are identified because they are simply named "shell".

4.3 OAT File Extraction

To determine which files are mapped by the process, the contents of /proc/[PID]/maps are retrieved using the "cat" shell command and then parsed using regular expressions. To dump specific regions of process memory, a custom C program was written and compiled using the Android NDK. The C program uses ptrace to attach to the process and dump a portion of its memory to a file in internal

storage. While the C program could be executed with the JNI, it would result in the code executing under normal privileges. Since accessing the memory of another application requires root privileges, the program is included with the assets of the application and is copied to /data/local/tmp to be executed through shell commands when needed.

4.4 DEX File Extraction

File parsing is accomplished using a custom Java class. The class provides the necessary functionality needed across four functions, the first to open the file for parsing, another to skip a specified number of bytes forward, one to read the value of the next four bits as an unsigned integer, and a function that extracts a specified region of the file to internal storage.

4.5 Conclusion

In this section, we discussed the implementation specific details of VirtuSleuth. This information is useful to those interested in replicating the tool or any of the algorithms mentioned in the previous section. Having understood the design and implementation of VirtuSleuth, the next chapter will involve evaluating the efficacy of the tool and exploring its limitations.

5 Evaluation

This section addresses the final two objectives of our study, and can be divided into two main parts: demonstrating the increased stealth provided by app virtualization and assessing the accuracy and practicality of VirtuSleuth.

5.1 Experimentation Setup

All experiments were conducted on a Windows 11 machine and the Android Studio x86 Android 7.0 emulator. For the Android backdoor, we opted for Meterpreter, an extensible payload within the Metasploit Framework penetration testing toolkit. Meterpreter is ideal due to its prevalence in real-world attacks and the importance of its detection for mobile device security. For the analysis of backdoor stealth, we utilised MobSF. MobSF is an automated mobile application security assessment framework capable of performing static and dynamic malware analysis. MobSF is ideal due to its features and widespread recognition in the mobile security community.

5.2 App Virtualization Stealth

Our methodology in evaluating backdoor stealth involves comparing standard backdoors with app virtualization-delivered backdoors in a sandbox environment to analyse IOCs. The MobSF sandbox tracks HTTP requests, Android framework API calls, system logs via logcat, and system state via dumpsys. In the context of Android, logcat is a command-line tool that offers a means to view and debug logs generated by the system and applications running on the device [51], while dumpsys is another command-line tool that enables the dumping of system state and diagnostic information about various system services and components [52]. We predict that the virtualized backdoor will exhibit less IOCs compared to the standard backdoor.

We created two applications to represent both attack vectors. The first application is a plain Meterpreter application, generated using the MSFvenom [53] console and assigned the package name com.metasploit.stage. The second application consists of a single empty activity that virtualizes Meterpreter in the background. This application has a package name of com.example.trustedcontainervirtualapp and emulates com.metasploit.stage.

After executing both applications in the sandbox, we observed the following:

- Network request and API call data were identical in both scenarios. This is expected since the API calls and network requests sent by the plugin application must be proxied by the host application, which is the application being analysed by the sandbox.
- System logs showed the ActivityManager system service logging instances of starting and stopping the main activity of com.metasploit.stage in both scenarios. This outcome is expected because the host application, responsible for managing the lifecycle of the plugin application, must launch and terminate the plugin's main activity.
- The dumpsys sections related to package, activity, and meminfo exhibited traces of com.metasploit.stage when the backdoor was delivered normally but not when virtualized. The absence of backdoor traces in these sections can be attributed to the fact that the virtualized application is not installed within the operating system. As a result, the system components responsible for monitoring installed applications do not track it.

The aforementioned dumpsys sections list information about installed applications, running activities, and memory usage details for each running application. Figure 4.1 shows a sample of the dumpsys package output when the backdoor is installed as a standard application. The absence of virtualized backdoor activities in these sections make it more difficult for investigators to detect the presence of the malicious activity on the device as there are fewer IOCs to detect. This finding supports our hypothesis and establishes the increased stealth provided by app virtualization.

```
...
Package [com.metasploit.stage] (c120184):
  userId=10089
  pkg=Package{390d6d com.metasploit.stage}
  codePath=/data/app/com.metasploit.stage-1
  resourcePath=/data/app/com.metasploit.stage-1
  legacyNativeLibraryDir=/data/app/com.metasploit.stage-1/lib
...
```

Figure 5.1: Backdoor IOC

5.3 VirtuSleuth Evaluation

In evaluating VirtuSleuth, our methodology focused on assessing the two most crucial aspects within the malware detection context: the accuracy in detecting virtualized application processes and the time taken for detection and bytecode extraction. Our setup involved virtualizing ten benign and ten malicious applications, measuring the time taken to detect the virtualized processes and extract their bytecode, as well as the number of engines that identified the bytecode as malicious. Tables 4.1 and Tables 4.2 present the results for the benign and malicious applications respectively.

