Leaky Apps: Large-scale Analysis of Secrets Distributed in Android and iOS Apps

David Schmidt
University of Vienna
Faculty of Computer Science
Doctoral School Computer Science
Christian Doppler Laboratory AsTra
Vienna, Austria
d.schmidt@unvie.ac.at

Sebastian Schrittwieser
University of Vienna
Faculty of Computer Science
Christian Doppler Laboratory AsTra
Vienna, Austria
sebastian.schrittwieser@univie.ac.at

Edgar Weippl
University of Vienna
Faculty of Computer Science
Vienna, Austria
edgar.weippl@univie.ac.at

Abstract

Mobile apps store various types of secrets to support their functionalities. These include API keys, and cryptographic material to authenticate users and access backend services. Once distributed, attackers can reverse-engineer the apps, and these secrets become accessible, posing risks such as data leaks, and service abuse.

In this paper, we conduct a large-scale analysis of 10,331 Android and iOS apps to study how secrets are embedded in mobile apps. Our methodology involves extracting and validating credentials from app bundles and comparing the types and frequency of embedded secrets across Android and iOS to identify systematic differences between the two ecosystems. To assess temporal dynamics, we re-analyze apps released in 2023 after their updates in 2024.

Our findings show that apps not only leak secrets required for functionality but also unintentionally include sensitive information like markdown documentation, and dependency management files.

We discovered 416 functional credentials across 65 services, including 13 Git credentials that grant access to 218 public and 2,440 private repositories. Our analysis reveals that iOS apps are more likely to expose secrets, although information leaks exist in both Android and iOS apps. Finally, we show that even if developers remove embedded credentials in later versions, they frequently forget to revoke them, leaving the credentials exploitable.

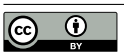
CCS Concepts

• **Security and privacy → Software and application security**.

Keywords

Mobile app security; static analysis; app measurements.

ACM Reference Format:
David Schmidt, Sebastian Schrittwieser, and Edgar Weippl. 2025. Leaky Apps: Large-scale Analysis of Secrets Distributed in Android and iOS Apps. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765033



CCS '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765033

1 Introduction

Smartphone apps have become essential in daily life, with approximately 6.9 billion smartphone users worldwide depending on over 8.9 million apps in 2023 [44]. These apps handle our communications, finances, social interactions, and personal health data, deeply integrating themselves into both our personal and professional lives. However, this widespread reliance creates significant security risks, particularly when it comes to protecting sensitive information embedded within apps. Security researchers recently uncovered that 13 widely-used mobile apps, some downloaded millions of times, exposed sensitive cloud credentials [43]. These credentials can enable attackers to manipulate or steal user data, potentially resulting in severe privacy breaches.

The issue arises because developers embed sensitive data, such as API tokens and authentication credentials, directly into the app's code. While some of these secrets are intentionally included for necessary functionality, developers also include sensitive information inadvertently due to oversight or rushed development cycles. An illustrative example is Snapchat, which unintentionally leaked portions of its source code through its iOS app [17]. Once an app reaches users' devices, its embedded secrets become vulnerable to extraction through reverse engineering, often referred to as a Man-At-The-End (MATE) threat model [24].

Given the scale of mobile app usage globally, the impact of compromised app secrets is substantial. This threat is underscored by its inclusion in the OWASP Mobile Top 10 of 2024, which ranks *Improper Credential Usage* as the highest security risk [71].

In the past, researchers studied the exposure of secrets in public code repositories. Meli et al. [57] performed a large-scale longitudinal analysis on secrets in GitHub repositories using regular expression-based pattern matching. Jungwirth et al. [50] analyzed secrets in dotfiles, which are often used as configuration files and hidden by default in most Operating Systems (OSes). Jin et al. [49] scanned GitHub for Internet of Things (IoT) cloud policies to identify misconfigured cloud services. Further, several popular rule-based analyses, such as Gitleaks[37] and TruffleHog [96], help to detect potential leaks of secrets in code repositories.

Investigating secrets in mobile apps differs from examining code repositories, primarily because repositories are explicitly intended for sharing code, making developers potentially more cautious about exposing secrets than in apps. Additionally, repositories contain source code, whereas mobile apps distribute app bundles containing compiled code, making secrets less immediately visible.

Still, the situation of secrets in mobile apps is under-explored. Earlier research primarily examined the leakage of Personally Identifiable Information (PII) from mobile apps [20, 67, 82, 81] or targeted specific types of secrets in the app's executed code [105, 107, 58]. Zhou et al. [105] employed static data flow analysis to detect AWS and email credentials in Android apps. Zuo et al. [107] proposed LeakScope, an Android app analysis methodology based on string Value Set Analysis (VSA) to identify wrongly configured AWS, Google, and Microsoft cloud credentials. Mendoza et al. [58] extracted app-to-web communication to detect vulnerabilities like web API hijacking. However, these works did not consider the wide range of potentially shared information in mobile apps or information unintentionally included in app bundles.

Our study differs from prior research in three important aspects: (1) We do not limit our analysis to the app's executable code. Snapchat previously leaked source code by accidentally packaging it with the iOS app bundle. Such unintentionally included data might hold secrets undetectable through code analysis methods. (2) Android is not the only widely used mobile OS. Despite iOS having a 58% market share in the US [45], large-scale studies of secret exposure within iOS apps remain notably absent in the literature. (3) Earlier studies typically provided only a snapshot, lacking an understanding of how secrets evolve over time, for example, whether developers later remove or revoke them.

To get a complete picture, we first study the content of app bundles, addressing *RQ1: What files do mobile apps contain?*. Those can range from binaries and configuration files to unintentionally included items, like build scripts.

After understanding what files developers distribute in their apps, we focus specifically on their contents. We use a regular expression-based detection approach, similar to those successfully used to identify secrets in code repositories. Through this, we answer *RQ2: What secrets do developers distribute in mobile apps?*

Next, we investigate platform-specific differences by performing a large-scale, comparative analysis across 10,331 Android and iOS apps. This allows us to answer *RQ3: How does the situation differ between Android and iOS apps?*

Developer responses significantly affect the security impact when secrets become public. Effective responses include revoking compromised tokens and releasing updates. To understand how developers handle such exposures over time, we updated our original 2023 dataset in 2024 and investigate *RQ4: How did the situation change between 2023 and 2024?*

In summary, we make the following key contributions:

- We performed a comprehensive analysis of files distributed within 10,331 Android and iOS apps, and were able to show that developers share sensitive information, e.g., dependency files, and documentation, by accident in the apps;
- We identified 416 valid credentials across 65 services, including highly critical findings such as 13 valid Git credentials;
- To the best of our knowledge, we conducted the first large-scale analysis, specifically examining secrets in both Android and iOS apps. Our results underscore the importance of studying apps on both platforms, as we frequently identified issues exclusive to one platform;
- We were able to highlight that even after developers remove credentials from apps, they often neglect to revoke them, leaving these credentials exposed to misuse.

Artifacts. We publish our code and analysis artifacts to make our study reproducible and to enable future work: https://github.com/CDL-AsTra/leaky_apps.

2 Threat Model, Secret Definition, and Mitigation

Threat Model. Attackers may analyze downloaded apps to extract embedded secrets in a MATE attack scenario. Once an app containing secrets is published, developers must assume that attackers can discover them. Malicious actors might download and analyze a vulnerable version while it is available. Further, older app versions remain accessible through third-party stores [6, 7], the Androzoo dataset [3], or even directly from the Google Play Store [10] or Apple App Store [2].

Secret Definition. In our paper, we consider following information as secret: (1) Information whose exposure might result in financial loss for app creators, e.g., API tokens with usage-based billing or proprietary source code; (2) Information compromising user privacy, e.g., tokens granting access to online databases; (3) Information exploitable by attackers to target users or developers, like documentation containing internal URLs useful for social engineering.

Mitigation. Our research aims to identify and responsibly disclose secrets found in apps. We also aim to provide insights that help developers detect security issues before publishing their apps.

When developers include tokens in apps, they must consider that attackers might extract them. Thus, tokens should have access strictly limited to the required scope. Further, app developers should monitor tokens closely and react immediately to any misuse, such as unintended use of Google Map tokens that could lead to costs due to pay-per-use charges.

If secrets are exposed or misused, developers should revoke them promptly and assess whether more secure alternatives to embedding them in their apps exist. They should review the scope of the credentials and investigate previous activities to detect malicious usage. Developers must also have an update strategy prepared beforehand, as without proper planning, revoking credentials could disrupt older app versions.

Applying code obfuscation might make discovering secrets more difficult, but it does not prevent manual analysis. Therefore, developers should never rely solely on obfuscation to protect sensitive data, especially when exposure could cause significant harm.

3 Methodology

We designed a static analysis methodology for large-scale analysis of Android and iOS apps to gain insights into the files and data they distribute. Figure 1 provides a high-level overview of our approach.

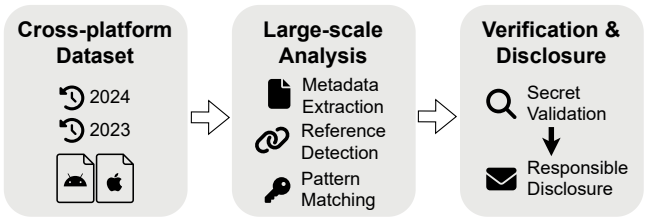


Figure 1: Overview of our methodology. We performed a large-scale analysis of 10,331 Android and iOS apps, collected twice, once in 2023 and in 2024. We validated the identified credentials and automatically disclosed them responsibly.

3.1 Large-scale Analysis

Our analysis pipeline includes three components written in Python: (1) file metadata extraction, (2) file reference detection, and (3) secret extraction using regular expressions.

Metadata Extraction. To study the contents of Android and iOS apps, we extract data from APK (Android) and IPA (iOS) files, both of which are bundled as archives [75]. This allows for a unified processing approach across both platforms. We also analyze split APK files, which Android uses to package language resources, screen densities, and native libraries [5]. We process them the same way as the main APK.

For each file within an app, we store its size, name, file path, suffix, and MIME type. This metadata supports downstream analyses, such as identifying files that may have been included unintentionally.

Reference Detection. We identify filenames referenced within other files to infer their potential use in the app. When a filename appears in other files, e.g., the app's binary, it typically indicates the file is accessed at runtime. This allows us to estimate how different files are used based on these references.

