Mobile Malware: Android Malware Analysis and Framework Development for ARM Analysis Environments

CMP403: Honours Dissertation

Mykola (Nick) Nesterenko

BSc (Hons) Ethical Hacking, 2022/2023

Abstract

This research project presents the development of a novel Framework for Mobile Malware Analysis, designed as a pre-configured Virtual Machine (VM) to run on ARM-based hardware. Alongside the developed framework, a structured methodology, known as the Unified Mobile Malware Analysis (UMMA), has been established. The UMMA methodology encapsulates several integral steps to facilitate a comprehensive examination of mobile malware, including sample acquisition, pre-static analysis, static analysis, automated analysis, and dynamic analysis.

The developed VM framework integrates a suite of carefully chosen tools, including the Mobile Security Framework (MobSF), AndroGuard, AndroPyTool, Apktool, and others. Each tool contributes unique capabilities, creating a robust system for Android malware examination when used in conjunction with the UMMA methodology.

This project has been partially successful. While it primarily focuses on Android malware, it offers a systematic approach to mobile malware analysis and anticipates the enforcement of the EU Digital Markets Act in 2024, which will mandate side-loading capabilities for iOS. For future work, the adaptation of the methodology to accommodate iOS malware analysis is suggested.

To foster a knowledge-sharing community, the developed framework and the UMMA methodology have been made accessible to the public via a dedicated website. Continual public access will allow for ongoing feedback and iterative improvements, ensuring the methodology and the framework remain relevant in the ever-evolving landscape of mobile malware. The development of this structured methodology and dedicated framework represents a significant contribution to the field of mobile malware analysis.

Contents

1 INTRODUCTION

1.1 BACKGROUND

1.1.1 The Significance of Mobile Devices: Android

In today's digital landscape, smartphones have emerged as the most trusted computing devices for the majority of individuals. These modern devices store vast amounts of sensitive user information, including location history recorded by navigation apps, confidential personal data like photos, which may contain images of legal identification documents and proprietary business information protected by non-disclosure agreements. Moreover, smartphones also grant users access to multi-factor authentication applications and the ability to store passwords for various accounts including mobile banking and much more. In fact, it is easier to mention information that users don't store on their mobile devices than to list everything. This vast pool of information, consolidated within a single device, presents an enticing attack vector for threat actors.

Among mobile devices, some have garnered more popularity than others. According to StatCounter, nearly 70% of the sold smartphones ship with Android Operating System (OS) as of April 2023 (StatCounter, 2023). Contrary to its primary competitor (Apple's iOS), Android tends to be more prone to malware infections due to the decentralised nature of application distribution through a variety of marketplaces like Google Play and Samsung's Galaxy and most importantly, side-loading. Side-loading is the ability to install applications directly using Android package (.apk) files, bypassing all security precautions provided by the application marketplace vendors.

1.1.2 Why Analysing Mobile Malware is Important?

Similarly to any other platform, the complexity and sophistication of mobile malware are constantly increasing, posing significant challenges to security researchers and practitioners. As the situation of Android's security issues extends beyond the ability to sideload software from unauthorised application marketplaces. Google's application store, Google Play, employs a default service named Google Play Protect, which is designed to "*run a safety check on apps from the Google Play Store before you download them*" (Google, n.d.). Despite its built-in malware filtering, Google Play Protect has proven to be unreliable over the years. A study by AV-Test concluded that the detection rate of Play Protect was only 33.1%, compared to dedicated AV solutions which on average have >97% detection rates (AV-Test, 2020). This would mean that there could be more malware distributed in Google Play alone as not every user utilises AVs, leading to a strong need for a more in-depth analysis of modern mobile malware.

In addition to the facts and vulnerabilities mentioned earlier, mobile software has a distinct set of design principles which differ from the desktop counterparts. For example, Both Android and iOS employ techniques like the definition of *permissions* which are used to specify certain capabilities of an application. Moreover, despite the differences from the typical desktop application fundamentals, Android (and iOS) lacks a structured methodology for Mobile Malware Analysis, proposing a tailored methodology could assist security researchers improve the analysis process and spread awareness of threats posed by mobile malware.

1.1.3 The European Digital Markets Act

The need for a structured methodology for mobile malware analysis would also be beneficial for combating the future threats proposed by the upcoming European laws and regulations. The Digital Markets Act (DMA) was proposed to ensure a fair and open digital markets (European Commission, 2022), to limit the monopolistic approach to software distribution, regardless of the platform. Some of the notable segments of the DMA include:

1. Enforce side-loading on restricted platforms (like iOS).
2. Enforce the ability to install and promote alternative application marketplaces.
3. Enforce the ability to uninstall any pre-installed software or app.

This means that after the act would be introduced completely by 2024, the distribution of unauthorised software will become less complex. While this regulation would be highly beneficial for both businesses and customers by providing a fair market, side-effects like a potential threat actor popularising an infected application distribution platform could pose a significant risk and bring malware to security caesious iOS as well. A structured unified mobile malware analysis methodology would help prepare for the incoming threats and aid security research and detection mechanisms development by providing an efficient approach to understanding mobile malware.

1.1.4 Benefits of ARM Technology

Finally, to provide an even better environment for the comprehension and investigation of mobile malware analysis, the development of a Framework would be beneficial. A framework in a form of a pre-configured Virtual Machine (VM) would provide an opportunity for novice security researchers to explore threats developed for mobile platforms.

When developing a framework, future proving could also be advantageous. Advanced Reduced Instruction Set Machines (ARM) based computing units are becoming significantly more capable and widespread nowadays. Not only ARM is used by the dominant majority of mobile devices, but personal computers and cloud computing platforms are also starting to leverage such technology. One of the main advantages of ARM is the reduced production cost of hardware as well as significantly improved power efficiency. For example, AWS promises "*up to 60% less energy consumption*" when using ARM-based *Graviton* EC2 Instances compared to traditional x86 EC2 instances while achieving the same performance (AWS, n.d.).

With the obvious advantages of ARM, there is one significant disadvantage which is the lack of compatibility which will require additional steps to run old tools. For the purposes of this project, MacBook Air with M2 ARM CPU was selected due to the current popularity of Apple's Silicon as an interpretation of ARM-based desktop computers.

1.2 RESEARCH QUESTIONS

This project set out to investigate and discuss the following research questions:

- **Is it possible to develop a Framework for Mobile Malware Analysis in a form of a pre-configured Virtual Machine designed to run on ARM-based hardware?**
  I. To accompany the developed Framework, is it possible to structure Mobile Malware Analysis into a standardised methodology?
  II. Could existing tools for manual Mobile Malware Analysis be automated to speed up and simplified for novice security researchers?

1.3 AIMS AND OBJECTIVES

This project aims to develop a pre-configured mobile malware analysis framework in the form of a virtual machine, which will then be shared for public use. This framework aims to popularise mobile malware analysis among security researchers and students. A pre-configured VM would provide a ready-to-use and secure environment for malware analysis, thereby simplifying and streamlining the analysis process. The VM should be optimised to operate on ARM-based Apple Silicon computers.

To accompany the framework, it is planned to propose a unified methodology for mobile malware analysis. This methodology will be formulated by examining existing malware analysis methodologies and attributes of mobile applications that could be of high value for understanding the logic behind the analysed sample. By combining a structured methodology with a VM containing the essential set of required tools for malware analysis, the entire malware analysis process could be further improved.

Finally, this project also aims to automate parts of the analysis process using scripting techniques in combination with a selected toolset.

2 Literature Review

2.1 Android Malware: Statistics

Recently, one of the leading Anti-Virus (AV) software providers, *Kaspersky*, published an article on *Securelist.com* (Kaspersky's information security website) titled "The mobile malware threat landscape in 2022" (Shikova, 2023). This article discusses recent statistics and trends in detail, providing substantial information about today's Android malware situation, which corroborates the assertions made in Section 1.1.1.

To begin with, the article provided insight into the detected malware installers statistics for the 2019-2022 period, which outlined a decline in the number of detected malware installers. As of May 2023, Kaspersky does not provide full anti-virus and detection capabilities on iOS, therefore the mentioned detection rate referred to Android OS specifically, (Figure 1):

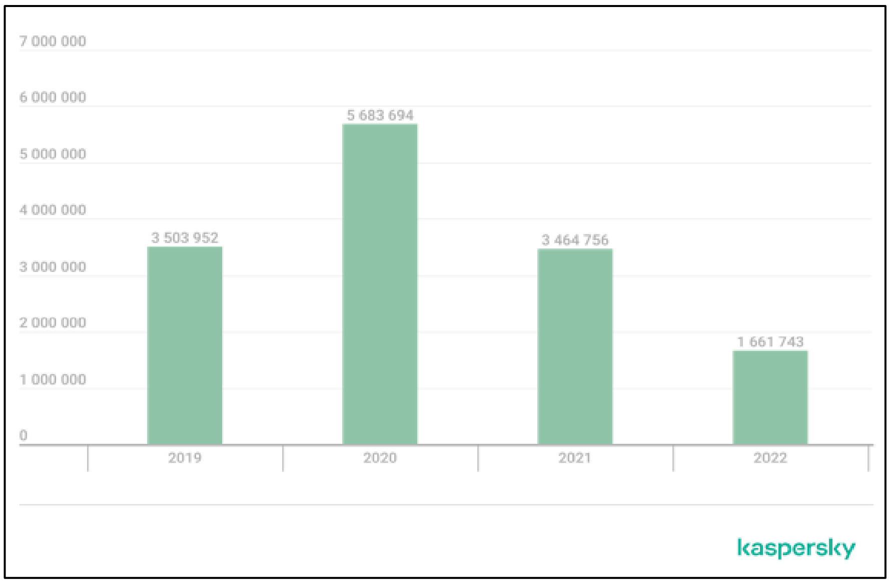


Figure 1: Number of detected malicious installation packages in 2019–2022 (Shikova, 2023)

The next notable statistic was related to the number of detected banking Trojan-class malicious software. Unlike the total number of the detected installers, the number of mobile banking Trojans detections rose significantly, precisely, a "year-on-year increase of 100% and the highest figure in the past six years", (Figure 2):

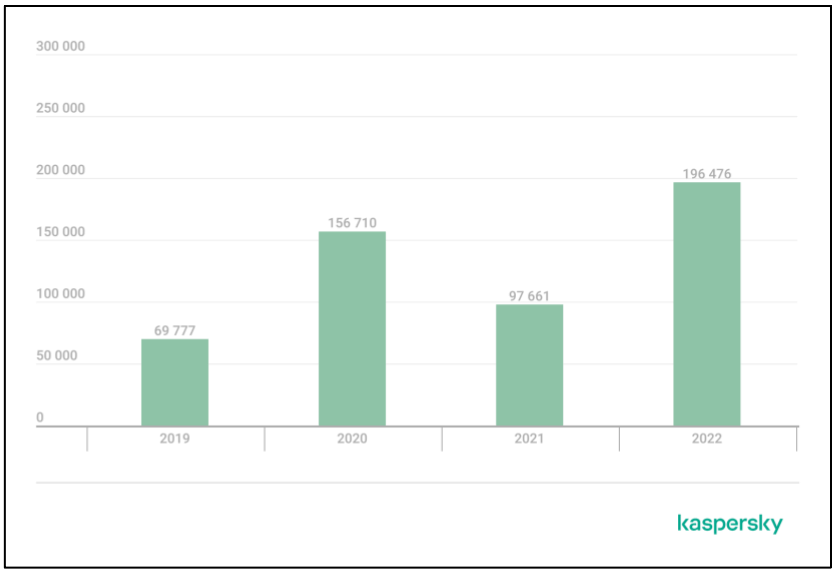


Figure 2: The number of mobile banking Trojan installers detected by Kaspersky in 2019–2022 (Shikova, 2023)

This statistical data could suggest that potential threat actors are aware of the increasing popularity of mobile banking applications and are thus focusing their efforts on exploiting these platforms. This shift in focus might be due to the potentially high returns from successful attacks on financial institutions or individuals' banking details, further underscoring the need for robust security measures in Android systems and improving the understanding of the logic behind mobile malware.