Table 4.1 Results of Benign Samples

Application	Version	Time Taken (ms)	VirusTotal Score
Chrome	111.0.5563.58	824	1/60
Discord	169.15 - Stable	1875	1/60
Facebook	405.1.0.28.72	833	1/60
Firefox	111.0	763	1/60
Gmail	2023.02.19.515 548686.Release	1263	1/60
Instagram	274.0.0.26.90	1498	1/60
MEGA	7.7	1201	1/60
Messenger	400.0.0.11.9	1399	1/60
Twitter	9.79.0-release.0	1958	1/60
Whatsapp	2.23.5.78	950	1/60

Table 4.2 Results of Malware Samples

Application	Version	Time Taken (ms)	VirusTotal Score
BrazKing	3.0.0	645	8/60
CopyCat	1	727	3/60

Application	Version	Time Taken (ms)	VirusTotal Score
CryptoStealer	1.2.0.3	621	8/61
Ermac	1.0	632	7/60
Meterpreter	1.0	625	21/61
PhotoEditor	1.0.1	839	8/60
PixStealer	1.4	702	16/60
PremiumSMS	1.0	666	15/59
Sova	1.0	770	9/61
TaxPayer	1.0	672	13/60

In all twenty instances, the tool correctly identified the process being virtualized. On average, the time needed to detect the process and obtain the DEX files was 973.15 milliseconds, and never surpassed two seconds. The swift detection and extraction of virtualized application bytecode results are highly promising, underscoring VirtuSleuth's potential for real-time deployment. In terms of malware detection, the average score for benign samples is 1 engine out of 60, likely due to false positives, while malware samples yielded a substantially higher average score of 10.8 engines out of 60. This substantial increase for malware samples is not surprising as we are dealing with the raw application bytecode, which includes the malicious code sequences that can be recognized by malware detection engines. Overall, these results are very positive, and reinforce VirtuSleuth's practicality.

5.4 Limitations

While VirtuSleuth is effective in detecting and analysing virtual applications, it serves as a proof of concept with some limitations. Some of these limitations, such as the lack of automatic scanning, compatibility with only Android 7, and the requirement of internet connectivity for analysis, could be addressed if VirtuSleuth were to be offered as a product or service. However, some limitations are inherent to the process:

- **Root Privileges**: The tool requires root privileges to function. Obtaining root access on a device involves risks and some devices cannot be rooted.
- **Malware Execution**: The virtual application must have already executed before it can be detected. This means that any potential damage caused by the malware may have occurred before the application is detected and stopped.
- **Android Developments**: Android and ART are constantly evolving, which will necessitate ongoing research and maintenance. The parts most likely to require continuous updates are those that involve parsing of OAT files and extraction of DEX files, as these are tied to the specific versions of Android in use.
- **Post-Mortem Intrusion Detection**: VirtuSleuth is not designed to perform post-mortem analysis, such as through disk forensics with Autopsy. Without being able to identify attacks after they have occurred, it becomes difficult to determine extent of a breach retrospectively.

The first limitation comes from the fact that elevated permissions are required to access the memory space of other applications and could be resolved if the application is bundled with the system. However, this can only be done by device vendors and requires firmware customisation. The second limitation is inherent to the fact that the virtualized process must be running for it to be detected. However, potential damages can be mitigated by analysing any system services that the malware has interacted with prior to detection, and implementing countermeasures to undo any potential damages. The third limitation is a characteristic of the ever-evolving nature of technology and also affects other anti-malware applications and solutions. The fourth limitation relates to the fact that VirtuSleuth is not designed to conduct post-mortem analyses. Integrating complementary tools that are proficient in disk forensics or working on enhancing VirtuSleuth's capabilities for a more comprehensive intrusion detection approach could alleviate this issue.

5.5 Conclusion

This chapter was centred around evaluating the stealth of app virtualization and the efficacy of VirtuSleuth. While VirtuSleuth showed promising results, we also acknowledged its limitations. This analysis will help us understand where further development and research can be directed, a topic discussed in the next chapter.

