SoK: Hardening Techniques in the Mobile Ecosystem — Are We There Yet?

Magdalena Steinböck
TU Wien
magdalena.steinboeck@seclab.wien

Jens Troost
VU Amsterdam
j.e.troost@vu.nl

Wilco van Beijnum
University of Twente
wilcovanbeijnum@gmail.com

Jan Seredynski
VU Amsterdam
janseredynski@gmail.com

Herbert Bos
VU Amsterdam
h.j.bos@vu.nl

Martina Lindorfer
TU Wien
martina@seclab.wien

Andrea Continella
University of Twente
a.continella@utwente.nl

***Abstract*—Irrespective of the security and isolation guarantees offered by the mobile operating system, the Mobile Application Security Verification Standard (MASVS) recommends app developers to implement hardening techniques for *self*-protection—to prevent tampering and leakage, detect jailbreaks, etc. Despite regulations incentivize developers toward implementing self-protection, our understanding of the use of hardening techniques is still very limited—especially regarding differences, if any, between the two main mobile ecosystems. In this paper, we systematize knowledge on the use and analysis of hardening techniques, covering, *for the first time*, both Android and iOS apps.**

To this end, we present HALY, a framework to analyze the adoption of hardening techniques. Using HALY's static and dynamic analysis, we analyze 2,646 popular apps available on both Android and iOS, and measure the prevalence of hardening techniques. Contrary to expectation, apps on iOS *underperform* in self-protection, implementing only half of the recommended hardening techniques compared to their Android counterparts—challenging the long-held belief that iOS is simply "more secure." Equally surprising, while privacy-sensitive apps implement more self-protection, many apps implement hardening techniques on only one of the two OSes. Furthermore, as many common techniques are easy to individually bypass, the additional security is questionable. Overall, almost all apps implement some hardening techniques, but as many as 24.1% (Android) and 73.6% (iOS) implement fewer than half of the recommended ones, and we only found 26 apps on Android to implement all eight and only one app on iOS adopt all seven analyzed techniques.

1. Introduction

As smartphones routinely handle our most sensitive data, there is a growing threat of attacks that bypass the system's isolation and sandboxing mechanisms meant to prevent malicious apps from accessing the data of other apps—for instance by rooting or jailbreaking the device. Such attacks target not just the confidentiality of data, but may also threaten an app's integrity, as exemplified by the popularity of cheating frameworks in mobile games [79]. In addition, security-sensitive apps, such as banking and shopping apps, are valuable targets for such attacks due to their finance-related nature [98].

In response, many apps rely on *Runtime App Self-Protection (RASP)*, to hinder reverse engineering, debugging, and tampering. In particular, the Mobile Application Security Verification Standard (MASVS) [60], published by OWASP, specifies multiple hardening techniques that apps are recommended to implement (MASVS-RESILIENCE) and defined a corresponding testing profile (R) for security verification. Also, new regulations regarding IT security in the EU and elsewhere now require developers to implement adequate (self-)protection. One such regulation, the EU Digital Operational Resilience Act (DORA) [21], enforced from January 2025, prescribes operational resilience testing of all ICT systems for financial entities. Hence, one expects financial apps to implement self-protection to comply with DORA.[1]

On the surface, these are positive developments. Regulations and recommendations serve as powerful incentives to persuade developers to implement hardening techniques. Moreover, compliance reassures developers that their apps meet security standards and are less likely to face legal claims, though vulnerabilities may still exist despite adherence to standards [54].

The questions are, first, if and which type of these techniques are actually used, second, if there are differences between Android and iOS, and third, if the apparent faith in today's hardening techniques is justified. OWASP [63] showed that many techniques are easy to bypass, giving developers and users a false sense of security, known as "security theater" [77].

Unfortunately, despite several studies investigating specific characteristics of self-protection (see Table 1), our community still lacks a comprehensive, yet in-depth, understanding of the hardening techniques, their adoption across the mobile ecosystems, and the suitable approaches to track them. A crucial gap in our knowledge concerns the differences, if any, between Android and iOS: is hardening more common on one of these platforms, are the most common techniques the same, and are there differences between the same apps on Android and iOS? Analyzing all recommended hardening techniques on 2,646 apps available on both platforms, this paper is the first to provide a comparative analysis of the two ecosystems with empirical evidence to answer these questions.

1. Similarly, the EU Cyber Resilience Act (CRA) [20], enforced in 2027, legally defines security requirements for hardware and software.

TABLE 1: Classes of hardening techniques and state-of-the-art approaches to identify their adoption (* Android only).

	AppJitsu [98]	Pradeep et al. [68]	Kellner et al. [45]	Ibrahim et al. [42]	Evans et al. [28]	Reaves et al. [71]	Ruggia et al. [75]	HALY
Analysis type	Dynamic	Static & Dynamic	Static & Dynamic	Static & Dynamic	Static & Dynamic	Static	Dynamic	Static & Dynamic
OS support	🤖	🤖 & 🍎	🍎	🤖	🤖	🤖	🤖	🤖 & 🍎
Total number of apps	455	5,079	3,482	163,773	35	46	41,710	5,292
Benign	100%	100%	100%	100%	100%	100%	50.72%	100%
Anti-tampering </>	✔	✖	✖	✔	✖	✖	✔	✔
Hooking detection </> 🛡	✔	✖	✖	✖	✖	✖	✔	✔
Debug detection 🛡	✔	✖	✖	✖	✖	✖	✔	✔
Emulation detection 🛡	✔	✖	✖	✖	✖	✖	✔	✔
Root/Jailbreak detection 🛡	✔	✖	✔	✖	✔	✖	✔	✔
Keylogger protection 👁	✖	✖	✖	✖	✖	✖	✖	✔
Screenreader protection 👁	✖	✖	✖	✖	✖	✖	✖	✔*
Secure connections 👁	✖	✔	✖	✖	✖	✔	✔	✔

Unlike existing work that examines hardening through a narrow lens (specific techniques [28], [42], [45], [68], specific apps [16], [71], or specific platforms [75], [98]), we cast our net wide and, especially, focus on both Android and iOS. Our study includes all access-related hardening techniques that prevent attacks at run time (e.g., detection of jailbreaks, or hooking) described in the literature. We categorize them into a taxonomy based on whether they protect the app's *integrity*, against threats from the run-time *environment*, or protect *input/output (I/O)* channels. This means that code transformation such as encryption and obfuscation, which form a separate field of study, are out of scope. It is also not our goal to analyze the *implementation* of such techniques (which is done elsewhere [80]) or to research new techniques against next-generation attacks based on side channels or hardware bugs. Similarly, our analysis focuses on benign apps. Li et al. [48] have shown that benign apps implement significantly more evasion techniques than malicious apps. We do include a case study on malware for completeness.

We present an automated *HArdening anaLYzer,* HALY, which tracks the implementation and adoption of these techniques, using both static and dynamic analysis. Using HALY, we successfully analyze a large dataset of more than 2,646 *cross-platform* [83] apps on each platform. Our empirical results yield important insights and challenge long-held beliefs. First, contrary to initial expectations, apps on iOS *underperform* in self-protection, implementing roughly half as many techniques as found in their Android counterparts. Second, we find that many of the most common techniques are easy to circumvent. Additionally, while most self-protection measures in popular apps are easy to analyze, this is not the case for more sophisticated schemes that marry hardening with code transformation—here, more research is needed. Finally, we show that the use of hardening techniques differs among app *categories*, that security-sensitive apps implement more techniques, and that there are remarkable inconsistencies between different versions of the same app, as we detect self-protections in many apps on only one OS. Overall, our results suggest that 0.2% (Android) and 1.4% (iOS) of the apps do not implement *any* of the recommended mechanisms, 75.9% and 51.5% at least 3, and only 26 apps (Android) and 1 app (iOS) adopt *all*.[2]

2. These results are lower bounds, as we tuned our approach to minimize false positives, even at the cost of possible false negatives (e.g., due to custom techniques or high-level protections).

In summary, we make the following contributions:

- We survey and systematize self-protection techniques on Android & iOS according to our taxonomy.
- We present HALY, a tool to track adoption of hardening techniques and study their use in 2,646 apps available on both platforms, highlighting differences.
- We present insights (affecting users, developers, and researchers) about the state of mobile self-protection. We identify areas for further research and question the overall security of today's apps.

Availability. Dataset, results, and code are available at https://github.com/utwente-scs/haly-hardening-analyzer.

2. Hardening Against Run-time Threats

2.1. Threat Model

We assume attackers try to analyze or manipulate an app on a compromised device. Developers use hardening techniques to protect the app's data even on such devices, but also to comply with legislation. For example, apps often include intellectual property (IP) such as music or movies that, in the absence of sufficient protection, attackers may extract without a license. Similarly, without adequate run-time protections in mobile games and finance apps, users can cheat or purchase in-app currency without payment, and attackers can leak sensitive financial details. To comply with legislation such as GDPR, developers need to ensure the protection of sensitive user data at all times. On a jailbroken device, system-wide security features can be disabled, putting user data at risk of theft from malicious actors. Hence, apps need sufficient protection mechanisms to ensure security and privacy of their user's data and prevent data leakage even on compromised devices. In particular, we assume attackers pursue their malicious purposes by attempting to compromise the target app's *integrity*, *environment*, or *data (I/O)*.

2.2. Hardening Techniques

OWASP's Mobile Application Security Verification Standard (MASVS) [60] lists requirements for apps to implement and the Mobile Application Security Testing Guide (MASTG) [63] provides guidance on how to test their implementation. MASVS is also the standard Android apps are tested against in case apps opt for the

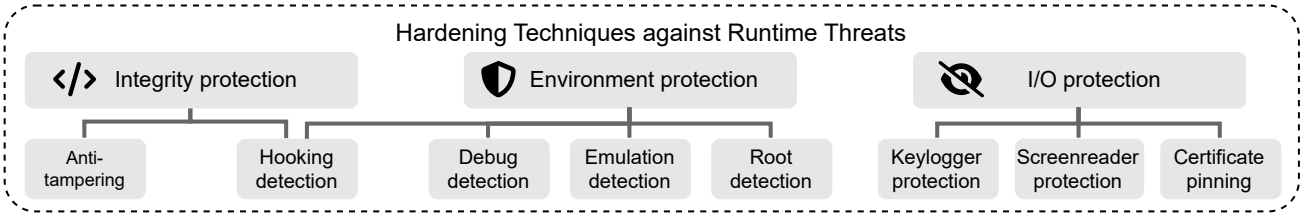

Figure 1: A taxonomy of hardening techniques protecting mobile apps from run-time threats.

independent security review reflected in Google Play's data safety section [35]. In addition, related work has studied the prevalence of selected hardening techniques [28], [45], [68], [71], [75], [98], as well as further highlighted additional techniques, together with their impact, that can compromise the overall run-time security of mobile apps [5], [18], [30], [44], [49], [51]. Based on our review of these sources we collected eight highly relevant types of app hardening techniques (or "RASPs") (see Table 1). The OWASP MASVS dedicates a whole section to the resilience of apps in regards to different approaches to defense-in-depth. It presents four categories: Resilience 1 (integrity validation of the platform), Resilience 2 (anti-tampering mechanisms), Resilience 3 (anti-static analysis mechanisms), and Resilience 4 (anti-dynamic analysis mechanisms). In our framework, we detect three resilience categories, i.e., Resilience 1 (root, emulation, screenshot, and keylogger detection), Resilience 2 (tamper detection and certificate pinning), and Resilience 4 (debug and hooking detection). We did not include Resilience 3, as it discusses code transformation techniques such as obfuscation. These target static analysis only and form their own field of studies – hence, they are out-of-scope for this work. However, we perform an analysis of packers present in our Android dataset, which can indicate the presence of code-transformation techniques.

Following our threat model, as a first step in systematization, we classify them into three broad categories (see Figure 1): (1) *integrity protection* (anti-tampering such as integrity checks, signing, hooking manipulations), (2) *environment protection* (detection of hooking, debugging, emulation, jailbreaks), (3) *I/O protection* (keylogger and screen reading protection, certificate pinning).

Integrity: anti-tampering protection. Even on a non-instrumented device, attackers can decompile, modify, and recompile apps to inject malicious behavior or disable security mechanisms, and then can spread their apps via alternative app stores. Especially on Android, app repackaging is easy to implement. To mitigate this threat, apps should adopt anti-tampering techniques, e.g., through integrity checks and signing [42], [60], [75], [98].

Integrity & Environment: hooking detection. Attackers alter the behavior of apps with hooking frameworks such as Frida [88], Cydia Substrate [76], or Xposed [74]. By hooking functions or system calls (syscalls), they read and modify parameters and return values, or even replace functions. Apps should therefore look for signs of hooking [60], [75], [98].