2.2 STANDARDISED MALWARE ANALYSIS METHODOLOGIES

As was mentioned in Chapter 1, the analysis of malware, specifically targeting Android devices, has become increasingly important in recent years. The significance of gathering understanding regarding mobile device threat analysis and detection, in general, was also elevated since the rate of adoption of Mobile Banking was increasing over the past decade, according to *EnterpriceAppToday*, in 2022 "**nearly 89%** of bank account holders in the US use mobile online banking for managing their accounts", (Pawar, 2022). Considering that people are storing gigabytes of personal information on their mobile phones alongside mobile banking applications, the combination of factors poses a major risk if one's smartphone is compromised with a malicious piece of software.

To optimise the process of malware analysis, a variety of methodologies and frameworks have been proposed to guide researchers in their efforts to systematically analyse and mitigate the risks posed by malware. These methodologies aim to provide a structured approach to understanding the behaviour, propagation, and impact of malware, thus facilitating the development of effective countermeasures. In this context, two notable and widely adopted methodologies for malware analysis are the Malware Analysis Reverse Engineering (MARE) (Nguyen & Goldman, 2010) and the Systematic Analysis of Malware Artifacts (SAMA) (Bermejo Higuera, et al., 2020).

The MARE and SAMA methodologies, while sharing a common goal of providing a structured approach to malware analysis, differ in their specific components. Understanding these methodologies would assist the development of a tailored approach that addresses the unique challenges posed by Android malware analysis.

2.2.1 Malware Analysis Reverse Engineering (MARE) Methodology

Up until the beginning of the second decade of the 21st century, there was no standardised approach to malware analysis. In 2010 the first attempt to create a structured Malware Defence (MD) timeline was developed, the segment of which was the Malware Analysis Reverse Engineering (MARE) methodology. MD timeline consists of the following stages where stages 2-5 represented the MARE methodology which were underlined (Nguyen & Goldman, 2010):

1. Malware Infection
2. **Detection.**
3. **Isolation and Extraction.**
4. **Behavioural Analysis.**
5. **Code Analysis and Reverse Engineering.**
6. Pattern Recognition.
7. Malware Inoculation and Remedy.

The first stage of the MARE methodology – *Detection*, involved scanning the malware sample using Antivirus (AV) software as well as using online based detection tools such as *VirusTotal* to understand if the malware was known or unknown (zero-day), based on the cryptographic hash, usually using MD5 or SHA256 algorithms (Nguyen & Goldman, 2010). However, in recent years, this process began to show its limitations with advanced threats which can dynamically change the structure of the malware (Kiachidis & Baltatzis, 2021). Such malware samples are classified as polymorphic malware.

Next, the *Isolation and Extraction* phase aimed to securely isolate the malware to prevent further propagation, extract the malware, and transfer it to a protected environment such as a virtual machine (VM) which was set up for the behaviour analysis.

The *Behavioural Analysis* phase involved the observation of changes and actions the malware performed on the infected system for its malicious intent. An analyst would consider file manipulation, registry tempering, library modification, and initiated network connections while taking snapshots before and after malware execution for further documentation and recreation of findings. MARE's behaviour analysis could potentially be referred to as ***Dynamic Analysis*** which is a more common term used throughout the malware analysis community. However, MARE's behaviour analysis does not include the use of debuggers, which limits the information gathered from the code logic itself.

Finally, the *Code Analysis & Reverse Engineering* phase relied mainly on the understanding of the assembly language behind the malware (the process is often referred to as ***Static Analysis***). This stage leveraged debugging, disassembling and decompilation techniques to understand the logic behind the malicious code as well as unveiling hidden details or segments of code which were not executed during the behaviour analysis phase. Code analysis and reverse engineering stage is closely tied with the behaviour analysis as this could potentially speed up the process of understanding the malware code structure (Nguyen & Goldman, 2010). When code and behaviour analyses are combined, this process is usually referred to as ***Hybrid Analysis***.

2.2.2 Systematic Approach to Malware Analysis (SAMA) Methodology

After 10 years after the release of MARE, the second attempt to create a standardised structured methodology of malware analysis was introduced named *Systematic Approach to Malware Analysis (SAMA)*, in response to the increasing complexity of attack vectors and threat actors' evolving malware design techniques and propagation tactics. SAMA's main aim was to standardise the procedure and provide a framework for analysing modern malware (Bermejo Higuera, et al., 2020). This methodology retained the four stages from MARE however, there were observable name-related and structural differences (Table 1):

Table 1: Differences between MARE and SAMA Methodologies (Bermejo Higuera, et al., 2020)

MARE	SAMA
Detection	Initial Actions
Isolation and Extraction	Classification
Behavioural Analysis	Static and Dynamic Code Analysis
Code Analysis and Reverse Engineering	Behavioural Analysis

The Initial Actions phase of SAMA methodology focused on preparing the analysis environment, including documenting the state and form of the systems and creating secure snapshots of virtual or physical systems to allow reversion to a clean state as well as conducting a comparison of created/altered files before and after malware execution. This preliminary phase was considered mandatory for the analysis procedure (Bermejo Higuera, et al., 2020).

In the Classification phase, it was proposed by SAMA methodology to undertake basic static analysis steps to determine if a further analysis was necessary. At this stage the code analysis itself would not be examined, the main goal of this stage was to determine if the sample is benign or malicious. This would then assist in the decision-making process of whether to continue the manual analysis of the sample or not. The steps of this phase also included transferring the sample to the analysis environment, creating hashes (also known as malware sample *Fingerprinting*) for identification, gathering information from open-source intelligence (OSINT) sources such as previously mentioned *VirusTotal*, analysing the strings contained inside of the malware sample, and assessing the presence of obfuscation techniques.

The Code Analysis phase involved advanced static and dynamic analysis techniques, such as disassembly and debugging, to examine the sample's code. This is one of the most significant and complex stages of the SAMA methodology. Code Analysis and Reverse Engineering stage leverages techniques such as disassembly and debugging. Since debugging is a dynamic process, the testing environment would be reverted to the initial state at the end of Code Analysis phase before moving to Behavioural analysis.

Finally, the Behavioural Analysis phase employed dynamic and memory analysis techniques. The sample would be executed in a safe environment, and system changes and logs would be recorded and analysed. Memory dumps are also collected for further investigation to identify the rest of the information which would be beneficial to evaluate the persistence mechanisms and other information about the sample's logic and motivation (Kiachidis & Baltatzis, 2021).

2.2.3 Another Methodology-like Example: Android Malware Analysis Workflow

The book titled *Mastering Malware Analysis* details an entire chapter about the malware samples developed for Android OS (Kleymenov & Thabet, 2022). Although the book does not provide a structured methodology, an example of Android malware analysis workflow was described which would be highly valuable for putting together a standardised methodology for Android malware analysis. The book's example details the following steps; however, it should be noted that the book states that "each case is different":

1. Sample Acquisition.

2. Decompilation/Disassembly.
3. Reviewing the App Manifest.
4. Code Analysis.
5. Deobfuscation and Decryption.
6. Behavioural Analysis.
7. Debugging.

In the given example of an Android sample analysis workflow, the process would begin with sample acquisition, which involved obtaining samples from various sources, such as average users, third-party websites, or *Google Play* using tools like *APK Downloader*. Decompilation and disassembling of apps were performed to facilitate code analysis, and in cases where anti-reverse engineering techniques were employed, the code was disassembled to amend tampering logic.

The review of the *App Manifest* was to gain insight into the sample's functionality, such as *permissions* requested, available *components*, the main activity, and the *Application subclass*. It was suggested to use IDEs or other tools with convenient user interfaces for Code analysis, similar to steps from MARE and SAMA. Deobfuscation and decryption would be undertaken if obfuscation was detected, utilising existing deobfuscators or generic method renaming tools which were also detailed in *Mastering Malware Analysis* (Kleymenov & Thabet, 2022).

The behavioural analysis would be conducted by executing the sample in an emulator with behavioural analysis tools enabled, while emulator detection techniques could be identified and excluded from the sample, again the commonalities with both MARE and SAMA can be observed. Debugging should be employed to understand complex functionality, particularly when malware interacted heavily with the operating system (Kleymenov & Thabet, 2022). Emulators supporting snapshot creation were recommended for the efficient reproduction of situations during the analysis process.

2.3 Currently Available Tools (Q2, 2023)

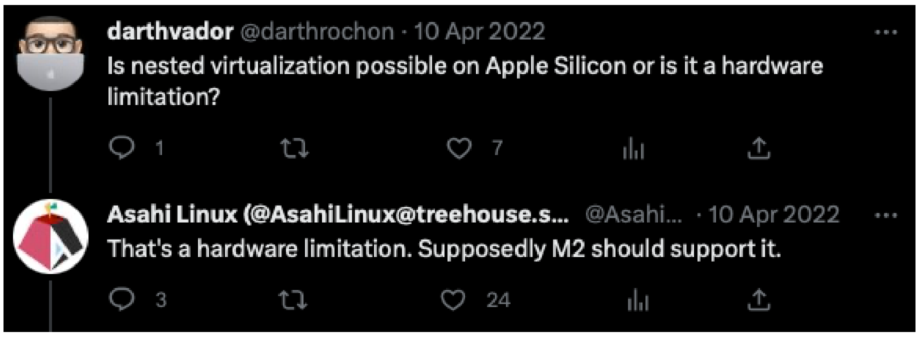
In the realm of malware analysis, the employment of various tools played a crucial role in facilitating the process of understanding, detecting, and mitigating threats posed by malicious software. These tools were designed to address specific aspects of the analysis process, including virtualisation, static analysis, and dynamic analysis techniques. As was mentioned in Chapter 1, in order to develop a framework, it was decided to combine a list of tools necessary for basic Android malware analysis in a virtual machine (VM). This section talks about the known tools that are widely used for the purposes of virtualisation on Apple Silicon Macs and the tools required to analyse Android-specific malicious applications.

2.3.1 Virtualisation on Apple Silicon

Prior to the end of 2020 when the first laptop with the Apple Silicon M1 CPU which used ARM instruction set, Apple computers used to be running Intel CPUs which used the x64 instruction set. The introduction of ARM brought both advantages like improved power efficiency, however, one major drawback was observed in a form of limited compatibility with some functionality that x64 based processors used to have.

The first encountered problem was virtualisation. As of April 2023, the nested virtualisation on Apple Silicon Mac is not available, which would be highly beneficial for use in this project as having an

Android emulator running inside of an isolated Virtual Machine would insure best malware analysis practices. There was no direct answer found as of why this was the case, some community members suggest this being a hardware limitation and others believe that it is the software problem, meaning that one could potentially implement such features. For example, the official Twitter account of Asahi Linux, a project that currently attempts to reverse engineer the drivers made by Apple for the new Apple Silicon Macs, suggested the following (Figure 3):


darthvador @darthrochon · 10 Apr 2022
Is nested virtualization possible on Apple Silicon or is it a hardware limitation?
1 7

Asahi Linux (@AsahiLinux@treehouse.s... @Asahi... · 10 Apr 2022
That's a hardware limitation. Supposedly M2 should support it.
3 24

Figure 3: Asahi Linux developer Tweet regarding nested virtualisation on Apple Silicon (Asahi Linux , 2020)

During the era of Intel Macs, virtualisation was not a problem, Apple even provided a specific tool which allowed to install other operating systems on their hardware called ***Boot Camp***. "With Boot Camp, you can install Microsoft Windows 10 on your Mac, then switch between macOS and Windows when restarting your Mac" (Apple, 2020). This feature is not available for the latest Apple hardware due to the use of an ARM instruction set and proprietary drivers.