6 Conclusion

In this study, we set forth with four key objectives aimed at exploring the potential of volatile memory forensics in combating stealthy, virtualized Android backdoors. We successfully developed an Android application capable of virtualizing a known backdoor, demonstrating the potential for malicious exploitation and addressing our first objective. We then achieved our second objective by creating VirtuSleuth, a defensive tool designed to counteract app virtualization stealth. The development of VirtuSleuth presented unique challenges in extracting virtualized process bytecode from volatile memory. We managed to overcome these challenges by utilizing native code for memory dumping and implementing a custom parsing algorithm for the extraction of DEX files.

Our third objective involved demonstrating the increased stealth of app virtualization by comparing the IOCs generated by a backdoor when executed as a standard Android application versus when virtualized. Our findings showed a decrease in IOCs related to dumpsys logs, confirming app virtualization's increased stealth. Finally, we evaluated VirtuSleuth's feasibility and practicality by measuring the detection accuracy and the time taken for detection and extraction of bytecode, successfully fulfilling our fourth objective. VirtuSleuth not only detected every virtualized application, but never exceeded two seconds for the detection and subsequent extraction. Overall, the results were very positive, and show the tool's potential for real-time deployment.

These positive outcomes highlight the potential benefits of incorporating volatile memory forensics in Android security frameworks, and our results serve as a stepping stone towards a future where malware detection engines can effectively counteract app virtualization stealth techniques.

6.1 Future Work

While the original objectives have been achieved, there is always room for further development and improvement. The scope of the malware detection capabilities can be broadened by incorporating more advanced detection techniques. For instance, machine learning algorithms could be applied at the bytecode analysis stage to enable

better pattern recognition of malicious activities and improve the classification accuracy of a wider variety of malware.

Additionally, cross-platform compatibility could be explored. While our current focus has been on Android, app virtualization and related security issues are not limited to this platform. Expanding our tool's detection and analysis capabilities to other platforms such as Windows would significantly increase its applicability and impact. However, such an expansion would involve overcoming platform-specific challenges such as different file systems, binary formats, and system APIs.

Finally, as app virtualization techniques continue to evolve, it will be crucial to stay ahead of these developments and adapt the tool accordingly. This might involve updating the detection algorithms to account for new virtualization methods, enhancing the DEX file extraction mechanism to support new Android versions, and exploring new strategies for identifying and mitigating the risks associated with app virtualization.

References

[1] A. Turner, “How Many Apps In Google Play Store? (Apr 2023),” Feb. 17, 2022. https://www.bankmycell.com/blog/number-of-google-play-store-apps/

[2] “36 Malicious Android Apps Found on Google Play, Did You Install Them?,” PCMAG. https://www.pcmag.com/news/36-malicious-android-apps-found-on-google-play-did-you-install-them

[3] “Check Point Research: Third quarter of 2022 reveals increase in cyberattacks and unexpected developments in global trends,” Check Point Software, Oct. 26, 2022. https://blog.checkpoint.com/2022/10/26/third-quarter-of-2022-reveals-increase-in-cyberattacks/

[4] “Comparison of antivirus software - Wikipedia,” en.wikipedia.org. https://en.wikipedia.org/wiki/Comparison_of_antivirus_software

[5] “Play Protect,” Google Developers. https://developers.google.com/android/play-protect

[6] T. Burt, “Microsoft report shows increasing sophistication of cyber threats,” Microsoft on the Issues, Sep. 29, 2020. https://blogs.microsoft.com/on-the-issues/2020/09/29/microsoft-digital-defense-report-cyber-threats/

[7] “Device protection in Windows Security,” support.microsoft.com. https://support.microsoft.com/en-us/windows/device-protection-in-windows-security-afa11526-de57-b1c5-599f-3a4c6a61c5e2

[8] “What is Application Virtualization? | VMware Glossary,” VMware, Jan. 20, 2022. https://www.vmware.com/topics/glossary/content/application-virtualization.html

[9] “Fake WhatsApp app downloaded more than one million times,” BBC News, Nov. 06, 2017. Accessed: Dec. 11, 2022. [Online]. Available: https://www.bbc.com/news/technology-41886157

[10] “Kernel overview,” Android Open Source Project. https://source.android.com/docs/core/architecture/kernel

[11] “Application Sandbox,” Android Open Source Project. https://source.android.com/docs/security/app-sandbox

[12] “fork(2) - Linux manual page,” man7.org. https://man7.org/linux/man-pages/man2/fork.2.html

[13] M. Ivanisevic, “What the Zygote!?,” Medium, Mar. 21, 2018. https://medium.com/@voodoomio/what-the-zygote-76f852d887d9