Environment: debug detection. Attackers often attach debuggers to reverse engineer apps. Native debugging may use tools such as `ptrace`, while Android also supports debugging of Java apps using the Java Debug Wire Protocol (JDWP) [80]. As a mitigation, apps should detect and block known debug methods [60], [75], [98].

Environment: emulation detection. As attackers similarly use emulators for reverse engineering, apps should implement checks for artifacts (e.g., differences in hardware and software configurations) that indicate the presence of an emulator [43], [50], [52], [60], [75], [89], [98].

Environment: root and jailbreak detection. Rooting Android devices or jailbreaking iOS devices weakens their security, as it allows apps to break out of their sandboxed environment and access data belonging to other apps. Also, attackers use jailbreaks to bypass an app's restrictions, e.g., to unlock paid content. Thus, apps should verify if the device is rooted or jailbroken [28], [45], [60], [75], [84], [98].

I/O: keylogger protection. As malicious keyboards can serve as keyloggers and collect all typed inputs, apps should use a custom virtual keyboard for sensitive fields [18], [44], [49].

I/O: screenreader protection. Attackers can obtain sensitive information by using a malicious app that captures or records the screen. To prevent sensitive data from being exposed, apps should detect screenreaders and block screen captures [5], [30], [44], [49], [51].

I/O: secure connections. Finally, a Machine-in-the-Middle (MitM) attack may intercept an app's network traffic to steal credentials, session tokens or other information. To prevent this, apps should adopt encrypted protocols (i.e., TLS) and consider implementing certificate pinning—including the certificate (hash) of the trusted backend in their code [8], [19], [29], [60], [68], [71], [75].

3. HALY: Framework Design

To further systematize the use of these techniques, HALY unifies the analysis of Android and iOS apps to track the hardening techniques discussed in Section 2—using static analysis to track techniques that only trigger under specific constraints, hard to satisfy at run time, and dynamic analysis to handle cases that static analysis cannot (e.g., obfuscated apps), and validate its results (mitigating false positives). For fair comparison, we implement the same detectors for both platforms. Overall, HALY consists of five stages (see Figure 2), mirrored for both platforms: pre-processing; static analysis; dynamic analysis; technique tracking; and post-processing.

3.1. Pre-processing

HALY first downloads the target app from its respective store, using gplay-downloader [11] for the Google

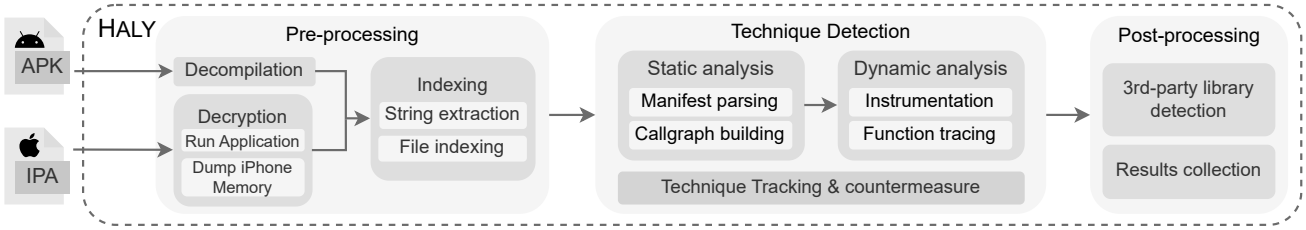

Figure 2: System overview of HALY. Our approach follows the stages in Section 3 for both Android and iOS.

Play Store and IPATool [6] for the Apple App Store, and additionally collects metadata, such as the app's category in the store. HALY then pre-processes the apps for analysis. It disassembles Android apps into their smali code using apktool [87]. For iOS apps, encrypted by default, it installs and runs them on a real device, and then, using Frida [88], dumps their decrypted binary from memory at run time. Finally, HALY extracts and indexes all strings from binaries—to use later for searching for specific artifacts (e.g., of hooking frameworks).

3.2. Static Analysis

Next, HALY statically disassembles the app's code and identifies functions, relying on Radare2 [69] to build a callgraph and find method calls in native binaries. We also use Radare2 to identify supervisor call (`SVC`) instructions. Apps can use `SVC` instructions to directly invoke syscalls from native code, bypassing the interception of hooking frameworks. In addition, HALY uses Code Search [38] to find occurrence of specific strings, such as file or app names, as well as to detect Java method calls in the smali code of Android apps. Finally, we parse the `AndroidManifest.xml` file in Android apps and the `Info.plist` file in iOS apps to extract metadata, such as the app name, versions, and requested permissions.

3.3. Dynamic Analysis

During dynamic analysis, HALY uses Frida to hook into various methods and syscalls. For some syscalls and methods, it additionally modifies the arguments or return values to hide from the app that the device is rooted/-jailbroken and running instrumentation software. We also hook the addresses of the `SVC` instructions identified during static analysis and look up the corresponding syscalls by looking up the `SVC` instruction number and process them in the same way as 'normal' syscalls. Unfortunately, relying solely on the invoked functions to track hardening techniques means that we cannot track variables, such as the `Build.MODEL` variable which some Android apps use to check if they run on an emulator. To the best of our knowledge, Frida cannot track run-time accesses to these variables, or to what values they are compared. Fortunately, we do track their usage during static analysis.

3.4. Tracking Hardening Techniques

On top of the static and dynamic analyses, HALY provides modular detectors, each of which is responsible for tracking a specific hardening technique. They collectively represent most, if not all, state-of-the-art techniques for detecting self-protection on both platforms based on our literature review (see Section 2.2). Where possible, we also implement generic countermeasures (see Section 3.3), to ensure that apps do not terminate as soon as root or instrumentation is detected.

Some of the techniques below may serve more than one purpose. For instance, detecting the Cydia Substrate framework [76] can be an indicator for hooking detection but also for root detection, as the framework can only be installed on rooted devices. However, our taxonomy takes this into consideration and thus, the detection of Cydia Substrate counts towards both integrity protection and environment protection. Hence, numbers regarding our classification may not add up to 100%.

Integrity: anti-tampering protection. On Android, HALY traces functions related to retrieving and validating app signatures and the usage of the (now deprecated) SafetyNet API [36] or Play Integrity API [34] (its successor, introduced in 2022 [32]). These APIs check device integrity, including emulator detection, and apps invoking them are classified as implementing anti-tampering, as well as root, hooking, and emulation detection.

On iOS, one cannot check the signature of an app directly. All apps from the App Store are re-signed with Apple's signature, and iOS devices only allow execution of Apple-signed binaries, unless the check is disabled by a jailbreak. However, apps could be re-signed and executed on a jailbroken device, which is why apps could test whether an `embedded.mobileprovision` file is present [91]. Apple does provide an App Attest Service though to verify apps are unmodified [9] and HALY detects the usage of this service.

Integrity & Environment: hooking detection. HALY first checks for the detection of hooking frameworks, such as Xposed [74], Cydia Substrate [76], and Frida [88], using static analysis, by scanning for the names of these frameworks or related apps in the app's code or text files. During dynamic analysis, HALY tracks if the app attempts to access files related to them, or 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 blurry. We therefore classify any mention of "substrate" as hooking detection and any other checks related to Cydia as root detection. Apps on iOS can also use `_dyld_get_image_name()` [63] to see if any modules related to hooking frameworks are loaded.

Environment: debug detection. On iOS, programs can use the syscalls `ptrace` and `sysctl` to check if the app is being traced. Further, they can use `getppid` to validate the PID of the parent process, which should be 1 if the

app is started by the launcher. HALY looks for and hooks these syscalls to monitor their use. On Android, HALY also looks for `ptrace`, in addition to Java methods such as `Debug.isDebuggerConnected()`.

Environment: emulation detection. On iOS, an app can inspect environment variables to test whether it is running on an emulator or simulator. For example, the iOS simulator contains the environment variable `SIMULATOR_MAINSCREEN_WIDTH` and the Corellium emulator loads the environment variable `SANDBOX_TOKENS`. On Android, an app can use various variables from the `Build` class such as `Build.MODEL` to understand whether it is running on an emulator. An app can also check if certain files that are only present on an emulator exist, such as `/Applications/Xcode.app` on an iOS simulator or `/dev/socket/genyd` on a Genymotion Android emulator. HALY tracks the usage of all these variables and files.

Environment: root and jailbreak detection. Apps may check for the presence of certain files not present on stock devices, or file/directory permissions different from those on unmodified devices. During static analysis, HALY checks if the app contains any mention of these filesystem artifacts, and during dynamic analysis, HALY hooks file access syscalls to track them. Some apps also check if other apps related to rooting/jailbreaking are installed. During static analysis, HALY looks for mentions of these apps, and during dynamic analysis, tracks requests for information about these apps by hooking `openURL()` and `canOpenURL()` on iOS, and hooking methods of the `PackageManager` and `Intent` classes on Android.

I/O: keylogger detection. To prevent keylogging, apps can show their own keyboard at input fields, or check if the active keyboard is part of an allowlist. HALY tracks the usage of functions related to hiding the system keyboard from an input field and functions for getting the active keyboard—on Android: `EditText.setShowSoftInputOnFocus()`, `InputMethodManager.getEnabledInputMethodList()`; on iOS with: `UIView.inputView()`, and `UIResponder.textInputMode()`.

I/O: screenreader detection. Bar unreliable workarounds, on iOS, there is no method to block screenshots or recordings. On Android, an app can set any of its components as "secure" to block screenshots. HALY checks for this using `SurfaceView.setSecure()` or `Window.setFlags()` [63].

I/O: secure connections. During static analysis, HALY uses a methodology similar to that of Pradeep et al. [68] to detect certificate pinning by checking for the inclusion of certificates or their hashes in the app. During dynamic analysis, it tracks the usage of known pinning functions of popular libraries. Furthermore, it intercepts all network traffic during the dynamic analysis of the app.

3.5. Post-processing

At the end of the analysis, HALY aggregates all results, and extracts the following additional pieces of information to put the results into context:

App categories and permissions. We correlate the adoption of hardening techniques to the app's category (according to the app store metadata), as well as their requested privacy- and security-sensitive permissions.

First-party vs. third-party. We classify hardening techniques as first-party or third-party implementations by first identifying all third-party libraries. In particular, we categorize any library that is present in at least n apps as a third-party dependency. In our analysis, we used $n = 5$, which should be sufficiently conservative. We chose this threshold as there may be apps from the same developer sharing parts of their implementation. For native libraries, we use their file name as an identifier for the process, while we use the code path for Java libraries (e.g., `com.google.android.gms`). As further discussed in Section 7, HALY might miss third-party libraries if they obfuscate their names or code paths or occur fewer than n times. Finally, we verify whether the identified third-party libraries implement any of the tracked hardening techniques by analyzing the stack trace during dynamic analysis. After filtering out all system libraries in the stacktrace, we select the top item in the stacktrace as the source of the hardening technique. This allows us to distinguish between hardening techniques implemented by developers and those provided by third-party libraries.

Confidence measure & usefulness. Finally, our tool assigns a confidence score to every finding, allowing users to assess the reliability of the detected hardening techniques. The confidence score for each finding is assessed based on its potential alternative use cases—of which we manually verified that they positively help identify the hardening technique, e.g., in root detection, searching for a specific jailbreak name is a strong indicator, whereas calling `fork` may serve other purposes as well. Further, HALY provides metadata for each finding – for static findings, we include details such as the source, offsets, and function names, and for dynamic findings we include backtraces, argument values, and contextual information. While our approach prioritizes confident findings to maintain a conservative detection strategy, it is possible to use a more permissive setting to increase detections at the risk of higher false positives. We do not differentiate whether the confident detection was obtained during static and dynamic analysis, meaning that if a method is confidently detected only in one analysis method, we still count it as detection of the hardening technique. As we aimed to minimize false positives, we only take into account confident detections and our results should be interpreted as a lower bound.

4. Framework Validation

In this section, we validate the HALY framework by assessing its accuracy on ground truth datasets and comparing the results to the well-known Mobile Security Framework (MobSF) [3].

Experimental setup. For our Android experiments, we use a Pixel 3 running Android 12 (released in October 2021), which we root by installing APatch [14]. APatch is running kernel patch version 0.10.5. For the emulator, we use Android 11 with Google APIs, version 30, the latest version of the Android Emulator that supports ARM on x86. For both the rooted device and the emulator, we

install `frida-server`. Additionally, we applied anti-detection patches for improved stability on the rooted phone [41]. For iOS, we use an iPhone 8 running iOS 16.4.1 (released in March 2023), which we jailbreak using the rootful palera1n jailbreak [64], install the Sileo package manager [81] and use it to install `frida-server`.

