HALY: Automated Evaluation of Hardening Techniques in Android and iOS Apps

Wilco van Beijnum
University of Twente
5 July 2023

Supervisor: Andrea Continella
Second supervisor: Ralph Holz

***Abstract*—Although mobile operating systems employ a variety of features to sandbox and isolate apps, these are not always sufficient. Because of this, app developers are recommended to implement their own security checks. In this work, we investigate the prevalence of hardening techniques in mobile apps. We design and develop HALY, an open-source framework that can detect the implementation of eight hardening techniques in apps by combining automated static and dynamic analysis. We use HALY to analyze 1,836 popular Android and iOS apps and present the general prevalence of these hardening techniques, as well as prevalence in relation to several factors, such as app store category and access to privacy-sensitive permissions. Our research is the first work that combines research into the prevalence of multiple hardening techniques with analysis of multiple mobile platforms, namely Android and iOS. We conclude that hardening techniques are more prevalent on Android than on iOS, and that apps with more privacy-sensitive permissions implement more hardening techniques. Furthermore, we find that many apps implement hardening techniques on only one of the two OSes and that third-party libraries contribute significantly to the prevalence of hardening techniques. Overall, our study reveals that respectively 0.9% and 2.7% of the analyzed Android and iOS apps lack *all* the recommended self-protection mechanisms, 31.4% and 78.1% implement less than half of the studied hardening techniques, and only 1.5% of Android apps and no iOS apps adopt all the techniques that we studied.**

I. INTRODUCTION

Nowadays, many people rely on their phones for a variety of tasks, including sensitive operations such as finances. Mobile operating systems (OSes) such as Android and iOS employ various isolation and sandboxing mechanisms so that potentially malicious apps cannot access sensitive data of other apps. However, both malware and users themselves can try to circumvent these protective measures, for instance by rooting or jailbreaking a device. This can affect both the confidentiality of the data, as well as its integrity. For instance, a malicious app could steal sensitive data from other apps, or a cheating framework could tamper with game data [38]. In such a scenario, an app needs to rely on its own detection methods to make sure that it can protect itself and its data. To this end, several hardening techniques, sometimes also referred to as *Runtime Application Self-Protection (RASP)*, are available to protect an app from being reverse engineered, debugged, or otherwise tampered with. The OWASP Foundation has published a list of these hardening techniques in their Mobile Application Security Verification Standard (MASVS) [27], which has to be followed by apps on the Google Play Store when they opt for independent security review [15]. In particular the *Resilience* category of MASVS deals with validation of the platform and the implementation of anti-tampering and anti-static and -dynamic analysis techniques.

Even though, evidently, the implementation of these hardening techniques is important, there has been surprisingly little research into the prevalence of these techniques. There has been some research into the usage of hardening techniques for both Android and iOS separately [37], [52]. However, very different approaches were used, and different hardening techniques were investigated, making a direct comparison between these works difficult. Other existing works focused only on a specific hardening technique, such as root detection [11], [21], anti-tampering using SafetyNet [19], certificate pinning [32], or secure connections [34]. Furthermore, many works analyzed a relatively small dataset of a few dozen to a few hundred apps [11], [34], [52]. We present a summary of existing work in Table I.

Clearly, there is a lack of research into the adoption of different types of hardening techniques by both Android and iOS apps. Thus, it is not clear how common the general adoption of hardening techniques is, and how apps on Android and iOS compare to each other in this regard.

In this work, we collect and analyze a wide range of hardening techniques available in both the Android and iOS ecosystems. Excluding only code transformation techniques such as encryption and obfuscation, we design and implement an open-source automated *HArdening anaLYzer*, HALY, which tracks the implementation and adoption of as many hardening techniques as we could find. We study these hardening techniques in both Android and iOS apps, using both static and dynamic analysis.

In particular, HALY can detect the presence of eight different hardening techniques on Android, and seven different hardening techniques on iOS. Thus, it can detect more hardening techniques than related work, while also supporting both iOS and Android.

To gain a better understanding of the prevalence of hardening techniques, we use our framework to perform analysis on a large dataset of Android and iOS apps. To this end, we scrape the top 100 apps of each Apple App Store category and link these apps to the Android version of each app using a combination of automatic and manual matching. This results in a dataset of 1,843 popular apps that are available on both

TABLE I. COMPARISON OF HALY WITH RELATED ANALYSIS FRAMEWORKS AND THE HARDENING TECHNIQUES THEY CAN DETECT. * ANDROID ONLY.

	AppJitsu [52]	**Pradeep et al.** [32]	**Kellner et al.** [21]	**Ibrahim et al.** [19]	**Evans et al.** [11]	**Reaves et al.** [34]	**HALY**
Analysis type	**Dynamic**	**Static & Dynamic**	**Static & Dynamic**	**Static & Dynamic**	**Static & Dynamic**	**Static**	**Static & Dynamic**
OS support	Android	Android & iOS	iOS	Android	Android	Android	Android & iOS
Number of apps	455	5,079	3,482	163,773	35	46	3,672
Anti-tampering protection	Yes	No	No	Yes	No	No	Yes
Hooking detection	Yes	No	No	No	No	No	Yes
Debug detection	Yes	No	No	No	No	No	Yes
Emulation detection	Yes	No	No	No	No	No	Yes
Root/jailbreak detection	Yes	No	Yes	No	Yes	No	Yes
Keylogger protection	No	No	No	No	No	No	Yes
Screenreader protection	No	No	No	No	No	No	Yes*
Secure connections	No	Yes	No	No	No	Yes	Yes

iOS and Android. On each OS, we successfully analyze 1,836 of these apps, for a total of 3,672 apps.

After analyzing the dataset, we find that the prevalence of hardening techniques is higher on Android than on iOS. We also find that the prevalence of hardening techniques differs between app categories, and that apps with more privacy-sensitive permissions implement more hardening techniques. Furthermore, we find that many apps implement certain hardening techniques on only one of the two OSes. Lastly, we find that third-party libraries contribute significantly to the prevalence of hardening techniques.

Overall, we find that 0.9% and 2.7% of the analyzed Android and iOS apps do not implement *any* of the hardening techniques we studied. Furthermore, only 1.5% of Android apps and no iOS apps adopt all of the studied techniques.

In summary, we make the following contributions:

- We design and develop HALY, an open-source framework to analyze the usage of hardening techniques on both Android and iOS.
- We create a large, labeled dataset of popular apps that are available on both Android and iOS.
- We analyze the prevalence of hardening techniques in these apps, the results of which provide new insights into the similarities and differences between Android and iOS.
- We provide new insights into the relation between the presence of hardening techniques in apps and the usage of privacy-sensitive permissions, as well as their prevalence in first- and third-party code.

Our dataset, results, and source code of our framework are available at https://github.com/utwente-scs/haly-hardening-analyzer.

II. BACKGROUND

A. Threat model

There are several threats that apps can face that can move a developer to adopt hardening techniques in their apps [52]. First of all, the app can be executed by a malicious actor in a controlled environment set up for analysis and reverse engineering in order to find exploits in the app. For instance, a malicious actor might want to investigate if there is a vulnerability in a shopping app that allows one to place free orders. Secondly, the app can be executed on an end-user device with weakened security. For example, the device might have malware that tries to steal a session token from a bank app. A user can also willingly weaken security. For instance, a user might root their phone to be able to install game cheats or jailbreak their phone to be able to customize the home screen. We consider malicious attackers that attempt to reverse engineer and analyze an app and its behavior in a controlled environment, in order to obtain sensitive information, exploit vulnerabilities in the app, or break Intellectual Property (IP).

To protect against these threats, developers can implement several hardening techniques, which we describe in the next section. It should be noted that most hardening techniques can be bypassed. While some techniques require skilled actors to bypass, others can easily be bypassed using tools available on the internet. If developers want to protect their apps from these threats, they should thus implement *multiple* hardening techniques, with redundant checks for each hardening technique.

B. Hardening techniques

There are many techniques available to harden an app, i.e., make it more resilient to attacks such as reverse engineering and malware. Sihag et al. have performed a survey of the techniques currently in use [39]. They focus on the Android ecosystem in their paper, but most techniques can be applied to iOS too, although their implementation might differ. The OWASP Foundation has also provided a list of resilience requirements in their Mobile App Security Verification Standard (MASVS) that an app should implement [27]. They state that apps should validate the integrity of the platform and implement anti-tampering, anti-static analysis, and anti-dynamic analysis techniques. In this chapter, we discuss existing hardening techniques that are used for hardening legitimate apps and are relevant to this research. Table I shows which of these techniques have been explored in previous research.

Anti-tampering protection. Even on an unmodified and non-instrumented device, an app cannot assume to be running in a safe environment. An attacker can decompile, modify and recompile an app to add malicious behavior or bypass security mechanisms and spread this app outside the official mobile app stores. Especially on Android, re-packaging an app is relatively easy because apps can be installed from outside the Google Play Store on unmodified devices. To make it harder to re-package an app, it can use anti-tampering techniques. By performing an integrity check at the start of the app, such as validating the signature of the app, the app can be sure to be unmodified. It should be noted that this check can also be removed by adversaries. However, the implementation of anti-tampering protection does require a more skilled adversary to circumvent, especially when combined with other hardening techniques.

Hooking detection. Aside from modifying a binary itself, an attacker can also modify an app, or extract information from it, by using a hooking framework such as Frida [43], Cydia Substrate [36], or Xposed [35]. With these frameworks, one can specify functions or system calls (syscalls) to hook into, and read and modify arguments and return values at runtime, or completely replace a function with a new implementation.

Debug detection. Debuggers can be used by developers to resolve bugs in their apps. However, they can also be used by adversaries to reverse engineer an app. Native debugging can be performed using tools such as `ptrace` [37]. Android also allows for debugging of Java apps using the Java Debug Wire Protocol (JDWP) [39]. To thwart the effort of attackers, apps can block these debug methods or quit when detecting them.

Emulation detection. Attackers can make use of emulators to run and debug apps in a controlled and easy-to-modify environment, making reverse engineering easier. However, it is very difficult to emulate a device in such a way that it is indistinguishable from a real device. By checking for these differences in hardware and software configuration, an app can detect if it is running on an emulator [20], [22], [24], [44].