To streamline this analysis step, we exclude media files and general configuration files such as manifests or UI layouts since their usage is implicitly managed by the OS.

Pattern Matching. Pattern matching offers two key advantages over other approaches. First, it allows a unified methodology across Android and iOS, facilitating direct comparison across platforms. Second, it enables the detection of secrets in files not actively used by the app, as well as those written in other programming languages such as JavaScript (JS) or code from cross-platform frameworks.

We use TruffleHog [96] to detect secrets, adapting it from its original purpose of scanning code repositories. After reviewing its detection patterns, we extended TruffleHog with seven additional rules sourced from Gitleaks [37] to enhance its detection coverage. We include the complete list of rules as part of our artifact [89]. To improve efficiency, we separated secret verification from the initial detection process. This prevents repeated and unnecessary validation requests when the same secret appears across multiple apps. The decoupled approach allows us to pre-filter results before verification, as detailed in Section 3.2.1.

As a preprocessing step to pattern matching, we run `strings` [35] on all non-text files to extract printable strings. For Android apps, we additionally use JADX [36], a decompiler that converts DEX bytecode into Java source code, which aligns more closely with code found in public repositories.

3.2 Verification and Disclosure

3.2.1 Secret Validation. We developed our analysis methodology with attention to ethical standards and responsible disclosure practices. Validating detected secrets is necessary to avoid False Positive (FP) reports that could cause unnecessary effort for developers. Remote validation introduces the risk of incurring costs if a service charges per request. However, since we issue only one request per token and most services bill by volume, we considered the practical risk to be low. Further, it aligns with validation strategies done by related work of secret detection [29, 107].

We further reduce requests by eliminating FPs before remote validation through rule-based heuristics. We discard results if a single detection rule flags 15 or more potential secrets in one file. This threshold is based on our observation that some rules may match unrelated patterns, such as hash values. We determined this cutoff empirically and discuss the link between detection frequency and actual credentials in Section 5.1.

We only perform remote validation after this filtering step. We selected endpoints that return distinct status codes depending on token validity, see [89] for the list of services. All validation requests do not alter data on remote servers. Further, we do not retain any response content. Only status codes and error messages are used to assess whether a credential is valid.

3.2.2 Responsible Disclosure. We responsibly disclosed all findings by contacting developer email addresses listed on the Google Play Store, which we retrieved along with app packages. For reporting of findings in iOS apps, we also used the corresponding Google Play developer addresses as the Apple App Store did not offer explicit developer contact information until February 2025 (we disclosed our findings in January 2025), and our dataset links each iOS app to a matching Android version, see Section 3.3.

For each finding category, we used a template message to ensure clarity and consistency. These templates include a description of the issue and proposed mitigation steps (see Appendix A for an example). When a finding affected both the Android and iOS versions of the same app, we combined the results into one report. However, we did not merge findings across different apps, even if they were linked to the same developer account. All reports were sent via our university's mail server.

3.3 Cross-platform Dataset

To study both Android and iOS apps at scale, we used an updated version of the cross-platform dataset introduced by Schmidt et al. [86]. Their dataset includes both popular and randomly chosen apps. Using the matching method of Steinböck et al. [94], and the Google migration API, they identified 10,862 iOS apps with corresponding Android versions.

Since our analysis requires decrypted iOS apps and the original dataset included encrypted versions, we used frida [4] to decrypt their dataset. We executed frida on an iPhone 8 running iOS 16, which we had jailbroken with palera1n [72]. This step reduced the usable set of iOS apps to 10,331 as decryption failed due to anti-debugging protections [83, 106] and iOS version mismatch. We refer

Table 1: File categories found in mobile apps. We categorized files based on their suffix. The File columns show the number of files per category, while the App columns indicate the number of apps containing at least one file of the category. Note that, although every app requires a binary, the relative number is below 100%, as our analysis failed for 31 Android and 5 iOS apps due to corrupted archives.

	File	App	File	App
ai model	3,293	755 (7.31%)	2,772	491 (4.75%)
archive	17,634	7,977 (77.21%)	20,395	1,767 (17.11%)
audio	196,420	9,710 (93.99%)	219,218	5,309 (51.39%)
backup	9,333	67 (0.65%)	7,294	46 (0.45%)
binary	3,194,424	10,300 (99.70%)	965,111	10,326 (99.96%)
code	171,413	2,989 (28.93%)	238,994	2,574 (24.92%)
config	10,969,802	10,300 (99.70%)	2,905,998	10,326 (99.96%)
cryptography	10,770	1,801 (17.43%)	12,098	2,293 (22.20%)
database	46,120	4,922 (47.64%)	161,343	4,803 (46.50%)
game	220,644	3,484 (33.72%)	292,021	3,456 (33.46%)
image	7,927,423	10,297 (99.67%)	3,138,408	10,319 (99.89%)
split	87,939	1,066 (10.32%)		
spreadsheet	26,208	347 (3.36%)	11,311	372 (3.60%)
system	494,928	10,249 (99.21%)	2,145,176	10,326 (99.96%)
text	258,319	7,631 (73.87%)	178,725	5,683 (55.01%)
video	68,333	2,734 (26.46%)	70,015	1,992 (19.28%)
web	281,587	7,004 (67.80%)	247,478	6,724 (65.09%)
other	751,212	8,628 (83.51%)	690,166	3,333 (32.26%)

to this set of apps as *2023 dataset*, as the apps were downloaded from the Play Store and App Store in 2023.

To capture how the inclusion of secrets in mobile apps evolves over time, we re-downloaded the latest app versions from the dataset in October 2024 from the official stores. Of the original set, 8,702 Android apps and 9,212 iOS apps were still available. We refer to the updated version as *2024 dataset*.

3.4 Data Analysis

Significance Test. To measure the significance of differences between Android and iOS versions of the same app, as well as differences between app versions from 2023 and 2024, we perform dependent t-tests and calculate effect sizes using Cohen's d. For t-tests comparing 2023 and 2024, we include only apps that remained available in 2024. For all statistical tests, we define the null hypothesis H_0 as *"There is no difference between the two groups"*, and the alternative hypothesis H_1 as *"There is a difference"*. We reject H_0 when the resulting p-value is below the significance level of 0.05.

Case Studies. We select case studies manually based on factors such as the app's relevance, revealing names, e.g., file or directory.

4 App Content

All numbers in this section refer to the 2023 dataset, which was chosen for better comparison due to its equal number of Android and iOS apps. We provide insights into the changes between the datasets from 2023 and 2024 in Section 7.

To understand what types of files apps contain, we categorized files based on their suffix, see Table 1. We derived these categories by merging two existing GitHub projects [28, 26] and manually reviewing the 500 most frequently used suffixes that had not yet been categorized. In total, we mapped 876 suffixes to 17 file categories.

As expected, all successfully analyzed apps contained binary code, configuration files, and system files. The overall percentage is slightly lower than 100% due to analysis failures in 31 Android and five iOS apps caused by corrupted app bundles.

In the following, we report on file types that are typically not bundled with Android or iOS apps but still appeared in our analysis.

4.1 Binaries

4.1.1 Windows. We found `.exe` and `.dll` binary suffixes, which are typically associated with Windows, in 321 Android and 228 iOS apps. Manual analysis showed that 266 of these apps were built using Microsoft's cross-platform framework Xamarin [62]. These Windows-related files contain Ahead-of-time (AOT) and Just-in-time (JIT) compiled code designed for execution on mobile devices [60, 61]. Similarly, we discovered `.aspx` files in 1,450 Android apps (14.04%) and one iOS app. All occurrences on Android were linked to the Mono project [64], which is designed for cross-platform development and also used by Xamarin [60].

In other cases, the binaries came from cross-platform libraries and included Windows executables alongside the mobile versions.

4.1.2 Android and Java on iOS. We found `.apk`, `.dex`, and `.jar` files not only in Android apps but also in 57 iOS apps (0.55%). None of these are directly executable on iOS [19]. Of these apps, 43 used MobiVM [63], a framework that allows Java-based development for iOS via AOT compilation. Among the remaining 14, we found `.dex` and `.jar` files added by libraries in 11 apps. In two apps, the included `.apk` files contained animation assets rather than compiled code. The last app included a `gradle-wrapper.jar` file.

We also searched in Android apps for files with the MIME type `x-mach-binary`, which typically indicates binaries for Apple platforms. We found them in 196 Android apps (1.90%). Libraries included these files to run on multiple OSes.

Case Study: PayPal Business. The PayPal business iOS app [73] contained the `gradle-wrapper.jar` file belonging to a project that used gradle to generate Java code files with default values. Comments hinted that they also used the resulting code files to generate Swift files for iOS.

The project also contained URLs of their internal Git and artifactory. Additionally, it revealed a bug-tracking URL. Attackers could use that information for targeted social engineering attacks.

4.2 Source Code and Scripts

4.2.1 App Code. We found source code files in 28.93% of Android and 24.92% of iOS apps. However, not all code files are equally relevant. We distinguished between potential app code, e.g., Java, Kotlin, Swift, or C++, and scripts, e.g., Python or Shell. To focus on developer-authored code rather than libraries, we excluded files appearing in more than two apps. We chose this threshold because

our dataset contains both Android and iOS versions. Further, we only considered code from programming languages with at least ten files per app. We empirically determined this threshold after observing that apps with fewer files generally include them as components of libraries or package management systems. For instance, we found apps containing `Package.swift`, which we separately discuss in Section 4.2.3.

We found code files meeting our criteria in 73 Android (0.71%) and 34 iOS apps (0.33%). Java source files were the most common on Android, appearing in 63 Android apps (0.61%). Additionally, one iOS app (0.01%) also contained Java code files. On iOS, Swift files were most prevalent, found in 23 apps (0.22%).

Case Study: Audible. In the Audible app [9], we found Swift code labeled `AlexaKit`, used to integrate Amazon Alexa. Notably, it included debugging code that sent error logs to two email addresses. This code was removed in the version we downloaded in 2024.

Case Study: Banking App. The iOS banking app of Raiffeisen Romania [79] included 152 Swift files containing the code of app features. This exposed code could aid in reverse engineering and increase the risk of targeted social engineering attacks, as files included developer names in their header comments.

4.2.2 Scripts. We separated *web* related scripts, e.g., JavaScript or TypeScript, as these are commonly used in mobile apps to render content via WebViews or similar components. Beer et al. [14] reported that 80% of Android apps use Custom Tabs, another form of in-app browsing. We found web-related files in 7,004 Android (67.80%) and 6,724 iOS apps (65.09%). The numbers may be lower than expected because apps can load web resources directly from the Internet without bundling them in the app package.