[14] “Memory mapping — The Linux Kernel documentation,” linux-kernel-labs.github.io. https://linux-kernel-labs.github.io/refs/heads/master/labs/memory_mapping.html

[15] "Understanding the Linux /proc/id/maps File," baeldung.com. https://www.nmapping — The Linux Kernel documentation,” linux-kernel-

labs.github.io. https://linux-kernel-labs.github.io/refs/heads/master/labs/memory_mapping.html

[16] “Platform Architecture,” Android Developers. https://developer.android.com/guide/platform

[17] “Application Fundamentals | Android Developers,” Android Developers, 2019. https://developer.android.com/guide/components/fundamentals

[18] “apk (file format),” Wikipedia, Apr. 08, 2023. https://en.wikipedia.org/wiki/Apk_(file_format)

[19] Wikipedia Contributors, “Java virtual machine,” Wikipedia, Apr. 24, 2019. https://en.wikipedia.org/wiki/Java_virtual_machine

[20] “Dalvik (software),” Wikipedia, Feb. 23, 2023. https://en.wikipedia.org/wiki/Dalvik_(software) (accessed Apr. 13, 2023).

[21] “Android - dx,” www.linuxtopia.org. https://www.linuxtopia.org/online_books/android/devguide/guide/developing/tools/android_othertools_dx.html (accessed May 12, 2023).

[22] “Android Runtime (ART) and Dalvik,” Android Open Source Project. https://source.android.com/docs/core/runtime

[23] "oat_79.pdf," romainthomas.fr. https://romainthomas.fr/oat/oat_79.pdf (accessed Apr. 13, 2023).

[24] “Home,” LIEF, Jul. 18, 2021. https://lief-project.github.io/

[25] Lody, “VA产品说明&开发指导,” GitHub, Apr. 13, 2023. https://github.com/asLody/VirtualApp/blob/master/README_eng.md

[26] “Droid Plugin,” GitHub, Dec. 09, 2022. https://github.com/DroidPluginTeam/DroidPlugin

[27] C. C. Editor, “backdoor - Glossary | CSRC,” csrc.nist.gov. https://csrc.nist.gov/glossary/term/backdoor

[28] “DexClassLoader,” Android Developers. https://developer.android.com/reference/dalvik/system/DexClassLoader (accessed May 12, 2023).

[29] “Using Java Reflection,” Oracle.com, 2018. https://www.oracle.com/technical-resources/articles/java/javareflection.html

[30] L. Zhang et al., “App in the Middle,” Proceedings of the ACM on Measurement and Analysis of Computing Systems, vol. 3, no. 1, pp. 1–24, Mar. 2019, doi: https://doi.org/10.1145/3322205.3311088.

[31] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio, “Detecting Android malware using sequences of system calls,” Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile, Aug. 2015, doi: https://doi.org/10.1145/2804345.2804349.

[32] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, “DroidMat: Android Malware Detection through Manifest and API Calls Tracing,” IEEE Xplore, Aug. 01, 2012. https://ieeexplore.ieee.org/abstract/document/6298136/

[33] V. Rastogi, Y. Chen, and X. Jiang, “Catch Me If You Can: Evaluating Android Anti-Malware Against Transformation Attacks,” IEEE Transactions on Information Forensics and Security, vol. 9, no. 1, pp. 99–108, Jan. 2014, doi: https://doi.org/10.1109/tifs.2013.2290431.

[34] G. Meng et al., “Mystique: Evolving Android Malware for Auditing Anti-Malware Tools,” doi: https://doi.org/10.1145/2897845.2897856.

[35] L. Richter, “Common Weaknesses of Android Malware Analysis Frameworks.” Accessed: Apr. 07, 2023. [Online]. Available: https://ayeks.de/images/blog/2015-06-16-android-analysis-frameworks/analysis_frameworks_paper.pdf

[36] A. Ruggia, E. Losiouk, L. Verderame, M. Conti, and A. Merlo, “Repack Me If You Can: An Anti-Repackaging Solution Based on Android Virtualization,” Annual Computer Security Applications Conference, Dec. 2021, doi: 10.1145/3485832.3488021.

[37] Y. Wu, J. Huang, B. Liang, and W. Shi, “Do not jail my app: Detecting the Android plugin environments by time lag contradiction,” Journal of Computer Security, vol. 28, no. 2, pp. 269–293, Mar. 2020, doi: https://doi.org/10.3233/jcs-191325.