Root and jailbreak detection. A rooted Android device or jailbroken iOS device is a strong indicator of an insecure environment for an app. With such a device, the user of the device obtains control over the OS as a root user. This allows apps to break out of their normal sandboxed environment. They can then, for instance, read app data belonging to other apps or modify system files. Since this could reveal sensitive information processed by the app, many hardened apps cannot be executed on rooted or jailbroken devices or show a warning.

Keylogger protection. A malicious third-party keyboard can function as a keylogger and send any input typed by the user to an adversary. To prevent this from happening, an app can show a custom virtual keyboard for sensitive fields, optionally only in case a non-trusted keyboard is used.

Screenreader protection. An adversary can obtain sensitive information by using a malicious app that captures or records the screen. To prevent sensitive data from being exposed, apps can detect screenreaders and block screen captures.

Secure connections. An app does not only need to protect itself and its local data from adversaries but also the data communicated between the app and its servers. For this, it is important that data is encrypted when sent over a network, such that a man-in-the-middle attack cannot take place. To further increase security, an app can implement certificate pinning. By including a (hash of a) trusted certificate in the app's source code, it can make sure it is communicating with the party it is expected to communicate with, even in case an adversary has injected a self-generated CA certificate into the OS root store.

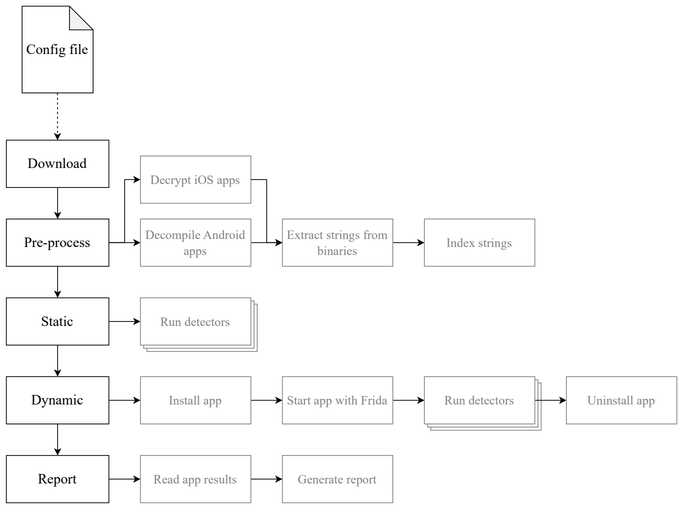

Fig. 1. System design overview

III. System design

In this work, we design an analysis framework that supports the detection of the hardening techniques discussed in Section II in both Android and iOS apps. To be able to take advantage of the strengths of both analysis techniques, our framework, HALY, combines static and dynamic analysis. Using static analysis, we can detect certain implementations that are difficult to detect during dynamic analysis, as well as hardening techniques that might only be executed during a specific situation that might not be encountered during dynamic analysis. Furthermore, during dynamic analysis, we can detect the execution of implementations—even if an app uses obfuscation. Combining these two techniques allows us to gain a better understanding of the general prevalence of hardening techniques in apps, mitigating potential false positives and negatives.

Overall, our framework consists of five main stages, namely downloading the apps, pre-processing them for analysis, static analysis, dynamic analysis, and reporting of results. An overview of the stages can be found in Fig. 1. Below, we describe the design of each of the stages.

A. *Downloading and pre-processing.*

Before HALY can analyze an app, it of course needs to download the app from the app store. It also collects metadata from the app store such as the app's category. After this is finished, it performs some pre-processing. Since iOS apps are encrypted when they are downloaded, an iOS app has to be decrypted first, so that our framework can effectively perform static analysis on it. Some initial data is then extracted from the app which can be used later during the static analysis. Furthermore, HALY creates an index of the pre-processed files so it can quickly search through them during static analysis.

B. Static analysis.

After downloading and pre-processing the apps, static analysis is performed. HALY disassembles the binaries of the apps and investigates relevant function calls. Furthermore, HALY searches for string patterns in the application, and parses the manifest file of the app to extract information about the app, such as the list of required permissions.

C. Dynamic analysis.

After static analysis has completed, HALY performs dynamic analysis of the app. HALY instruments the app and intercepts syscalls and other functions to detect what hardening techniques an app implements. During this process, our framework collects the parameters, return values, and stacktraces of the intercepted functions. This data can then be used at a later stage to calculate statistics on the prevalence of hardening techniques in apps, as well as how apps implemented these techniques.

D. Technique tracking.

Both static and dynamic analysis make use of a modular system of detectors, where each detector is responsible for detecting a specific hardening technique. This way, our framework can easily be expanded with detectors for more hardening techniques. In addition to the detection of hardening techniques, HALY also has a module that extracts general app information, such as its name, version, and permissions. Since we are using a rooted/jailbroken, instrumented phone for our dynamic analysis, we want to prevent the app that is being analyzed from fingerprinting our environment, to prevent an app from terminating after the first hardening technique is triggered. To this end, the detectors also implement countermeasures to circumvent the app's hardening techniques and trick the app into assuming it is executed in an unmodified environment.

E. Post-processing and reporting.

After analyzing the apps, the results of all the individual apps are merged into a report. Our framework uses the data from the app store and the results of the analysis to relate the detected presence of hardening techniques to various variables such as the app's category and permissions. Furthermore, it also categorizes the hardening technique implementations it detected as first-party or third-party implementations, depending on whether the specific implementation also occurs in other apps or only the analyzed app.

IV. SYSTEM IMPLEMENTATION

In this section, we describe the stages of our system and their implementation in more detail.

A. Downloading and pre-processing

In the download stage, HALY downloads the apps in the dataset from their mobile app store. Android apps are downloaded from the Google Play Store using gplay-downloader [5] and iOS apps are downloaded from the Apple App Store using IPATool [2]. After downloading, HALY pre-processes the apps for analysis. First, Android apps are disassembled to smali code using Apktool [42], and iOS apps are unzipped. Then, iOS apps are decrypted using a Frida [43] script that installs and opens the app on an iPhone and then dumps the decrypted binaries from the phone's memory. The encrypted binaries in the unzipped directory are then replaced by these decrypted binaries. After this, all human-readable strings are extracted from all binary files using Radare2 [33], and placed in text files alongside the binary files. Finally, Code Search [18] is used to index all text files, so HALY can quickly search through these files using regular expressions during static analysis.

B. Static analysis

During static analysis, HALY uses Code Search to search for the occurrence of specific strings, such as file or app names, as well as Java method calls in the smali code. Furthermore, Radare2 is used to find method calls in native binaries. HALY also uses Radare2 to identify SVC instructions in the binaries. SVC instructions can be used to directly invoke syscalls from native code, bypassing normal intercepts of hooking frameworks. Lastly, our framework also extracts some information from specific files by parsing them directly, such as the `AndroidManifest.xml` file on Android and the `Info.plist` file on iOS, which contain information such as the app's name and version.

C. Dynamic analysis

During dynamic analysis, HALY uses Frida to hook into various methods and syscalls. For some syscalls and methods, it also modifies the arguments or return value to try to hide to the app being analyzed that our phone is rooted/jailbroken and running Frida. Furthermore, the memory addresses of the SVC instructions that we found during static analysis are hooked. Whenever the SVC instruction is executed, HALY identifies the corresponding syscall by looking up the SVC's instruction number and processes the call in the same way as a 'normal' syscall. Unfortunately, solely relying on called functions to track the hardening techniques that apps adopt does have a downside, namely that we cannot track variables. For example, in Android apps, `Build` variables such as `Build.MODEL` are often used to check if the app is running on an emulator. Although HALY does track the usage of these variables during static analysis, to the best of our knowledge, Frida cannot track if these variables are accessed, and if so, what values they are compared against.

D. Technique tracking

As described in Section III, HALY makes use of a modular system of detectors. HALY makes use of the following detectors to track each hardening technique:

Info. During both static and dynamic analysis, HALY saves some basic information about the app. This includes information like the app's name, version, and requested permissions.

Anti-tampering protection. On iOS, there is no way to directly check the signature of an app. All apps from the Apple App Store are re-signed by Apple's signature, and iOS devices only allow for the execution of Apple-signed binaries, unless this check is disabled by a jailbreak. Because of this, checking the signature is not useful on non-jailbroken devices either way. Apple does provide an App Attest Service that should check if the app is unmodified [3]. HALY can detect the usage of this service.

On Android, HALY tracks functions related to retrieving and validating the signature of the app, as well as the usage of the SafetyNet Attestation API [16] or its successor Play Integrity API [14]. It should be noted that these APIs also perform more extensive checks on the integrity of the device, such as checking that the device is not an emulator. We thus classify apps that implement these APIs as having anti-tampering and root, hooking, and emulation detection.

Hooking detection. There are several hooking frameworks available, such as Xposed [35], Cydia Substrate [36], and Frida [43]. During static analysis, HALY checks if the names of these frameworks, or related apps or files, occur in the app's code or text files. During dynamic analysis, HALY tracks access attempts for files related to these frameworks, and if the app checks whether apps related to these frameworks are installed. Since jailbreaks almost always come with Cydia Substrate (or 'tweak') support, the distinction between hooking and root detection on iOS is a bit difficult. We have chosen to classify any mentions of "substrate" as hooking detection while classifying any other checks related to Cydia as root detection. Apps on iOS can also use `_dyld_get_image_name()` to check if there are any modules related to hooking frameworks loaded into the memory.

Debug detection. On iOS, the syscalls `ptrace` and `sysctl` can be used to check if the app is being traced. Furthermore, `getppid` can be used to validate the process ID of the parent process, which should always be 1 if the app is started by the launcher. HALY looks for and hooks these syscalls to see if they are used by the app. On Android, HALY also looks for `ptrace`, in addition to various Java methods such as `Debug.isDebuggerConnected()`.

Emulation detection. On iOS, an app can check the environment variables to verify whether it is running on an emulator or simulator. For example, the iOS simulator has the environment variable `SIMULATOR_MAINSCREEN_WIDTH` and the Corellium emulator has the environment variable `SANDBOX_TOKENS`. On Android, an app can use various variables from the `Build` class such as `Build.MODEL` to validate that it is not running on an emulator. Furthermore, an app can also check if certain files exist on the device that only exist on an emulator, such as `/Applications/Xcode.app` on an iOS simulator or `/dev/socket/genyd` on a Genymotion Android emulator. HALY tracks the usage of these variables and access of these files.