Lua, Python, and Shell Scripts. We found scripts in 144 Android (1.39%) and 294 iOS apps (2.85%). The most common scripts were shell scripts, which we discovered in 40 Android (0.39%) and 158 iOS apps (1.53%), followed by Lua scripts in 70 Android (0.68%) and 91 iOS apps (0.88%), and Python scripts in 24 Android (0.23%) and 50 iOS apps (0.48%).

Developers often use shell scripts during development and sometimes forget to remove them before release. For example, we found build scripts in the iOS versions of the iRobot [48] and Microsoft Whiteboard [59] apps. These scripts did not contain credentials, as they loaded secrets via environment variables, a best practice to prevent leaks in code repositories [12]. As these cases show, this approach also helps prevent secret exposure in mobile apps. Unlike public code repositories, where code is shared intentionally, these script leaks likely happened unintentionally.

We found similar cases involving Python scripts. For instance, the Android version of the Expedia app [30] included `github_utils.py`, used for opening pull requests, and to retrieve domain information `fetch_site_configs.py`.

On iOS, the Firefox app [66] bundled a *SyncIntegrationTests* directory containing six Python test scripts. The game JellyBlast [97] included Python scripts used to enable debug features for users.

Unlike Python and shell scripts, Lua scripts are often part of the app's core code, as they are used by mobile development libraries such as MoonSharp for Unity [65] and Corona SDK [22]. In addition to previously reported Lua usage, we found compiled Lua scripts in 174 apps, 80 Android apps (0.77%) and 94 iOS apps (0.91%).

The role of these scripts becomes clearer when comparing their presence across platforms. Of the 181 apps that included shell scripts on at least one platform, only 17 (9.39%) had them on both. For Python, 63 apps included scripts on at least one platform, but only 11 (17.46%) did so on both. In contrast, 51 apps (46.36%) included Lua scripts on both platforms, and 71 apps (68.93%) had compiled Lua scripts on both. This pattern indicates that, typically, Lua scripts are intentionally included as part of the app's functionality, while Python and shell scripts are more likely to have been bundled unintentionally during development.

Table 2: Number of dependency management files in our dataset. We summarized all file types that occurred in fewer than 25 apps and label them as *Other*. The number of findings in 2024 is related to the reduced number of 8,702 available Android and 9,212 iOS apps. We uploaded the full table [88].

	2023		2024	
C/C++/C#	7 (0.07%)	91 (0.88%)	5 (0.06%)	69 (0.75%)
CMakeLists.txt	3 (0.03%)	56 (0.54%)	2 (0.02%)	43 (0.47%)
Other	4 (0.04%)	40 (0.39%)	3 (0.03%)	32 (0.35%)
Dart	12 (0.12%)	9 (0.09%)	16 (0.18%)	18 (0.20%)
Go		3 (0.03%)		3 (0.03%)
Java	233 (2.26%)	7 (0.07%)	172 (1.98%)	6 (0.07%)
build.gradle	4 (0.04%)	6 (0.06%)	2 (0.02%)	6 (0.07%)
pom.xml	229 (2.22%)	1 (0.01%)	169 (1.94%)	
Other	1 (0.01%)		1 (0.01%)	3 (0.03%)
Python	3 (0.03%)	4 (0.04%)	2 (0.02%)	7 (0.08%)
Ruby	6 (0.06%)	13 (0.13%)	5 (0.06%)	12 (0.13%)
Swift	3 (0.03%)	256 (2.48%)	4 (0.05%)	263 (2.85%)
Package.swift	3 (0.03%)	28 (0.27%)	4 (0.05%)	52 (0.56%)
Podfile		92 (0.89%)		78 (0.85%)
*.podspec		110 (1.06%)		100 (1.09%)
Other		48 (0.46%)		28 (0.30%)
Web	121 (1.17%)	336 (3.25%)	117 (1.34%)	234 (2.54%)
bower.json	50 (0.48%)	55 (0.53%)	40 (0.46%)	43 (0.47%)
package.json	107 (1.04%)	320 (3.10%)	104 (1.20%)	222 (2.41%)
package-lock.json	21 (0.20%)	25 (0.24%)	16 (0.18%)	24 (0.26%)
Other	20 (0.19%)	22 (0.21%)	19 (0.22%)	18 (0.20%)
Total	372 (3.60%)	697 (6.75%)	310 (3.56%)	588 (6.38%)

4.2.3 Dependency Management. We found dependency management files in 372 Android apps (3.6%) and 697 iOS apps (6.75%). An overview of these results is shown in Table 2.

These files can reveal the specific versions of libraries used, which helps attackers identify outdated components with known vulnerabilities [27, 76]. They may also reference internal repositories. If these references are misconfigured, they can enable dependency confusion attacks in which an attacker publishes a package with the same name to a public repository, such as NPM [70]. If the dependency manager prioritizes public sources, it may fetch the

malicious package instead of the intended internal one. This can lead to remote code execution as the attacker is able to include a pre-installation script in the package [15].

CocoaPods. CocoaPods [18] is a popular dependency management system for Swift and Objective-C [85]. In our analysis, we identified three file types that disclose dependency details: *Podfile* files appeared in 92 apps (0.89%), *Podfile.lock* in 24 apps (0.23%), and **.podspec* files in 110 apps (1.06%). The *Podfile* defines which libraries (pods) the app uses, their versions, and the repositories from which they should be fetched. During installation, CocoaPods generates a *Podfile.lock* to store the exact versions used, ensuring consistent builds. The *.podspec* files describe individual libraries, specifying their name, version, source, and dependencies.

To assess the risk of dependency confusion attacks, we checked whether pod names used in analyzed apps were unclaimed in the public CocoaPods repository. We found that 81 iOS apps (0.78%) referenced at least one of 97 available pod names. Attackers could register them, enabling the execution of malicious code on developer devices or build servers when installing or updating dependencies.

Carthage and SwiftPM. Carthage [16] and SwiftPM [95] are alternative package managers for Swift and Objective-C. Unlike CocoaPods, they do not rely on a central dependency repository which makes them immune to dependency confusion attacks. However, other risks remain, e.g., attackers might use library and version information to identify outdated libraries with known vulnerabilities.

We found *Package.swift* files in 3 Android (0.03%) and 28 iOS apps (0.27%). More apps contained *Package.swift* files than were flagged as including Swift source code. This discrepancy results from our classification method, which requires at least ten unique files in a single programming language to consider an app as containing source code, which not all of these apps had.

Similar to *Podfile.lock*, the *Package.resolved* file records the dependency versions which we found in 7 apps (0.07%). Further, 17 iOS apps (0.16%) contained *Cartfiles* and 4 (0.04%) *Cartfile.resolved*.

Gradle and Maven. We also discovered Android-related package management files, including *pom.xml*, *build.gradle*, and *gradle.lockfile*. Specifically, *pom.xml* appeared in 229 Android apps (2.22%) and 1 iOS app (0.01%), *build.gradle* in 4 Android (0.04%) and 6 iOS apps (0.06%), and *gradle.lockfile* in 1 Android app (0.01%).

The Maven *pom.xml* files contained only library descriptions of well-known libraries, while the *build.gradle* files mostly included automation scripts rather than dependency declarations. As a result, their potential to expose sensitive information was limited.

Web. Unlike Java and Swift-related package managers, which we mostly found on a single platform, package managers of various web technologies appeared on both platforms. As discussed in Section 4.2, apps frequently embed web code to streamline cross-platform development, reducing the need to implement features twice. Still, we observed a higher prevalence of web-related dependency management files in iOS apps, for example, *package.json* was found in 107 Android (1.04%) and 320 iOS apps (3.10%).

To assess the risk of dependency confusion, we analyzed *npm* package names and found 17 that were unregistered in the public repository, spanning six apps. These included three finance apps, two from Disney, and one health-related app.

4.3 Misc

4.3.1 Dotfiles and Dotdirectories. Hidden dotfiles are often used for configuration data in development environments but are rarely needed in production apps. Jungwirth et al. [50] found that 73.6% of code repositories leak potentially sensitive data through dotfiles.

Dotfiles. In contrast to code repositories, dotfiles in released apps likely result from oversight. Still, we found them in 126 Android apps (1.22%) and 869 iOS apps (8.41%). The most frequent were `.gikeep` (701 files across 21 Android and 202 iOS apps), `.DS_Store` (378 files in 33 Android and 161 iOS apps), and `.gitignore` (300 files in 21 Android and 202 iOS apps).

Dotdirectory. We found dotdirectories in 8 Android apps (0.08%), containing a total of 1,401 files, and in 922 iOS apps (8.93%), containing 3,190 files. The most common was `.AppLovinQualityService`, present in 768 iOS apps. AppLovin is a monetization library available for both Android and iOS [8]. This directory consistently included two files, called: `AppLovinQualityService.json` and `AppLovinServiceRanges.json`. Despite their suffix, these files contain binary data with high entropy (e.g., 7.9), indicating encryption.

Other dotdirectories included `.monotouch` (found in 61 iOS apps), `.vscode` (2 Android and 29 iOS apps), and `.swiftpm` (19 iOS apps).

Case Study: Epic Seven. In an iOS game app [93], we found an Apache subversion directory `.svn`, which contained metadata about previous code changes.

4.3.2 Markdown. Markdown is a popular markup language for formatting text [56]. We found Markdown files in 840 Android (8.13%) and 1,004 iOS apps (9.72%). These files appeared for three main reasons: (1) apps used them to store text that they render at runtime; (2) they were included as part of third-party dependencies—e.g., Markdown files in the `node_modules` directory were present in 30 apps; (3) developers used them for internal documentation but forgot to exclude them from production builds. We classified these cases based on file location and whether the files were referenced in binaries or resources.

Most Markdown files originated from third-party dependencies, found in 734 Android apps (7.10%) and 636 iOS apps (6.16%). Files likely included unintentionally were found in 91 Android apps (0.88%) and 398 iOS apps (3.85%).

Case Study: Decathlon Connect. In the *Decathlon Connect* app [25], we found Markdown files used for internal documentation, including component descriptions and onboarding materials. These files pose a security risk, as they contained developer names, email addresses, and internal URLs, information that could aid in reverse engineering or targeted social engineering attacks.