For the dynamic analysis, we start a proxy (Squid [82] for Android and mitmproxy [25] for iOS) on our machine, and configure the devices to use the proxy using WiFi-hotspots and tap interfaces to intercept the network traffic at run time. We collect traffic using `tcpdump` with a filter on the IP address of the devices.

We set a 10-minute timeout and an 8 GB memory limit per file for static analysis. Since benign apps execute hardening techniques (detection of rooting/jailbreaks, hooking, debugging, emulation, etc.) early in the execution, we use a one-minute timeout for dynamic analysis, which is in line with other work on dynamic analysis of benign apps [46]. Ruggia et al. [75] found that benign apps mostly performed environment checks very early in the execution (at $<$10% into their four minute execution). Further, Pradeep et al. [68] found that most TLS connections are established within 30 seconds from startup.

Ground truth. To validate HALY's results and errors, we first develop proof-of-concept Android and iOS apps (part of our artifact) that (a) implement no hardening techniques, as well as apps that (b) trigger all the studied hardening techniques, and verify that HALY produces correct results in both scenarios.

Next, we validate HALY against 5 open-source Android apps, notably four levels of OWASP Crackme apps [61], and the proof-of-concept app that implements multiple different evasion strategies by Ruggie et al. [75]. Similarly, to validate our iOS analysis, we use the two available OWASP Crackmes [62] for iOS, iGoat [95], as well as the sample apps from the SDKs FreeRASP [86] and the library IOSSecuritySuite [73], which both provide hardening techniques for iOS apps. Table 2a presents our Android evaluation results, while Table 2b covers iOS. During static analysis, HALY correctly identifies 95.0% of the implemented RASPs of our validation dataset on Android, and 76.3% on iOS. When also considering dynamic results, HALY correctly identifies 95.0% of the implemented RASPs in our Android validation dataset on an emulator, 90.0% on a rooted Android device, and 92.3% on iOS. In our experiments, we only encountered two false positives on Android on a rooted device. False negatives are expected in cases where developers implement custom security solutions or use techniques that serve multiple purposes beyond hardening. For instance, the emulation detection of all undetected iOS apps is implemented by retrieving all environment variables. While HALY detects this process, it does not classify it as confident, as the same approach can be used for, e.g., testing purposes [67].

Additionally, we ran HALY on 15 highly security-sensitive Android apps that implement sophisticated self-protection, which have been provided and manually analyzed by our industry partner *Keysight Riscure Security Solutions* [1] specifically for that purpose. These apps are known to implement hardening techniques, and make use of obfuscators, packers, and a host of other *software protection tools (SPTs)*. Of the 15 apps, 14 completed static analysis, and only 4 completed dynamic analysis. Static analysis is shown to be affected by these SPTs, having an accuracy of 38.8%. Of the 4 apps that completed dynamic analysis, HALY correctly identified 54.7% of the implemented RASPs. Similarly to the OWASP Crackme apps, most errors were false negatives (62%). These can be expected in this set of apps, due to the high level of protections used, and some apps require interaction with the app to trigger certain checks. These results suggest that combining hardening techniques with other protective measures significantly raises the bar for automated analysis. Sections 5.4 and 7 elaborate on these findings.

Comparison to MobSF. To demonstrate the effectiveness of HALY, we evaluated our ground truth with MobSF [3] and compare its findings to our tool's results. MobSF detects hardening techniques by using regex-based pattern matching via libsast [2]. While our framework also utilizes pattern matching during static analysis, it further observes the behavior of apps during runtime and monitors function/API calls. As static analysis is undecidable, many cases can only be handled using dynamic analysis. Also, MobSF's dynamic analysis for iOS apps is limited to Corellium VM [24], a closed source and paid service provider, whereas HALY supports dynamic analysis on any jailbreakable iOS device. For Android, MobSF correctly identified root detection in all five apps, and certificate pinning in one app (out of two). However, it did not detect other hardening techniques, leading to a high amount of false negatives and a false negative rate (FNR) of 68%. For our iOS apps, MobSF did not detect any hardening technique, resulting in a FNR of 100%.

Case study: malware. To understand the effect malicious applications have on HALY, we analyzed a small dataset of 10 malware samples we obtained from AndroZoo [7]. These samples were classified with AVClass2 [78] as belonging to four different families: *necro*, *joker*, *spyagent*, and *smsthief*. Wang et al. [90] analyzed the behaviour of these families in depth—providing ground truth for our validation. We selected these families manually with the goal to have a variety of evasive techniques and minimize the risks of the malware.

Two families, *necro* and *joker*, are known to employ obfuscation and packing, along with other environmental evasive techniques. Meanwhile, *spyagent* and *smsthief*, do not use obfuscation or packers. *Smsthief* does not perform any environment checks or other evasive techniques, while *spyagent* performs environmental checks.

The setup for this case study was limited to an emulator, to avoid the issue of persistence. A list of MD5 hashes of the samples as well as our results can be found in Table 12 (in the Appendix). HALY was successful in detecting hardening techniques for five of the malware samples with static analysis, and three with dynamic analysis. Regarding the quality of the detections, static analysis yielded many confident environmental checks for *joker* and *spyagent* samples, as expected. In contrast, *smsthief* samples showed no confident detections, with only one unconfident anti-debug detection, which aligns with expectations. For the *necro* samples HALY was unsuccessful in detecting any hardening technique, both during static and dynamic analysis. This family shows the limitations of HALY in the face of obfuscation and packers, along with emulation detection, discussed further in Section 7.

TABLE 2: Ground truth evaluation results on Android and iOS. TP = true positive, FP = false positive, TN = true negative, FN = false negative, ACC = accuracy, PRE = precision, FPR = false positive rate, FNR = false negative rate. $^+$ indicates results from dynamic analysis on an emulator, while * denotes a jailbroken physical device.

(a) Results of our ground truth analysis for Android (5 apps).

	Static Analysis				Dynamic Analysis								Total							
	TP	FP	TN	FN	TP$^+$	FP$^+$	TN$^+$	FN$^+$	TP*	FP*	TN*	FN*	TP	FP	TN	FN	ACC	PRE	FPR	FNR
Anti-Debugging	5	0	0	0	3	0	0	2	3	0	0	2	5	0	0	0	1.00	1.00	–	0.00
Anti-Tampering	2	0	2	1	1	0	2	2	1	0	2	2	2	0	2	1	0.80	1.00	0.00	0.33
Anti-Hooking	3	0	2	0	2	0	2	1	3	2	0	0	3	2	0	0	0.60	0.80	1.00	0.00
Anti-Emulator	1	0	4	0	1	0	4	0	1	0	4	0	1	0	4	0	1.00	1.00	0.00	0.00
Anti-Root	5	0	0	0	5	0	0	0	5	0	0	0	5	0	0	0	1.00	1.00	–	0.00
Anti-Keylogger	0	0	5	0	0	0	5	0	0	0	5	0	0	0	5	0	1.00	–	0.00	–
Anti-Screenreader	0	0	5	0	0	0	5	0	0	0	5	0	0	0	5	0	1.00	–	0.00	–
Cert. Pinning	1	0	3	1	1	0	3	1	1	0	3	1	1	0	3	1	0.80	1.00	0.00	0.50
Total (%)	42.5	0.0	52.5	5.0	32.5	0.0	52.5	15.0	35.0	5.0	0.0	12.5	42.5	5.0	47.5	5.0				

(b) Results of our ground truth analysis for iOS (5 apps).

	Static Analysis				Dynamic Analysis				Total							
	TP	FP	TN	FN	TP*	FP*	TN*	FN*	TP	FP	TN	FN	ACC	PRE	FPR	FNR
Anti-Debugging	2	0	2	1	3	0	2	0	3	0	2	0	1.00	1.00	0.00	0.00
Anti-Tampering	0	0	2	3	3	0	2	0	3	0	2	0	1.00	–	0.00	0.00
Anti-Hooking	1	0	2	2	3	0	2	0	3	0	2	0	1.00	1.00	0.00	0.00
Anti-Emulator	0	0	1	3	0	0	2	3	0	0	2	3	0.40	–	0.00	1.00
Anti-Root	3	0	1	0	3	0	1	1	3	0	1	0	1.00	1.00	0.00	0.00
Anti-Keylogger	0	0	5	0	0	0	5	0	0	0	5	0	1.00	–	0.00	–
Anti-Screenreader	0	0	5	0	0	0	5	0	0	0	5	0	1.00	–	0.00	–
Cert. Pinning	1	0	4	0	0	0	4	1	1	0	4	0	1.00	1.00	0.00	0.00
Total (%)	18.4	0.0	57.9	23.7	30.0	0.0	57.5	12.5	33.3	0.0	59.0	7.7				

TABLE 3: Number and percentage of apps using each hardening technique, detected via static and dynamic analysis on rooted devices and an emulator. Brackets indicate low-confidence detections. Screenreader protection is Android-only. Most techniques are more common on Android, where nearly all apps use anti-debugging, emulation, and root detection. On iOS, root detection is the most widely used. We also show how many apps use at least one technique per category.

	Static Analysis	Dynamic Analysis		Static Analysis	Dynamic Analysis
		Rooted	Emulator		
Anti-Tampering </>	589 (22.26%) [2,048 (77.40%)]	1 (0.04%) [2,339 (88.40%)]	1 (0.04%) [2,300 (86.92%)]	108 (4.08%) [0]	872 (33.0%) [0]
Hooking Detection </>	1,419 (53.63%) [0]	118 (4.46%) [2,514 (95.01%)]	69 (2.61%) [2,065 (78.04%)]	399 (15.08%) [2,140 (80.88%)]	524 (19.80%) [1,994 (75.36%)]
Debug Detection	2,470 (93.35%) [173 (6.54%)]	1,877 (70.94%) [0]	1,734 (65.53%) [0]	528 (19.95%) [2,052 (77.55%)]	204 (7.71%) [2,151 (81.29%)]
Emulator Detection	2,618 (98.94%) [0]	58 (2.19%) [0]	37 (1.40%) [0]	8 (0.30%) [0]	12 (0.45%) [2,585 (97.69%)]
Root Detection	2,285 (86.36%) [0]	2,196 (82.99%) [0]	1,973 (74.57%) [0]	2,544 (96.15%) [31 (1.17%)]	659 (24.91%) [1,937 (73.20%)]
Keylogger Protection	996 (37.64%) [342 (12.93%)]	598 (22.60%) [0]	104 (3.93%) [0]	1,398 (52.83%) [0]	197 (7.45%) [0]
Screenreader Protection	375 (14.17%) [1,082 (40.89%)]	34 (1.28%) [0]	26 (0.98%) [0]	-	-
Certificate Pinning	676 (25.55%) [0]	4 (0.15%) [1,284 (48.53%)]	5 (0.19%) [1,109 (41.91%)]	753 (28.46%) [0]	11 (0.42%) [1,411 (53.33%)]
Integrity Protection </>	1,601 (60.5%)	**Env. Protection**	2,638 (99.7%)	**I/O Protection**	1,758 (66.4%)
Integrity Protection </>	1,365 (51.6%)	**Env. Protection**	2,561 (96.8%)	**I/O Protection**	1,891 (71.5%)

5. Empirical Study

We validate HALY's capabilities and use it to analyze popular Android and iOS apps to assess their adoption of hardening techniques. First, we study the prevalence of hardening techniques in the two ecosystems. Second, we directly compare Android apps with their respective iOS counterparts, revealing insights into how developers adapt or change their techniques according to the target OS. Finally, we investigate whether hardening is commonly implemented by third-party libraries and correlate adoption to the use of sensitive permissions.

5.1. Dataset

In order to analyze and compare the usage of hardening techniques between Android and iOS we focus on apps that exist on both platforms, i.e., cross-platform apps. We use the dataset of Steinböck et al. [83], consisting of the 3,322 most popular widely used cross-platform apps collected in 2023 that have both an Android and an iOS version. The dataset provides a 1:1 mapping of apps across both platforms, e.g., the Facebook app for Android and for iOS. Note, however, that they could still be developed by different development teams.