Root and jailbreak detection. There are many ways to check if a device is rooted or jailbroken. An app can, for example, check if certain files exist that are not present on stock devices, or check if file or directory permissions are different than they are on an unmodified device. During static analysis, HALY checks if the app contains any mention of these files, and during dynamic analysis, HALY hooks syscalls related to file access to track this. An app can also check if certain apps are installed related to rooting/jailbreaking. During static analysis, HALY looks for mentions of these apps, and during dynamic analysis, it tracks if any information about these apps is requested by hooking `openURL()` and `canOpenURL()` on iOS, and hooking methods of the `PackageManager` and `Intent` classes on Android.

Keylogger detection. To prevent keylogging, apps can show their own keyboard at an input field, or check if the active keyboard is part of a whitelist. HALY checks for the usage of functions related to hiding the system keyboard from an input field and functions for getting the active keyboard. On Android, functions like `EditText.setShowSoftInputOnFocus()` and `InputMethodManager.getEnabledInputMethodList()` can be used for this. On iOS, apps can use functions like `UIView.inputView()` and `UIResponder.textInputMode()`.

Screenreader detection. On iOS, there is no easy method available to block screenshots or -recordings. There are a few workarounds available that implement this, but it is difficult to detect these in an automated way. On Android, an app can set a part of the app as being "secure", which blocks screenshots. HALY checks for the setting of this secure flag using `SurfaceView.setSecure()` or `Window.setFlags()`.

Secure connections. During static analysis, HALY uses a similar methodology to Pradeep et al. [32] to detect certificate pinning. HALY checks for the inclusion of certificates or certificate hashes in the app. During dynamic analysis, HALY tracks the usage of known pinning functions of popular libraries functions. Furthermore, it also intercepts all network traffic during the dynamic analysis of the app. This traffic is then inspected to investigate whether the app makes use of plaintext network traffic or insecure TLS connections. Pradeep et al. found that background traffic can occur on iOS just after installing an app, so before starting the dynamic analysis on iOS, HALY waits two minutes for this traffic to cease. Note that, unlike Pradeep et al., we have not performed differential analysis, and only check for known pinning functions of popular libraries during dynamic analysis.

E. Post-processing and reporting

After the analysis is complete, our system starts a Flask server [29] that presents the aggregated results, as well as the analysis results of individual apps, to the user. During this stage, HALY also identifies if the detected hardening techniques are first-party or third-party implementations. For this, we first identify all third-party libraries. We categorize any library that is present in at least five apps as a third-party implementation. For native libraries, we use the file name as the identifier for this process, while we use the code path for Java libraries (e.g., `com.google.android.gms`). For each hardening technique implementation, we then identify if it is in this list of third-party libraries. During dynamic analysis, we use the stacktrace for this. After ignoring all system libraries in this stacktrace, we select the top item from the backtrace as the source of the hardening technique. It should be noted that the actual number of third-party implementations of hardening techniques may be higher since HALY would be unable to determine that a library is third-party in cases where obfuscation is used that changes the names or code paths of the library.

V. Experimental results

We now present our dataset of popular Android and iOS apps, describe our validation process, and use HALY to analyze our dataset of apps to assess the adoption of hardening techniques. First, we investigate the general prevalence of hardening techniques on both Android and iOS. Then, we directly compare the Android and iOS versions of apps to each other to gain insight into the development process of these hardening techniques in companies. Afterward, we investigate whether hardening techniques are implemented by third-party libraries or first-party code, and study the correlation between the presence of hardening techniques and the usage of privacy-sensitive permissions. Finally, we also present some insights into the adoption of TLS and the usage of plaintext network requests.

A. Dataset

To compare Android and iOS as fairly as possible, and also gain insight into the development methodology of companies, we construct and analyze a dataset of apps that are available on both iOS and Android. First, we select the top 100 apps in each category in the United States Apple App Store, which results in a dataset of 2,382 apps. Note that some apps are present in the top 100 of multiple categories. The apps are then linked to their Android versions. For this, we first use appranking.com. They provide a mapping between Android and iOS apps, but only for a relatively small subset of our dataset. We then use alternativeto.net, a crowdsourced website, to further find Android versions of the iOS apps in our dataset. Here, we crawl the first 100 pages of AlternativeTo's Android apps list and verify whether our iOS apps are included here. Finally, we manually link the remaining apps to their Android version. To this end, we query the Google Play Store for the iOS app's name and publisher, and manually select the corresponding Android app from the first five results, or none if none matches. Finally, we manually validate the dataset of Android and iOS apps obtained from the above three sources. The resulting dataset consists of 1,843 apps that have both an Android and iOS version.

In the dataset mentioned above, seven apps failed our static analysis on Android. We decided to remove both the iOS and Android version of these apps from our dataset since our dynamic analysis relies on data gathered during static analysis. This means we analyzed 1,836 apps per OS, so 3,672 apps in total.

B. Experimental setup

For our experiments, we use an iPhone 8 running iOS 16.4.1, which is jailbroken using the rootful palera1n jailbreak [28]. After this, we install the Sileo package manager [40] and use it to install `frida-server`, which we then start using SSH. Furthermore, we use a Pixel 3a running Android 12, which we root by installing Magisk [47]. We then install the Magisk module MagiskFrida [45], as well as Universal SafetyNet Fix [23] to pass SafetyNet.

After configuring our phones for analysis, we start a Squid proxy [41] on the computer running the dynamic analysis and configure the phones to use this proxy, such that we can intercept network traffic of the devices during the dynamic analysis. We intercept traffic using `tcpdump` with a filter on the IP address of the phone. We do not perform any man-in-the-middle attacks.

For our testing, we configure a timeout of 10 minutes and a memory limit of 8 GB per binary for the static analysis. Note that these limits are set for individual binary files within an app, and not for analyzing the app as a whole. Furthermore, we use a timeout of one minute for the dynamic analysis.

C. Validation

To minimize false positives and false negatives, we validate our dataset, framework, and results to the best of our abilities. As mentioned above, our dataset of input apps is entirely manually validated, and we are thus confident that we have managed to create a large dataset with a correct mapping between Android and iOS versions of popular apps. To validate our framework, we develop and analyze an Android and iOS app that triggers no hardening techniques (a blank Android Studio or Xcode project), as well as an Android and iOS app that triggers all hardening techniques. For all of these, HALY produces the expected results.

We also investigate false positives. During the analysis, HALY marks any low-confidence detections of hardening technique as uncertain. These cases are specifically discussed below. Because we can identify if function calls are low-confidence or not, false positives can only occur during the pattern-matching in our static analysis phase. We manually investigate all of the relevant results for false positives, and adjust our patterns and filtering accordingly. After this, we did not find any false positives in our pattern-matching results.

Finally, HALY is unable to fully analyze some (components of) apps, either using static or dynamic analysis. These issues are discussed below.

Static analysis. There are 11 Android and 36 iOS apps that contained one or more binaries that failed to complete static analysis because our timeout or memory limit was reached. On Android, the binaries that failed analysis are all third-party libraries, namely `libwebviewchromium.so`, `libUE4.so`, and `libili2cpp.so`. The first two libraries are present in one app each, while the last library is present in nine apps. On iOS, the binaries that failed analysis are a bit more diverse. The larger number of failed analyses can be explained by the fact that iOS binaries are generally larger and more complex than native Android libraries since most Android apps implement the majority of their logic using Java. There are 23 iOS apps in which the analysis of the main binary failed. Furthermore, there are 20 plugins and two frameworks that failed static analysis. There are two plugins that are present in two iOS apps and failed static analysis for both of them, namely `FileProvider.appexx` and `ShareExtension.appexx`. The rest of the frameworks and plugins were only present in one app.

Dynamic analysis. There are 46 Android apps and 70 iOS apps that failed to complete dynamic analysis without crashing or exiting before the timeout. We manually investigated why these apps might be crashing. First of all, there are 33 Android apps and 37 iOS apps that also crashed or exited when we executed them manually on our test devices. This indicates

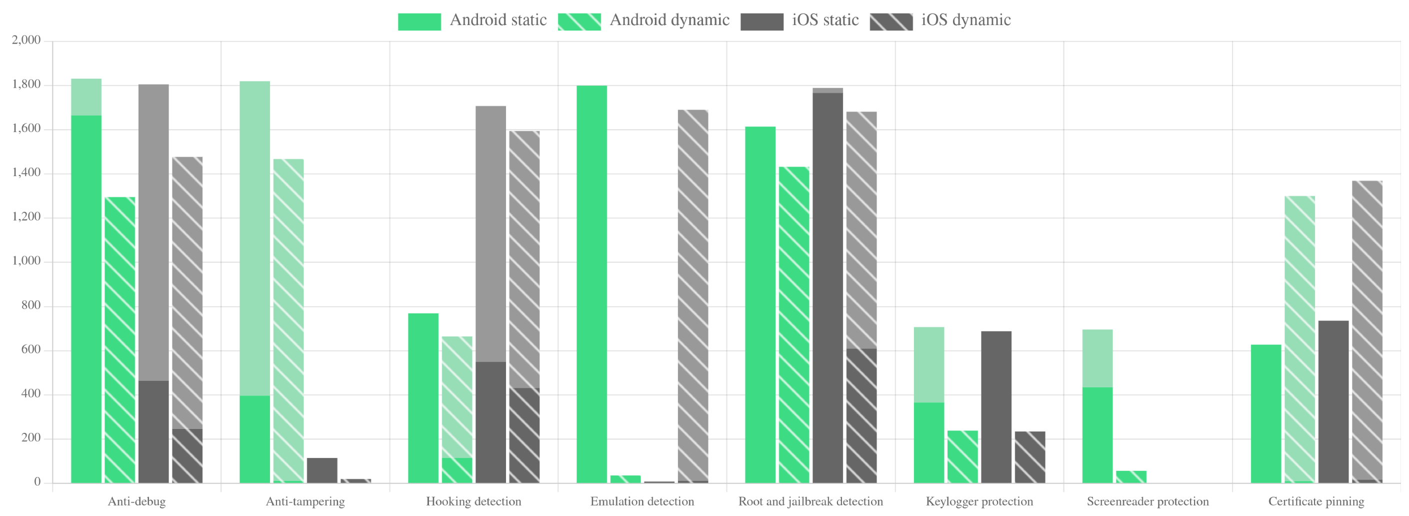

Fig. 2. Number of apps that each hardening techniques is detected for, using both static and dynamic analysis. The partially transparent bars indicate unconfident detections of the hardening technique. Note that screenreader protection was only analyzed for Android.

that these apps are not compatible with our devices, contain a bug that causes them to crash, or contain advanced root or jailbreak detection techniques that our framework is unable to circumvent. Furthermore, three Android apps exit as soon as a Frida session is connected to them, which indicates these apps have advanced Frida detection techniques that our framework is unable to circumvent. Finally, there are 10 Android apps and 33 iOS apps that crash whenever HALY tries to analyze them. This could be caused by the hooks we implemented, or general bugs in Frida. We were unable to determine the exact cause of these crashes.