Case Study: Scan & Translate+. In the *Scan & Translate+* [1] app, we found general documentation that included a link to an internal repository, as well as a Markdown file named `REMOTE_SERVICES.md`. This file listed various services along with their credentials, including those used for premium features. If abused, these credentials could allow unauthorized access to paid services, resulting in financial costs for the app provider.

4.3.3 AI assets. The rise of AI and machine learning has introduced new security risks, like model stealing, where attackers extract and

Table 3: Coded content of responses to the responsible disclosure. Two researchers manually coded the responses. Afterwards, they compared their codebooks, merged them, and discussed disagreements. We received 77 non-automated responses, 37 about the app content, and 40 about secrets.

Responses	Files	Secrets	Total
Aware of issue		2 (5.00%)	2 (2.60%)
Collaboration		1 (2.50%)	1 (1.30%)
Fixed	5 (13.51%)	8 (20.00%)	13 (16.88%)
Fix to expensive	1 (2.70%)		1 (1.30%)
Forward to team	16 (43.24%)	21 (52.50%)	37 (48.05%)
No critical issue	1 (2.70%)	1 (2.50%)	2 (2.60%)
No issue	1 (2.70%)	1 (2.50%)	2 (2.60%)
Old version/legacy		4 (10.00%)	4 (5.19%)
Questions	9 (24.32%)	3 (7.50%)	12 (15.58%)
Resubmit	4 (10.81%)		4 (5.19%)
Security address	6 (16.22%)	1 (2.50%)	7 (9.09%)
Will fix	3 (8.11%)	6 (15.00%)	9 (11.69%)

reuse trained models [40, 47]. While running AI tasks on-device improves privacy by avoiding data transmission to remote servers, extraction of models bundled with an app is straightforward. We found AI-related files in 755 Android apps (7.31%) and 491 iOS apps (4.75%). Not all of these are sensitive, many are publicly-available, pre-trained models offering specific features. For example, a Google model for barcode scanning [41] appeared in 347 apps, and a face detection model [42] in 160 apps. However, we found unique models in 262 apps, which may include custom-trained ones. For apps available on both platforms, we counted each model only once.

4.4 Responsible Disclosure

We contacted developers when their apps contained source code (in January 2025) or dependency management files (in March 2025). We did not automatically report the presence of markdown files, scripts, and dotfiles, as manual inspection showed that these rarely contained sensitive data. If any of these files included valid credentials, we disclosed them as described in Section 5.3. Additionally, we did not report `pom.xml` and `build.gradle` files, as manual review indicated they carried minimal risk (see Section 4.2.3). Nevertheless, even though such files might not be sensitive, their inclusion increases app size and storage use on the user's devices.

In total, we sent 661 disclosure emails on findings in the 2024 dataset: 117 regarding exposed source code, 535 about dependency management files, and 9 covering both. We received mail delivery failures for 69 messages. Within two weeks, 37 developers replied with non-automated responses. In Table 3, we provided an overview of the responses. These were evaluated by two researchers who independently analyzed their content. Overall, we had a positive impression of the responses: five (13.51%) stated that they had already fixed the issue, and three (8.11%) mentioned that they plan to fix it. However, one developer responded that implementing a fix would be too expensive (2.7%), another argued that the issue was not critical enough (2.7%), and one stated that it posed no issue in their case (2.7%).

Table 4: Number of valid and invalid credentials in context of the number of credential candidates of the same type detected per file. To minimize validation requests, we did not attempt to validate credentials when we found 15 or more credentials of the same type in a single file.

# Credential-type per File	Valid	Invalid	Valid Ratio
1	327	5,374	5.74%
2	55	1,393	3.80%
3	11	611	1.77%
4	16	501	3.09%
5	3	262	1.13%
6	2	284	0.70%
7		162	
8		126	
9	2	181	1.09%
>9 & <15		854	

Takeaways

To answer RQ1: *What files do mobile apps contain?*, we found:

- Mobile apps include numerous file types, including unexpected ones, e.g., Windows binaries and shell scripts;
- Files likely included unintentionally, such as source code, dependency management files, or internal documentation written in Markdown. Those files can expose sensitive security and privacy-related information;
- Even when these files do not pose security risks, they increase app size and storage usage on user's devices.

5 Secrets

After examining what types of files mobile apps include, we now address *RQ2: What secrets do developers distribute in mobile apps?*

In this section, all credential counts refer to the combined datasets from 2023 and 2024. We include both years to capture all valid credentials present at the time of analysis. A separate breakdown by year is provided in Section 7 to explore temporal trends.

5.1 Hardcoded Credentials

Our rule-based detection identified 26,380 potential credentials. We applied a heuristic that discards any file containing 15 or more credentials of the same service type, as we observed that rules producing numerous findings within a file typically represent FPs, e.g., hash values mistakenly matching token patterns. To minimize unnecessary validation requests for ethical reasons, we explored different thresholds by incrementally increasing the cut-off and manually reviewing files containing potentially valid secrets that would have been excluded. This reduced the number of candidates to 10,164. Table 4 summarizes the distribution of valid and invalid credentials based on the number of same-type credentials found per file. The majority of valid credentials (327) came from files containing only one instance of a specific credential type. In contrast, only 23 valid credentials were found in files with more than three

Table 5: Number of *valid* and *invalid* credentials discovered for selected services. We summarized the other 55 services for which we found at least one valid credential as *Other*. We provide the full table online [90].

	Valid	Invalid	Valid Ratio
AWS	58	85	40.56%
Alibaba	10	9	52.63%
Azure	1	52	1.89%
FTP	1		100%
GCP	1	24	4%
GitHub	9	15	37.5%
RazorPay	3	30	9.09%
Squareup	3		100%
Stripe	4	4	50%
Yelp	1	16	5.88%
Other	325	1.546	17.37%

of the same type, demonstrating our heuristic's effectiveness in filtering out FPs.

The 10,164 potential credentials identified came from 258 different categories. After completing the validation requests, we confirmed 416 as valid (4.09%), spanning 65 services. In Table 5, we provide details for 10 services with confirmed valid credentials. The FP rate varied significantly by credential type. This has three reasons: (1) Some credential types are rarely used in mobile apps compared to code repositories (for which TruffleHog was originally developed). For instance, none of the 46 Dockerhub tokens were valid, while all three Squareup tokens were. Squareup provides financial services for mobile payments. Thus, its scope aligns with mobile apps. (2) All Squareup tokens used a consistent prefix (`sq0idp-`), making them easier to detect accurately. (3) Some tokens consist of multiple parts, as seen with YouTube and AWS credentials. For both, one part can be identified easily due to its prefix, while the other lacks a clear pattern, leading to misclassification of unrelated strings.

The valid credentials stem from 200 Android (1.94%) and 292 iOS apps (2.83%). These totals do not match the overall number of valid credentials because 29 credentials appeared in multiple apps ($\bar{x}$ = 1.12, sd = 0.56, max = 8). Android and iOS versions of the same app were counted only once. In addition, we found 106 credentials in 117 apps across both platforms, often due to developers reusing the same credentials. Overall, 67 apps contained more than one valid credential ($\bar{x}$ = 1.18, sd = 0.52, max = 5).

Popular Apps. In Table 6, we break down our findings by the number of downloads reported on the Play Store. Each app was counted only once, even if the finding occurred in both its Android and iOS versions. Since the App Store does not provide installation data, we used Play Store downloads as a proxy for iOS apps. The median minimum installation count of apps with secrets was 1 million (MAD 990,500). Notably, two apps had over 1 billion installs, showing that even widely used apps are affected by credential leaks.

5.1.1 Git Credentials. For Git credentials, we additionally performed a GET request to check repository access and assess the potential impact. Valid tokens could allow attackers to push malicious code, regardless of whether the repository is public or private. Private repositories pose an additional risk by exposing internal company data, enabling cloning and republishing of apps.

Table 6: Number of apps with valid credentials. We categorized the apps by their Android installation count and grouped categories below 1,000 installations into *0-1,000*. Further, we show in ∪ the number of apps where we found a credential in the Android or iOS app.

			∪
1,000,000,000+	1 (0.01%)	1 (0.01%)	2 (0.02%)
500,000,000+	0 (0.00%)	4 (0.04%)	4 (0.04%)
100,000,000+	8 (0.08%)	8 (0.08%)	11 (0.11%)
50,000,000+	7 (0.07%)	12 (0.12%)	14 (0.14%)
10,000,000+	26 (0.25%)	50 (0.48%)	59 (0.57%)
5,000,000+	18 (0.17%)	31 (0.30%)	38 (0.37%)
1,000,000+	29 (0.28%)	48 (0.46%)	60 (0.58%)
500,000+	14 (0.14%)	22 (0.21%)	30 (0.29%)
100,000+	35 (0.34%)	44 (0.43%)	59 (0.57%)
50,000+	11 (0.11%)	10 (0.10%)	15 (0.15%)
10,000+	12 (0.12%)	16 (0.15%)	22 (0.21%)
5,000+	1 (0.01%)	3 (0.03%)	3 (0.03%)
1,000+	14 (0.14%)	17 (0.16%)	21 (0.20%)
0 - 1,000	24 (0.23%)	26 (0.25%)	35 (0.34%)
Total	200 (1.94%)	292 (2.83%)	373 (3.61%)

We identified 13 valid Git credentials: nine for GitHub, three for BitBucket, and one for Gerrit. We also found nine GitLab tokens matching the prefix `glpat-`, but none were valid. This may be due to their use in self-hosted GitLab instances. However, we found no private GitLab URLs in the files containing the identified tokens.

These 13 valid tokens granted access to 218 public and 2,440 private repositories. One token alone had access to 1,140 private and 17 public repositories. Another appeared particularly sensitive, providing access to six private repositories belonging to a bank.

We manually investigated the causes for including Git credentials in mobile apps. The two main reasons were: (1) leftover code in the app intended to trigger continuous integration workflows, and (2) the inclusion of dependency management files.

5.1.2 Files Containing Credentials. Table 7 presents an overview of the file categories in which we identified hardcoded credentials. We used the category *Web* when we found credentials in JS files; *Resources* for those located in `Info.plist`, `AndroidManifest.xml`, or other Android resource files; *Config* for configuration files not tied to the operating system, such as `.json` or `.xml`; *Cross-platform* for files associated with cross-platform or game libraries; *Binary* for common compiled Android or iOS binary files; and *Unintended* for files that are typically not expected in released apps, such as `.gitlab-ci.yml`, `.xcconfig`, or shell scripts.