The virtualisation in the early days of Apple Silicon lifecycle was challenging in general, even large hypervisor software providers such as VMware released their version of VMware Fusion optimised to work with Apple Silicon processors only after nearly two years after the initial release of the Apple M1 MacBook Air, the first Apple device to ship with the proprietary chipset. On the other hand, an updated version of the hypervisor called ***Parallels*** was revealed in April 2021 which supported the new Apple CPUs, which was one of the first hypervisors to achieve that (Dobrovolskiy, 2021). In fact, Parallels hypervisor was demoed on Apple's Worldwide Developer Conference 2020. The Parallels hypervisor and was also reputable prior to the release of Apple Silicon, a researcher that compared the available hypervisors for Macs that featured Intel CPUs and came to the conclusion "that Parallels Desktop performs the best followed by VMware Fusion and the least performing Hypervisor is Oracle VirtualBox" (Pandey, 2020).

There were other attempts to make VM hypervisors for Apple Silicon such as ***UTM*** which is based on QEMU and is open source. However, despite being free to use, UTM would not be the ideal choice due to the instabilities related to hardware acceleration compared to Parallels.

2.3.2 Android Emulation on Apple Silicon

As was mentioned in Section 2.2.1, the nested virtualisation was not available during the development of this project, therefore the next best option was chosen for emulation of Android Devices as a secondary virtualised device.

The official IDE for Android development from Google, Android Studio was optimised to work with Apple Silicon. The built-in emulator was also optimised and allowed the desired "snapshot" feature for malware analysis.

2.3.3 Malware Analysis Tooling: Android

One of the most popular static analysis tools for Android applications is ***JADX***, a free and open-source decompiler that translates Android APK files and DEX bytecode back into human-readable Java source code similar to *IDA Pro* or *Ghidra* decompilers. With JADX, analysts can effectively inspect the code logic of Android applications to identify any malicious intent, obfuscation techniques, or potential vulnerabilities. Its user-friendly graphical user interface and command-line options make JADX a versatile tool for various analysis scenarios while still being accurate. According to research, JADX performed decompilation with a weighted average of 0.02% failure rate, compared to other decompilers with failure rates of around 1% (Mauthe, et al., 2021).

Another known static analysis tool in the Android ecosystem is ***APKTool***, which facilitates the reverse engineering of Android application packages (APKs) by decoding resources and disassembling the application's bytecode (Batyuk, et al., 2011). APKTool is a command-line tool that can extract resources from an Android APK file, decompile the APK code into Smali bytecode, and decode the manifest file into a human-readable XML format. It allows technical users to analyse the code structure, resources, and permissions used by an Android application and can be used for tasks such as debugging, modification, or recompiling an APK file. According to the *Mastering Malware Analysis book,* "*Smali (assembler in Icelandic) is the name of the assembler tool that can be used to compile Dalvik instructions to the bytecode and, in this way, build full-fledged DEX files*." (Kleymenov & Thabet, 2022).

There are many other tools which can be used for Android malware analysis, the creation of the basic toolset required for assembling a framework were be explored and discussed in the following chapters.

3 METHODOLOGY

3.1 OVERVIEW

This chapter is dedicated to the design and refinement of a proposed methodology for mobile malware analysis. It sets out with a draft version of this methodology, taking into account the existing MARE and SAMA methodologies as well as the Android malware analysis workflow presented in the preceding Chapter 2. The methodology is subsequently sharpened throughout this chapter, in parallel of the exploration of the practical aspects of setting up an analysis environment and exploring tools for malware analysis.

A key part of this chapter is the establishment of the mobile malware framework and analysis environment in general. Due to the limitations regarding nested virtualisation on Apple Silicon, the environment is set up as a Parallels virtual machine. This choice aligns with the considerations discussed in Chapter 2, particularly regarding the relative stability and performance of Parallels compared to other hypervisor options for Apple Silicon.

Next, the focus shifts to the identification and configuration of the necessary tools for basic Android malware analysis. This encompasses both static and dynamic analysis tools, and the selection process is informed by the requirements of the proposed methodology as well as the constraints of the analysis environment and ARM compatibility. The tools selected will be integral to the various stages of the methodology, from initial actions and classification through to code and behavioural analysis.

The final part of this chapter addresses the potential for automation within the proposed methodology. In the field of malware analysis, automation can not only improve efficiency but also reduce the risk of human error such as missing important steps of the initial actions etc. The use of scripting techniques is explored to automate repetitive or time-consuming tasks, such as the extraction and classification of malware samples.

The following hardware was used for framework development, however, any other Apple Silicon based Mac will be suitable for recreation of the results:

- MacBook Air M2, 2022 8CPU/8GPU 16Gb RAM. macOS Ventura 13.1.1

3.2 PROPOSING THE METHODOLOGY

Although the MARE (Malware Analysis and Reverse Engineering) and SAMA (Structured Approach to Malware Analysis) methodologies offer valuable insights for general malware analysis, they are primarily centred on the analysis of Windows-based malware. Therefore, their direct application to mobile malware analysis is limited. However, certain aspects of these methodologies can be repurposed and incorporated into a custom methodology specifically tailored for mobile malware analysis.

The draft methodology proposed in this section consists of three fundamental stages: sample acquisition, static analysis, and dynamic analysis. It is important to stress that this is a preliminary version, and it will be further refined and expanded throughout this chapter.

1. **Sample Acquisition**: This stage involves the collection of mobile malware samples for analysis. The process of acquiring samples can be challenging due to the diverse sources and techniques used by attackers to distribute malware. However, it is a crucial first step, as the quality and relevance of the samples directly impact the subsequent analysis stages. In the context of Android malware analysis, this may involve the sourcing of potentially malicious Android applications (APK files) from app stores, infected devices, or online malware databases. This chapter explores the extraction of .apk files.
2. **Static Analysis**: Once the samples are acquired, the next step is static analysis. This stage involves examining the malware without executing it, focusing on attributes that can be gleaned from the surface. For Android malware, this could mean dissecting the APK file to analyse the manifest file, permissions, source code, and other embedded resources. Static analysis allows us to understand the potential behaviour of the malware and to identify any suspicious or malicious indicators. It serves as a foundational stage for more in-depth analysis, informing the subsequent dynamic analysis phase.
3. **Dynamic Analysis**: The final stage is dynamic analysis, which involves executing the malware in a controlled environment to observe its behaviour and interactions with the system. This stage is crucial in understanding the real-time operations of the malware, including the changes it makes to the system, the network communications it initiates, and any obfuscated or hidden behaviour that might not be evident during static analysis. In the context of Android malware, dynamic analysis might involve running the malicious APK in a virtualised environment and using tools to monitor its activity.

The selection of these stages aligns with certain elements of the MARE and SAMA methodologies, which also emphasise the importance of a systematic and layered approach to malware analysis. However, this draft methodology aims to adapt these stages to the unique context and challenges of mobile malware analysis, specifically for Android devices. This preliminary methodology serves as a starting point, and its structure and components will continue to be improved and expanded throughout this chapter.

3.3 SETTING UP THE ENVIRONMENT

The initial stage of setting up the mobile malware analysis environment involves the selection of an operating system to be virtualised. The chosen hypervisor for this process is Parallels Desktop 18 Standard Edition for Mac, due to its superiority over other free alternatives such as UTM, as discussed in Chapter 2. Parallels Desktop is the market leader in this field and enjoys a high level of popularity within the developer community. In fact, it could be considered a basic essential for any ARM Mac user, particularly given the discontinuation of support for Apple Boot Camp with the release of Apple Silicon. As of May 2023, there is no alternative to the booting of a different OS natively on Apple Silicon Macs, making Parallels a necessary tool. Furthermore, Parallels offers a student discount, making it a cost-effective choice for academic purposes.

3.3.1 Selection OS for the Framework

Parallels offers a seamless VM installation process, with several default operating system choices: *Windows 11, Ubuntu Server ARM, Fedora ARM, Kali ARM, Debian ARM, and macOS*. Among these, Windows 11 was immediately excluded due to a couple of reasons. Firstly, the availability of open-source tools for malware analysis is generally better on Linux systems. Secondly, the proliferation of malware on Windows platforms poses a significant security risk, especially in the case of an Android malware dropping Windows malware. Such a situation would be far less likely with Linux systems, which have significantly fewer malware developed for them.

MacOS could have also been a viable option, particularly considering Apple's Rosetta technology that enables the emulation of x86 applications with minimal performance loss. However, as of May 2023, Parallels does not support snapshots on Apple Silicon Macs, which significantly limits its usefulness in a malware analysis environment where being able to revert to a known clean state is important.

The final choice, therefore, lay between Ubuntu and Kali. Upon closer examination, Kali was selected due to the superior availability of packages in its Advanced Package Tool (apt) repository, especially once the qemu-user-static package is installed. This package allows the user to execute foreign binaries, expanding the range of software that can be run. In Kali, arm64 and amd64 packages are located in the same apt repository, which simplifies configuration compared to Ubuntu Server ARM where different repository URLs would need to be added. Another advantage would be the fact that Kali has many useful unitalities preinstalled such as *Wireshark* for analysis of network communications. Kali also offers many packages in its repository due to the popularity of Kali on Raspberry Pi which also unutilised ARM technology and could theoretically be beneficial during the framework development.

3.3.2 Installing the VM

Setting up ARM Kali VM in Parallels 18 was a matter of three clicks providing a user-friendly experience for any user. To create a Kali VM, navigate to Control Center and click on the "+" icon at the right top of the window (Figure 4). After the Installation Assistant window appears, select "*Download Kali Linux*" and press "*Continue*" which will download and install the VM (Figure 5).