D. *Prevalence of hardening techniques*

In Fig. 2, we present an overview of the prevalence of each of the hardening techniques that we analyzed. For each hardening technique, the chart shows in how many apps HALY detected this technique on Android and iOS, both for static and dynamic analysis. The partially transparent bars indicate low-confidence detections, which we further elaborate on below. On a general level, we can see that the prevalence of hardening techniques differs greatly between different techniques, as well as different OSes. Fig. 3 shows the cumulative distribution of the number of techniques adopted by Android and iOS apps, excluding screenreader protection, which is only supported for Android. Respectively, 0.9% and 2.7% of the analyzed Android and iOS apps do not implement any of the studied hardening techniques, 68.6% and 21.9% implement at least half of the tested techniques, and only 1.5% of Android apps and no iOS apps adopt all the recommended techniques.

Anti-tampering protection. There are two ways to implement anti-tampering protection. First of all, a developer can use an attestation framework. On Android, HALY detects the SafetyNet API in 9.5% of apps when using static analysis, but only in 0.5% during dynamic analysis. Its successor Play Integrity API is only detected in 1.4% of apps during static analysis, and not detected at all during dynamic analysis. However, the Play Integrity API has only been released since 2022. On iOS, HALY finds the App Attest Service in 6.3% of apps using static analysis and 1.1% of apps during dynamic analysis. The lower adoption of the App Attest Service could be explained by its relatively recent release in 2020 [31]. Interestingly, there are many apps that include an attestation service, which is however not detected during dynamic analysis. One would expect an app to execute attestation at the earliest possible moment. This could indicate that many apps planned to implement attestation, but did not complete the implementation, or that they only perform attestation in specific scenarios.

On Android, an app can also use the signature of the app to check if it is unmodified. HALY finds the `PackageManager::hasSigningCertificate()` function, which can be used to validate the app's signature, in 12.1% of apps using static analysis, but we only detect it in one app during dynamic analysis. Furthermore, we detect requesting of the app's signature in 99.1% of apps during static analysis, and 80.0% during dynamic analysis. However, manually looking through the results, this seems to be often used for fingerprinting purposes, and not necessarily for signature validation. For instance, we found that 28.6% of apps performed a signature retrieval from a function or class with the word "fingerprint" in its name.

Hooking detection. Using static analysis, we find hooking detection in 41.7% of Android apps and 29.9% of iOS apps. Interestingly, detection during dynamic analysis is much lower for Android apps, namely around 6.4%. We manually investigated this difference. We find that around 25.3% of apps include the library `com.appsflyer`, which performs both Frida and Xposed detection. We find that this library accesses the file `/proc/<pid>/maps` during dynamic analysis, which can be used to find traces of Xposed or Frida. However, this file can also be read for a different reason, which is why HALY marks it as an uncertain detection. In total, we detect access to this file in 23.9% of apps. In 12.8% of apps, we detect that this accessing originates from the `com.appsflyer` library. In the other results, we see also see a large number of false positives. Meanwhile, on iOS, hooking is detected in 23.5% of apps during dynamic analysis. The high number of uncertain results on iOS can be explained by the fact that around 91.4% of iOS apps use

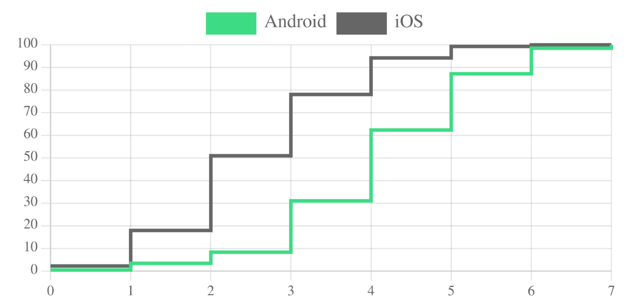

Fig. 3. CDF of the number of studied hardening techniques implemented by apps.

the _dyld_get_image_name or _dyld_image_count functions, which are quite commonly used to check for traces of hooking frameworks. However, these functions can also be used for other purposes. Note that most jailbreaks on iOS come with Cydia Substrate to allow users to install so-called 'tweaks', which use hooking to modify apps such as the home screen. Of the apps that implement jailbreak detection, 34.1% also implement detection of this hooking framework.

Unsurprisingly, detection of hooking frameworks that are only available on Android are only detected in Android apps. Xposed detection is present in 32.6% of Android apps. References to the lesser-known Android hooking frameworks Zygisk and Riru were only found in two apps and one app respectively. There are still quite a few Android apps that check for the Cydia Substrate framework, even though Cydia Substrate has not been in active development on Android after 2013 [17]. We find references to this framework in 12.6% of Android apps. Naturally, the detection of Cydia Substrate is much more prevalent on iOS, where we find it in 29.8% of apps. Using static analysis, we also reveal that Android apps implement detection of Frida more often than iOS apps, namely 26.9% vs. 3.8%. This difference can mostly be explained by the aforementioned com.appsflyer library.

Debug detection. Almost all Android apps implement some kind of debug detection. 82.9% of Android apps use the Debug::waitingForDebugger() or Debug::isDebuggerConnected() function. 44.5% of Android apps check if the developer settings are enabled, and 28.5% check if ADB is enabled. Furthermore, we find usage of ptrace in 37.8% of Android apps and 3.1% of iOS apps. 24.4% of iOS apps use the getppid function. Finally, HALY finds the sysctl function in 98.4% of apps, which can be used to check for the P_TRACED flag, but also for other purposes.

Emulation detection. Emulator detection is very common on Android and is present in 97.6% of Android apps. Build variables are used in 97.2% of apps to identify emulators. This also explains the large difference between static and dynamic analysis results, since the usage of these variables cannot be detected during dynamic analysis, as mentioned in Section IV. The most common Build checks can be found in Table II. Aside from the Build variable, 9.8% of Android apps check for the existence of emulator-related files.

Contrary, on iOS, emulator detection is very rare. During static analysis, HALY detects five apps that check for the iOS simulator using the /Applications/Xcode.app directory, and two apps that inspect the SIMULATOR_SHARED_RESOURCES_DIRECTORY environment variable. Furthermore, we only find one app that mentions Corellium-related files. However, none of these apps are present in our dynamic analysis results, which only include 12 apps that check for the CI_NO_CM and CI_PRINT_PROGRAM environment variables, which we identified as environment variables present on a Corellium emulator but not on a physical iPhone. The large number of uncertain results for dynamic analysis on iOS are retrievals of all environment variables. HALY cannot detect if these variables are then used for emulation detection or something else, but considering our other results, we consider the latter a more probable hypothesis for most of these retrievals.

TABLE II. MOST COMMON BUILD CHECKS ON ANDROID

Variable	Value	Occurence
FINGERPRINT	contains(generic)	89.3%
TYPE	equals(eng || userdebug)	82.8%
TAGS	contains(dev-keys || test-keys)	82.3%
TAGS	contains(test-keys)	69.7%
TAGS	contains(dev-keys)	69.1%
PRODUCT	contains(sdk)	66.3%
DEVICE	startsWith(generic)	61.1%
BRAND	startsWith(generic)	54.6%
MANUFACTURER	contains(Genymotion)	53.6%
HARDWARE	contains(goldfish || ranchu)	42.5%

The lack of emulator detection on iOS could be explained by the relatively recent developments in this area. The emulation of iOS devices is only widely possible since Corellium, the only iOS emulator available so far, opened its services to the public in 2021 [9]. Furthermore, before the release of the M1 processor, which uses the ARM architecture, it was impossible to run apps from the Apple App Store on an iOS Simulator from Apple, since the simulator only ran on x86 platforms, while iOS apps from the App Store run on the ARM architecture. In 2021, Giertler released a blog post on how to get ARM app binaries working on the iOS Simulator [12].

Root and jailbreak detection. We find root detection in 91.1% of Android apps and 96.5% of iOS apps. During dynamic analysis, we find that 77.6% of Android apps and 89.3% of iOS apps check for the existence of root-related files or validate the permissions of system directories, and 8.5% of Android apps and 3.3% of iOS apps check if root-related apps are installed. The most common apps and files apps check for can be found in Table III and Table IV respectively.

TABLE III. COMMON DETECTIONS OF ROOT- AND JAILBREAK-RELATED APPS.

Android	Occur.	iOS	Occur.
eu.chainfire.supersu	6.8%	cydia://	3.0%
com.noshufou.android.su	6.8%	sileo://	1.7%
com.koushikdutta.superuser	6.7%	undecimus://	1.6%
com.thirdparty.superuser	6.7%	blackra1n://	0.1%
com.topjohnwu.magisk	4.5%	Icy://	0.1%
com.devadvance.rootcloak	4.2%	RockApp://	0.1%
com.devadvance.rootcloakplus	4.2%		
com.ramdroid.appquarantine	3.9%		
com.noshufou.android.su.elite	3.8%		
com.yellowes.su	3.8%		

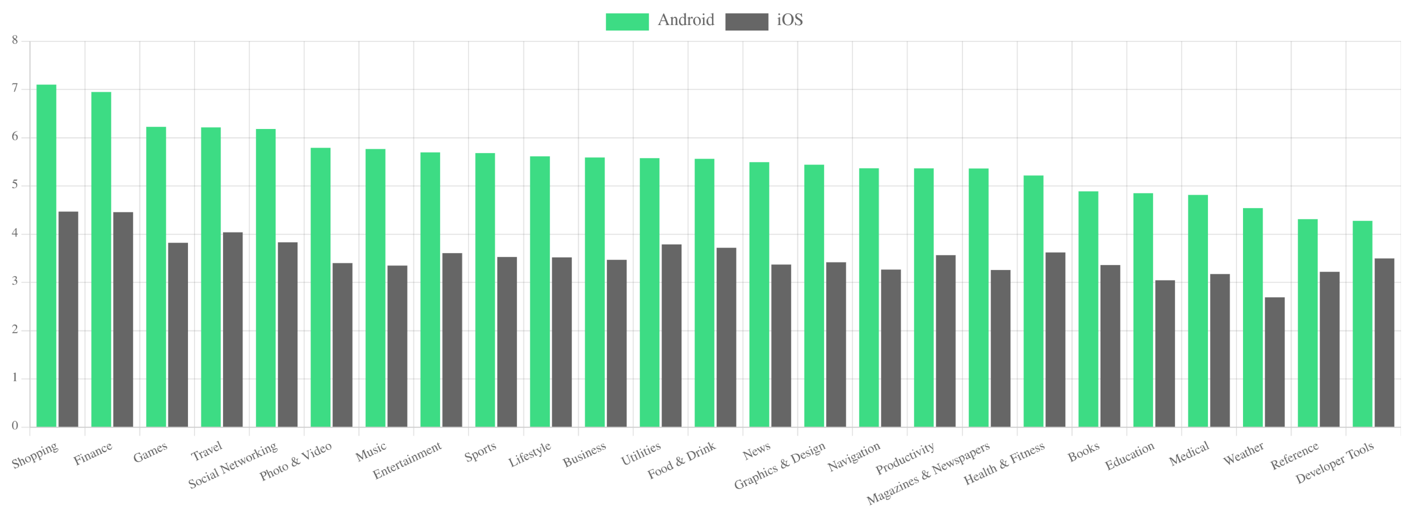

Fig. 4. Average number of hardening techniques implemented in an app per category.