Among these apps, 19 (0.6%) apps failed our static analysis on Android because their manifest file could not be parsed, preventing us from obtaining critical information for our analyses. Further, we were unable to decrypt 190 (5.7%) iOS apps because decryption requires the app to run on a jailbroken device for memory dumping. Some apps failed to install due to version incompatibilities, as our jailbreak was limited to iOS 16, while others were terminated by the device for excessive memory usage since our approach loaded all data into memory for dumping. During dynamic analysis, 80 (2.4%) Android apps failed on our physical device and 160 (5.4%) on the emulator, out of which 34 (1.0%) apps fail on both. Additionally, 318 (9.6%) iOS apps failed dynamic analysis. The most common reason for dynamic analysis failure was that apps

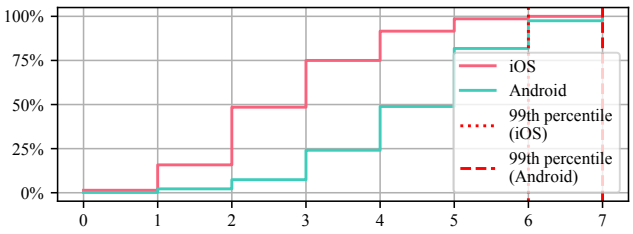

Figure 3: CDF of studied hardening techniques implemented by apps. We found that Android apps implement more categories of hardening techniques than iOS apps.

crashed too many times during the analysis, which can be caused by our Frida hooks. Reducing the run time of the analysis could prevent this issue in some cases, but we still excluded these apps from our results. Other reasons for dynamic analysis failures were timeouts or that the app crashed upon opening, even without Frida hooks present, which might be caused by version mismatches. We remove both the Android and iOS version of all apps that failed one part of our analysis from our dataset. However, analysis failures could also be an indicator for advanced protections, although our framework tries to circumvent them, which might influence our dataset to contain less hardened apps. Ultimately, we excluded a total number of 676 apps and our final dataset contains 2,646 apps per OS, and 5,292 apps in total.

5.2. Prevalence of Hardening Techniques

In Table 3, we present an overview of the prevalence of each of the RASPs that we analyzed. For each hardening technique, the table shows the number of Android and iOS apps that HALY detected as implementing the specific technique, both using static and dynamic analysis. Within square brackets, we also indicate the number of low-confidence detections, which we further elaborate on below. On a general level, we observe that the prevalence of hardening techniques differs significantly between different techniques, as well as different OSes. Figure 3 shows the cumulative distribution function of the number of different techniques adopted by Android and iOS apps—excluding screenreader protection, which is only supported for Android. Respectively, 0.2% and 1.4% of the analyzed Android and iOS apps do not implement any of the studied hardening techniques. While most apps on Android implement more than 4 different techniques, on iOS almost half of our analyzed apps implement at most 2 different hardening techniques. Only 26 Android apps and one iOS app adopt all of the analyzed techniques. Since our framework may not detect high-level protections or unconventional and custom implementations, these results should be treated as lower bounds.

Integrity: 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 finds the SafetyNet API in 4.2% of apps using static analysis, but only in one during dynamic analysis. The Play Integrity API, the successor of SafetyNet, is detected in 9.3% of apps during static analysis, and not detected at all during dynamic analysis. This indicates that some of the apps still rely on the older framework. However, the Play Integrity API has only been released since 2022. We also find 12 Android apps that contain both, SafetyNet API and Play Integrity API. On iOS, HALY finds Apple's App Attest Service in 4.1% 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 [66]. It is interesting that 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 fully integrate it, or that they only perform attestation in specific scenarios. Apps can also use their signatures to detect tampering. On Android, we track the `PackageManager::hasSigningCertificate()` function, which validates signatures in 10.4% of apps according to our static analysis, but we only detect it in 3 apps during dynamic analysis. On iOS, 32.6% of apps access their `embedded.mobileprovision` file, which may indicate repackaging [91].

Integrity & Environment: hooking detection. Using static analysis, we find hooking detection in 55.3% of Android apps. Interestingly, detection during dynamic analysis is much lower for Android apps, namely around 4.5% when executed on a rooted device and 2.6% on an emulator. We manually investigated this difference. We find that 875 Android apps (around 33.1%) of apps include the library `com.appsflyer`, which performs both Frida and Xposed detection. We discuss details about this and other third-party libraries and their influence on the prevalence of RASPs in Section 5.5. Meanwhile, on iOS, hooking is detected in 15.1% of apps during static analysis, and 19.8% during dynamic analysis. The high numbers of uncertain results on iOS can be explained by the fact that around 95.7% of iOS apps use the function `_dyld_image_count`, and 74% use both the `_dyld_image_count` and `_dyld_get_image_name` functions, which are quite commonly used to check for traces of hooking frameworks [63]. 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, 31.4% 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 46.9% of Android apps. In contrast, HALY identifies artifacts related to the lesser-known Android hooking frameworks Zygisk and Riru in two apps and three apps, 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 [37]. We identify artifacts for this framework in 24.7% of Android apps. Naturally, detection of Cydia Substrate is much more prevalent on iOS, where HALY identifies adoption in 30.3% of apps. Finally, using static analysis, we reveal that Android apps implement detection of Frida more often than iOS apps, namely 44.9% vs. 2.9%. This difference can mostly be explained by the popularity of

the aforementioned `com.appsflyer` Android library.

Environment: debug detection. Almost all (94.0%) Android apps implement some kind of debug detection. 88.3% of Android apps use the `Debug::waitingForDebugger()` or `Debug::isDebuggerConnected()` function. 61.9% of Android apps check if the developer settings are enabled, and 53.1% check if `adb` is enabled. Furthermore, we find usage of `ptrace` in 58.4% of Android apps and 2.0% of iOS apps, and 19.6% of iOS apps use the `getppid` function.

Environment: emulation detection. Emulator detection is very common on Android (present in 98.9% of the apps). `Build` variables are used in all but one case to identify emulators. This also explains the large difference between static and dynamic analysis results, since usage of these variables cannot be detected during dynamic analysis (see Section 3). The most common `Build` checks can be found in Table 5 (Appendix). `Build` variables are also used for non-hardening purposes, such as compatibility checks. HALY cannot differentiate between these uses. Aside from the `Build` variable, 2.5% of Android apps check for the presence of emulator-related files.

Contrary, on iOS, emulator detection is very rare (0.8%). During static analysis, HALY identifies 6 apps that check for the iOS simulator using the `/Applications/Xcode.app` directory, and one app that inspects the `SIMULATOR_SHARED_RESOURCES_DIRECTORY` environment variable. However, none of these apps are present in our dynamic analysis results, which only include 10 apps that check for both the `CI_NO_CM` and `CI_PRINT_PROGRAM` environment variables, and two that check only for `CI_NO_CM`, 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 caused by the retrieval of all environment variables. HALY cannot detect if these variables are then used for emulation detection or something else, e.g., testing purposes [67], but considering our other results, we consider the latter a more probable hypothesis for most of these retrievals.

Environment: root and jailbreak detection. We find root detection in 92.3% of Android apps and 96.7% of iOS apps. During dynamic analysis, we find that 84.5% of Android apps and 24.9% of iOS apps check for the existence of root-related files or validate the permissions of system directories, and 13.3% of Android apps and 6.8% of iOS apps check if root-related apps are installed. The most common apps and files that apps check for can be found in Tables 6 and 7 (in the Appendix), respectively.

Since jailbreaks for different iOS versions have distinct names, we can investigate if an app checks for specific jailbreaks. We observe that apps looking for specific jailbreaks mostly check for blackra1n and unc0ver in 5.5% and 1.4% of iOS apps, respectively. We also find a few apps that check for a selection of lesser-known jailbreaks, such as Pangu and Taurine. Interestingly, we find no apps with detection for specific jailbreaks for iOS 15 (released in September 2021) or newer, indicating that jailbreak detection in apps is not updated very often after new jailbreaking techniques are released. However, 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.

I/O: keylogger protection. On Android, we find that 49.4% of Android apps and 56.2% of iOS apps query enabled input methods, which can be used to check if a trusted keyboard is used. Further, 0.5% of Android apps disable showing the keyboard for some input fields and 4.5% of iOS apps change the `inputView` of an input field, which can be used to show a custom keyboard. 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 are a lower bound, as the input view for which a custom keyboard is shown is not visible on the startup screen.

I/O: screenreader protection. We found screenreader protection in 14.6% of Android apps. HALY detected that 22 apps set a view as "secure" to prevent screenshots or screen-recording. The low dynamic analysis results may be caused by limited code coverage (e.g., if apps only use this flag for views with sensitive information and not for the main screen shown at startup). Uncertainty in static analysis results is caused by HALY not always being able to determine if the secure flag or another flag is enabled. On iOS, there is no API to prevent screenshots or screen-recording (and we do not detect custom implementations).

I/O: certificate pinning. HALY detects certificates in 25.6% Android apps and 28.6% iOS apps—either included as a certificate file or as the hash of a certificate. We find a certificate file in 19.0% of Android apps and 28.2% of iOS apps. We find certificate hashes in 21.9% of Android apps and 8.6% 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 the app calls a certificate pinning or connection security-related function, but it could confirm that certificate pinning was used in less than 1% of cases.

Takeaways. We showed that:

- Only 26 Android apps with (and 66 without) screen-reader protection, and just 1 iOS app implement all analyzed hardening techniques.
- More than 50% of Android apps implement at least 4 RASPs, while on iOS more than 47.1% implement at most 2.
- Some RASPs are more prevalent than others: we found root and jailbreak detection in most Android and iOS apps, while emulator and anti-debug detection are more prevalent on Android.

5.3. Android Emulator vs. Rooted Device

We run HALY on both an Android emulator and a rooted device to investigate the effectiveness of HALY in both environments. In Table 3 it can be seen that the prevalence of hardening techniques is 23.7% higher on the rooted device than on the emulator. Through a Chi-squared test, we find that the differences in prevalence is statistically significant for all hardening techniques except

for screenreader protection and anti-tampering. We use a significance p-value of 0.05. We find that screenreader protection, anti-tampering and emulation detection have a p-value of 0.36, 0.11 and 0.04 respectively and find p-values of near 0.0 for the remaining techniques Overall, this indicates that HALY is more effective at detecting self-protection on a rooted device than on an emulator—either due to apps checking for emulators or compatibility.

5.4. Applications with Advanced Protections

Although not the focus of our study, SPTs have been seen to influence the effectiveness of HALY, as discussed in Section 4. Table 8 and 9 (in the Appendix) show the most common obfuscators and packers for Android in the popular apps dataset [83], our dataset as evaluated, and our industry dataset (see Section 4), which are *DexGuard 9.x* and *Arxan* (obfuscators), as well as *DexProtector* and *Promon Shield* (packers). We observed other obfuscators and packers in the industry dataset, which are not detected by *APKiD* [72], thus their presence in the other datasets is unknown. The overall prevalence of obfuscators and packers is low, with 6.27% and 0.26% respectively.

We noticed that apps with obfuscators or packers are unlikely to complete the full analysis: almost all apps with packers applied crash. We believe these to be advanced applications, and that these SPTs hinder analysis [27]. This applies to the industry dataset as well, which presents a high false negative rate and many crashes (Section 4).

The apps that completed our analysis implement on average 7.13 (obfuscators) and 6.29 (packers) hardening techniques. Globally, the average number of hardening techniques implemented is 5.91. This indicates that advanced security-focused applications apply obfuscators and packers on top of other hardening techniques.

5.5. Android vs. iOS Adoption

We now zoom out and investigate whether the prevalence of hardening techniques differs between the Android and iOS versions of the same apps. This gives an indication if the implemented hardening techniques are strictly dependent on the OS or if there are, e.g, company-wide policies to implement certain techniques. We also provide insights into the prevalence of hardening techniques in relation to the category of apps, the sensitive permissions they request, as well as the libraries they integrate.

On Android, 45.1% (1,194) of apps implement all three categories of hardening techniques, 36.6% (968) implement exactly two categories, and 18.1% (479) of apps implement only one category of hardening techniques. On iOS, 38.5% (1,019) of apps implement all three categories, while 44.2% (1,170) of iOS apps implement two categories and 15.9% (420) implement one category. This suggests that Android apps implement a wider range of RASPs than iOS apps. More specifically, for each category in our taxonomy we observed:

Integrity Protection. We identified 1,601 (60.5%) apps on Android and 1,365 (51.6%) apps on iOS implementing anti-tampering or anti-hooking techniques. Only 450 (17.0%, Android) and 386 (14.6%, iOS) implement both.

Environment Protection. On both ecosystems we found that over 95% (2,638 on Android; 2,561 on iOS) of our analyzed apps implement at least one environment protection technique, and 1,424 (53.8%) on Android but only one iOS app implement all.