We found the majority of credentials in binary files, 133 on Android (61.86%) and 184 on iOS (59.93%). However, the second most frequent category is *Web*, with 36 credentials found in Android (16.74%) and 47 in iOS apps (15.31%).

Table 7: File categories of valid credentials. *Web* indicates that we found credentials in JS files. *Resources* refers to credentials found in `Info.plist`, `Manifest`, or Android resource files. *Cross-platform* denotes files related to cross-platform or game libraries. Note that we found two Android and two iOS credentials in two file categories each. Overall, we found 106 credentials on both platforms.

			∪
Binary	133 (61.86%)	184 (59.93%)	277 (66.59%)
Config	26 (12.09%)	15 (4.89%)	33 (7.93%)
Cross-platform	17 (7.91%)	22 (7.17%)	25 (6.01%)
Resources	5 (2.33%)	30 (9.77%)	35 (8.41%)
Unintended		11 (3.58%)	11 (2.64%)
Web	36 (16.74%)	47 (15.31%)	50 (12.02%)
∑ Credentials	215	307	416

This result highlights limitations in traditional static analysis methods, such as VSA, which would usually overlook credentials embedded in web assets or require substantial customization to detect them. The situation is similar for credentials in cross-platform libraries, as each library may require a tailored analysis approach. Moreover, we found 11 credentials (3.58%) in files likely included unintentionally, making them especially difficult to detect using standard approaches.

We also identified 106 credentials shared across both platforms, with 93 (87.74%) located in the same file category. For the remaining 13 (12.26%), discrepancies in file locations were observed. For example, a Twitter consumer key was found in a `Prototype.xcconfig` on iOS, while on Android, it was stored directly in the app's bytecode. Similarly, an Infura key appeared in an Android bytecode but in a JS file on iOS. The other differences were limited to binary, resource, and config file categories.

5.2 JSON Web Tokens (JWTs) and Private Keys

Unlike hardcoded credentials discussed in the previous section, we did not test the validity of found JSON Web Tokens (JWTs) and private keys. The presence of private keys in mobile apps inherently violates their purpose, which is to remain confidential. The security impact depends on how the key is used in the app. For example, in a payment app [74], the private key was solely used for obfuscation, e.g., encrypting payloads before transmission. While knowing the key simplifies protocol reverse engineering, it does not have further security implications.

We identified 212 private keys across 433 Android (4.19%) and 180 iOS apps (1.74%). Among these, 29 keys appeared in more than one app, and six were found in over 100 apps. These frequently recurring keys originated from default or test values included in libraries such as the Google HTTP Client Library [39]. For this evaluation, we counted findings in the Android and iOS versions of the same app as a single instance.

The situation for JWTs is similar. These tokens are commonly used for user authentication and authorization, and leaking them can enable unauthorized access. In total, we found 1,378 tokens across 1,018 Android (9.85%) and 569 iOS apps (5.51%).

A significant portion of this discrepancy stems from a single token in the 2024 dataset, included by the Unity library [98], which appeared in 645 Android apps. Excluding this token, the number of affected Android apps drops to 385 (3.73%), while the iOS count remains unchanged, resulting in more iOS apps with at least one token. We also identified 75 tokens present in multiple apps, primarily due to reuse by the same developer.

To further analyze the JWTs, we parsed their contents. Of the 1,378 tokens, 859 (62.34%) did not have an expiration date set, 69 (5.01%) were still valid for more than ten years, and 392 (28.45%) were expired. We also searched for the keyword *admin* and found it in 202 tokens (14.66%), only 11 of which were already expired.

In some cases, the credentials appeared unused, suggesting they may have been unintentionally included for debugging or legacy reasons. Nonetheless, their presence can aid manual analysis and help uncover potential security or privacy issues. Automatically determining the purpose of a private key or JWT is challenging, as it requires understanding or executing the surrounding code, which we leave to future work, see Section 8.

5.3 Responsible Disclosure

We automatically sent 422 emails to developers in January 2025 to report hardcoded credentials. Of these, 12 messages (2.84%) failed to deliver. In total, we received 40 non-automated responses.

We provided insights into the responses in Table 3. Positively, eight (20%) responded that they fixed it already, and six (15%) that they will fix it. Interestingly, four (10%) answered that the issue comes from legacy code or an old app version, and two (5%) mentioned that they are aware of the issue.

Takeaways

To answer RQ2: *What secrets do developers distribute in mobile apps*, we showed:

- We found valid credentials for 65 different services, including unexpected ones such as GitHub;
- Credentials appeared not only in app binaries but also in files likely included unintentionally. We identified 11 such cases;
- Developers embedded private keys and JWTs in their apps. Only 27.16% of the tokens had already expired, and 18.52% contained indications of administrator privileges.

6 Platform Differences

To answer *RQ3: How does the situation differ between Android and iOS apps?*, we compare the contents of the apps and the credentials found. We highlight key differences and explore potential factors contributing to discrepancies.

6.1 Files

As shown in Table 1, Android and iOS apps share similar distributions across most file categories. However, *archives* and *text* categories deviate from this trend, appearing more frequently in Android apps than in iOS apps (77.21% vs. 17.1%; $p < 0.01$, $d = 1.16$ and

Table 8: Comparison of valid credentials found in Android and iOS apps. We display services with large differences separately. ∩ indicates credentials discovered on both platforms. We also report the p-value of a dependent t-test and the effect size (d) calculated using Cohen's d. We provide the complete table online [87].

Services with Major Differences			∩	p	d
AWS	31	47	20	<0.01	-0.02
Flickr	10	5	2	0.29	0.01
Github	4	6	1	0.41	-0.01
Infura	5	12	4	0.02	-0.02
OpenAI	1	13	0	<0.01	-0.02
OpenWeather	5	13	3	<0.01	-0.02
SlackWebhook	17	49	13	<0.01	-0.03
Other	142	162	63	0.01	-0.02

Table 9: Comparison of valid credentials found in apps with at least 100 million installations from 2023. ∩ indicates credentials discovered in an app's Android and iOS version.

100,000,000+ in 2023			∩
AWS	3	3	2
Alchemy		2	
BrowserStack		1	
Infura	1	3	1
LaunchDarkly		1	
PubNubSubscriptionKey	1	1	1
SlackWebhook		1	
TwitterConsumerkey	1	3	
URI		1	
Total	6	16	4

73.86% vs. 55%; $p < 0.01$, $d = 0.33$, respectively). The higher prevalence of archive files in Android is primarily due to the OkHttp3 library which includes the file `publicsuffixes.gz` present in 72.75% of Android apps. Similarly, the increase of text files results from the presence of the `androidsupportmultidexversion.txt` in 47.13% of Android apps.

Source Code, Scripts, Markdown. As detailed in Section 4.2, our analysis identified 73 Android apps (0.71%) and 34 iOS apps (0.33%) containing source code files that may reveal app code ($p < 0.01$, $d = 0.04$). In contrast, a greater share of iOS apps included potentially unintended files: scripts (1.39% vs. 2.85%, $p < 0.01$, $d = -0.09$), markdown files (8.13% vs. 9.72%, $p < 0.01$, $d = -0.05$; non third-party 1.64% vs. 4.09%, $p < 0.01$, $d = -0.11$), dotfiles (1.22% vs. 8.41%, $p < 0.01$, $d = -0.25$), and dotdirectories (0.08% vs. 8.92%, $p < 0.01$, $d = -0.31$). We observed similar trends in the 2024 dataset, as we discuss in Section 7.

Factors Contributing to Platform Differences. Two main factors explain these differences. First, Android apps are more accessible for analysis, e.g., due to the distribution of app bundles via third-party platforms such as APKPure [7] and APKMirror [6]. As a consequence, previous research primarily focused on Android. Second, Apple encrypts iOS app binaries [102], which may cause confusion among developers about the encryption status of additional files in the app bundle. One developer responding to our disclosure was surprised that Apple did not take any measures to protect dependency management files that had been unintentionally bundled with their app.

However, even if Apple would encrypt all files, they must be decrypted at runtime. This makes it still possible to extract and analyze the content using jailbroken iOS devices.

6.2 Hardcoded Secrets

Overall, fewer Android apps contained valid credentials compared to iOS apps (1.94% vs. 2.83%, $p < 0.01$, $d = -0.04$). This difference is also reflected in the absolute number of valid credentials found: 230 in Android and 352 in iOS apps. We identified 106 credentials shared across platforms, originating from 117 apps that included at least one common credential in both their Android and iOS versions.

We identified fewer credentials than apps that share them across the platforms because developers use the same credentials in multiple apps, see Section 5.1.

Services. When we look at the services with valid credentials, we learn that a significant difference results from five services. iOS apps contained more valid credentials for AWS, Infura, OpenAI, OpenWeather, and SlackWebhook. In contrast, Android apps included more valid credentials for Flickr. The remaining services yielded 142 valid credentials in Android apps and 162 in iOS apps. We provide an overview in Table 8. The table also shows the number of credentials found on both platforms, underscoring the importance of analyzing both Android and iOS apps. For instance, of the nine valid GitHub credentials, five were found exclusively in iOS apps and three only in Android apps. Consequently, we would have missed a significant number if we had only analyzed a single platform.

The most pronounced difference occurred with SlackWebhook credentials. We identified 49 such credentials in iOS apps, 13 of which also appeared in the Android version, while only four were exclusive to Android ($p < 0.01$, $d = -0.03$). SlackWebhooks allow apps to send messages to Slack channels, with the severity of exposed credentials depending on the channel's purpose. In general, we consider this a lower-risk issue. The AWS findings also showed a notable difference across platforms. We found 11 credentials exclusively in Android apps, 27 exclusively in iOS apps, and 20 shared between both ($p < 0.01$, $d = -0.02$).

Files with Credentials. A comparison of the file types containing credentials revealed two key differences between the two OSes. On Android, credentials were more frequently found in configuration files (12.09% vs. 4.89%), whereas on iOS, they appeared more often in system resource files (2.33% vs. 9.77%). Additionally, we identified 11 credentials in files likely included unintentionally in 9 iOS apps. These included files such as `.gitlab-ci.yml`, `*.xcconfig`, and shell scripts. We provide a detailed breakdown in Table 7.