Since jailbreaks for different iOS versions have distinct names, we can also investigate if an app checks for a specific jailbreak. We observe that apps with checks for specific jailbreaks mostly only check for the blackra1n and unc0ver jailbreaks, which were both quite popular jailbreaks. Checks for these jailbreaks were found in 10.7% and 4.0% of iOS apps respectively. Furthermore, there are a few apps that check for a larger selection of lesser-known jailbreaks, such as Pangu and Electra. Interestingly, we find no apps with detection for specific jailbreaks for iOS 13 or newer, indicating that jailbreak detection in apps is not updated very often after it has been implemented. Note that many apps use a more generic approach to detect jailbreaks, for instance by checking if a third-party app store is installed, if directory permissions differ from a non-jailbroken iPhone, or if a binary such as `apt` exists on the phone.

Keylogger protection. On Android, we find that 20.1% of Android apps and 38.4% of iOS apps query which input methods are enabled, which can be used to check if a trusted keyboard is used. Furthermore, 1.4% of Android apps disable showing the keyboard for some input fields and 6.8% of iOS apps change the `inputView` of an input field, which can be used to show a custom keyboard for the input field. The uncertainty in static analysis results for Android is caused by HALY not always being able to determine if a function is used to enable or disable showing the keyboard and which exact settings are retrieved. The dynamic results could be lower because the input view for which a custom keyboard is shown is not visible on the startup screen.

TABLE IV. COMMON DETECTIONS OF ROOT- AND JAILBREAK-RELATED FILES. * THESE DIRECTORIES ALSO EXIST ON NORMAL DEVICES, BUT MIGHT HAVE DIFFERENT PERMISSIONS.

	Android files	iOS files
1	/system/app/Superuser.apk	/private*
2	/system/xbin/su	/private/var/mobile/Containers*
3	/system/bin/su	/Applications/Cydia.app
4	/sbin/su	/bin/bash
5	/data/local/xbin/su	/Applications/RockApp.app
6	/data/local/bin/su	/Applications/Icy.app
7	/data/local/su	/Applications/blackra1n.app
8	/system/sd/xbin/su	/Applications/FakeCarrier.app
9	/system/bin/failsafe/su	/Applications/IntelliScreen.app
10	/su/bin/su	/Applications/MxTube.app

Screenreader protection. HALY detected 24.3% of Android apps set a view as "secure" to prevent screenshots or screen recording of that view. The much lower dynamic analysis results could be caused by apps only using this flag for views with sensitive information, and not for the main screen that is shown when the app starts. Furthermore, uncertainty in static analysis results is caused by HALY not always being able to determine if the secure flag or another flag is enabled. Unfortunately, there is no API available on iOS to prevent screenshots or screen recording, and HALY is not able to detect custom implementations of this.

Certificate pinning. HALY detects some form of pinning in many apps. We find certificates in 34.1% of Android apps and 40.3% of iOS apps. These certificates can be included as a certificate file or as the hash of a certificate. We find a certificate file in 22.5% of Android apps and 39.4% of iOS apps. Furthermore, we find certificate hashes in 18.3% of Android apps and 18.0% of iOS apps. It is unfortunately quite difficult to say if an app uses certificate pinning when utilizing detection analysis. In many cases, HALY can detect that a certificate pinning or connection security related function is called, but it was only able to confirm that certificate pinning was used in less than 1% of cases.

After looking at each hardening technique in detail, we want to zoom out and provide some insight into the prevalence of hardening techniques in relation to the category of apps, as well as the privacy-sensitive permissions they request.

App store categories. One would expect hardening techniques to be more prevalent in apps within certain categories, depending on the amount and sensitivity of privacy-sensitive information typically handled by apps in a category. In Fig. 4 we present the average number of hardening techniques implemented by apps in each category. Here, we considered a hardening technique as being present in an app if it was

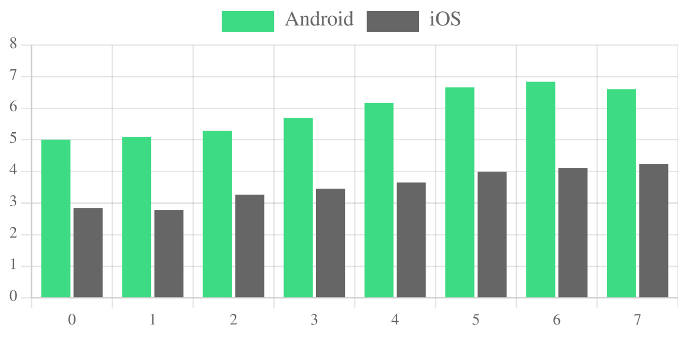

Fig. 5. Average number of hardening techniques implemented in an app depending on the number of privacy-sensitive permissions.

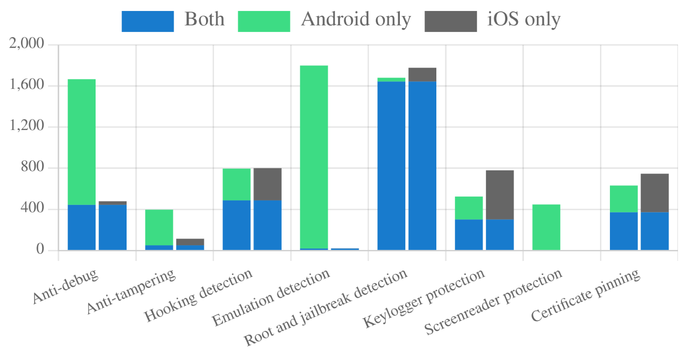

Fig. 6. Consistency of hardening techniques between OSes. Shows for how many apps, both the Android and iOS version or only the iOS or the Android version implement the hardening technique, or if neither version implements it.

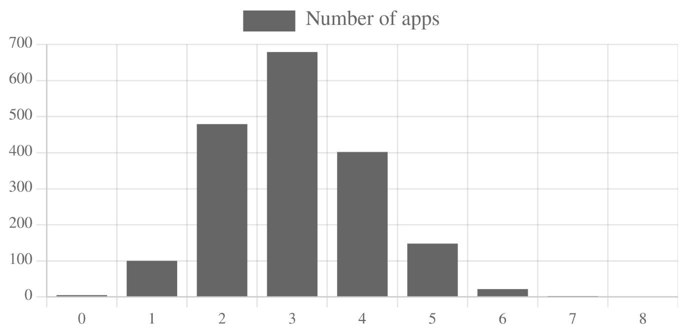

Fig. 7. Number of hardening techniques that are only implemented on one of the two OSes.

detected during static or during dynamic analysis. In general, the OSes follow a similar pattern. Especially the Finance and Shopping categories have a high average of implemented hardening techniques. In this graph, we can also clearly see that the number of implemented hardening techniques is generally lower on iOS than on Android. It should be noted here that HALY cannot detect screenreader protection on iOS, so this slightly skews these results towards Android apps having more hardening techniques. Removing screenreader protection from the results decreases the average number of hardening techniques on Android, but does not significantly change the prevalence of hardening techniques in different categories relative to each other.

Apps with privacy-sensitive permissions. Apps with access to privacy-sensitive data likely want to keep such information safe. To investigate this relation, we have identified eight categories of privacy-sensitive permissions, namely calendar, camera, contacts, location, microphone, health sensors, storage, and HomeKit access, where the last permission category is only relevant for iOS. For each app, we calculate the number of categories of privacy-sensitive permissions the app uses. In Fig. 5, we present the average number of hardening techniques for apps depending on the number of these privacy-sensitive permission categories. On both OSes, we can see a clear trend, namely that apps with more privacy-sensitive permissions also implement more hardening techniques. There are no results for iOS apps with all eight privacy-sensitive permission categories since there are no iOS apps in our dataset that request permissions from all eight categories.

E. Adoption comparison: Android vs. iOS

It is interesting to know whether the prevalence of hardening techniques differs between the iOS and Android versions of the same apps. This gives an indication if there are any company-wide policies to implement certain hardening techniques or if the implemented hardening techniques are strictly dependent on the OS. In Fig. 6, we present whether hardening techniques are implemented on both the Android and iOS versions of apps, or only on one of the two. Interestingly, we can see that there are quite a few apps that implement a hardening technique only on one of the two OSes, even for hardening techniques that are prevalent on both OSes. In Fig. 7, we present the difference between the number of hardening techniques implemented on the iOS and Android version of apps. Here, we exclude the screenreader protection hardening technique, since HALY cannot detect this on iOS. We can see that, for most apps, the variance in implemented hardening techniques between their Android and iOS version ranged from one to three. We find only five apps that implement the same hardening techniques on both OSes.

Overall, our results show a significant inconsistency in the implementation of the hardening techniques among the iOS and Android versions of the same apps, indicating either a disjunction among developers for the two OSes, together with a gap in the developer's expertise for a certain OS, or that certain techniques (e.g., anti-debug and emulation detection) are strictly related to and more documented for only one of the two OSes.