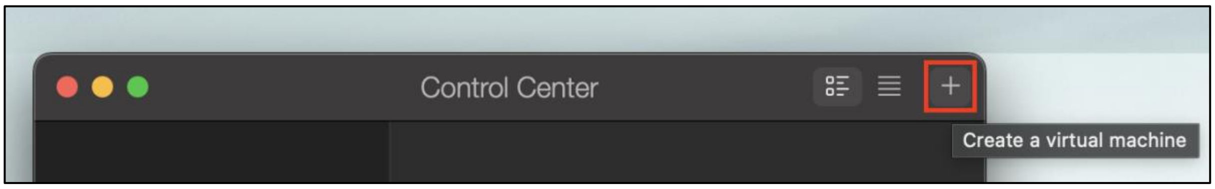


Figure 4: Adding new VM in Parallels 18

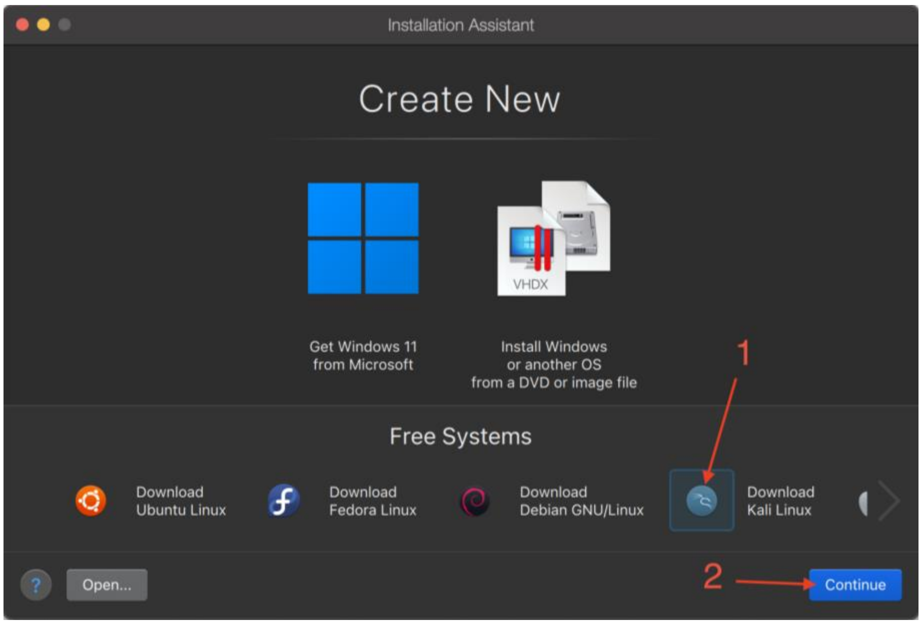
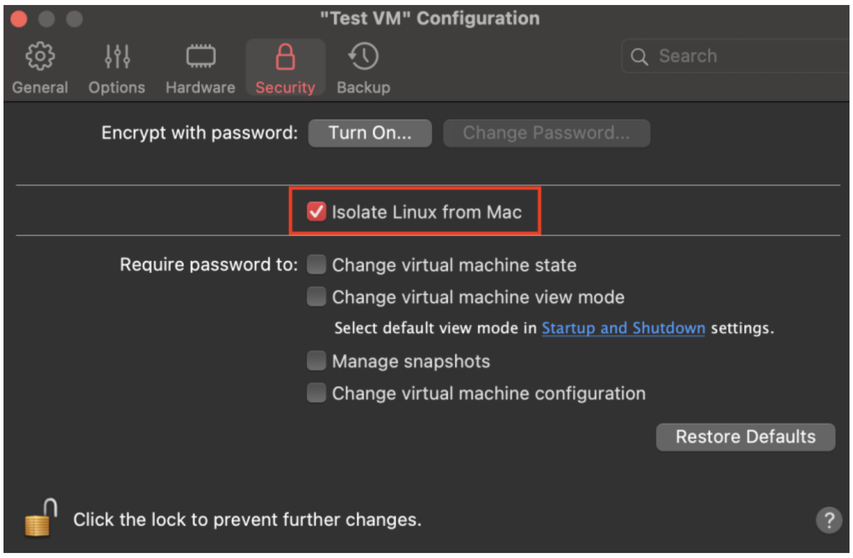


Figure 5: Choosing the Operating System for installation

3.3.3 Isolating the VM

After setting the password for the Kali VM (which is "**qwerty**") and the Parallels VM tools were installed automatically, the VM was shutdown to configure the security preferences of the VM. Parallels 18 provides the option to *Isolate Linux from Mac* (Figure 6):



Encrypt with password: Turn On... Change Password...

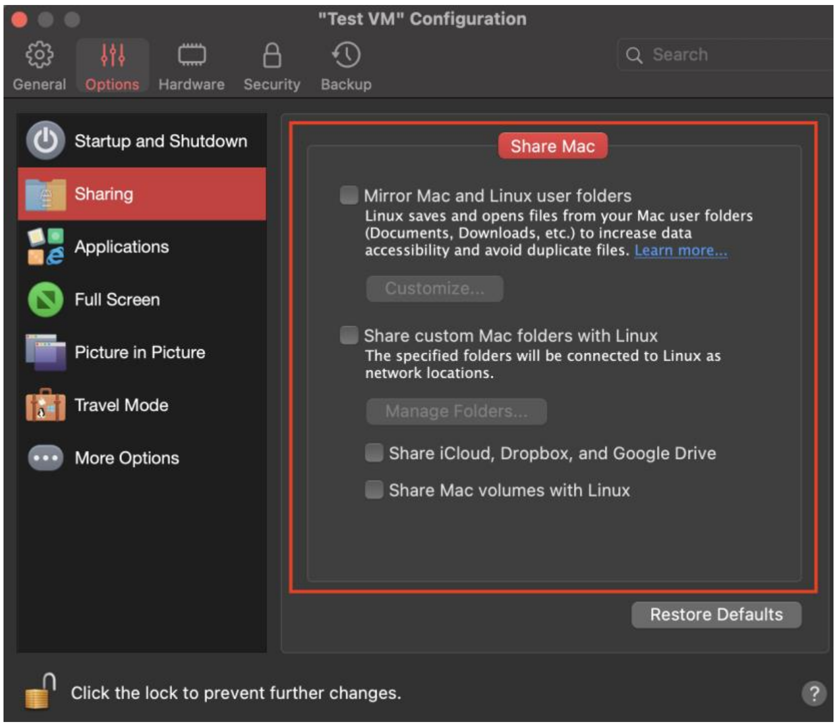
[✓] Isolate Linux from Mac

Require password to:
[ ] Change virtual machine state
[ ] Change virtual machine view mode
Select default view mode in Startup and Shutdown settings.
[ ] Manage snapshots
[ ] Change virtual machine configuration

Click the lock to prevent further changes.

Figure 6: Ability to isolate Parallels VM

However, this option would not be ideal for the user experience as the shared text buffer is disabled for the fully isolated VM. Instead, it was decided to keep isolation off and instead configure all the filesharing option to off. This would ensure acceptable levels of analysis environment isolation while keeling the access to the convenience of the shared copy/paste buffer between the host machine and the ARM Kali VM (Figure 7):



[ ] Mirror Mac and Linux user folders
Linux saves and opens files from your Mac user folders (Documents, Downloads, etc.) to increase data accessibility and avoid duplicate files. Learn more...

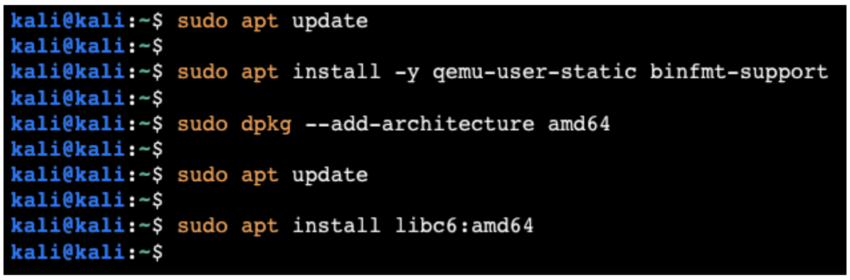
[ ] Share custom Mac folders with Linux
The specified folders will be connected to Linux as network locations.

[ ] Share iCloud, Dropbox, and Google Drive
[ ] Share Mac volumes with Linux

Click the lock to prevent further changes.

Figure 7: Selected Security Configuration

Finally, to complete the setup process, **qemu-user-static** was installed in accordance with Kali documentation which was supposed to aid in *running x86 code on ARM devices* (OffSec Services Limited, 2019). This would provide a wider range of available packages for the purposes of this project and development of Mobile Malware Analysis Framework (Figure 8):

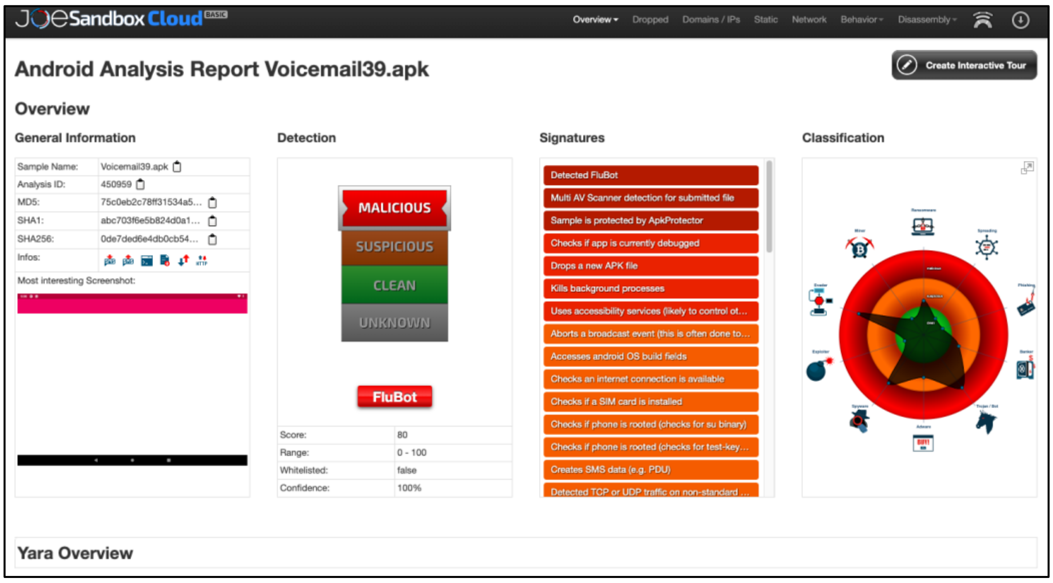


```
kali@kali:~$ sudo apt update
kali@kali:~$
kali@kali:~$ sudo apt install -y qemu-user-static binfmt-support
kali@kali:~$
kali@kali:~$ sudo dpkg --add-architecture amd64
kali@kali:~$
kali@kali:~$ sudo apt update
kali@kali:~$
kali@kali:~$ sudo apt install libc6:amd64
kali@kali:~$
```

Figure 8: Allowing ARM Kali compatibility with x86 code (OffSec Services Limited, 2019)

3.4 Setting the Benchmark for the Framework

With the goal of developing a comprehensive and effective mobile malware analysis framework, it is necessary to establish a performance benchmark. Among the numerous online automated malware analysis tools available, *JoeSandbox Cloud* stands out due to its remarkable ability to dissect Android malware (JoeSecurity, n.d.). This tool performs a thorough examination of Android malware samples by leveraging both static and dynamic analysis techniques, thereby providing a comprehensive report on the analysed sample, (Figure 9):



Android Analysis Report Voicemail39.apk

Overview

General Information

Sample Name:	Voicemail39.apk
Analysis ID:	450959
MD5:	75c0eb2c78ff31534a5…
SHA1:	abc703f6e5b824d0a1…
SHA256:	0de7ded6e4db0cb54…
Infos:	

Most interesting Screenshot:

Detection

Score:	80
Range:	0 - 100
Whitelisted:	false
Confidence:	100%

Signatures

Detected FluBot
Multi AV Scanner detection for submitted file
Sample is protected by ApkProtector
Checks if app is currently debugged
Drops a new APK file
Kills background processes
Uses accessibility services (likely to control ot…
Aborts a broadcast event (this is often done to…
Accesses android OS build fields
Checks an internet connection is available
Checks if a SIM card is installed
Checks if phone is rooted (checks for su binary)
Checks if phone is rooted (checks for test-key…
Creates SMS data (e.g. PDU)
Detected TCP or UDP traffic on non-standard …

Classification

Yara Overview

Figure 9: JoeSandbox Report Preview (Joe Security LCC, n.d.)

However, such robust capabilities come at a considerable cost. The full version of JoeSandbox Cloud has a hefty price tag, with some users reporting quotes of $12,000 per annum (No_Shift_Buckwheat, 2022). The free version, on the other hand, while more accessible, is limited in its functionality and requires individual application submissions from each interested user. Given these constraints, it is not a viable option for many users, particularly those in academia or other areas with budgetary constraints.