Popular Apps. In 2019, Google launched a bug bounty program for Android apps with over 100 million installations [78], including third-party apps. However, the program was discontinued in August 2024, just before we collected our dataset in October [78].

We report the number of valid credentials found in the 2023 dataset for apps with at least 100 million installations in Table 9. With two findings exclusively in Android apps, 12 only in iOS apps, and four in both, the platform difference is significant ($p = 0.02$, $d = -0.1$). Two of the iOS findings resulted from unintentionally included files. In one case, we discovered a SlackWebhook token within a commented-out block of a Swift file in a game app [68]. In another, a BrowserStack token appeared in a Java test file included with the Wattpad app [100] as part of test automation code.

Discussion. At first glance, Android apps seem to exhibit fewer credential exposures. However, we also found several instances where credentials were present only in the Android version. One likely reason is the better accessibility of Android app bundles, which makes them easier to analyze and thus has historically received more research attention. Additionally, Google's bug bounty program likely had a positive impact on popular apps. In Section 7, we revisit this topic to examine how findings shifted after the program ended in 2024. Another explanation for platform-specific issues is that different development teams may be responsible for each version, resulting in inconsistencies in secure coding practices. Overall, our research underscores the importance of analyzing both platforms, as many findings were exclusive to a single platform.

Takeaways

To answer RQ3: *How does the situation differ between Android and iOS apps?*, we showed:

- Unintentionally included files were more common in iOS apps. A potential explanation is Apple's closed environment, which limits accessibility and complicates external security analysis;
- The pattern extended to credential exposures, with more valid credentials identified in iOS apps;
- Despite identifying more issues in iOS apps, the situation on Android is only slightly better. We also found cases where findings appeared exclusively in the Android version of the app.

7 Changes in 2024

To answer *RQ4: How did the situation change between 2023 and 2024?*, we examine shifts in the inclusion of files such as code and scripts, and highlight differences related to credentials. To ensure comparability of relative values, we normalize the results based on the updated 2024 dataset sizes of 8,702 Android and 9,212 iOS apps.

7.1 Files

A comparison of file categories between the 2023 and 2024 datasets revealed no major overall changes. However, we observed a notable increase in iOS apps flagged for potentially including source code: from 62 in 2023 (0.67%) to 156 in 2024 (1.51%). This increase was more pronounced on iOS ($p < 0.01$, $d = -0.04$), while numbers for Android remained relatively stable ($p = 0.62$, $d = 0.01$), with 66 in 2023 (0.76%) and 76 in 2024 (0.74%).

Table 10: Comparison of files included in mobile apps across platforms and collection years. The findings in 2024 are relate to the reduced number of 8,702 Android and 9,212 iOS apps.

	2023		2024	
Dotdirectory	8 (0.08%)	922 (8.92%)	11 (0.13%)	979 (10.63%)
Dotfile	126 (1.22%)	869 (8.41%)	122 (1.40%)	1,006 (10.92%)
Code	73 (0.71%)	34 (0.33%)	66 (0.76%)	62 (0.67%)
Markdown	840 (8.13%)	1,004 (9.72%)	918 (8.89%)	1,071 (10.37%)
Scripts	144 (1.39%)	294 (2.85%)	129 (1.48%)	232 (2.52%)

Similarly, the occurrences of dotfiles and dotdirectories increased more notably in the iOS dataset. In 2023, we identified dotfiles in 126 Android (1.22%) and 869 iOS apps (8.41%). In 2024, the relative number on Android slightly rose to 122 (1.40%, $p = 0.54$, $d = -0.01$), while it did clearly on iOS to 1,006 (10.92%, $p < 0.01$, $d = -0.07$). Dotdirectories increased from 8 (0.08%) to 11 in Android apps (0.13%, $p = 0.25$, $d = -0.01$), while on iOS, the number grew from 922 (8.92%) to 979 (10.63%, $p < 0.01$, $d = -0.04$), as detailed in Table 10.

For dependency management files, the number of findings stayed mostly constant, as shown in Table 2. The most notable change concerned *package.json* files, which dropped on iOS from 330 in 2023 (3.1%) to 222 in 2024 (2.41%, $p < 0.01$, $d = 0.05$). In contrast, the numbers for Android remained relatively constant, with 107 in 2023 (1.04%) and 104 in 2024 (1.2% $p = 0.47$, $d = -0.01$).

7.2 Hardcoded Credentials

In Figure 2, we present the number of hardcoded credentials that were valid at the time of testing. Notably, we identified 95 credentials in the 2023 dataset that, despite their removal from the app version downloaded in 2024, remained valid. For example, in the 2023 dataset, the Android version of the app `com.viber.voip` included AWS credentials within a native library. Although these were removed in the 2024 version, the credentials themselves remained valid. This finding is particularly concerning given the app's large user base, with over one billion installations. A possible explanation for this practice is legacy support. Developers may remove credentials from newer app versions but refrain from revoking them to prevent breaking functionality in older versions, which could otherwise become non-functional.

Further, we found 99 credentials newly introduced in the 2024 app versions. The total number of valid credentials remained nearly constant, with 317 valid credentials in 2023 and 321 in 2024 ($p = 0.01$, $d = -0.02$). However, relative to the number of apps analyzed, the numbers slightly increased in 2024, as 1,629 Android and 1,119 iOS apps were not available anymore at the time of the dataset update. The number of valid credentials on Android decreased from 175 in 2023 to 161 in 2024, but the relative share increased from 1.69% to 1.85% ($p = 0.01$, $d = -0.03$). We observed a similar trend for iOS: 236 valid credentials in 2023 and 230 in 2024 (2.28% vs. 2.5%, $p = 0.17$, $d = -0.01$). In general, we assumed a slight increase as we

expected some developers to remove or revoke leaked credentials from newer builds. Still, others also introduce new credentials that have not yet been revoked.

Credential Types. We observed an increase in OpenAI credentials, rising from three in 2023 to 11 in 2024 ($p = 0.03$, $d = -0.67$). The reason is likely the increased popularity of generative AI [53].

The number of valid Git credentials also increased from 7 in 2023 to 12 in 2024 ($p = 0.01$, $d = -0.2$). The percentage of valid Git credentials rose from 25% to 36.36%. One possible reason could be that developers tend to revoke Git credentials once they become aware of their exposure.

End of Google's Bounty Program. To examine the impact of the end of Google's bug bounty program in August 2024, we compared the number of valid credentials found in apps with over 100 million installations across both years.

In the 2023 dataset, we identified valid credentials in 5 Android and 11 iOS apps exceeding 100 million installations. In 2024, the number of affected iOS apps slightly decreased to 8, while the number for Android increased to 7. The difference between platforms in 2023 was statistically significant ($p = 0.02$, $d = -0.1$), but this was no longer the case in 2024 ($p = 0.29$, $d = -0.05$).

Takeaways

To answer RQ4: *How did the situation change between 2023 and 2024?*, we showed:

- The types of files included in mobile apps remained mostly constant across both years;
- Even if developers removed credentials from apps, it does not necessarily mean they also revoked them;
- We observed a slight increase in credential exposure in popular Android apps, possibly linked to the termination of Google's bug bounty program.

8 Limitations and Future Work

Our analysis faces limitations inherent to static analysis. For example, we are unable to detect encrypted credentials or files that are dynamically downloaded during runtime. Future work could address such limitations by triggering decryption routines in apps, e.g., with VSA or dynamic forced code execution. Also, pattern-based secret detection faces limitations. Basak et al. [11] reported a high number of FPs and False Negatives (FNs). We eliminate all FPs from our pattern matching results by remotely validating our findings. Thus, our results represent a lower bound, highlighting the severity of our findings.

Developer responses to our findings regarding dependency management and included source code revealed a lack of awareness that such files could be packaged with production apps. One developer expressed surprise that Apple does not obfuscate or encrypt these files. This suggests that future research should investigate developers' mental models related to file inclusion in mobile apps and explore strategies to improve their awareness.

We did not attempt to detect misconfigurations related to information that developers intentionally include in apps. Prior work has demonstrated the risks of misconfigured services, such as publicly

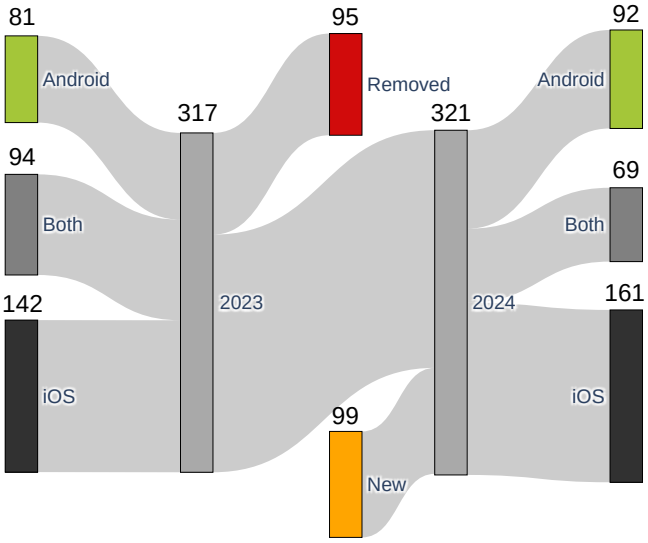


Figure 2: Valid credentials discovered in Android and iOS apps. The numbers above the bars indicate the amount of credentials discovered. The *removed* bar represents credentials present exclusively in the 2023 app version. Conversely, the *new* bar highlights credentials detected solely in 2024.

accessible AWS S3 buckets [21]. Detecting such misconfigurations typically requires knowledge of resource identifiers, e.g., bucket names and regions, which mobile apps may reveal. Similarly, Jin et al. [49] showed that improperly configured IoT access policies can lead to data leakage. They identified such issues through analysis of online code repositories. Future work could extend this by inferring misconfigurations from app content.

Future research could target context-dependent secrets, such as JWTs. For these cases, dynamic analysis or AI-based methods could be used to interpret application logic and better understand the context in which values are used.

9 Related Work

Secrets in Git Repositories. In previous research, pattern-matching approaches have been frequently used to detect secrets in Git repositories [92, 31, 57, 50, 101]. Meli et al. [57] conducted a large-scale longitudinal analysis of public GitHub repositories to identify secrets and private key leaks. Koishybayev et al. [51] highlighted the security risks of CI/CD pipelines executing arbitrary code from untrusted sources and proposed an early warning system to detect security risks, including the exposure of secrets in Git repositories.