[38] T. Luo, C. Zheng, Z. Xu, and X. Ouyang, “ANTI-PLUGIN: DON’T LET YOUR APP PLAY AS AN ANDROID PLUGIN.” Available: https://paper.bobylive.com/Meeting_Papers/BlackHat/Asia-2017/asia-17-Luo-Anti-Plugin-Don%27t-Let-Your-App-Play-As-An-Android-Plugin-wp.pdf

[39] L. Shi, J. Fu, Z. Guo, and J. Ming, “‘Jekyll and Hyde’ is Risky,” Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services, Jun. 2019, doi: https://doi.org/10.1145/3307334.3326072.

[40] M. Alecci, R. Cestaro, M. Conti, K. Kanishka, and E. Losiouk, “Mascara: A Novel Attack Leveraging Android Virtualization,” arXiv:2010.10639 [cs], Oct. 2020, Available: https://arxiv.org/abs/2010.10639

[41] A. Ruggia, E. Losiouk, L. Verderame, M. Conti, and A. Merlo, “Repack Me If You Can: An Anti-Repackaging Solution Based on Android Virtualization,” Annual Computer Security Applications Conference, Dec. 2021, doi: 10.1145/3485832.3488021.

[42] Y. Chen et al., “InstaGuard: Instantly Deployable Hot-patches for Vulnerable System Programs on Android,” Proceedings 2018 Network and Distributed System Security Symposium, 2018, doi: https://doi.org/10.14722/ndss.2018.23124.

[43] Simeone Pizzi, “VirtualPatch: fixing Android security vulnerabilities with app-level virtualization,” thesis.unipd.it, 2022, Available: https://thesis.unipd.it/handle/20.500.12608/32823

[44] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. Von Styp-Rekowsky, “Open access to the Proceedings of the 24th USENIX Security Symposium is sponsored by USENIX Boxify: Full-fledged App Sandboxing for Stock Android Boxify: Full-fledged App Sandboxing for Stock Android.” Available:

https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-backes.pdf

[45] J. Sylve, "Lime-Linux memory extractor", in Proc. 7th ShmooCon Conf., 2012.

[46] S. Song, B. Kim, and S. Lee, “The Effective Ransomware Prevention Technique Using Process Monitoring on Android Platform,” Mobile Information Systems, vol. 2016, pp. 1–9, 2016, doi: https://doi.org/10.1155/2016/2946735.

[47] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, “GUITAR,” Computer and Communications Security, Oct. 2015, doi: https://doi.org/10.1145/2810103.2813650.

[48] A. Ali-Gombe, S. Sudhakaran, A. Case, and G. Iii, “DroidScraper: A Tool for Android In-Memory Object Recovery and Reconstruction.” Accessed: Apr. 21, 2023. [Online]. Available: https://www.usenix.org/system/files/raid2019-ali-gombe.pdf

[49] J. Bellizzi, M. Vella, C. Colombo, and J. Hernandez-Castro, “Responding to Targeted Stealthy Attacks on Android Using Timely-Captured Memory Dumps,” IEEE Access, vol. 10, pp. 35172–35218, 2022, doi: https://doi.org/10.1109/ACCESS.2022.3160531.

[50] J. Bellizzi, M. Vella, C. Colombo, and J. C. Hernandez-Castro, “Real-Time Triggering of Android Memory Dumps for Stealthy Attack Investigation,” Lecture Notes in Computer Science, pp. 20–36, Nov. 2020, doi: https://doi.org/10.1007/978-3-030-70852-8_2.

[51] “Logcat command-line tool | Android Studio,” Android Developers. https://developer.android.com/tools/logcat (accessed Apr. 13, 2023).

[52] “dumpsys | Android Studio,” Android Developers. https://developer.android.com/tools/dumpsys (accessed Apr. 13, 2023).

[53] “MSFvenom - Metasploit Unleashed,” OffSec. https://www.offsec.com/metasploit-unleashed/msfvenom/

[54] “Android malware ‘Triada’ has evolved to be incorporated into pre-shipped devices according to Google measures,” GIGAZINE, Jun. 07, 2019. https://gigazine.net/gsc_news/en/20190607-triada-found-phones-before-shipped/ (accessed May 19, 2023).

[55] “Hyperdetect,” Bitdefender. https://www.bitdefender.com/business/gravityzone-platform/hyperdetect.html (accessed May 19, 2023).

[56] “Understanding Android Malware Families: Adware and Backdoor (Article 5) - IT World Canada,” www.itworldcanada.com, Jun. 01, 2021. https://www.itworldcanada.com/blog/understanding-android-malware-families-part-5-adware-backdoor/447798

[57] “Configuring ART,” Android Open Source Project. https://source.android.com/docs/core/runtime/configure