The objective of this project, therefore, is to develop a framework that can at least remotely match the capabilities of JoeSandbox Cloud. By setting this tool as the benchmark, the aim is to create a solution that is both affordable and effective, capable of delivering comprehensive malware analysis results. Achieving a framework with a performance comparable to JoeSandbox Cloud, without the associated cost, would be a clear indication of the project's success. Thus, JoeSandbox Cloud is marked as the benchmark for the proposed framework.

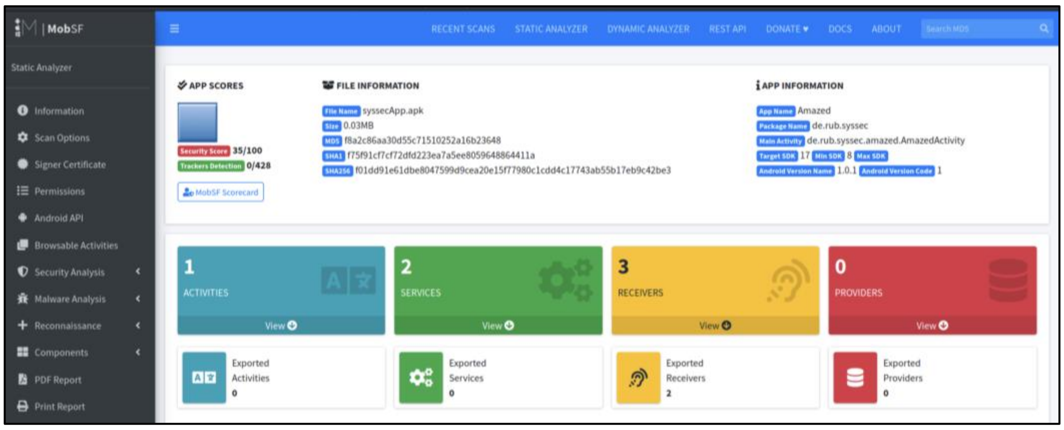
3.5 Selecting and Installing Required Tools

The selection of appropriate tools is a critical part of developing a comprehensive mobile malware analysis framework. These tools, much like the individual components of a machine, each play an important role in the overall function of the framework. Chosen tools must not only be compatible with the chosen operating system and hardware environment, but also robust enough to facilitate the malware analysis process of the proposed methodology.

3.5.1 Automated Analysis: Mobile Security Framework (MobSF)

One of the key tools selected for the framework was the Mobile Security Framework (MobSF), which was chosen as a local alternative to JoeSandbox Cloud, which was discussed in Section 3.4. MobSF offers similar functionality to the online-based counterpart, with the added advantage of being able to run locally. However, it is important to note that, like any other tool, MobSF is not without its limitations and challenges. For example, MobSF v3.6.3 lacks the functionality to look up existing malware databases such as VirusTotal. When analysis malware, the research of open malware databases would be highly beneficial specially for classification and understanding if the sample is known or not. **The integration of this step into the proposed methodology should be considered**.

MobSF stands as a versatile tool in the realm of mobile malware analysis, offering a wide range of functionalities. It facilitates both static and dynamic analysis of Android applications. In static analysis, it dissects APK files to extract manifest and certificate information, examines code for potential security issues, and identifies potentially malicious activities, services, and permissions. In dynamic analysis, it executes the application in a controlled environment to observe its behaviour. This includes monitoring file operations, network activity, and inter-process communication. MobSF offers a comprehensive analysis, assisting in the detection and understanding of malware behaviour and could be highly beneficial for the developed framework, especially considering that MobSF even offers similar user interface through a web browser (Figure 10):



APP SCORES
Security Score 35/100
Trackers Detection 0/428

FILE INFORMATION
File Name syssecApp.apk
Size 0.03MB
MD5 f8a2c86aa30d55c71510252a16b23648
SHA1 f75f91cf7cf72dfd223ea7a5ee8059648864411a
SHA256 f01dd91e61dbe8047599d9cea20e15f77980c1cdd4c17743ab55b17eb9c42be3

APP INFORMATION
App Name Amazed
Package Name de.rub.syssec
Main Activity de.rub.syssec.amazed.AmazedActivity
Target SDK 17 Min SDK 8 Max SDK
Android Version Name 1.0.1 Android Version Code 1

1 ACTIVITIES
2 SERVICES
3 RECEIVERS
0 PROVIDERS

Exported Activities 0
Exported Services 0
Exported Receivers 2
Exported Providers 0

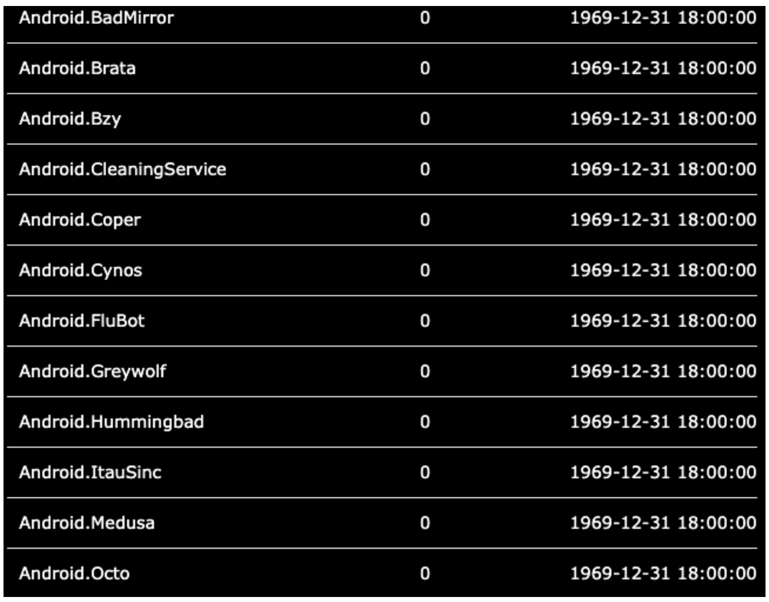
Figure 10: Working MobSF Report Preview

Nevertheless, the installation of MobSF in the ARM Kali VM was not a straightforward process. It involved a complex combination of dependencies and required multiple weeks of trial and error to find the perfect configuration. Due to the complexity of the process, and considering the scope of this project, a detailed description of each step of the installation process would be impractical. Some of the steps included the refracturing of the requirements.txt file, modification of the Python virtual environment, installing substitute dependencies and much more. The focus of this chapter is to provide an overview of the main tools used in the framework, and as such, the intricacies of the installation processes for each tool are beyond the scope of this chapter, although some of the setup challenges were discussed in Chapter 4. However, it is important to note that MobSF version 3.6.3 was successfully installed and is fully functional in the developed virtual machine.

3.5.2 Sample Acquisition

The ability to locate and acquire malware samples for analysis is a key aspect of any mobile malware analysis framework. With the right samples, analysts can learn about the latest threats, understand their behaviours, and develop effective countermeasures.

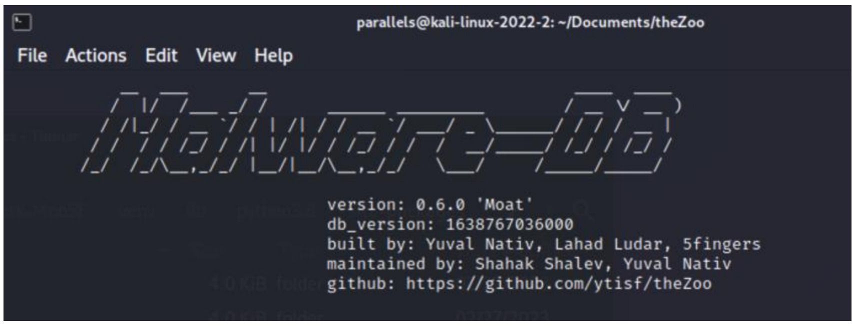
Despite the availability of numerous malware databases such as *vx-underground* (Figure 11), the availability of mobile malware samples is relatively limited. This is largely due to the fact that mobile malware is less widespread and less popular among security researchers, who often focus on threats to operating systems used in corporate operations and critical national infrastructure. However, the significance of mobile malware should not be underestimated, especially with the rapid advancement in mobile hardware that is bringing their performance closer to that of compact form-factor laptops.



Android.BadMirror	0	1969-12-31 18:00:00
Android.Brata	0	1969-12-31 18:00:00
Android.Bzy	0	1969-12-31 18:00:00
Android.CleaningService	0	1969-12-31 18:00:00
Android.Coper	0	1969-12-31 18:00:00
Android.Cynos	0	1969-12-31 18:00:00
Android.FluBot	0	1969-12-31 18:00:00
Android.Greywolf	0	1969-12-31 18:00:00
Android.Hummingbad	0	1969-12-31 18:00:00
Android.ItauSinc	0	1969-12-31 18:00:00
Android.Medusa	0	1969-12-31 18:00:00
Android.Octo	0	1969-12-31 18:00:00

Figure 11: vx-underground Android samples (vx-underground, n.d.)

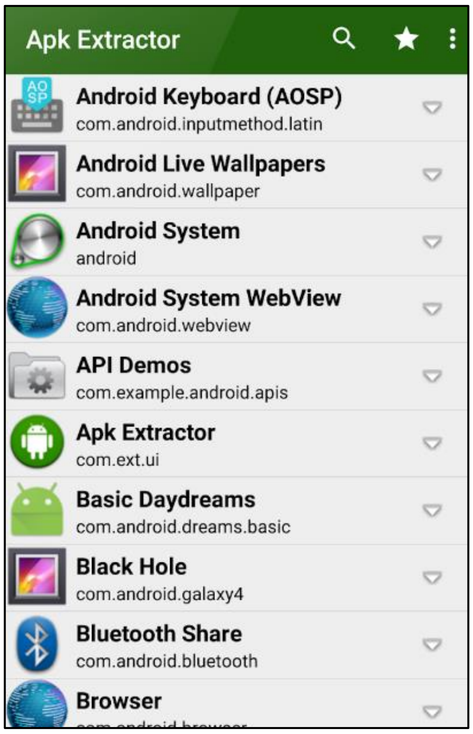
To facilitate the acquisition of malware samples for this project, the utility theZoo was installed in the framework. TheZoo is a project that houses and curates a live, openly available, malware repository. Despite some library compatibility issues, the installation process for theZoo was considerably more streamlined compared to the MobSF installation (Figure 12). theZoo houses a number of Android malware samples, which can be useful for training users of the framework and familiarising them with the characteristics and behaviours of Android malware.


parallels@kali-linux-2022-2: ~/Documents/theZoo
File Actions Edit View Help

```
version: 0.6.0 'Moat'
db_version: 1638767036000
built by: Yuval Nativ, Lahad Ludar, 5fingers
maintained by: Shahak Shalev, Yuval Nativ
github: https://github.com/ytisf/theZoo
```

Figure 12: theZoo working

In addition to theZoo, an application available on Google Play, named *Apk Extractor* by *meher*, was identified as a potential source of sample acquisition. This application allows for the extraction of .apk files from any installed application on a physical device or Android emulator, including those downloaded from Google Play itself. This opens up a vast range of potential samples for analysis, including both benign and potentially malicious applications. As such, Apk Extractor by Meher represents an important tool in the arsenal of the proposed mobile malware analysis framework although it would have to be installed manually by each researcher on their emulator or physical device.