F. First-party vs. third-party implementations

In Fig. 8, we present the prevalence of hardening techniques in third-party libraries vs. first-party code. Note that apps may have both first-party implementations, as well as third-party libraries that implement hardening techniques. Interestingly, we can see that the majority of hardening techniques on Android originate from third-party libraries, while on iOS, the majority is implemented in first-party code. Tables V and VI show the most popular libraries that implement any hardening techniques, respectively for Android and iOS. There are three Android libraries that contribute a lot to this difference. Surprisingly, the hardening techniques that are

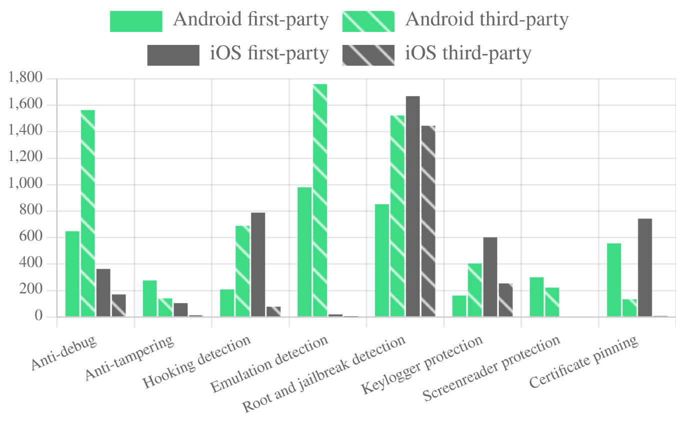

Fig. 8. Prevalence of hardening techniques in first-party vs. third-party code.

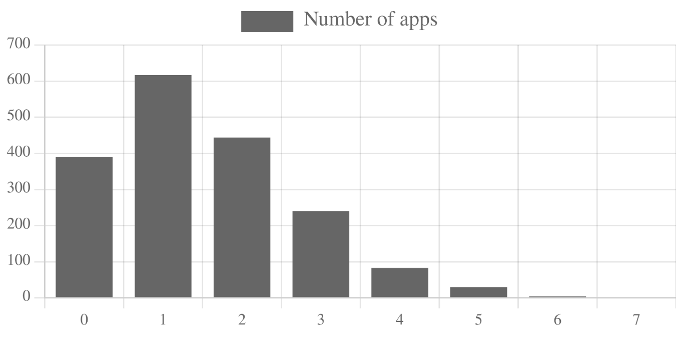

Fig. 9. Number of privacy-sensitive permissions that are only requested on one of the two OSes.

detected in these libraries differ per app. This might be caused by different versions of the libraries, or different parts of it being used.

The most-used Android library we find is *Google Mobile Services*, which is detected in 96.6% of Android apps. We find emulation detection in this library in 92.3% of apps and debug detection in 31.8% of apps. Next, the *Firebase* library is found in 89.0% of apps, which we find to implement root detection in 48.1% of apps, debug detection in 47.4% of apps, and emulation detection in 41.8% of apps. Finally, the *AppsFlyer* library, which is present in 25.9% of apps, implements hooking detection in 25.2% of apps.

Interestingly, there are no iOS libraries that jump out as implementing a hardening technique for any significant number of apps, apart from the root detection hardening technique. The main libraries responsible for this seem to be *Firebase* and *GoogleUtilities*, the latter of which is a utilities library for Firebase, among other libraries. Firebase-related libraries are detected in 38.0% of iOS apps. We find that these libraries are responsible for root detection in 37.1% of apps.

We find several libraries that seem to be specifically focussed on hardening apps, however, none of them are implemented by a significant number of apps. Some examples of these libraries are *RootBeer* (3.1% of Android apps), *TrustDefender* (2.7% of Android apps), *FraudForce* (2.3% of iOS apps), and *Forter* (1.7% of iOS apps).

G. *Usage of privacy-sensitive permissions.*

We also investigated how consistent apps are in the required privacy-sensitive permissions across the two OSes, the results of which we present in Fig. 9. We can see that many apps request different permissions across their iOS and Android versions. We find that only 21.2% of apps use the same privacy-sensitive permissions on both OSes. Most apps have one or two privacy-sensitive permissions that are only used on one of the two OSes.

We created a mapping between iOS and Android permissions by combining the protected resources [4] in the iOS documentation with the manifest permissions [13] in the Android documentation. Note that there is no perfect one-to-one match between Android and iOS permissions, so an app might have the same functionality on both OSes but still require different permissions. Furthermore, while on Android, the developer explicitly defines the needed permissions, no such system is present on iOS. Here, permissions are requested dynamically when a function is called that needs a permission. Developers do, however, need to provide a description of why they request a privacy-sensitive permission in the plist file. We detect the presence of these descriptions. An app might also define the permission in the manifest, or provide a description in the plist file, but not actually use the permission.

We manually investigated the apps showing a difference

TABLE V. MOST POPULAR THIRD-PARTY ANDROID LIBRARIES WITH IMPLEMENTED HARDENING TECHNIQUES. * FULL NAME: AUDIENCE_NETWORK.COM.FACEBOOK.ADS.REDEXGEN.X.

Library	**Occurrence**	**Anti-debug**	**Anti-emulation**	**Anti-hooking**	**Anti-keylogger**	**Pinning**	**Anti-root**	**Anti-tamper**
com.google.android.gms	96.6%	62.7%	92.3%	1.4%	—	—	51.3%	0.5%
com.google.firebase	89.0%	47.5%	41.8%	5.7%	—	—	48.3%	5.7%
com.google.android.play	57.9%	—	38.8%	—	—	—	—	—
com.facebook	52.9%	1.2%	36.5%	0.5%	—	—	16.1%	0.1%
okhttp3	38.3%	—	—	0.1%	—	0.3%	0.2%	—
com.squareup.picasso	30.1%	—	—	—	—	—	—	—
com.appsflyer	25.3%	—	—	25.2%	—	—	0.6%	—
libcrashlytics-common.so	15.6%	15.6%	—	—	—	—	—	—
bo.app	14.5%	—	—	—	—	—	14.5%	—
audience_network.com*	13.6%	13.4%	—	—	—	—	—	—
com.applovin	11.1%	0.8%	11.0%	1.0%	0.2%	—	0.9%	0.1%

TABLE VI. MOST POPULAR THIRD-PARTY IOS LIBRARIES WITH IMPLEMENTED HARDENING TECHNIQUES.

Library	**Occurrence**	**Anti-debug**	**Anti-emulation**	**Anti-hooking**	**Anti-keylogger**	**Pinning**	**Anti-root**	**Anti-tamper**
GoogleUtilities	37.7%	—	—	—	—	—	34.3%	—
GoogleDataTransport	32.4%	0.1%	—	—	—	—	23.9%	—
FirebaseCore	30.9%	—	—	—	—	—	24.1%	—
FirebaseCrashlytics	27.7%	—	0.1%	—	—	—	25.4%	—
FBSDKCoreKit	26.1%	—	—	—	—	—	7.5%	—
FBLPromises	16.8%	—	—	—	—	—	0.1%	0.3%
FirebaseCoreDiagnostics	15.4%	0.1%	—	—	—	—	15.3%	—
GTMSessionFetcher	15.2%	—	—	—	—	—	15.2%	—
FirebaseRemoteConfig	15.4%	—	—	—	—	—	14.0%	—
SDWebImage	11.6%	0.1%	—	—	—	0.1%	11.5%	—

of at least five permissions. In all cases, the iOS version of the app requested more permissions than the Android version. An example of one of these apps is *Audiomack*. One of the permissions present on the iOS version of this app but not on the Android version, is location access. On iOS, this permission is used to "show popular music in your area".

H. Adoption of TLS.

During dynamic analysis, we capture the network traffic of apps. From this data, we find that 5.8% of Android apps and 58.5% of iOS apps contact a webserver over unencrypted HTTP. On iOS, most of these connections are used for Online Certificate Status Protocol (OCSP) stapling or other verification of certificates. It should be noted that OCSP stapling commonly takes place using plaintext HTTP since requiring a TLS connection with OSCP validation for OCSP itself can create a deadlock. OCSP responses are signed to prevent modification. The most commonly accessed domains without encryption are *r3.o.lencr.org* from Let's Encrypt and *ocsp.digicert.com* from DigiCert. If we filter out all plaintext network traffic related to OCSP stapling, we are left with 8.7% of iOS apps using plaintext HTTP, which is still more than Android, but significantly less than 58.5%. We observe that, aside from OCSP traffic, images are the most common resource requested using plaintext HTTP (around 43% of non-OSCP requests), followed by API requests (around 31%), followed by fonts (around 17%).

Encryption of network traffic. The `TLS_AES_128_GCM_SHA256` cipher is the most used cipher for TLS connections, namely in 59.1% of connections on Android and 64.5% on iOS. This cipher is considered *Modern* by Mozilla. The second and third most used ciphers were `ECDHE-RSA-AES128-GCM-SHA256` (18.8% on Android, 14.6% on iOS) and `TLS_AES_256_GCM_SHA384` (9.9% on Android, 16.5% on iOS), which are considered *Modern* and *Intermediate* respectively by Mozilla. Less than 0.3% of connections on Android and less than 0.2% of connections on iOS used a cipher considered *Old* by Mozilla.

VI. Related work

In this section, we discuss related work and highlight the differences with our work. Here, will show that our analysis is not just more extensive, but also leads to results that often deviate significantly from earlier work thanks to our unique detection analysis approach. An overview of the most closely related studies and their features can be found in Table I.

The most closely related work to this research is RaspScan [37] and AppJitsu [52]. RaspScan is a framework for iOS that uses static and dynamic analysis to detect several hardening techniques using a similar methodology to our work. For static analysis, the framework radare2 [33] is used and for dynamic analysis, RaspScan makes use of Frida [43]. It can detect the use of hooking, debug, and jailbreak detection using static and dynamic analysis, as well as the use of SSL pinning using static analysis, by detecting the use of specific strings and syscalls. Furthermore, it also replaces return values of syscalls to bypass jailbreak and hooking detection, such that analysis can take place on a jailbroken iPhone running Frida.
J. Seredynski found, using his RaspScan framework, that around 73% of iOS apps implemented jailbreak, debug, or hooking detection. Debug detection was found in around 10% of apps by RaspScan, and hooking detection using `_dyld_get_image_name()` and `_dyld_image_count()` was found by RaspScan in around 60% of apps. Certificate pinning is found in around 15% of apps by RaspScan. Here, they only implemented the detection of pinning-related functions, and not (the hashes of) certificates. Even though our work generally uses a similar methodology, we observe a significantly higher prevalence of all these hardening techniques. This could be caused by differences in our dataset, but also our more extensive detection of hardening techniques.