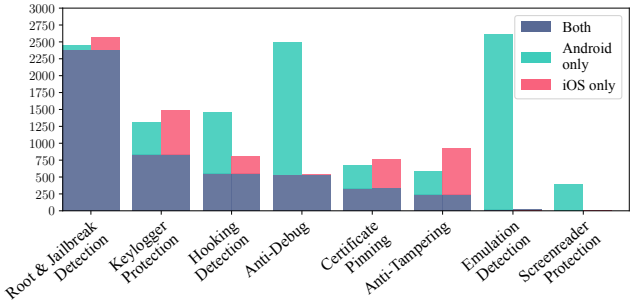

Figure 4: Consistency of hardening techniques between OSes. We show for how many apps each hardening technique is implemented by (1) both the Android and iOS versions, or (2) only by the iOS or the Android version. Only root detection is implemented in almost every app with little difference among the OSes. The other techniques show clear differences.

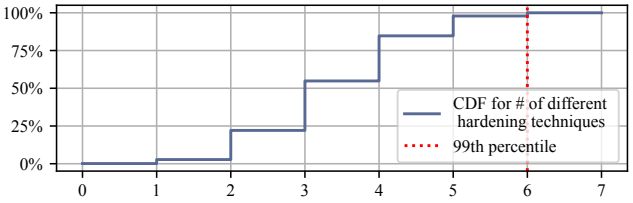

Figure 5: CDF of hardening techniques implemented on only one platform. We find a high discrepancy in the implemented RASPs between the two OSes. Almost no app implements the same techniques on Android and iOS.

I/O Protection. We found less Android than iOS apps that implement I/O protection techniques: 1,758 (66.4%) implement at least one and 91 (3.4%) all three techniques on Android while on iOS 1,891 (71.5%) implement at least one and 353 (13.3%) both analyzed techniques.

Cross-platform consistency. Figure 4 shows whether hardening techniques are implemented on both OS versions or just one. We include the results of both static and dynamic analysis. Interestingly, we can see that there are quite a few apps that implement a hardening technique only on one platform, even for hardening techniques that are prevalent on both OSes, such as hooking detection, certificate pinning, and keylogger protection. Anti-tampering, hooking and emulator detection are almost exclusively implemented on Android but not on the iOS counterpart. In Figure 5, we present the CDF of the number of hardening techniques apps implement on only platform. Here, we exclude the screenreader protection, since HALY cannot detect this on iOS. We found that for roughly 55% of our analyzed apps, the variance in implemented hardening techniques between their Android and iOS version ranged from 1 to 3. We find only three apps that implement the same techniques on both OSes.

Overall, our results reveal significant inconsistencies in hardening techniques between Android and iOS versions of the same apps. This suggests either a disconnect between developers for each platform, differences in expertise, or that certain techniques (e.g., anti-debugging,

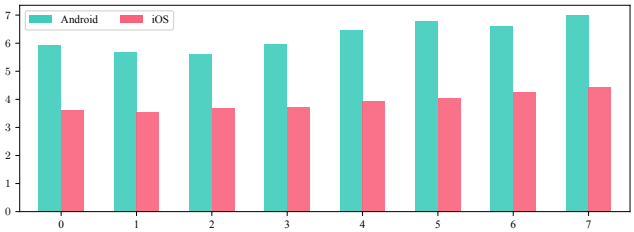

Figure 6: Average number of hardening techniques implemented in an app depending on the number of privacy-sensitive permissions. Apps requesting more sensitive permissions tend to implement more hardening techniques.

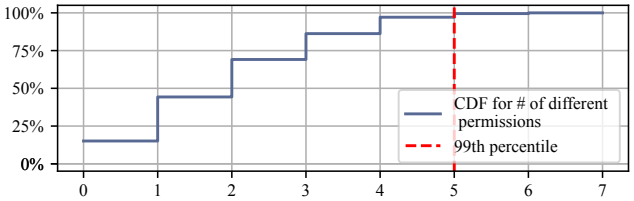

Figure 7: CDF of privacy-sensitive permissions requested on only one OS. Only around 15% request the same permissions, and most apps differ in the number of requested sensitive permissions by one to two.

emulator detection) are more relevant and documented for one OS. The lack of emulator detection on iOS may be due to the relatively recent advancements in iOS emulation. Corellium, the first publicly available iOS emulator, opened its services in 2021 [26]. Additionally, it was only in late 2021 that researchers found ways to run ARM app binaries on the iOS Simulator [31].

App store categories. One would expect hardening techniques to be more prevalent in apps within certain categories, based on the amount and sensitivity of privacy-sensitive information they typically handle (e.g., finance, games or shopping apps). Table 4 shows the average hardening techniques per app based on sensitive permission categories. We considered a hardening technique as being present in an app if it is detected during either static or dynamic analysis. In general, the OSes follow a similar pattern. Especially in the Finance and Shopping categories, we observe a high average number of implemented hardening techniques. We can also clearly see that the number of implemented hardening techniques is generally lower on iOS than on Android. Since HALY cannot detect screenreader protection on iOS, this slightly skews these results. Removing screenreader protection lowers the average hardening techniques on Android but barely affects their relative prevalence across categories.

Apps with sensitive permissions. Apps that access privacy- or security-sensitive data likely want to keep such information safe. To investigate this relation, we have identified eight categories of sensitive permissions: calendar, camera, contacts, location, microphone, health sensors, storage, and HomeKit access—where the last category is only relevant for iOS. For each app, we compute the number of categories of sensitive permissions the app uses. In Figure 6, we present the average number of hardening techniques for apps depending on the number of these sensitive permission categories. On both OSes, we can see a positive correlation between the number of

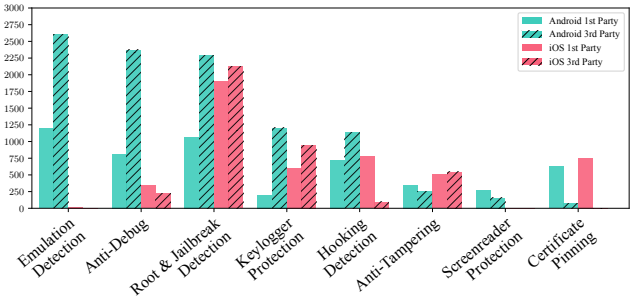

Figure 8: Prevalence of hardening techniques in first-party vs. third-party code. On Android, third-party code implements more protections. On iOS, root detection is the most widely used third-party code.

privacy-sensitive permissions requested by an application and the number of hardening techniques it implements, i.e., apps with more sensitive permissions also implement more hardening techniques. While this may suggest that developers of privacy-sensitive apps recognize the risk and employ more protections, some apps still lack adequate hardening despite requesting many sensitive permissions.

We also investigated how consistent apps are in requiring sensitive permissions across OSes. Figure 7 shows the distribution of the number of sensitive permissions that are only requested on one OS. We observe that many apps request different sets of permissions across their Android and iOS versions. Indeed, only 15.1% of apps request the same sensitive permissions on both OSes. Most apps request one or two sensitive permissions only on one OS.

We created a mapping between iOS and Android permissions by combining the protected resources [10] in the iOS documentation with the manifest permissions [33] in the Android documentation. Note that there is no perfect one-to-one match between Android and iOS permissions, so an app might provide the same functionality on both OSes but still require different permissions. Furthermore, while on Android the developer explicitly defines the needed permissions in the manifest file, on iOS permissions are requested dynamically when apps invoke a function that needs a permission. Developers, however, need to provide a description of why they request a sensitive permission in the plist file. We detect the presence of these descriptions. Nonetheless, an app might declare a permission in the manifest, or provide a description in the plist file, but not actually use the permission at run time.

First-party vs. third-party implementations. Figure 8, presents the prevalence of hardening techniques in third-party libraries vs. first-party code (note that apps may have both). Interestingly, most hardening techniques on Android originate from third-party libraries, while on iOS, most are in first-party code. Tables 10 and 11 (Appendix) show the most popular libraries that implement hardening and show that three Android libraries contribute to much of this difference. The hardening techniques detected in them surprisingly differ per app, although this may be caused by different versions of the libraries.

The most-used Android library we find is *Google Mobile Services* (`com.google.android.gms`), detected in 96.3% of Android apps. Emulation detection in this library is in 96.1% of apps in our dataset and debug detection in 76%. Next, we find the *Firebase*

TABLE 4: Average number of hardening techniques implemented in an app per category. The category "Other" includes categories with less than 1% of the total number of analyzed apps. Shopping, Social and Finance on average have the most hardening techniques on Android and iOS. Screenreader is included in the average on Android, but not on iOS.

	Shopping	Social	Finance	Travel	Communication	Business	Games	Maps	Sports	Lifestyle
	6.76	6.67	6.55	6.43	6.21	6.07	6.00	5.97	5.92	5.79
	4.41	3.98	4.10	3.99	4.00	3.58	3.96	3.74	3.56	3.93
	Photography	**News**	**Entertainment**	**Music & Audio**	**Productivity**	**Health & Fitness**	**Tools**	**Books & Ref.**	**Other**	
	5.68	5.60	5.56	5.49	5.40	5.32	5.16	5.09	5.98	
	3.48	3.23	3.75	3.46	3.61	3.33	3.15	3.45	3.81	

(`com.google.firebase`) library in 55.4% of apps. It implements root detection in 55.2%, debug detection in 53.7%, and emulation detection in 53.0% of apps. Finally, *AppsFlyer* (`com.appsflyer`), present in 33.1% of apps, implements hooking detection for all of those apps.

No iOS library implements a hardening technique for significant numbers of apps, apart from root detection. Here, the main libraries are *Firebase* and *GoogleUtilities*, a utilities library for Firebase. We find Firebase-related libraries in 42.0% of iOS apps, and they perform root detection in all of them. We find that *UnityFramework* offers keylogger detection in 31.9% of apps. Several libraries seem to be intended for hardening apps, but are not very prevalent: *RootBeer* (2.0% on Android), *TrustDefender* (0.6% on Android), *FraudForce* (0.4% on iOS) and *ForterSDK* (0.6% on iOS).

On Android, we found 5 libraries responsible for 6 or more protections: `com.google.android.gms`, `com.facebook`, `com.google.firebase`, `com.inmobi`, and `com.amazon`. On iOS, the only library with more than 5 protections is UnityFramework.

Takeaways. Comparing hardening technique prevalence on Android and iOS, we found:

- The adoption of RASPs shows inconsistencies between the OSes, as most apps (82.0%) have two to four hardening techniques that are only implemented on one OS (typically on Android).
- The adoption of hardening techniques differs between app categories on both OSes. Shopping, Social and Finance apps implement most.
- Apps with more privacy-sensitive permissions implement more hardening techniques. Furthermore, the usage of privacy-sensitive permissions often differs between Android and iOS apps.
- The difference in the prevalence of RASPs is largely caused by third-party libraries. 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.

6. Discussion

We discuss insights for users, developers, and researchers and compare our results to prior studies.

6.1. Insights

We reveal important insights, questioning the overall security of today's apps, calling for action from mobile developers, and pinpointing areas for future research.

Effectiveness of hardening techniques. We showed, that almost all apps implement at least one hardening technique. However, few apps use a broad set of techniques, and individual protections can be bypassed, as demonstrated in our framework. In fact, the effectiveness of self-protection depends on implementation and whether multiple layered techniques are used, as recommended by OWASP [63]. For instance, anti-tampering protection can be bypassed by either patching the responsible code or using hooking frameworks like Frida or Xposed to overwrite the return values of functions that check signatures [63]. Ruggia et al. [75] built a sandbox that can effectively bypass emulator detection on Android, while OWASP [63] further mentions the possibility of patching the apps to overwrite methods that give away the running environment. Similarly, Pradeep et al. [68] showed that certificate pinning can be bypassed by using Frida to disable certificate validation on both Android and iOS. All in all, the presence of a single, or a few, hardening techniques says little about effectiveness of self-protection methods, questioning the overall security of today's apps. Worse, the current adoption may give both users and developers a false sense of security.

Android vs. iOS. Apps on iOS significantly underperform in self-protection when compared to the respective Android versions. 73.6% of iOS apps implementing at most 3 of the recommended techniques—opposed to 24.1% on Android; and only 26 Android and 1 iOS app adopting all. As such, we call for action from developers, especially on iOS, to implement a more diverse set of techniques.

Effectiveness of analysis approaches. Several prior studies leveraged and improved static and dynamic approaches to analyze mobile apps and track the adoption of hardening techniques. However, while most of the self-protection measures in today's popular apps are easy to analyze, we show that this is not the case for some highly sophisticated mechanisms present in our industry dataset, which intertwine hardening techniques with code obfuscation. Besides, apps in this dataset are more likely to crash upon detection of a tampered environment. Thus, despite the recent advances, more research is needed.

6.2. Trends

We compare our results to the related work present in Table 1 to show how the adoption of hardening techniques developed over time. As most prior work focuses on Android, we compare only Android results—except for certificate pinning, which has also been studied on iOS.