Android Keyboard (AOSP)
com.android.inputmethod.latin
Android Live Wallpapers
com.android.wallpaper
Android System
android
Android System WebView
com.android.webview
API Demos
com.example.android.apis
Apk Extractor
com.ext.ui
Basic Daydreams
com.android.dreams.basic
Black Hole
com.android.galaxy4
Bluetooth Share
com.android.bluetooth
Browser

Figure 13: Apk Extractor by meher (meher, 2021)

3.5.3 Pre-Static Analysis

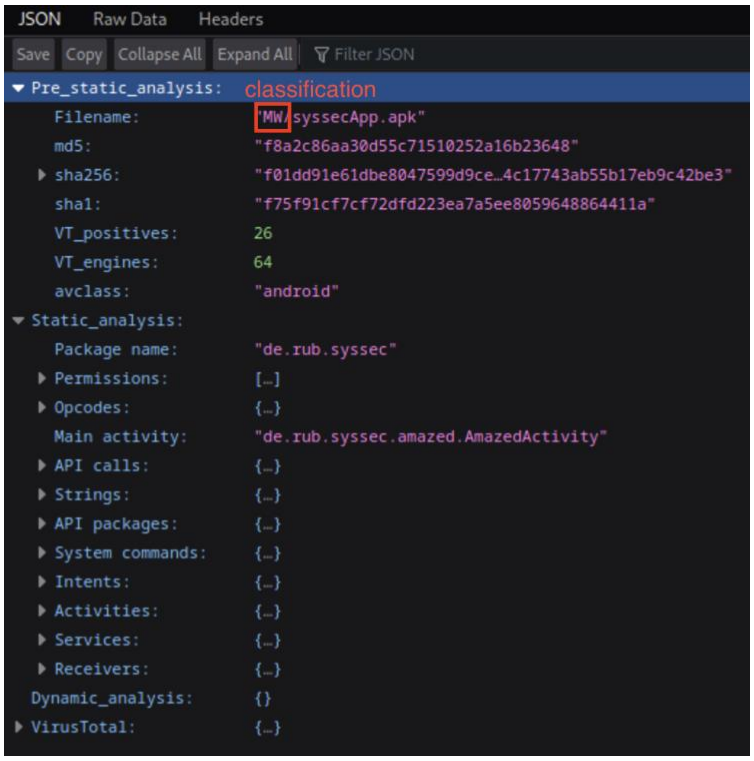
As noted in Section 3.5.1, one of the limitations of MobSF, in contrast to Joe Sandbox, is its inability to classify malware. After a thorough examination of the functionality of Joe Sandbox, it was concluded that malware classification would be a crucial part of the proposed methodology.

While there are several tools that utilise the VirusTotal API for classification purposes, further examination indicated that there may be room for improvement in this regard. During this exploration, AndroPyTool was discovered. This is a tool that automates aspects of Android malware analysis. Written in Python 2 and dependent on some 32-bit binaries, it is incompatible with Apple Silicon. An attempt was made to translate AndroPyTool to Python 3 using the 2to3 tool and manual coding. Although some functionality was achieved, it became evident that reconstructing the entirety of this open-source project would be a time-consuming task.

Fortunately, a Docker image of AndroPyTool was available, which was optimised for x86 architectures but was also executable on ARM Kali. Docker is a platform that uses OS-level virtualisation to deliver software in packages called containers. Containers are isolated from each other and bundle their own software, libraries, and configuration files, allowing for software to run reliably when moved from one computing environment to another.

AndroPyTool was identified as a valuable tool for the pre-static and static analysis stage (Martín, et al., 2018). It has the capacity to fetch VirusTotal reports, perform fingerprinting, and conduct basic automated static analysis, generating a JSON report at the end of the process. This tool utilises *AndroGuard*, another tool that was installed in the framework. AndroGuard is a full Python tool providing Android application analysis by reverse engineering the DEX format and APK files, offering capabilities such as decompiling to smali, generating call and control flow graphs, or getting information on used permissions and defined activities or services. Hence, AndroPyTool, in conjunction with AndroGuard, would greatly enhance the capabilities of the framework in the pre-static analysis phase.

Here is an example of what AndroPyTool can generate, (Figure 14):



```
▼ Pre_static_analysis:
    Filename:           "MW/syssecApp.apk"
    md5:                "f8a2c86aa30d55c71510252a16b23648"
  ▶ sha256:             "f01dd91e61dbe8047599d9ce…4c17743ab55b17eb9c42be3"
    sha1:               "f75f91cf7cf72dfd223ea7a5ee8059648864411a"
    VT_positives:       26
    VT_engines:         64
    avclass:            "android"
▼ Static_analysis:
    Package name:       "de.rub.syssec"
  ▶ Permissions:        […]
  ▶ Opcodes:            {…}
    Main activity:      "de.rub.syssec.amazed.AmazedActivity"
  ▶ API calls:          {…}
  ▶ Strings:            {…}
  ▶ API packages:       {…}
  ▶ System commands:    {…}
  ▶ Intents:            {…}
  ▶ Activities:         {…}
  ▶ Services:           {…}
  ▶ Receivers:          {…}
  Dynamic_analysis:     {}
▶ VirusTotal:           {…}
```

Figure 14: AndroPyTool JSON report example

3.5.4 Static Analysis and Reverse Engineering

The importance of static analysis in the malware analysis process cannot be underestimated. Static analysis plays a significant role in gaining insights into the malware's underlying structure and functionality without executing the malicious code. This section aims to discuss some of the widely used static analysis tools that have proven effective in examining Android malware.

AndroPyTool, as mentioned in the previous section, is equipped to perform most of the static analysis tasks. These include dumping strings, listing permissions, and mapping API calls, which are all essential for understanding the logic behind malware. Indeed, several research studies have demonstrated the value of API calls and permissions in identifying whether a sample is benign or malicious, making these aspects paramount in machine learning-based Android malware detection.

In addition to AndroPyTool, *JADX*, a widely recognised reverse engineering tool, was installed as part of the framework. As mentioned in Section 2.3.3, JADX offers best-on-market decompilation capabilities for Android software, akin to what Hopper Disassembler provides for software developed for Apple platforms. It enables manual reverse engineering, allowing analysts to delve deeper into the inner workings of malware. One of the standout features of JADX is its ability to create Frida snippets. Frida is a dynamic instrumentation toolkit that allows you to inject JavaScript to explore native apps on Windows, macOS, Linux, iOS, Android, and QNX. These Frida snippets generated by JADX can be of tremendous value during dynamic analysis, as they offer insights into the behaviour of the malware during execution. Luckily, JADX could be installed by using *apt install* command.

3.6 DYNAMIC ANALYSIS: OVERCOMING THE LACK OF NESTER VIRTUALISATION

The Mobile Security Framework (MobSF) offers an extensive range of dynamic analysis functionalities, such as Frida scripting and the ability to execute all APIs of the APK file. With these capabilities, the only remaining requirement is to provide MobSF with a suitable execution environment. One straightforward approach is to connect a physical Android device, but this may not always be a feasible option, as not every analyst has access to a dedicated Android device for malware analysis.

An alternative was considered: to install Android Studio in the developed Framework VM. However, this approach was met with a significant obstacle. Due to the lack of nested virtualisation support on Apple Silicon, the Android emulator included in Android Studio could not be run within the virtual machine

Nevertheless, a workaround was found to overcome this limitation. Android Studio can be installed outside of the VM, directly on the host machine. Once installed, the *Android Debug Bridge* (ADB) port, which is typically 5555, can be forwarded using the "*nc*" command. This port forwarding allows the VM to connect to the Android emulator running on the host machine, effectively enabling MobSF to perform dynamic analysis.

In addition to this, the Android Debug Bridge (ADB) utility itself provides a variety of tools for dynamic analysis. These tools can facilitate tasks such as monitoring system events, manipulating incoming calls and messages, and much more. Through a combination of these methods, it's possible to construct a comprehensive dynamic analysis environment, even in the absence of nested virtualisation support on the host machine.

The full and updated instructions of how to set this up was placed as a video demo on the website of the author of this project:

https://cybernester.com/androidmalware.html

3.7 Automating some of the Processes with Python

Automation plays a pivotal role in streamlining the malware analysis process, enhancing efficiency, and maintaining consistency in accordance with the proposed methodology. For this project, a Python 3 script called *android_analysis* was developed to automate some of the routine tasks involved in malware analysis.

The script carries out several steps, starting with the setup of output and temporary folders for the analysis process. It then copies the APK file to be analysed to the temporary folder. Once the initial setup is done, it executes a Docker command to run AndroPyTool on the APK file, performing preliminary analysis and obtaining a VirusTotal report.

Once the analysis is complete, the script ensures that the user has the necessary permissions over the files and folders generated during the analysis. It then copies the results to the output folder located on the VM's desktop and deletes the temporary folder.

Finally, the script opens JADX-GUI for the selected AP and MobSF in new terminal windows, starting the GUI-based analysis process. It also opens the JSON analysis file and the localhost URL where the MobSF interaction is hosted in Firefox for easy access.

The script is available in Appendix A for further reference. The automation implemented through this script significantly reduces the manual steps involved in setting up and initiating malware analysis, thus enhancing the overall productivity of the process. You can also watch the video demo using the following URL:

https://cybernester.com/androidmalware.html

The script requires two arguments:

1. -vtk – VirusTotal API key, which is issued upon VirusTotal account creation
2. -f – the file path to the desired .apk file.

4 Results

4.1 Final Methodology: The Unified Mobile Malware Analysis (UMMA)

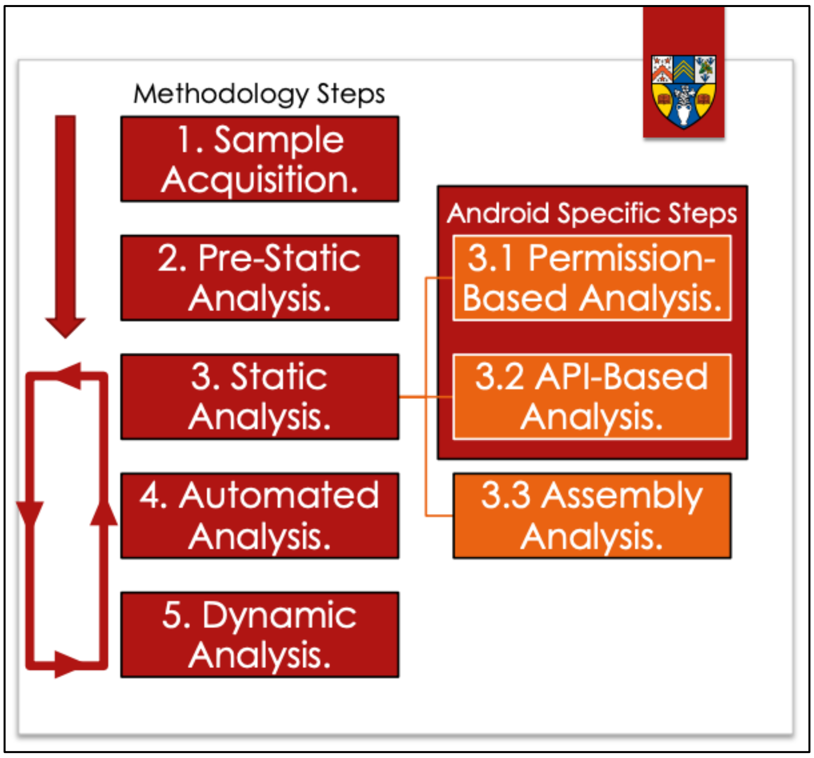


Figure 15: The Unified Mobile Malware Analysis (UMMA) methodology

The culmination of the research and development undertaken in configuring the Framework VM and exploring the available tools has led to the formation of a systematic approach towards mobile malware analysis. We present this new approach as the Unified Mobile Malware Analysis (UMMA) methodology (Figure 15), which incorporates several integral steps, each designed to facilitate a comprehensive examination of mobile malware.