AppJitsu [52] is a framework for Android that only uses dynamic analysis. It can detect the use of signature verification, as well as hooking, debug, emulation, and root detection. Contrary to RaspScan and our work, it does not use hooking during analysis but instead uses multiple environments to perform differential analysis. It detects hardening techniques by checking if an app runs differently on a phone or emulator, and with or without root, Frida, or debug tools. They found that 19% of apps have anti-tampering protection, which is close to our result of at least 21% of Android apps. Emulator detection was found in 25% of apps, 36% of apps failed to run on a rooted emulator, and 49% of apps failed to run on a rooted emulator with Frida running. Finally, 36% of apps failed to run on an emulator with a debugger attached. These are all much lower percentages than we find in this work. A plausible explanation for this is that apps might implement hardening techniques for statistical purposes, to differentiate between debugging and normal code, or to fingerprint the device, without actually terminating the execution of the app or otherwise changing its behavior. This is further supported by the fact that we observed that a large portion of the detected hardening techniques on Android originates from third-party libraries that are not necessarily used for security purposes.

Multiple studies have been performed on the prevalence of individual hardening techniques. Pradeep et al. [32] performed a comparison of certificate pinning and connection security between iOS and Android apps. They use static analysis to analyze a dataset of 5,079 apps and find certificates and certificate hashes in the app binaries, indicating the use of TLS pinning, which they find in 19.7% of Android apps and 33.4% of iOS apps, which is a significantly lower percentage than we find in this work. They further enhance these results by utilizing differential dynamic analysis. By first running the app in a normal environment and then with mitmproxy and a certificate added to the root store, the traffic can be compared to detect TLS pinning. Using this methodology, they find that only 6.7% of Android apps and 11.4% of iOS apps use TLS pinning. They also investigated the usage of the Network Security Configuration file to implement certificate pinning and observed that 1.8% of popular Android apps used this method. We find a slightly higher number, namely 3.1%.

Ibrahim et al. [19] performed research on the prevalence of SafetyNet in Android apps. They analyze 163,773 popular apps. They first use static analysis to filter out apps without SafetyNet and then use dynamic analysis to identify apps that invoke SafetyNet attestation. They only find 62 apps that invoke SafetyNet attestation, and no apps that implement the SafetyNet API in a fully correct way. Even though we found a higher number of apps that implement SafetyNet using static analysis, our dynamic analysis also suggests that most apps have not completely correctly implemented SafetyNet.

Kellner et al. [21] analyzed jailbreak detection in 34 banking apps on iOS. During their study, they also researched the general prevalence of jailbreak detection in a dataset of 3,482 popular apps. They find that 59% of popular apps and 53% of banking apps implement jailbreak detection, and that the prevalence of jailbreak detection in banking apps thus does not exceed the average prevalence in popular apps.

Evans et al. [11] performed a study on the prevalence of root detection in Android apps. They analyzed 16 security apps and 19 enterprise mobile device management apps. They find that 13 security apps and 15 device management apps with root detection, but also manage to develop a framework that can bypass root detection in all of these apps. Since this study is from 2015, the prevalence of root detection in these apps might have changed in the meantime.

Reaves et al. [34] have investigated the communication security of branchless banking applications. They investigate 46 Android apps and find that the majority of these apps fail to properly protect the financial information they are handling.

Although this is not the focus of this research, much research has also been performed into the usage of obfuscation and packing to thwart the reverse engineering of malicious or legitimate apps. There are many studies on developing identification techniques for obfuscation in apps [6], [7], [25], [26], [30]. Furthermore, Wang et al. [46] specifically investigate the prevalence of obfuscation in popular iOS apps. They analyze 6,600 apps and identify 601 versions of 539 unique apps that implement obfuscation. Another method to make it harder to analyze apps is the use of packing. Furthermore, there are several studies on the identification and reversing of packing [10], [48]–[51]. Xue et al. [49] also used their packing detector Happer to investigate the prevalence of packing in legitimate Android apps and found 1,710 apps that used packing in a dataset of 24,031 apps.

It is also worth mentioning that there are several analysis frameworks available that can be used by security researchers to investigate the security of apps, as well as the usage of certain hardening techniques. A popular open-source framework for this purpose is MobSF [1].

In conclusion, this work differs from related work by implementing the detection of more hardening techniques and utilizing static and dynamic analysis to investigate the prevalence of these techniques on both Android and iOS using a large dataset of apps. Thus, our work provides a more extensive insight into the prevalence of hardening techniques, and the difference between both OSes.

VII. Limitations & Future work

In this work, we have studied the prevalence of several hardening techniques in both Android and iOS apps. However, there are some limitations to our work, and some interesting possibilities for future research.

Technical limitations. Our analysis approach brings some technical limitations with it. Since we depend on the detection of the usage of specific method calls and fields in apps, our framework might miss certain hardening techniques that are implemented in an unconventional way. Furthermore, our static analysis is not always able to determine if a method calls corresponds to the implementation of a hardening technique if this depends on the arguments passed to the method, or how the return value is processed. Finally, our dynamic analysis cannot detect accessing of variables, and cannot detect how an app processes the return value of a function. These limitations can partially be resolved by using more advanced static analysis techniques. Furthermore, future work can look into how the results of static analysis can be used during dynamic analysis to determine if a certain method call is indeed related to a hardening technique.

Differential analysis. Since we only perform detection analysis instead of differential analysis, our framework is unable to determine how an app responds to the detection of a hardening technique. It could, for instance, only be used during the collection of statistics about the device, or to fingerprint the device. This hypothesis also explains the difference between our results and the results of related work that use a differential analysis methodology. To overcome this limitation, our framework could be expanded by adding differential analysis and combining the results of both dynamic analysis techniques. This way, one can investigate very specifically what hardening techniques an app implements, as well as the effect this has on the app's behavior.

Privacy leakage. In our work, we investigated the relation between access to privacy-sensitive permissions and the usage of hardening techniques and noticed a correlation between them. Self-protection techniques can be a double-edged sword as they can also be adopted by apps to complicate their analysis and hide malicious behavior. For example, apps have been shown to adopt obfuscation, or other hardening techniques, to hide the leaking of private information [8]. It would thus be interesting to further investigate the relation between the usage of hardening techniques and the processing of privacy-sensitive information, as well as privacy leakage. This way, we can gain a better understanding of the motivation for developers to implement hardening techniques, namely if they do so to protect sensitive information, or to hide leakage of it.

Detectable hardening techniques. There are two more hardening techniques that were not the focus of this study, namely code obfuscation and device binding [39]. Code obfuscation can take many forms, and the implementation of device binding is highly app-specific, which makes them both difficult to detect automatically. Thus, investigating the prevalence of these techniques requires more research, and was considered out-of-scope for this work.

Hardening bypassing. To be able to run our dynamic analysis, HALY tries to bypass root and hooking detection. However, these bypasses are not perfect, and some apps are still able to detect that the device is rooted or that Frida is running. Because of this, there may be a few apps that did not execute all their hardening checks. Furthermore, some apps failed dynamic analysis since they quit after detecting Frida. Since this only happens for a small number of apps in our dataset (less than 5% of the apps), this does not affect the significance of our findings. However, more research into the detection and circumvention of these advanced hardening techniques would be interesting for future work.

Dataset. Although we analyzed a large dataset of apps, our dataset was limited to popular apps that are available on both Android and iOS. It would be interesting to analyze a larger dataset that also includes (far) less popular apps, as well as apps only available on one of the OSes, and investigate how this changes the results for the prevalence of hardening techniques.

App exploration. During our analysis, we did not use any app exploration. We assume that an app wants to run its hardening techniques at the earliest possible stage and that we will thus be able to detect most hardening techniques without any app interaction. Furthermore, Pradeep et al. [32] found that random interactions made no significant changes to the resulting network traffic. However, targeted app interactions such as creating an account could influence results and would be interesting for future work.

Validation. We were unable to manually verify all of our results. To the best of our abilities, we have minimized false positives and false negatives. However, a small number of false positives and negatives likely still remains.

First-party vs. third-party. In our work, we have investigated the difference in the prevalence of hardening techniques between first-party and third-party code. However, we have not investigated whether these libraries were added to harden the app or for other purposes. Future research should further investigate the usage of these libraries, as well as their purposes.

VIII. Key takeaways

Large dataset of iOS/Android apps. We created a high-quality labeled dataset of 1,843 apps that are available on both Android and iOS. This dataset can be used for future research into the prevalence of hardening techniques, as well as other research topics.

Prevalence of hardening techniques. We show that some hardening techniques are more prevalent than others. Specifically, root and jailbreak detection are present in most Android and iOS apps, and emulator and anti-debug detection are present in most Android apps.

Prevalence on iOS vs. Android We show that the prevalence of hardening techniques is lower on iOS than on Android.

Lack of full adoption. Our results show that, of the analyzed Android and iOS apps, respectively 31.4% and 78.1% of the analyzed Android and iOS apps implement less than half of the recommended hardening techniques. Furthermore, only 1.5% of Android apps and no iOS apps adopt all the techniques that we studied.

Prevalence in different categories. We show that the prevalence of hardening techniques differs between app categories. Specifically, we show that apps in the Finance and Shopping categories implement more hardening techniques than apps in other categories.

Prevalence in apps with privacy-sensitive permissions. We show that apps with more privacy-sensitive permissions implement more hardening techniques. This indicates that developers are more likely to implement hardening techniques if they handle privacy-sensitive information. Furthermore, we show that the usage of privacy-sensitive permissions often differs between Android and iOS apps.

Consistency of hardening techniques. We show that many apps implement hardening techniques on only one of the two OSes. More specifically, we show that most apps, namely 84.6%, have two to four hardening techniques that are only implemented on one OS. This can indicate that there are not always company-wide policies to implement certain hardening techniques, that developers have more knowledge on implementing hardening techniques for one of the OSes, or that some hardening techniques are more documented for only one of the two OSes.

First vs. third-party implementations. We show that the difference in the prevalence of hardening techniques between iOS and Android is largely caused by third-party libraries. Furthermore, we show that, on Android, hardening techniques are more often present in third-party libraries than in first-party code, while on iOS this is the other way around.