Moreover, researchers have explored machine learning techniques to improve secret detection in Git repositories [84, 33, 55, 46, 69]. For instance, Saha et al. [84] used machine learning to reduce false positives and expand the range of detectable secrets.

Others focused on developers' perspectives by analyzing why secrets leak, the challenges developers face in preventing exposure, and mitigation techniques [52, 77, 12].

Analyzing mobile apps for secrets comes with two major differences compared to analyzing code repositories: (1) The analyzed code format differs. While repositories primarily contain source

code, mobile apps are distributed in compiled binary form, making extraction and analysis more complex. (2) The purpose of these platforms differs. Repositories facilitate collaboration among developers, whereas apps are built for end-users.

Researchers have also applied regular expression-based detection beyond code repositories. Yadmani et al.[29] employed this method to uncover secrets stored on cloud storage servers. Similar to our work, they validated detected credentials and responsibly disclosed their findings. Dahlmanns et al.[23] applied this approach to Docker container images. However, as with code repositories, the purpose and format of data shared through cloud storage services and container images differ significantly from those of mobile apps.

Mobile Analysis. Previous work often focused on privacy aspects of mobile apps, in particular leaks of PII [20, 67, 82, 81] or permission models to protect personal data [80, 86, 103, 54]. In contrast, our study takes a different direction, we focus on secrets embedded in released mobile apps instead of privacy aspects.

Previous work on mobile app analysis researched the threat of hardcoded cryptographic keys and credentials in mobile apps [99, 32, 34, 58, 38]. Schrittwieser et al. [91] identified authentication bypass vulnerabilities in popular messenger apps based on protocol analysis. Zhou et al. [105] used data flow analysis to find email and Amazon AWS credentials in Android apps. Mendoza et al. [58] developed a methodology to identify credential misuse in two iOS SDKs and showed their real-world impact by analyzing 100 apps. Zuo et al. [107] analyzed Android apps to find potential data leaks of cloud APIs due to authentication and authorization issues.

In contrast to existing work, we studied Android and iOS apps to compare the situation on both platforms. Furthermore, we adopted a broader perspective rather than limiting our analysis to specific credentials or those strictly required by the app to function properly. This allowed us to identify sensitive information in files that developers likely included unintentionally, such as source code or documentation, and to examine files originating from cross-platform libraries and embedded web content.

Baskaran et al. [13] and Zhang et al. [104] studied secrets included in so-called super-apps which support third-party mini-programs to extend their functionality. In contrast, we conducted a large-scale analysis of Android and iOS apps without restricting our scope to the ecosystem of individual super-apps.

10 Conclusion

Our analysis revealed that mobile apps often contain unintentionally added files, exposing security- and privacy-sensitive information. These files included markdown documentation, source code, and dependency management files. In particular, dependency management files can pose a critical risk by enabling remote code execution on developer machines or build servers. We identified this threat for 114 dependencies declared in 87 apps (0.84%).

We also uncovered 416 valid credentials spanning 65 different services. Notably, this included 13 Git credentials that provided access to 218 public and 2,440 private repositories. These issues were not limited to niche apps: We identified two apps with over one billion installations, and the median installation count across affected apps was one million.

Overall, our findings showed a higher prevalence of such issues in iOS apps. However, we also documented findings exclusive to either the Android or iOS version of the same app. This underlines the importance of analyzing apps from both OSes.

In some cases, developers removed hardcoded credentials or unintended files from newer app versions. However, this alone does not eliminate the risk. Attackers may have already downloaded earlier versions or still have access to them. Developers should, therefore, carefully audit app bundles before release, and when removing credentials, they must also revoke them. However, this does not always happen. We found 95 credentials that had been removed from the 2024 app version but remained valid.

Acknowledgments

The financial support by the Austrian Federal Ministry of Economy, Energy and Tourism, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association is gratefully acknowledged.

References

[1] Aisberg Inc LLC. App Store – Scan & Translate+ Text Grabber. Archived at: https://archive.ph/qHLIP. https://apps.apple.com/us/app/scan-translate-text-grabber/id845139175.

[2] M. Alfhaily. Github – majd/ipatool – release v2.2.0. Archived: https://archive.ph/OUJbF. https://github.com/majd/ipatool/releases/tag/v2.2.0.

[3] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*. (May 2016). doi:10.1145/2901739.2903508.

[4] AloneMonkey. GitHub – frida-ios-dump. Commit: 56e99b2. https://github.com/AloneMonkey/frida-ios-dump.

[5] Android Developers. Build multiple APKs. Archived at https://archive.ph/s17jN. https://developer.android.com/build/configure-apk-splits.

[6] APKMirror. Free APK Downloads. Archived at https://archive.ph/h4ABV. https://www.apkmirror.com/.

[7] APKPure. Download APK on Android. Archived at https://archive.ph/oCCd5. https://apkpure.com/.

[8] AppLovin. Connect to audiences in-app, on mobile devices, and across CTV. Archived at https://archive.ph/JLpKX. https://www.applovin.com/.

[9] Audible, Inc. App Store – Audible: Audio Entertainment. Archived at https://archive.ph/G5CPz. https://apps.apple.com/us/app/audible-audio-entertainment/id379693831.

[10] M. Backes, S. Bugiel, and E. Derr. Reliable Third-Party Library Detection in Android and its Security Applications. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. (Oct. 2016). doi:10.1145/2976749.2978333.

[11] S. K. Basak, J. Cox, B. Reaves, and L. Williams. A Comparative Study of Software Secrets Reporting by Secret Detection Tools. In *Proceedings of the IEEE/ACM International Symposium on Empirical Software Engineering and Measurement (ESEM)*. (Dec. 2023). doi:10.1109/ESEM56168.2023.10304853.

[12] S. K. Basak, L. Neil, B. Reaves, and L. Williams. What Are the Practices for Secret Management in Software Artifacts? In *Proceedings of the IEEE Cybersecurity Development (SecDev)*. (Oct. 2022). doi:10.1109/SecDev53368.2022.00026.

[13] S. Baskaran, L. Zhao, M. Mannan, and A. Youssef. Measuring the Leakage and Exploitability of Authentication Secrets in Super-apps: The WeChat Case. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. (Oct. 2023). doi:10.1145/3607199.3607236.

[14] P. Beer, M. Squarcina, L. Veronese, and M. Lindorfer. Tabbed Out: Subverting the Android Custom Tab Security Model. In *Proceedings of the 45th IEEE Symposium on Security & Privacy (S&P)*. (May 2024). doi:10.1109/SP54263.2024.00105.

[15] A. Birsan. Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies. Archived at https://archive.ph/455ky. (Feb. 9, 2021). https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610.

[16] Carthage. GitHub – A simple, decentralized dependency manager for Cocoa. Archived at https://archive.ph/KVMkT. https://github.com/Carthage/Carthage.

[17] G. Cluley. Snapchat's source code leaked out, and was published on GitHub. Archived at https://archive.ph/OgcR6. (Aug. 8, 2018). https://www.bitdefender.com/en-gb/blog/hotforsecurity/snapchats-source-code-leaked-out-and-was-published-on-github.

[18] CocoaPods. CocoaPods.org. Archived at https://archive.ph/yQ24y. https://cocoapods.org/.

[19] Codename One. How to Build iOS Apps with Java. Archived at https://archive.ph/VdGWX. https://www.codenameone.com/blog/how-to-build-ios-apps-with-java.html.

[20] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna. Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis. In *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS)*. (Feb. 2017). doi:10.14722/ndss.2017.23465.

[21] A. Continella, M. Polino, M. Pogliani, and S. Zanero. There's a Hole in that Bucket! A Large-scale Analysis of Misconfigured S3 Buckets. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*. (Dec. 2018). doi:10.1145/3274694.3274736.

[22] Corona Labs Inc. Corona – Free Cross-Platform 2D Game Engine. Archived at: https://archive.ph/FmKN1. https://coronalabs.com/.

[23] M. Dahlmanns, C. Sander, R. Decker, and K. Wehrle. Secrets Revealed in Container Images: An Internet-wide Study on Occurrence and Impact. In *Proceedings of the 18th ACM ASIA Conference on Computer and Communications Security (ASIACCS)*. (July 2023). doi:10.1145/3579856.3590329.

[24] B. De Sutter, S. Schrittwieser, B. Coppens, and P. Kochberger. Evaluation Methodologies in Software Protection Research. *ACM Computing Surveys*, 57, 4. doi:10.1145/3702314.

[25] Decathlon. App Store – Decathlon Connect. Archived at: https://archive.ph/13znY. https://apps.apple.com/us/app/decathlon-connect/id1288552594.

[26] J. Dinneen. GitHub – File Extension Categoriser. Archived at https://archive.ph/DntMe. https://github.com/jddinneen/file-extension-categoriser.

[27] D. Domínguez-Álvarez, A. de la Cruz, A. Gorla, and J. Caballero. LibKit: Detecting Third-Party Libraries in iOS Apps. In *Proceedings of the 10th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. (Aug. 2015). doi:10.1145/3611643.3616344.

[28] Dyne.org. GitHub – Organised collection of common file extensions. Archived at https://archive.ph/LS08v. https://github.com/dyne/file-extension-list/.

[29] S. El Yadmani, O. Gadyatskaya, and Y. Zhauniarovich. The File That Contained the Keys Has Been Removed: An Empirical Analysis of Secret Leaks in Cloud Buckets and Responsible Disclosure Outcomes. In *Proceedings of the 46th IEEE Symposium on Security & Privacy (S&P)*. (May 2025). doi:10.1109/SP61157.2025.00009.

[30] Expedia. Play Store – Expedia: Hotels, Flights, Cars. Archived at https://archive.ph/aYx88. https://play.google.com/store/apps/details?id=com.expedia.bookings.

[31] C. Farinella, A. Ahmed, and C. Watterson. Git Leaks: Boosting Detection Effectiveness Through Endpoint Visibility. In *Proceedings of the 20th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. (Oct. 2021). doi:10.1109/TrustCom53373.2021.00103.

[32] J. Feichtner. A Comparative Study of Misapplied Crypto in Android and iOS Applications. In *Proceedings of the 16th International Conference on Security and Cryptography (SECRYPT)*. (July 2019). doi:10.5220/0007915300960108.

[33] R. Feng, Z. Yan, S. Peng, and Y. Zhang. Automated Detection of Password Leakage from Public GitHub Repositories. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE)*. (May 2022). doi:10.1145/3510003.3510150.