The steps of the UMMA methodology are as follows:

1. **Sample Acquisition**: The first step involves sourcing malware samples from various repositories or databases. The quality and relevance of these samples significantly impact the subsequent stages of analysis.
2. **Pre-Static Analysis**: This preliminary stage involves the use of tools like AndroPyTool for obtaining virus total reports, basic automated static analysis, and fingerprinting of the malware samples.
3. **Static Analysis**: This critical phase involves a detailed examination of the malware's structure and functionality without execution. This step further breaks down into three android specific sub-steps:

3.1 **Permission-based Analysis**: Here, the focus lies on the analysis of permissions requested by the malware. Unusual or excessive permissions can often indicate malicious intent.

3.2 **API-based Analysis**: This step involves studying the API calls made by the malware. Patterns of suspicious or harmful API calls can be indicative of malicious behavior.

3.3 **Assembly Analysis**: This stage requires the use of tools like JADX for reverse engineering and decompilation of the android software. The understanding of assembly code can provide insights into the malware's functionality.

4. **Automated Analysis**: Utilising the automated analysis features of tools like MobSF, this step aims to improve efficiency and consistency in the examination process.
5. **Dynamic Analysis**: The final step involves the execution of malware in a controlled environment to study its behavior and impact. Tools like MobSF, and ADB utility from Android Studio can be employed for this purpose.

It is essential to note that steps 3, 4, and 5 should not be viewed as strictly linear or sequential. In practice, these stages should be looped, forming an iterative cycle of analysis. Information discovered during one stage can provide valuable insights that aid the understanding and progress of other stages. For instance, data uncovered during dynamic analysis could facilitate a better understanding of assembly code and vice versa. This interplay between stages fosters a more thorough and nuanced understanding of the malware.

4.2 THE DEVELOPED FRAMEWORK

The developed framework is an integrated collection of tools, each contributing to a comprehensive environment for Android malware analysis. This environment takes the form of a Parallels ARM Kali virtual machine (VM), designed to operate efficiently on the Apple Silicon M1. It features a carefully selected suite of tools, each with unique capabilities that, when used in tandem, create a robust and efficient system for examining and understanding the intricacies of Android malware.

Prominent tools incorporated within the framework include the Mobile Security Framework (MobSF), AndroGuard, AndroPyTool, and Apktool, and many others. Each of these tools was chosen for its specific functionality and its ability to contribute to the overall malware analysis process:

- **MobSF** facilitates both static and dynamic analysis of Android applications, providing comprehensive reports on potential security issues, potentially malicious activities, and observed application behaviour during execution.
- **AndroGuard** is used for the examination of Android APKs, allowing for the extraction of permissions, activities, services, and receivers, among other features. It also provides the ability to visualise the control flow graph of the application's code, contributing to an understanding of the application's logic and potential points of vulnerability.
- **AndroPyTool** provides automated static analysis, fingerprinting, and VirusTotal reporting capabilities, offering a starting point for malware examination and classification.
- **Apktool** is used for reverse engineering third-party, closed, binary Android apps. It allows for the decoding of resources to nearly original form and the rebuilding of them after making modifications.
- Other tools which can assist the analysis process.

These tools, among others, were carefully integrated into the VM to provide a comprehensive toolkit for Android malware analysis. When used in conjunction with the proposed Unified Mobile Malware Analysis (UMMA) methodology, this framework is capable of providing thorough, systematic, and nuanced insight into the underlying logic and behaviour of Android malware.

4.3 METHODOLOGY AND FRAMEWORK AVAILABILITY

In the spirit of fostering a community of knowledge sharing among security researchers and students alike, the developed framework and the proposed Unified Mobile Malware Analysis (UMMA) methodology are made readily accessible. They have been uploaded to my personal website at the following website which was developed by the autor of this report and is hosted on GitHub which can provide a good communication platform with the use of Issues feature of GitHub:

https://cybernester.com/androidmalware.html.

On this dedicated page, visitors can download the ARM Kali virtual machine pre-configured with all the necessary tools for Android malware analysis as outlined in the methodology. The VM password is also provided to ensure full access to the features and functionalities of the framework.

Additionally, a detailed presentation of the UMMA methodology is available for anyone interested in understanding the proposed process for Android malware analysis. This step-by-step guide can help both beginners and seasoned researchers in their malware analysis efforts.

Moreover, video demos are provided to offer a more interactive and visual experience. These demos include a walkthrough of the developed Python script in action, a guided tour of the VM, and a comprehensive guide on setting up the Android emulator to work seamlessly with MobSF. The demos aim to provide a practical, hands-on approach to understanding and using the framework, ultimately easing the learning curve and increasing the efficiency of Android malware analysis.

5 Discussion

5.1 Unified Mobile Malware Analysis (UMMA) Methodology

One of the primary objectives of this project was to investigate the feasibility of structuring Mobile Malware Analysis into a standardised methodology. The resultant product is the Unified Mobile Malware Analysis (UMMA) methodology. The evaluation of this methodology is conducted in light of existing malware analysis methodologies, Malware Analysis Reverse Engineering (MARE) and Systematic Analysis of Malware Artifacts (SAMA), to determine its effectiveness and overall benefits.

The UMMA methodology comprises of the following steps:

1. Sample acquisition
2. Pre-Static Analysis
3. Static Analysis

3.1 Permission-based analysis
3.2 API-based analysis
3.3 Assembly Analysis

4. Automated Analysis
5. Dynamic Analysis

Unlike the MARE and SAMA methodologies, which have a linear process, the UMMA methodology is designed with a loop mechanism for steps 3 to 5. This signifies the iterative nature of Android malware analysis, recognising that new information uncovered at any stage can facilitate understanding and potentially advance other stages.

The UMMA methodology encompasses more Android specific steps than both MARE and SAMA. It incorporates the permission-based and API-based analysis, both unique to Android application analysis. These steps can reveal crucial insights regarding the potentially malicious actions that a malware-infected app can undertake.

The inclusion of an automated analysis stage is another key difference between the UMMA methodology and the traditional MARE and SAMA methodologies. Automated tools can expedite the process of analysis and reduce the amount of manual effort required. This, combined with the loop mechanism, can make the malware analysis process significantly more efficient.

However, the UMMA methodology does not explicitly include a stage for isolation and extraction of malware, which is a crucial aspect in both MARE and SAMA. It can be argued that this is an implicit step taken care of during the setup of the framework VM. Nevertheless, explicitly including it would provide a more comprehensive approach.

While the UMMA methodology does not involve a dedicated deobfuscation and decryption stage, like the Android Malware Analysis Workflow suggested in Mastering Malware Analysis, it is important to note that these processes are inherently considered in the static and dynamic analysis stages.

In summary, the UMMA methodology provides a more detailed and Android-specific approach to malware analysis compared to MARE and SAMA. Its iterative nature is more reflective of the actual malware analysis process, and the inclusion of automated analysis can enhance efficiency. However, it would benefit from explicitly including stages for isolation and extraction of malware, and possibly deobfuscation and decryption, to ensure a more comprehensive and robust methodology.

5.2 THE DEVELOPED FRAMEWORK

The Unified Mobile Malware Analysis (UMMA) methodology presents a comprehensive framework for Android malware analysis, incorporating several potent tools that assist in various stages of malware analysis. This methodology offers a unique, unified approach to combat the ever-growing threat of mobile malware by providing a user-friendly, efficient, and effective system.

The UMMA methodology has several significant advantages. First, it incorporates a basic set of tools necessary for Android malware analysis, such as the Mobile Security Framework (MobSF), JADX, AndroPyTool, and AndroGuard, among others. These tools, when combined, offer a powerful suite that can facilitate both static and dynamic analysis of Android applications, providing crucial insights into malware's behaviour and structure.

Secondly, the framework's design allows for easy importing of virtual machines into Parallels. This is particularly advantageous as it eases the setup process, which is typically a complex and time-consuming endeavour. The straightforward import process effectively reduces the barrier to entry, enabling more users, like police investigators, students, and incident response teams, to utilise the framework.

For instance, police investigators could use the framework for digital forensics, helping them trace cybercriminal activities, while students can gain hands-on experience in malware analysis, sharpening their skills and knowledge. Incident response teams, on the other hand, can use this framework to promptly analyse and respond to malware-related incidents, thus minimising potential damage.

However, the UMMA methodology does present some limitations. One of the main drawbacks is that the framework is more inclined towards Android malware analysis rather than mobile malware in general. While this specialisation allows for a deeper focus on Android malware, which forms a significant portion of mobile malware, it does limit the applicability of the framework in the broader scope of mobile malware analysis, particularly concerning other operating systems such as iOS.

Another challenge lies in maintaining and promoting the framework. With the rapidly evolving landscape of malware and corresponding analysis tools, it is crucial to continuously validate and update the tools included in the framework. This demands substantial effort and resources, as potential users and security researchers' suggested tools would need extensive validation before integration.

Additionally, the framework is primarily designed to run on Apple Silicon rather than ARM computers in general. While this may not pose a significant problem, given the .pvm files can be converted to other hypervisor VM formats, it does restrict the framework's accessibility to those who do not have access to Apple Silicon machines.

In summary, the UMMA methodology and the developed Framework represent a significant stride in Android malware analysis. Its strengths lie in its comprehensive suite of tools, user-friendly approach, and adaptability. Despite its limitations, the methodology and framework provide a robust platform for Android malware analysis, offering substantial value to various stakeholders, including police investigators, students, and incident response teams. With appropriate updates and enhancements, the UMMA methodology has the potential to become a cornerstone in the field of Android malware analysis. In light of the main project aim, the development of the Framework can be considered successful to an extent, while the basic toolset was configured for analysis of Android malware, the preparation of the VM to accommodate iOS malware was left unconfigured.

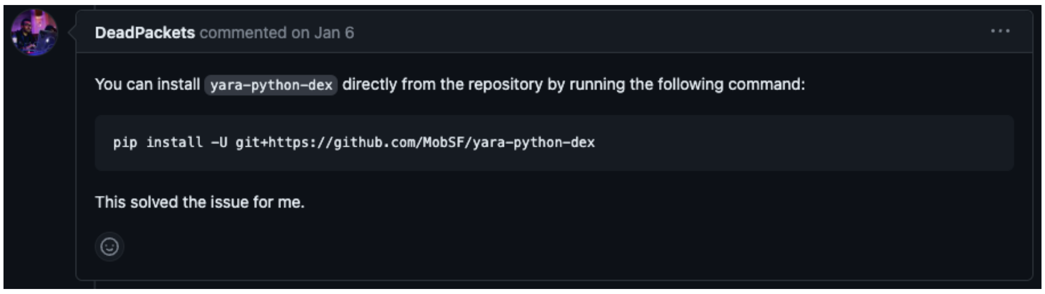
5.3 Compatibility and Setup Challenges

The development of the Unified Mobile Malware Analysis (UMMA) framework has been marked by several compatibility and setup challenges, largely due to the inherent complexities of ARM hardware and the relative unpopularity of mobile malware analysis within the security research community.

There's an undeniable advantage in analysing mobile malware, particularly given the exponential growth of mobile platforms. As smartphones become increasingly integrated into our daily activities, from banking to social networking, there's a corresponding increase in potential threats, thus highlighting the importance of rigorous mobile malware analysis.

However, this growing field has not received the same level of attention as desktop malware analysis. This lack of focus can be attributed to the long-standing centrality of desktop operating systems in malware analysis, largely because they have traditionally been the primary targets of cyber attackers, given their widespread use in corporate environments and critical infrastructures. Conversely, mobile devices have often been overlooked as less attractive targets, leading to a dearth of dedicated tools and resources for mobile malware analysis.