Total prevalence. Zungur et al. [98] found in 2021 that 25.1% of the analyzed financial apps implement no hard-

ening technique. Ruggia et al. [75] (2024) found only 9.2% of their apps to adopt not a single RASP and we find that only 0.2% of our Android apps to implement no hardening technique, suggesting that hardening techniques on Android have become prevalent.

Integrity: anti-tampering protection. In 2021, Zungur et al. [98] found that 47.5% of their dataset implements signature detection, and Ibrahim et al. [42] detected the invocation of SafetyNet in 0.3% of their analyzed apps. Ruggia et al. [75] (2024) detected the usage of any function related to SafetyNet in 35.7% of their benign dataset and the usage of Play Integrity in 0.11%. We find SafetyNet used for anti-tampering protection in 4.2% of our apps and Play Integrity in 9.3%. The increased adoption of Play Integrity indicates that many apps that previously used SafetyNet now migrated to the newer API.

Integrity & Environment: hooking detection. Zungur et al. [98] (2021) found hooking detection in 31.7% of their dataset. Ruggia et al. [75] (2024) state that, within their benign dataset, 13% check for process artifacts and 6.5% search for installed apps. We find over 55% of our apps to detect different hooking frameworks. However, a significant amount of these identifications is caused by libraries. This might indicate that popular libraries included hooking detection in recent years.

Environment: anti-debugging protection. Zungur et al. [98] (2021) found 43% of apps in their dataset to detect debugging. We observe debugger detection in almost 94% of Android apps, showing a significant increase in the usage of this technique.

Environment: emulation detection. In 2021, Zungur et al. [98] found emulation detection in 42.2% of their apps. Three years later, Ruggia et al. [75] detected such technique in 81% of their dataset, while we detect it in almost 99% of our apps. This suggests that nowadays apps are sensitive to the environment where they are run.

Environment: root detection. In 2015, Evans et al. [28] found that 80% of security and mobile device management apps detect root artifacts. In 2019, Kellner et al. [45] showed that 59% of apps across all categories and 66% of their investigated banking apps implement root detection. Ruggia et al. [75] (2024) found root detection in 61% of their benign apps. We observe root detection in 92.3% of our Android dataset, showing an increased adoption rate.

I/O: certificate pinning. In 2015, Reaves et al. [71] analyzed 7 Android apps and found 28 SSL/TLS vulnerabilities. In 2022, Pradeep et al. [68] identified certificate pinning in 6.7% and 11.4% of the analyzed Android and iOS apps. We find 25.6% of Android apps and 28.6% of iOS apps to implement certificate pinning, showing a significant adoption increase.

7. Limitations & Future Work

Technical limitations. Since we depend on tracking specific method calls and fields in apps, our framework might miss certain hardening techniques that are implemented in an unconventional way, e.g., using a classifier to distinguish between physical Android devices and known sandboxes [47], as well as high-level or custom implementations of hardening techniques. Also, our static analysis cannot always determine whether a method call corresponds to a hardening technique when this depends on the arguments passed to the method or how the return value is processed. Future research can investigate how to combine static and dynamic analysis to determine if a method call is related to a hardening technique and how to handle custom implementations automatically.

Differential analysis. Since we perform detection analysis, our framework is unable to determine how an app responds to the detection of a hardening technique, i.e., if it leads to behavioral changes. When testing HALY on both emulators and rooted devices, no concrete conclusion can be drawn. For this, we need to capture additional information (e.g., state snapshots). We plan to extend HALY with a differential analysis phase and combine the results of both approaches.

Privacy leakage. Self-protection can be a double-edged sword, as it can be adopted by apps to hide malicious behavior. Apps have been shown to adopt obfuscation to hide leaking private information [23]. Future research can investigate the relation between the usage of hardening techniques and the occurrence of privacy leakage.

App exploration. We did not use any app exploration. We assume that apps have an interest in running their hardening techniques at the earliest possible stage and that we are thus able to detect adoption without any app interaction. Furthermore, Pradeep et al. [68] found that random interactions made no significant changes to the resulting network traffic.

Detectable hardening techniques. Our framework does not focus on the analysis of code obfuscation and device binding [80]. Obfuscation targets static analysis and takes many forms. In apps, however, obfuscation mainly leads to false negatives during static analysis, which HALY overcomes with dynamic analysis. In applications like malware, where both obfuscation is applied and environment checking leads to a crash, HALY will most likely fail to fully analyse the application. The implementation of device binding is highly app-specific, which makes both difficult to detect automatically and investigating the prevalence of these techniques requires more research.

Hardening bypassing and advanced hardening techniques. Our bypasses for root and hooking detection are not perfect. Thus, there may be a few apps that did not execute all their hardening checks. Furthermore, a few apps failed dynamic analysis since they detected Frida, especially in the dataset of advanced Android apps (Section 5.4). This, however, occurs in a small portion of our popular app dataset, without affecting the significance of our findings. Further, if dynamic analysis fails, HALY still provides partial results based on static analysis.

Threats to validity. In general, determining with absolute certainty whether or not an app implements protection against specific threats is fundamentally impossible due (ultimately) to the halting problem. Instead, we report on the results of state-of-the-art techniques for the detection of self-protection, tuned for the avoidance of false positives where applicable. We assume that the wide range of detection techniques, both static and dynamic, and explicit (albeit non-exhaustive) validation efforts result in robust and meaningful results overall, even if a small number

of false positives/negatives may still be present for some apps. Hypothetical pervasiveness of more sophisticated self-protection that cannot be detected by any of our techniques would skew these results.

8. Related Work

While prior work has surveyed hardening techniques on Android [80], we are the first to comprehensively study app hardening techniques across mobile ecosystems. Table 1 summarizes the extent to which prior research considered these hardening techniques and on which datasets. Here, we summarize these studies.

Broader studies. AppJitsu [98] uses dynamic analysis to detect the presence of five hardening techniques in a few hundred Android apps. In contrast, HALY uses both static and dynamic analysis to detect seven techniques in thousands of apps on both Android and iOS. Since it is difficult to determine if an app's refusal to run on an analysis engine due to emulator detection, root detection, hooking detection, or a combination of these, its results are more limited than ours. Unlike HALY, DroidDungeon by Ruggia et al. [75] analyzes *evasion* techniques of malicious and benign Android apps using a probe-based sandbox, rather than the prevalence of all recommended hardening techniques across Android and iOS or the consistency thereof in cross-platform apps. Suo et al. [85] conducted a manual static analysis to identify specific implementations of hardening techniques by generating fingerprints. They developed a tool that combines static and dynamic analysis to detect these fingerprints in applications. However, unlike our work, their focus is limited to Android, while we evaluate a broader range of hardening techniques across multiple platforms. Sihag et al. [80] survey hardening techniques and obfuscation of benign and malicious Android apps, and elaborate their effectiveness based on existing literature. In contrast, our work conducts an empirical analysis of apps to assess the implementation of hardening techniques.

Android SafetyNet. Ibrahim et al. [42] analyze the popularity of Android's SafetyNet Attestation API, the predecessor of the Play Integrity API. Analyzing 163k apps, they find only 62 that invoke SafetyNet attestation and none that correctly implement the SafetyNet API in full. In our experiments, 111 apps clearly used SafetyNet, while attestation was present in only one—supporting their claim that apps do not implement SafetyNet properly.

RASP products. Haupert et al. [40] demonstrate two vulnerabilities in Promon Shield, one of the leading RASP products for Android, that made it possible to disable all offered protections statically and dynamically.

Root and jailbreak detection. Kellner et al. [45] dynamically analyze jailbreak detection in 34 banking apps and 3,482 popular apps on iOS. They find that 59% of the popular apps and a comparable fraction (53%) of the banking apps implement it. In contrast, our dynamic analysis finds jailbreak detection in just 25% of iOS apps. Similarly, Evans et al. [28] study the prevalence of root detection in 16 security apps and 19 enterprise mobile device management apps on Android and find 13 security apps and 15 device management apps that employ root detection. In our results, the popularity of root detection on Android is high (92.3%). Sun et al. [84] examined various root detection techniques and analyzed 182 Android apps, concluding that these methods are ineffective.

Debug and anti-tamper detection. Berlato et al. [13] study anti-debugging and anti-tampering adoption in 14k (2015) and 23k (2019) benign Android apps, finding that 59% lacked both protections and that apps in the newer dataset implemented more security measures.

Certificate pinning. Multiple studies analyze the use of *individual* hardening techniques. Fahl et al. [29] analyzed potential vulnerabilities against MITM attacks in 13,500 Android apps and found that around 8% are vulnerable. Pradeep et al. [68] compare the prevalence of certificate pinning in Android and iOS apps, using static analysis and differential dynamic analysis. Interestingly, they find that 19.7% (Android) and 33.4% (iOS) of apps include embedded certificates, while HALY discovers more apps that embed certificates on Android (25.6%) and less on iOS (28.6%), even though it uses a similar methodology for static analysis. Reaves et al. [71], investigating the communication security of 46 branchless banking apps on Android, found that most fail to protect the financial information properly.

Other protections. While not the focus of our study, related work has also studied the use of code transformation (obfuscation, encryption, and packing) to thwart reverse engineering [4], [12], [22], [39], [56], [57], [65], [70], [97]. Likewise, several studies focus on identification (and reversing) of packers [27], [92]–[94], [96].

Malicious behavior. While hardening techniques are commonly used to protect legitimate applications, they are also leveraged by malware to resist analysis and bypass security mechanisms. Various studies have explored how hardening techniques can enable malicious apps to evade detection [17], [47], [53], [55], [59], [89]. Similarly, related work has studied how sandboxes for Android can be made robust against evasion [15], [75] or how to effectively use real devices for analysis [58].

Summary. Compared to prior work, we cover more recommended hardening techniques [60], [63], more types of analysis, a large-scale dataset of apps, as well as offering cross-correlation across both mobile ecosystems.

9. Conclusion

We performed an in-depth comparative study of the adoption of self-protection in Android and iOS apps. First, we reviewed and categorized hardening techniques available in both mobile ecosystems. Next, we implemented a framework, HALY, which combines state-of-the-art static and dynamic analysis to track such techniques. We then analyzed 2,646 popular apps available on both OSes. Our results revealed that iOS apps underperform in adopting self-protection in comparison to their Android counterparts, that adoption differs across app categories, and that many apps implement hardening techniques on only one OS. Finally, we revealed that 24.1% (Android) and 73.6% (iOS) apps implement fewer than half of the recommended self-protection techniques, and only 26 Android app adopt all eight and only one iOS app adopts all seven.

Acknowledgments

We would like to thank our reviewers for their valuable comments and inputs to improve our paper. We would also like to wholeheartedly thank Uzzetti for her unlimited, never-ending support and inspiration. This work is based on research supported by the Vienna Science and Technology Fund (WWTF) and the City of Vienna [Grant ID: 10.47379/ICT22060 and Grant ID: 10.47379/ICT19056], the Austrian Science Fund (FWF) [Grant ID: 10.55776/F8515-N] and SBA Research (SBA-K1 NGC), a COMET Center within the COMET – Competence Centers for Excellent Technologies Programme and funded by BMIMI, BMWET, and the federal state of Vienna. The COMET Programme is managed by FFG. This work has also been partially supported by the Government of Canada's New Frontiers in Research Fund (NFRF), NFRFE-2019-00806, by the project P6 (Open Technology Programme No. 20475) funded by the Dutch Research Council (NWO), and by the Dutch Ministry of Economic Affairs and Climate Policy (EZK) through the AVR project "FirmPatch".

References

[1] Keysight Riscure Security Solutions. https://www.keysight.com/.

[2] Ajin Abraham. libsast. https://github.com/ajinabraham/libsast, 2025.

[3] Ajin Abraham, Magaofei, Matan Dobrushin, and Vincent Nadal. Mobile Security Framework (MobSF). https://github.com/MobSF/Mobile-Security-Framework-MobSF, 2023.

[4] Hojjat Aghakhani, Fabio Gritti, Francesco Mecca, Martina Lindorfer, Stefano Ortolani, Davide Balzarotti, Giovanni Vigna, and Christopher Kruegel. When Malware is Packin' Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2020.

[5] Efthimios Alepis and Constantinos Patsakis. Trapped by the UI: The Android Case. In *Proc. of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2017.

[6] Majd Alfhaily. IPATool. https://github.com/majd/ipatool, 2023.

[7] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proc. of the International Conference on Mining Software Repositories (MSR)*, 2016.