IX. Conclusion

In this work, we studied the adoption of hardening techniques in both Android and iOS apps. To this end, we first reviewed the common hardening techniques available to mobile app developers and then implemented an automated universal framework, HALY, that can automatically detect these hardening techniques on both mobile OSes, using a combination of static and dynamic analysis. We then used HALY to analyze a large dataset of 1,843 popular apps available on both OSes. Our results show that hardening techniques are more prevalent on Android than on iOS, and that adoption of hardening techniques differs between app categories, with categories such as finance and shopping implementing more hardening techniques than other categories. Furthermore, we show that apps with more privacy-sensitive permissions implement more hardening techniques and that the usage of privacy-sensitive permissions differs between Android and iOS apps. Finally, we show that many apps implement hardening techniques on only one of the two OSes, and that third-party libraries significantly contribute to the prevalence of hardening techniques in apps. Overall, our study shows that respectively 0.9% and 2.7% of the analyzed Android and iOS apps lack *all* the recommended self-protection mechanisms, 31.4% and 78.1% implement at less than half of the studied hardening techniques, and only 1.5% of Android apps and no iOS apps adopt all the techniques that we studied.

Acknowledgment

I would like to thank everyone who helped me with this research by providing support, ideas, and feedback. I would especially like to thank my supervisor Andrea Continella for his guidance and feedback. Furthermore, I would like to thank my second supervisor Ralph Holz, as well as Jan Seredynski for providing me with the source code of his master's thesis, RaspScan, and Jacco Brandt for explaining it to me. Finally, I would like to thank Magdalena Steinböck, Herbert Bos, Martina Lindorfer, and Erik van de Kouwe for their help during this thesis.

References

[1] A. Abraham, Magaofei, M. Dobrushin, and V. Nadal, "Mobile Security Framework (MobSF)," 2023. [Online]. Available: https://github.com/MobSF/Mobile-Security-Framework-MobSF

[2] M. Alfhaily, "IPATool," 2023. [Online]. Available: https://github.com/majd/ipatool

[3] Apple Inc., "Establishing your app's integrity | Apple Developer Documentation." [Online]. Available: https://developer.apple.com/documentation/devicecheck/establishing_your_app_s_integrity

[4] ——, "Protected resources | Apple Developer Documentation." [Online]. Available: https://developer.apple.com/documentation/bundleresources/information_property_list/protected_resources

[5] I. Avci, "gplay-downloader," 2023. [Online]. Available: https://github.com/ikolomiko/gplay-downloader

[6] A. Bacci, A. Bartoli, F. Martinelli, E. Medvet, and F. Mercaldo, "Detection of Obfuscation Techniques in Android Applications," in *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*, 2018.

[7] M. Conti, V. P., and A. Vitella, "Obfuscation detection in Android applications using deep learning," *Journal of Information Security and Applications*, vol. 70, 2022.

[8] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna, "Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[9] R. Daws, "Corellium enables iOS device virtualisation on individual accounts," 2021. [Online]. Available: https://www.developer-tech.com/news/2021/jan/26/corellium-enables-ios-device-virtualisation-individual-accounts/

[10] Y. Duan, M. Zhang, A. Bhaskar, H. Yin, X. Pan, T. Li, X. Wang, and X. Wang, "Things You May Not Know About Android (Un)Packers: A Systematic Study based on Whole-System Emulation," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[11] N. S. Evans, A. Benameur, and Y. Shen, "All your Root Checks are Belong to Us: The Sad State of Root Detection," in *Proceedings of the ACM International Symposium on Mobility Management and Wireless Access*, 2015.

[12] B. Giertler, "Hacking native ARM64 binaries to run on the iOS Simulator," 2021. [Online]. Available: https://bogo.wtf/arm64-to-sim.html

[13] Google LLC, "Manifest.permission | Android Developers." [Online]. Available: https://developer.android.com/reference/android/Manifest.permission

[14] ——, "Play Integrity API | Google Play." [Online]. Available: https://developer.android.com/google/play/integrity

[15] ——, "Provide information for Google Play's Data safety section." [Online]. Available: https://support.google.com/googleplay/android-developer/answer/10787469#independent_security_review

[16] ——, "SafetyNet Attestation API | Android Developers." [Online]. Available: https://developer.android.com/training/safetynet/attestation

[17] ——, "Cydia Substrate - Android Apps on Google Play," 2013, archived at https://web.archive.org/web/20170110195857/https://play.google.com/store/apps/details?id=com.saurik.substrate. [Online]. Available: https://play.google.com/store/apps/details?id=com.saurik.substrate

[18] ——, "codesearch: Fast, indexed regexp search over large file trees," 2020. [Online]. Available: https://github.com/google/codesearch

[19] M. Ibrahim, A. Imran, and A. Bianchi, "SafetyNOT: On the Usage of the SafetyNet Attestation API in Android," in *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2021.

[20] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu, "Morpheus: automatically generating heuristics to detect Android emulators," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2014.

[21] A. Kellner, M. Horlboge, K. Rieck, and C. Wressnegger, "False Sense of Security: A Study on the Effectivity of Jailbreak Detection in Banking Apps," in *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[22] J.-d. Lim, I.-k. Kim, N. Kim, B. Kang, and S.-j. Cho, "A Study on Android Emulator Detection Using Build Properties," in *Proceedings of the International Conference on Next Generation Computing*, 2022.

[23] D. Lin, "Universal SafetyNet Fix," 2023. [Online]. Available: https://github.com/kdrag0n/safetynet-fix

[24] J. Lin, C. Liu, and B. Fang, "Out-of-Domain Characteristic Based Hierarchical Emulator Detection for Mobile," in *Proceedings of the International Conference on Information Technologies and Electrical Engineering*, 2020.

[25] O. Mirzaei, J. M. de Fuentes, J. Tapiador, and L. Gonzalez-Manzano, "AndrODet: An adaptive Android obfuscation detector," *Future Generation Computer Systems*, vol. 90, 2019.

[26] A. Mohammadinodooshan, U. Kargén, and N. Shahmehri, "Robust Detection of Obfuscated Strings in Android Apps," in *Proceedings of the ACM Workshop on Artificial Intelligence and Security*, 2019.

[27] OWASP Foundation, Inc., "OWASP MASVS v2.0.0," 2023. [Online]. Available: https://github.com/OWASP/owasp-masvs/releases/tag/v2.0.0

[28] palera1n, "palera1n." [Online]. Available: https://palera.in/

[29] Pallets, "Flask," 2023. [Online]. Available: https://github.com/pallets/flask

[30] M. Park, G. You, S.-j. Cho, M. Park, and S. Han, "A Framework for Identifying Obfuscation Techniques applied to Android Apps using Machine Learning," *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, vol. 10, no. 4, 2019.

[31] M. Peterson, "iOS 14 introduces new 'App Attest' API to cut down on app fraud," 2020. [Online]. Available: https://appleinsider.com/articles/20/08/18/ios-14-introduces-new-app-attest-api-to-cut-down-on-app-fraud

[32] A. Pradeep, M. T. Paracha, P. Bhowmick, A. Davanian, A. Razaghpanah, T. Chung, M. Lindorfer, N. Vallina-Rodriguez, D. Levin, and D. Choffnes, "A comparative analysis of certificate pinning in Android & iOS," in *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2022.

[33] radare.org, "Radare2: Libre Reversing Framework for Unix Geeks," 2022. [Online]. Available: https://github.com/radareorg/radare2

[34] B. Reaves, J. Bowers, N. Scaife, A. Bates, A. Bhartiya, P. Traynor, and K. R. B. Butler, "Mo(bile) Money, Mo(bile) Problems: Analysis of Branchless Banking Applications," *ACM Transactions on Privacy and Security (TOPS)*, vol. 20, no. 3, 2017.

[35] rovo89, "Xposed framework," 2017. [Online]. Available: https://github.com/rovo89/Xposed

[36] SaurikIT, LLC, "Cydia Substrate." [Online]. Available: http://www.cydiasubstrate.com/

[37] J. Seredynski, "Is your iOS app secure? An automated security evaluation of AppStore applications," 2020.

[38] ——, "Demystifying typical mobile game cheats," https://www.guardsquare.com/blog/demystifying-typical-mobile-game-cheats, 2021.

[39] V. Sihag, M. Vardhan, and P. Singh, "A survey of android application and malware hardening," *Computer Science Review*, vol. 39, 2021.

[40] Sileo Team, "Sileo." [Online]. Available: https://getsileo.app/

[41] Squid, "Squid : Optimising Web Delivery." [Online]. Available: http://www.squid-cache.org/

[42] C. Tumbleson, "Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps." 2022. [Online]. Available: https://ibotpeaches.github.io/Apktool/

[43] O. A. Vadla Ravnås, "Frida • A world-class dynamic instrumentation framework." [Online]. Available: https://frida.re/

[44] T. Vidas and N. Christin, "Evading android runtime analysis via sandbox detection," in *Proceedings of the ACM symposium on Information, computer and communications security*, 2014.

[45] ViRb3, "MagiskFrida," 2023. [Online]. Available: https://github.com/ViRb3/magisk-frida

[46] P. Wang, Q. Bao, L. Wang, S. Wang, Z. Chen, T. Wei, and D. Wu, "Software Protection on the Go: A Large-Scale Empirical Study on Mobile

App Obfuscation," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2018.

[47] J. Wu, "Magisk," 2023. [Online]. Available: https://github.com/topjohnwu/Magisk

[48] L. Xue, Y. Yan, L. Yan, M. Jiang, X. Luo, D. Wu, and Y. Zhou, "Parema: An Unpacking Framework for Demystifying VM-Based Android Packers," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2021.

[49] L. Xue, H. Zhou, X. Luo, Y. Zhou, Y. Shi, G. Gu, F. Zhang, and M. H. Au, "Happer: Unpacking Android Apps via a Hardware-Assisted Approach," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2021.

[50] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu, "AppSpear: Bytecode Decrypting and DEX Reassembling for Packed Android Malware," in *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2015.

[51] Y. Zhang, X. Luo, and H. Yin, "DexHunter: Toward Extracting Hidden Code from Packed Android Applications," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2015.

[52] O. Zungur, A. Bianchi, G. Stringhini, and M. Egele, "AppJitsu: Investigating the Resiliency of Android Applications," in *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2021.