This lack of resources and tools specifically designed for mobile malware analysis poses a significant challenge when setting up the UMMA framework on ARM hardware. Many tools developed for traditional malware analysis are optimised for x86-based systems and lack support for the ARM architecture. This represents a considerable hurdle in the development and implementation of a comprehensive mobile malware analysis framework. For example, there was a challenge faced during the installation of MobSF, *yara-python* dependency version was not available for the in the ARM pip repository, adding extra required steps to overcome the compatibility problems, (Figure 16):


DeadPackets commented on Jan 6

You can install `yara-python-dex` directly from the repository by running the following command:

```
pip install -U git+https://github.com/MobSF/yara-python-dex
```

This solved the issue for me.

Figure 16: Example of issue faced fix provided by the community member

Addressing these challenges requires a significant shift within the IT security community. One of the primary requirements for a successful mobile malware analysis framework designed for ARM-based hardware is the porting of existing tools to Python3 and the elimination of 32-bit dependencies. By doing so, the community can significantly alleviate the compatibility and setup challenges associated with ARM hardware, thereby encouraging wider adoption and use of the framework. In turn, this could lead to a more balanced focus between mobile and desktop malware analysis, helping to develop more effective countermeasures against the evolving landscape of mobile malware threats.

6 CONCLUSION & FUTURE WORK

Throughout this paper, an array of existing research, tools, and technologies have been critically examined. The Unified Mobile Malware Analysis (UMMA) methodology, a novel approach to mobile malware analysis, was developed in parallel with the Virtual Machine (VM) framework.

This project set out to answer several research questions, notably:

- **Is it possible to develop a Framework for Mobile Malware Analysis in a form of a pre-configured Virtual Machine designed to run on ARM-based hardware?**
    I. To accompany the developed Framework, is it possible to structure Mobile Malware Analysis into a standardised methodology?
    II. Could existing tools for manual Mobile Malware Analysis be automated to speed up and simplified for novice security researchers?

A critical evaluation of the project in light of these questions indicates a semi-successful outcome. The developed framework and accompanying UMMA methodology primarily focus on mobile malware, presenting both a comprehensive, structured approach to mobile malware analysis and a positive step towards preparing for the enforcement of the EU Digital Markets Act in 2024. This legislation will mandate side-loading capabilities for iOS, which will likely result in an increase in mobile malware.

However, the UMMA methodology and the developed VM framework have proven to be effective and beneficial tools for the analysis of mobile malware. With the ability to acquire samples, conduct static and dynamic analysis, automate parts of the process, and delve into the intricacies of Android malware, these resources provide a robust and efficient system for understanding and combating mobile malware.

The methodology and framework were made available to the public on a dedicated website, fostering a community of knowledge sharing among security researchers and students. This public access allowed for the gathering of valuable feedback and suggestions from professional malware analysis engineers, further refining and improving the UMMA methodology and the VM framework.

Looking forward, there are several potential avenues for future work. One of the most significant is the adaptation of the methodology to accommodate iOS malware analysis, especially in light of the impending changes to iOS side-loading policies. Continual public access to the framework will allow for ongoing feedback and iterative improvements, ensuring the methodology and framework remain relevant and effective in the ever-evolving landscape of mobile malware.

The analysis of mobile malware is a critical facet of contemporary cybersecurity, and the development of a structured methodology and a dedicated framework for such analysis represents a significant contribution to the field. While there is always room for improvement and expansion, the UMMA methodology and the VM framework stand as robust, effective tools for the analysis of mobile malware, fostering a better understanding of these threats and facilitating the development of more effective countermeasures.

The future of mobile malware analysis is set to be challenging, dynamic, and crucially important. With the tools and methodologies developed in this project, researchers and security professionals will be better equipped to face these challenges and contribute to the ongoing effort to secure our digital world.

BIBLIOGRAPHY

Apple, 2020. *Install Windows 10 on your Mac with Boot Camp Assistant.* [Online]
Available at: https://support.apple.com/en-gb/HT201468
[Accessed 18 April 2023].

Asahi Linux , 2020. *Twitter: Asahi Linux regarding nested virtualisation on Apple Silicon.* [Online]
Available at: https://twitter.com/AsahiLinux/status/1513098987297185794?s=20
[Accessed 18 April 2023].

AV-Test, 2020. *Android Users Beware: This Is Why You Should Never Rely On Google's Own Malware Protection.* [Online]
Available at: https://www.forbes.com/sites/kateoflahertyuk/2020/03/10/android-users-beware-this-is-why-you-should-never-rely-on-googles-own-malware-protection/
[Accessed 13 May 2023].

AWS, n.d. *AWS Graviton Processor.* [Online]
Available at: https://aws.amazon.com/ec2/graviton/
[Accessed 15 May 2023].

Batyuk, L. et al., 2011. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications. *IEEE: 6th International Conference on Malicious and Unwanted Software.*

Bermejo Higuera, J. et al., 2020. Systematic approach to malware analysis (SAMA). *Applied Sciences,* 10(4), p. 1360.

Dobrovolskiy, N., 2021. *Parallels Desktop for Mac with Apple M1 chip.* [Online]
Available at: https://www.parallels.com/blogs/parallels-desktop-apple-silicon-mac/
[Accessed 18 Appril 2023].

Elayan, O. & Mustafa, A., 2021. Android malware detection using deep learning. *Procedia Computer Science,* Volume 184, pp. 847-852.

European Commission, 2022. *The Digital Markets Act: ensuring fair and open digital markets.* [Online]
Available at: https://commission.europa.eu/strategy-and-policy/priorities-2019-2024/europe-fit-digital-age/digital-markets-act-ensuring-fair-and-open-digital-markets_en
[Accessed 14 May 2023].

Google, n.d. *Use Google Play Protect to help keep your apps safe and your data private.* [Online]
Available at: https://support.google.com/googleplay/answer/2812853?hl=en-GB
[Accessed 13 May 2023].

Joe Security LCC, n.d. *Android Analysis Report Voicemail39.apk.* [Online]
Available at: https://www.joesandbox.com/analysis/450959/0/html
[Accessed 15 May 2023].

JoeSecurity, n.d. *JoeSandbox Cloud: Deep Malware Analysis.* [Online]
Available at: https://www.joesecurity.org/joe-sandbox-cloud#overview
[Accessed 15 May 2023].

Kiachidis, I. & Baltatzis, D., 2021. Comparative Review of Malware Analysis Methodologies. *International Journal of Network Security & Its Applications (IJNSA),* 13(6).

Kleymenov, A. & Thabet, A., 2022. *Mastering Malware Analysis: A malware analyst's practical guide to combating malicious software, APT, cybercrime, and IoT attacks.* s.l.:Packt Publishing Ltd.

Martín, A., Lara-Cabrera, R. & Camacho, D., 2018. Android malware detection through hybrid features fusion and ensemble classifiers: The AndroPyTool framework and the OmniDroid dataset. *Information Fusion,* Volume 52, pp. 128-142.

Mauthe, N., Kargén, U. & Shahmehri, N., 2021. A Large-Scale empirical study of Android app decompilation. *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER),* pp. 400-410.

meher, 2021. *Apk Extractor.* [Online]
Available at: https://play.google.com/store/apps/details?id=com.ext.ui&hl=en_GB&gl=US
[Accessed 15 May 2023].

Nguyen, C. Q. & Goldman, J. E., 2010. Malware analysis reverse engineering (MARE) methodology & malware defense (MD) timeline. *Information Security Curriculum Development Conference,* pp. 8-14.

No_Shift_Buckwheat, 2022. *r/Malware Joe's Sanbox Pricing.* [Online]
Available at: https://www.reddit.com/r/Malware/comments/s27592/joes_sandbox_pricing/
[Accessed 15 May 2023].

OffSec Services Limited, 2019. *Running x86 code on ARM devices.* [Online]
Available at: https://www.kali.org/docs/arm/x86-on-arm/
[Accessed 15 May 2023].

Pandey, R., 2020. Comparing VMware Fusion, Oracle VirtualBox, Parallels Desktop implemented as Type-2 hypervisors.

Pawar, P., 2022. *Most Insightful Digital Banking Statistics To Know in 2022.* [Online]
Available at: https://www.enterpriseappstoday.com/stats/digital-banking-statistics.html
[Accessed 17 April 2023].

Shikova, T., 2023. *The mobile malware threat landscape in 2022.* [Online]
Available at: https://securelist.com/mobile-threat-report-2022/108844/
[Accessed 13 May 2023].

StatCounter, 2023. *Mobile Operating System Market Share Worldwide.* [Online]
Available at: https://gs.statcounter.com/os-market-share/mobile/worldwide
[Accessed 4 May 2023].

vx-underground, n.d. *Directory: samples/Families/.* [Online]
Available at: https://samples.vx-underground.org/samples/Families/
[Accessed 15 May 2023].

APPENDICES

APPENDIX A: *ANDROID_ANALYSIS* SCRIPT

```
#!/usr/bin/env python3
import argparse
import os
import shutil
import subprocess

def check_and_delete_folders(folder_list):
    for folder in folder_list:
        if os.path.exists(folder):
            subprocess.run(f"sudo chown -R $(whoami) {folder}", shell=True, check=True)
            shutil.rmtree(folder)

def main(vtk, f):
    apk_name = os.path.basename(f)

    # Create output folder on Desktop
    output_folder = os.path.join(os.path.expanduser("~/Desktop"), apk_name.split('.')[0])

    # Create temporary folder
    temp_folder = os.path.expanduser("~/Documents/TempAndroid")

    # Check and delete output and temporary folders if they exist
    check_and_delete_folders([output_folder, temp_folder])

    os.makedirs(output_folder, exist_ok=True)
    os.makedirs(temp_folder, exist_ok=True)

    # Copy APK to temporary folder
    shutil.copy(f, os.path.join(temp_folder, apk_name))

    # Run Docker AndroPyTool command
    cmd = f"sudo docker run --volume={temp_folder}:/apks alexmyg/andropytool -s /apks/ -vt {vtk} -cl"
    subprocess.run(cmd, shell=True, check=True)
```

```python
    # Get folder permissions
    subprocess.run(f"sudo chown -R $(whoami) {temp_folder}", shell=True, check=True)

    # Copy results to output folder
    shutil.copytree(os.path.join(temp_folder, "Features_files"), os.path.join(output_folder, "Features_files"))
    shutil.copytree(os.path.join(temp_folder, "samples"), os.path.join(output_folder, "samples"))
    shutil.copytree(os.path.join(temp_folder, "VT_analysis"), os.path.join(output_folder, "VT_analysis"))

    # Delete temporary folder
    shutil.rmtree(temp_folder)

    # Run jadx-gui in a new terminal window
    cmd_jadx = f"gnome-terminal --title='jadx' -- jadx-gui {f}"
    subprocess.Popen(cmd_jadx, shell=True)

    # Run MobSF in a new terminal window
    mobsf_path = os.path.expanduser("~/Desktop/workingMobsf/Mobile-Security-Framework-MobSF")
    cmd_mobsf = f"gnome-terminal --title='MobSF' -- bash -c 'cd {mobsf_path} && ./run.sh 127.0.0.1:8000'"
    subprocess.Popen(cmd_mobsf, shell=True)

    # Open JSON file and localhost URL in Firefox
    cmd_firefox = f"gnome-terminal --title='firefox' -- firefox -new-tab -url
file://{output_folder}/Features_files/{apk_name.split('.')[0]}-analysis.json -url http://localhost:8000/"
    subprocess.Popen(cmd_firefox, shell=True)

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="APK analysis script")
    parser.add_argument("--vtk", type=str, required=True, help="VirusTotal API key, get one by making an
account at VirusTotal")
    parser.add_argument("--f", type=str, required=True, help="Path to the APK file you wish to analyse")
    args = parser.parse_args()

    main(args.vtk, args.f)
```