[8] Omar Alrawi, Chaz Lever, Mano Antonakakis, and Fabian Monrose. SoK: Security Evaluation of Home-Based IoT Deployments. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2019.

[9] Apple Inc. Establishing your app's integrity | Apple Developer Documentation. https://developer.apple.com/documentation/devicecheck/establishing_your_app_s_integrity.

[10] Apple Inc. Protected resources | Apple Developer Documentation. https://developer.apple.com/documentation/bundleresources/information_property_list/protected_resources.

[11] Ilker Avci. gplay-downloader. https://github.com/ikolomiko/gplay-downloader, 2023.

[12] Alessandro Bacci, Alberto Bartoli, Fabio Martinelli, Eric Medvet, and Francesco Mercaldo. Detection of Obfuscation Techniques in Android Applications. In *Proc. of the International Conference on Availability, Reliability and Security (ARES)*, 2018.

[13] Stefano Berlato and Mariano Ceccato. A Large-Scale Study on the Adoption of Anti-Debugging and Anti-Tampering Protections in Android Apps. *Journal of Information Security and Applications*, 2020.

[14] bmax121. Apatch. https://github.com/bmax121/APatch, 2024.

[15] Lorenzo Bordoni, Mauro Conti, and Riccardo Spolaor. Mirage: Toward a Stealthier and Modular Malware Analysis Sandbox for Android. In *Proc. of the European Symposium on Research in Computer Security (ESORICS)*, 2017.

[16] Marcus Botacin, Anatoli Kalysch, and André Grégio. The Internet Banking [in]Security Spiral: Past, Present, and Future of Online Banking Protection Mechanisms based on a Brazilian case study. In *Proc. of the International Conference on Availability, Reliability and Security (ARES)*, 2019.

[17] Alexei Bulazel and Bülent Yener. A Survey On Automated Dynamic Malware Analysis Evasion and Counter-Evasion: PC, Mobile, and Web. In *Proc. of the Reversing and Offensive-Oriented Trends Symposium (ROOTS)*, 2017.

[18] Junsung Cho, Geumhwan Cho, and Hyoungshick Kim. Keyboard or Keylogger?: A Security Analysis of Third-party Keyboards on Android. In *Proc. of the Annual Conference on Privacy, Security and Trust (PST)*, 2015.

[19] Jeremy Clark and Paul C. van Oorschot. SoK: SSL and HTTPS: Revisiting Past Challenges and Evaluating Certificate Trust Model Enhancements. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2013.

[20] European Commission. Cyber Resilience Act (CRA). https://digital-strategy.ec.europa.eu/en/policies/cyber-resilience-act, 2022.

[21] European Commission. Digital Operational Resilience Act (DORA). https://www.eiopa.europa.eu/digital-operational-resilience-act-dora_en, 2023.

[22] Mauro Conti, Vinod P., and Alessio Vitella. Obfuscation detection in Android applications using deep learning. *Journal of Information Security and Applications*, 70, 2022.

[23] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[24] Corellium. Corellium virtual hardware. https://www.corellium.com/.

[25] Aldo Cortesi, Maximilian Hils, and Thomas Kriechbaumer. mitmproxy. https://mitmproxy.org/.

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

[27] Yue Duan, Mu Zhang, Abhishek Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and Xiaofeng Wang. Things You May Not Know About Android (Un)Packers: A Systematic Study based on Whole-System Emulation. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[28] Nathan S. Evans, Azzedine Benameur, and Yun Shen. All your Root Checks are Belong to Us: The Sad State of Root Detection. In *Proc. of the ACM International Symposium on Mobility Management and Wireless Access (MobiWac)*, 2015.

[29] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security. In *Proc. of the ACM Symposium on Information, Computer and Communications Security (CCS)*, 2012.

[30] Yanick Fratantonio, Chenxiong Qian, Simon P Chung, and Wenke Lee. Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2017.

[31] Bogo Giertler. Hacking native ARM64 binaries to run on the iOS Simulator. https://bogo.wtf/arm64-to-sim.html, 2021.

[32] Google LLC. About the SafetyNet Attestation API deprecation | Android Developers. https://developer.android.com/privacy-and-security/safetynet/deprecation-timeline.

[33] Google LLC. Manifest.permission | Android Developers. https://developer.android.com/reference/android/Manifest.permission.

[34] Google LLC. Play Integrity API | Google Play. https://developer.android.com/google/play/integrity.

[35] Google LLC. Provide information for Google Play's Data safety section. https://support.google.com/googleplay/android-developer/answer/10787469#independent_security_review.

[36] Google LLC. SafetyNet Attestation API | Android Developers. https://developer.android.com/training/safetynet/attestation.

[37] Google LLC. Cydia Substrate - Android Apps on Google Play. https://play.google.com/store/apps/details?id=com.saurik.substrate, 2013. Archived at https://web.archive.org/web/20170110195857/https://play.google.com/store/apps/details?id=com.saurik.substrate.

[38] Google LLC. codesearch: Fast, indexed regexp search over large file trees. https://github.com/google/codesearch, 2020.

[39] Mahmoud Hammad, Joshua Garcia, and Sam Malek. A Large-Scale Empirical, Study on the Effects of Code Obfuscations on Android Apps and Anti-Malware Products. In *Proc. of the International Conference on Software Engineering (ICSE)*, 2018.

[40] Vincent Haupert, Dominik Maier, Nicolas Schneider, Julian Kirsch, and Tilo Müller. Honey, I Shrunk Your App Security: The State of Android App Hardening. In *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2018.

[41] hzzheyang. strongr-frida-android. https://github.com/hzzheyang/strongR-frida-android, 2024.

[42] Muhammad Ibrahim, Abdullah Imran, and Antonio Bianchi. SafetyNOT: On the Usage of the SafetyNet Attestation API in Android. In *Proc. of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2021.

[43] Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. Morpheus: Automatically Generating Heuristics to Detect Android Emulators. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2014.

[44] Anatoli Kalysch, Davide Bove, and Tilo Müller. How Android's UI Security is Undermined by Accessibility. In *Proc. of the Reversing and Offensive-Oriented Trends Symposium (ROOTS)*, 2018.

[45] Ansgar Kellner, Micha Horlboge, Konrad Rieck, and Christian Wressnegger. False Sense of Security: A Study on the Effectivity of Jailbreak Detection in Banking Apps. In *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[46] Konrad Kollnig, Anastasia Shuba, Reuben Binns, Max Van Kleek, and Nigel Shadbolt. Are iPhones Really Better for Privacy? A Comparative Study of iOS and Android Apps. In *Proc. on Privacy Enhancing Technologies (PETS)*, 2022.

[47] Brian Kondracki, Babak Amin Azad, Najmeh Miramirkhani, and Nick Nikiforakis. The Droid is in the Details: Environment-aware Evasion of Android Sandboxes. In *Proc. of the 29th Network and Distributed System Security Symposium (NDSS)*, 2022.

[48] Shuang Li, Rui Li, Shishuai Yang, and Wenrui Diao. Android's Cat-and-Mouse Game: Understanding Evasion Techniques against Dynamic Analysis. In *Proc. of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2024.

[49] Xinyue Liang and Jun Ma. A Study on Screen Logging Risks of Secure Keyboards of Android Financial Apps. In *Proc. of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022.

[50] Jae-do Lim, Il-kyu Kim, Namsu Kim, BooJoong Kang, and Seong-je Cho. A Study on Android Emulator Detection Using Build Properties. In *Proc. of the International Conference on Next Generation Computing*, 2022.

[51] Chia-Chi Lin, Hongyang Li, Xiao-yong Zhou, and XiaoFeng Wang. Screenmilker: How to Milk Your Android Screen for Secrets. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[52] Jie Lin, Chuanyi Liu, and Binxing Fang. Out-of-Domain Characteristic Based Hierarchical Emulator Detection for Mobile. In *Proc. of the International Conference on Information Technologies and Electrical Engineering (ICITEE)*, 2020.

[53] Dominik Maier, Mykola Protsenko, and Tilo Müller. A Game of Droid and Mouse: The Threat of Split-Personality Malware on Android. *Computers & Security*, 2015.

[54] Prianka Mandal, Amit Seal Ami, Victor Olaiya, Sayyed Hadi Razmjo, and Adwait Nadkarni. "Belt and suspenders" or "just red tape"?: Investigating Early Artifacts and User Perceptions of IoT App Security Certification. In *Proc. of the USENIX Security Symposium*, 2024.

[55] Guozhu Meng, Yinxing Xue, Chandramohan Mahinthan, Annamalai Narayanan, Yang Liu, Jie Zhang, and Tieming Chen. Mystique: Evolving Android Malware for Auditing Anti-Malware Tools. In *Proc. of ACM ASIA Conference on Computer and Communications Security (ACM AsiaCCS)*, 2016.

[56] O. Mirzaei, J. M. de Fuentes, J. Tapiador, and L. Gonzalez-Manzano. AndrODet: An adaptive Android Obfuscation Detector. *Future Generation Computer Systems*, 2019.

[57] Alireza Mohammadinodooshan, Ulf Kargén, and Nahid Shahmehri. Robust Detection of Obfuscated Strings in Android Apps. In *Proc. of the ACM Workshop on Artificial Intelligence and Security*, 2019.

[58] Simone Mutti, Yanick Fratantonio, Antonio Bianchi, Luca Invernizzi, Jacopo Corbetta, Dhilung Kirat, Christopher Kruegel, and Giovanni Vigna. BareDroid: Large-Scale Analysis of Android Apps on Real Devices. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2015.

[59] Sebastian Neuner, Victor Van der Veen, Martina Lindorfer, Markus Huber, Georg Merzdovnik, Martin Mulazzani, and Edgar Weippl. Enter Sandbox: Android Sandbox Comparison. In *Proc. of the IEEE Mobile Security Technologies Workshop (MoST)*, 2014.

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

[61] OWASP Foundation, Inc. OWASP MAS Crackmes Android. https://mas.owasp.org/crackmes/Android, 2024.

[62] OWASP Foundation, Inc. OWASP MAS Crackmes iOS. https://mas.owasp.org/crackmes/iOS, 2024.

[63] OWASP Foundation, Inc. OWASP MASTG. https://mas.owasp.org/MASTG/, 2024.

[64] palera1n. palera1n. https://palera.in/.

[65] Minjae Park, Geunha You, Seong-je Cho, Minkyu Park, and Sangchul Han. A Framework for Identifying Obfuscation Techniques applied to Android Apps using Machine Learning. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, 2019.

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

[67] Stephan Petzl. Accessing Launch Environment and Launch Arguments in Xcode UI Tests. https://www.repeato.app/accessing-launch-environment-and-launch-arguments-in-xcode-ui-tests/, 2024.

[68] Amogh Pradeep, Muhammad Talha Paracha, Protick Bhowmick, Ali Davanian, Abbas Razaghpanah, Taejoong Chung, Martina Lindorfer, Narseo Vallina-Rodriguez, Dave Levin, and David Choffnes. A Comparative Analysis of Certificate Pinning in Android & iOS. In *Proc. of the ACM Internet Measurement Conference (IMC)*, 2022.

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

[70] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks. In *Proc. of ACM ASIA Conference on Computer and Communications Security (ACM AsiaCCS)*, 2013.

[71] Bradley Reaves, Jasmine Bowers, Nolen Scaife, Adam Bates, Arnav Bhartiya, Patrick Traynor, and Kevin R. B. Butler. Mo(bile) Money, Mo(bile) Problems: Analysis of Branchless Banking Applications. *ACM Transactions on Privacy and Security (TOPS)*, 2017.

[72] RedNaga. APKiD. https://github.com/rednaga/APKiD, 2022.

[73] Wojciech Reguła. Iossecuritysuite. https://github.com/securing/IOSSecuritySuite.

[74] rovo89. Xposed framework. https://github.com/rovo89/Xposed, 2017.

[75] Antonio Ruggia, Dario Nisi, Savino Dambra, Alessio Merlo, Davide Balzarotti, and Simone Aonzo. Unmasking the Veiled: A Comprehensive Analysis of Android Evasive Malware. In *Proc. of ACM ASIA Conference on Computer and Communications Security (ACM AsiaCCS)*, 2024.

[76] SaurikIT, LLC. Cydia Substrate. http://www.cydiasubstrate.com/.

[77] Bruce Schneier. *Beyond Fear: Thinking Sensibly about Security in an Uncertain World*. Springer-Verlag, Berlin, Heidelberg, 2003.

[78] Silvia Sebastián and Juan Caballero. AVclass2: Massive Malware Tag Extraction from AV Labels. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2020.

[79] Jan Seredynski. Demystifying typical mobile game cheats. https://www.guardsquare.com/blog/demystifying-typical-mobile-game-cheats, 2021.

[80] Vikas Sihag, Manu Vardhan, and Pradeep Singh. A Survey of Android Application and Malware Hardening. *Computer Science Review*, 2021.

[81] Sileo Team. Sileo. https://getsileo.app/.

[82] Squid. Squid : Optimising Web Delivery. http://www.squid-cache.org/.

[83] Magdalena Steinböck, Jakob Bleier, Mikka Rainer, Tobias Urban, Christine Utz, and Martina Lindorfer. Comparing Apples to Androids: Discovery, Retrieval, and Matching of iOS and Android Apps for Cross-Platform Analyses. In *Proc. of the International Conference on Mining Software Repositories (MSR)*, 2024.

[84] San-Tsai Sun, Andrea Cuadros, and Konstantin Beznosov. Android Rooting: Methods, Detection, and Evasion. In *Proc. of the ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2015.

[85] Dewen Suo, Lei Xue, Runze Tan, Weihao Huang, and Guozi Sun. ARAP: Demystifying Anti Runtime Analysis Code in Android Apps. *arXiv preprint arXiv:2408.11080*, 2024.

[86] talsec. Freerasp. https://github.com/talsec/Free-RASP-iOS.

[87] Connor Tumbleson. Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps. https://ibotpeaches.github.io/Apktool/, 2022.

[88] Ole André Vadla Ravnås. Frida. https://frida.re/.

[89] Timothy Vidas and Nicolas Christin. Evading Android Runtime Analysis via Sandbox Detection. In *Proc. of ACM ASIA Conference on Computer and Communications Security (ACM AsiaCCS)*, 2014.

[90] Liu Wang, Haoyu Wang, Ren He, Ran Tao, Guozhu Meng, Xiapu Luo, and Xuanzhe Liu. MalRadar: Demystifying Android Malware in the New Era. *Proc. of the ACM on Measurement and Analysis of Computing Systems*, 2022.

[91] WithSecure. A Guide to Repacking iOS Applications. https://labs.withsecure.com/publications/repacking-and-resigning-ios-applications, 2018.

[92] Lei Xue, Yuxiao Yan, Luyi Yan, Muhui Jiang, Xiapu Luo, Dinghao Wu, and Yajin Zhou. Parema: An Unpacking Framework for Demystifying VM-Based Android Packers. In *Proc. of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2021.

[93] Lei Xue, Hao Zhou, Xiapu Luo, Yajin Zhou, Yang Shi, Guofei Gu, Fengwei Zhang, and Man Ho Au. Happer: Unpacking Android Apps via a Hardware-Assisted Approach. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2021.

[94] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu. AppSpear: Bytecode Decrypting and DEX Reassembling for Packed Android Malware. In *Proc. of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2015.

[95] Swaroop Yermalkar. OWASP iGoat. https://github.com/OWASP/iGoat-Swift, 2021.

[96] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. DexHunter: Toward Extracting Hidden Code from Packed Android Applications. In *Proc. of the European Symposium on Research in Computer Security (ESORICS)*, 2015.

[97] Min Zheng, Patrick P. C. Lee, and John C. S. Lui. ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-virus Systems. In *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2013.

[98] Onur Zungur, Antonio Bianchi, Gianluca Stringhini, and Manuel Egele. AppJitsu: Investigating the Resiliency of Android Applications. In *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2021.

Appendix

1. Extended Evaluation Results

Build checks. Table 5 shows the most common `Build` checks in our Android dataset. `FINGERPRINT`, `TAGS`, `TYPE` and `DEVICE` were present in over 70% of apps.

TABLE 5: Most common `Build` checks on Android.

Variable	Value	Occurrence
`FINGERPRINT`	contains(`generic`)	94.6%
`TAGS`	contains(`dev-keys` || `test-keys`)	85.0%
`TYPE`	equals(`eng` || `userdebug`)	81.5%
`TAGS`	contains(`test-keys`)	72.1%
`TAGS`	contains(`dev-keys`)	71.8%
`DEVICE`	startsWith(`generic`)	70.0%
`PRODUCT`	contains(`sdk`)	68.2%
`BRAND`	startsWith(`generic`)	66.2%
`MANUFACTURER`	contains(`Genymotion`)	65.7%
`HARDWARE`	contains(`ranchu`)	49.1%

Root- and jailbreak detection checks. Table 6 shows the most frequently searched apps associated with root and jailbreak detection. On Android, apps primarily check for the presence of SuperSu and similar apps, whereas on iOS they mostly look for package managers like Cydia. In Table 7 we list the most commonly searched files on both platforms. Similarly, on Android, apps predominantly search for files related to the `su`-binary, while on iOS, they look for artifacts indicative of jailbreaking (e.g., directories with different permissions) and package managers.

TABLE 6: Common detections of root- and jailbreak-related apps. The most commonly searched for app is *supersu* on Android and *Cydia* on iOS.

Android	Occur.	iOS	Occur.
`eu.chainfire.supersu`	13.4%	`Cydia`	28.3%
`com.noshufou.android.su`	13.4%	`Icy`	5.5%
`com.koushikdutta.superuser`	13.2%	`blackra1n`	5.5%
`com.thirdparty.superuser`	13.2%	`RockApp`	5.4%
`com.devadvance.rootcloakplus`	10.0%	`Sileo`	2.0%
`com.devadvance.rootcloak`	10.0%	`Undecimus`	1.9%
`com.noshufou.android.su.elite`	7.5%	`Zebra`	0.1%
`com.yellowes.su`	7.5%		
`com.ramdroid.appquarantine`	5.7%		
`com.topjohnwu.magisk`	2.0%		

Obfuscators and packers. Table 8 and Table 9 show the most frequently observed obfuscators and packers respectively in our full dataset, apps that completed analysis and the industry dataset. Overall, the prevalence of packers and obfuscators was low. The most common obfuscator in our datasets was *DexGuard 9.x* and the most common packers were *DexProtector* and *Promon Shield*.

Third-party libraries. Table 10 shows the most popular third-party libraries in our Android dataset. We found

TABLE 7: Common detections of root- and jailbreak-related files. On Android, most apps search for the su-binary, while on iOS, they look for apps and artifacts in relation to jailbreaking. * This directory also exist on normal devices, but might have different permissions.

	Android files	iOS files
1	/system/app/Superuser.apk	/private*
2	/system/xbin/su	/Applications/Cydia.app
3	/system/bin/su	/etc/apt
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	/system/library/launchdaemons
9	/system/bin/failsafe/su	/var/lib/cydia
10	/su/bin/su	/usr/libexec/sftp-server

TABLE 8: Most common obfuscators in the popular apps dataset [83], our dataset as evaluated, and the dataset from our industry partner (Industry Dataset). Note that Obfuscator-LLVM is grouped with all various versions that were found. *DexGuard 9.x* and *Arxan* are the most common obfuscators in the full and industry dataset, while *Kiwi encrypter* is most common in the analyzed dataset.

Name	Popular Apps	Analyzed	Industry
DexGuard 9.x	52	31	4
Arxan	52	29	1
Kiwi encrypter	44	35	0
Obfuscator-LLVM	39	19	1
DexGuard	28	22	2
Obfuscator-LLVM (string encryption)	19	16	0
DexProtector	12	2	1
ADVobfuscator	4	3	0
Alipay	4	4	0
Allatori demo	2	2	0
Gemalto	2	1	0
Hikari	1	1	0
Safeengine LLVM	1	1	0

TABLE 9: Most common packers in the popular apps dataset [83], our analyzed dataset, and the dataset from our industry partner. *DexProtector* and *Promon Shield* are the most common packers used.

Name	Popular Apps	Analyzed	Industry
DexProtector	14	3	1
Promon Shield	14	0	1
Ijiami	4	0	0
UPX	4	0	0
yidun	4	1	0
LIAPP	4	1	0
SecNeo.B	2	0	0
AppSealing	2	1	0
Bangcle	2	0	0
CrackProof	2	0	0
Bangcle (SecShell)	1	1	0
DxShield	1	0	0
SecNeo.A	1	0	0
AppGuard	1	0	0
Baidu	0	0	0

Google Mobile Services in over 96% of the apps, as well as *Firebase* and *Facebook* libraries in around half of our dataset, all implementing a range of hardening techniques. In Table 11 we list the most popular third-party libraries in our iOS dataset. In contrast to Android, most of them only implement root detection, except for *UnityFramework*, which also implements additional hardening techniques.

2. Case Study: Malware Samples

Table 12 shows the Android malware samples we evaluated including their hash values and family classification.

TABLE 10: Most popular third-party Android libraries with implemented hardening techniques (in %). *Google Mobile Services* is present in almost every app in our dataset and provides debug, emulation and root detection.

Library	Occ.	Anti-debug	Anti-emulation	Anti-hooking	Anti-keylogger	Cert. Pinning	Anti-root	Anti-tamper
com.google.android.gms	96.3	76.0	96.1	2.6	—	0.04	59.8	0.04
com.google.firebase	55.4	53.7	53.0	9.5	—	—	55.2	9.4
com.facebook	49.4	0.6	46.9	0.2	—	—	9.5	—
com.unity3d	47.4	42.6	0.2	—	—	—	26.9	—
audience_network.com.facebook.ads.redexgen.X	43.8	43.8	—	—	—	—	—	—
com.google.android.play.core	43.6	—	43.6	—	—	—	—	—
com.applovin	38.6	1.8	38.6	—	0.3	—	4.3	—
com.ironsource	37.8	37.8	—	—	—	—	2.3	—
libcrashlytics-common.so	35.5	35.5	—	—	—	—	—	—
com.appsflyer	33.1	—	—	33.1	—	—	—	—

TABLE 11: Most popular third-party iOS libraries with implemented hardening techniques (in %). All libraries implement root detection. Only *UnityFramework* implements a wider range of hardening techniques.

Library	Occ.	Anti-debug	Anti-emulation	Anti-hooking	Anti-keylogger	Cert. Pinning	Anti-root	Anti-tamper
GoogleUtilities	44.1	—	—	—	—	—	44.1	—
FirebaseCore	39.2	—	—	—	—	—	39.2	—
GoogleDataTransport	37.3	—	—	—	—	—	37.3	—
FirebaseCrashlytics	32.3	—	—	—	—	—	32.3	—
UnityFramework	32.0	3.4	—	1.5	31.9	—	28.2	0.1
FBSDKLoginKit	28.0	—	—	—	—	—	28.0	—
FirebaseRemoteConfig	20.0	—	—	—	—	—	20.0	—
FirebaseCoreDiagnostics	17.5	—	—	—	—	—	17.5	—
FirebaseMessaging	17.3	—	—	—	—	—	17.3	—
FBSDKCoreKit	16.6	—	—	—	—	—	16.6	—

TABLE 12: Analyzed malware samples with MD5 hashes, families and results (static/dynamic). Results in brackets are unconfident. Cells with a value are highlighted.

MD5	Family	Anti-debug	Anti-emulation	Anti-hooking	Anti-keylogger	Cert. Pinning	Anti-root	Anti-tamper
A910F32D6563FFA-425F192581154903B	joker	4/2 (1/0)	8/0 (0/0)	0/0 (0/0)	2/2 (0/0)	0/1 (0/1)	1/2 (0/0)	1/1 (1/1)
2C7AC97977EFA55-278FC4D5CF83D78B9	joker	5/0 (2/0)	3/0 (0/0)	0/0 (0/0)	1/0 (1/0)	0/0 (0/0)	1/0 (0/0)	1/0 (1/0)
274535A08A0D06F-8F1F3552C33FB8EEF	spyagent	0/0 (0/0)	0/0 (0/0)	0/1 (0/1)	0/0 (0/0)	0/2 (0/2)	0/0 (0/0)	0/0 (0/0)
547217D768CA6A4-6B00FC02F1AB7FB56	spyagent	3/0 (2/0)	28/0 (0/0)	0/0 (0/0)	1/0 (0/0)	0/0 (0/0)	21/0 (0/0)	1/0 (1/0)
79F681872CCC356-4D587EDB090A0A3E3	spyagent	2/0 (2/0)	8/7 (0/0)	4/3 (0/1)	1/0 (1/0)	0/2 (0/2)	2/19 (0/0)	1/1 (1/1)
D4F4B00C3DBCF19-71695E452DB3916B1	smsthief	1/0 (1/0)	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)
E992789EEAB04E0-B0E862BF9A4F40233	smsthief	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)
B25A4B632098DB2-AA0A075BDA3958B27	necro	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)
F21A53C259B48FE-3B73CBBED1C18BE0F	necro	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)
845A0C355381D4D-DF96874E5075F3356	necro	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)