[34] A. Forsberg and L. H. Iwaya. Security Analysis of Top-Ranked mHealth Fitness Apps: An Empirical Study. In *Proceedings of the 29th Secure IT Systems - Nordic Conference (NordSec)*. (Nov. 2024). doi:10.1007/978-3-031-79007-2_19.

[35] Free Software Foundation, Inc. Coreutils - GNU Core Utilities. Archived at https://archive.ph/9cQ2P. https://www.gnu.org/software/coreutils/.

[36] GitHub – jadx - Dex to Java decompiler. Version: 1.5.1. https://github.com/skylot/jadx.

[37] Gitleaks. GitHub – Gitleaks. Version: 8.21.2. https://github.com/gitleaks/gitleaks.

[38] L. Glanz, P. Müller, L. Baumgärtner, M. Reif, S. Amann, P. Anthonysamy, and M. Mezini. Hidden in Plain Sight: Obfuscated Strings Threatening Your Privacy. In *Proceedings of the 15th ACM ASIA Conference on Computer and Communications Security (ASIACCS)*. (Oct. 2020). doi:10.1145/3320269.3384745.

[39] Google. GitHub – Google HTTP Client Library for Java. Archived at: https://archive.ph/TQK53. https://github.com/googleapis/google-http-java-client.

[40] Google. ML Kit. Archived at https://archive.ph/oktoP. https://developers.google.com/ml-kit.

[41] Google. ML Kit – Barcode scanning. Archived at https://archive.ph/NmHdk. https://developers.google.com/ml-kit/vision/barcode-scanning.

[42] Google. ML Kit – Face detection. Archived at https://archive.ph/6J3yw. https://developers.google.com/ml-kit/vision/face-detection.

[43] Y. Guo and T. Dong. Exposing the Danger Within: Hardcoded Cloud Credentials in Popular Mobile Apps. Archived at https://archive.ph/bC9m3. (Oct. 22, 2024). https://www.security.com/threat-intelligence/exposing-danger-within-hardcoded-cloud-credentials-popular-mobile-apps.

[44] I. Gupta. Mobile App Industry Statistics 2023: Trends and Insights You Shouldn't Ignore. Archived at: https://archive.ph/0OjDB. https://ripenapps.com/blog/mobile-app-industry-statistics/.

[45] J. Howarth. iPhone vs Android User Stats (2024 Data). Archived at https://archive.ph/L1QOZ. (June 14, 2024). Retrieved Jan. 13, 2025 from https://explodingtopics.com/blog/iphone-android-users.

[46] Y. Huang, Y. Li, W. Wu, J. Zhang, and M. R. Lyu. Your Code Secret Belongs to Me: Neural Code Completion Tools Can Memorize Hard-Coded Credentials. *ACM on Software Engineering*, 1, FSE, (July 2024). doi:10.1145/3660818.

[47] Hugging Face. The AI community building the future. Archived at https://archive.ph/T6678. https://huggingface.co/.

[48] iRobot Corporation. App Store – iRobot Home. Archived at https://archive.ph/HKgmC. https://apps.apple.com/at/app/irobot-home/id1012014442.

[49] Z. Jin, L. Xing, Y. Fang, Y. Jia, B. Yuan, and Q. Liu. P-verifier: understanding and mitigating security risks in cloud-based iot access policies. In *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. (Nov. 2022). doi:10.1145/3548606.3560680.

[50] G. Jungwirth, A. Saha, M. Schröder, T. Fiebig, M. Lindorfer, and J. Cito. Connecting the .dotfiles: Checked-In Secret Exposure with Extra (Lateral Movement) Steps. In *Proceedings of the 21st International Conference on Mining Software Repositories (MSR)*. (Apr. 2024). doi:10.1109/MSR59073.2023.00051.

[51] I. Koishybayev, A. Nahapetyan, R. Zachariah, S. Muralee, B. Reaves, A. Kapravelos, and A. Machiry. Characterizing the Security of Github CI Workflows. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*. (Aug. 2022).

[52] A. Krause, J. H. Klemmer, N. Huaman, D. Wermke, Y. Acar, and S. Fahl. Pushed by Accident: A Mixed-Methods Study on Strategies of Handling Secret Information in Source Code Repositories. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security)*. (Aug. 2023).

[53] Lareina Yee. McKinsey – The state of AI: How organizations are rewiring to capture value. Archived at https://archive.ph/h26UW. (Mar. 12, 2025). https://www.mckinsey.com/capabilities/quantumblack/our-insights/the-state-of-ai/.

[54] X. Liu, Y. Leng, W. Yang, W. Wang, C. Zhai, and T. Xie. A Large-Scale Empirical Study on Android Runtime-Permission Rationale Messages. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. (Oct. 2018). doi:10.1109/VLHCC.2018.8506574.

[55] S. Lounici, M. Rosa, C. Negri, S. Trabelsi, and M. Önen. Optimizing Leak Detection in Open-source Platforms with Machine Learning Techniques. In *Proceedings of the 7th International Conference on Information Systems Security and Privacy*. (Feb. 2021). doi:10.5220/0010238101450159.

[56] Markdown Guide – Getting Started. Archived at https://archive.ph/s01xz. https://www.markdownguide.org/getting-started/.

[57] M. Meli, M. R. McNiece, and B. Reaves. How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories. In *Proceedings of the 26th Network and Distributed System Security Symposium (NDSS)*. (Feb. 2019). doi:10.14722/ndss.2019.23418.

[58] A. Mendoza and G. Gu. Mobile Application Web API Reconnaissance: Web-to-Mobile Inconsistencies & Vulnerabilities. In *Proceedings of the 39th IEEE Symposium on Security & Privacy (S&P)*. (May 2018). doi:10.1109/SP.2018.00039.

[59] Microsoft Corporation. App Store – Microsoft Whiteboard. Archived at https://archive.ph/dWhUu. https://apps.apple.com/us/app/microsoft-whiteboard/id1352499399.

[60] Microsoft Corporation. Mono interpreter on iOS and Mac Catalyst. Archived at https://archive.ph/YkA6b. https://learn.microsoft.com/en-us/dotnet/maui/macios/interpreter.

[61] Microsoft Corporation. Native AOT deployment on iOS and Mac Catalyst. Archived at https://archive.ph/MG4HP. https://learn.microsoft.com/en-us/dotnet/maui/deployment/nativeaot.

[62] Microsoft Corporation. Xamarin. Archived at https://archive.ph/29xDC. https://dotnet.microsoft.com/en-us/apps/xamarin.

[63] MobiVM. Archived at https://archive.ph/PU5uc. https://mobivm.github.io/.

[64] Mono Project. Mono open source ECMA CLI, C# and .NET implementation. Archived at https://archive.ph/Rtbuv. https://github.com/mono/mono/.

[65] Moonsharp. A Lua interpreter written entirely in C# for the .NET, Mono and Unity platforms. Archived at: https://archive.ph/pp6zg. https://www.moonsharp.org/.

[66] Mozilla. App Store – Firefox: Private, Safe Browser. Archived at https://archive.ph/LEKJo. https://apps.apple.com/us/app/firefox-private-safe-browser/id989804926.

[67] Y. Nan, Z. Yang, X. Wang, Y. Zhang, D. Zhu, and M. Yang. Finding Clues for Your Secrets: Semantics-Driven, Learning-Based Privacy Discovery in Mobile Apps. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS)*. (Feb. 2018). doi:10.14722/ndss.2018.23092.

[68] NAVER Z Corporation. App Store – ZEPETO: Avatar, Connect & Live. Archived at https://archive.ph/2fz7N. https://apps.apple.com/us/app/zepeto-avatar-connect-live/id1350301428.

[69] L. Niu, S. Mirza, Z. Maradni, and C. Pöpper. CodexLeaks: Privacy Leaks from Code Generation Language Models in GitHub Copilot. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security)*. (Aug. 2023).

[70] npm – Home. Archived at https://archive.ph/ImYpo. https://www.npmjs.com/.

[71] OWASP Foundation. OWASP Mobile Top 10. Archived at: https://archive.ph/fspDN. https://owasp.org/www-project-mobile-top-10/.

[72] palera1n. Version: v2.0.0-beta.8. https://palera.in/.

[73] PayPal, Inc. App Store – PayPal Business. Archived at https://archive.ph/C3sAq. https://apps.apple.com/us/app/paypal-business/id1053148887.

[74] Paytm. Secure UPI Payments. Archived at: https://archive.ph/rPnAX. https://paytm.com/.

[75] L. Quinn. What's the difference between IPA and APK? Archived at https://archive.ph/llOi9. (Feb. 3, 2022). https://lovequinn.medium.com/whats-the-difference-between-ipa-and-apk-eff81fb0c61b.

[76] K. Rahkema and D. Pfahl. SwiftDependencyChecker: detecting vulnerable dependencies declared through CocoaPods, carthage and swift PM. In *Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESOFT)*. (May 2022). doi:10.1145/3524613.3527806.

[77] M. R. Rahman, N. Imtiaz, M.-A. Storey, and L. Williams. Why Secret Detection Tools Are Not Enough: It's Not Just about False Positives - An Industrial Case Study. *Empirical Software Engineering*, 27, 3, (May 2022). doi:10.1007/s10664-021-10109-y.

[78] M. Rahman. AndroidAuthority – Google Play will no longer pay to discover vulnerabilities in popular Android apps. Archived at https://archive.ph/1RKjo. (Aug. 19, 2024). https://www.androidauthority.com/google-play-security-reward-program-winding-down-3472376/.

[79] Raiffeisen Bank Romania S.A. App Store – Noul Raiffeisen Smart Mobile. Archived at https://archive.ph/7GjQk. https://apps.apple.com/us/app/noul-raiffeisen-smart-mobile/id1255136212.

[80] J. Reardon, Á. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez, and S. Egelman. 50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*. (Aug. 2019).

[81] J. Ren, M. Lindorfer, D. J. Dubois, A. Rao, D. Choffnes, and N. Vallina-Rodriguez. Bug Fixes, Improvements, ... and Privacy Leaks - A Longitudinal Study of PII Leaks Across Android App Versions. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS)*. (Feb. 2018). doi:10.14722/ndss.2018.23143.

[82] J. Ren, A. Rao, M. Lindorfer, A. Legout, and D. Choffnes. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. (June 2016). doi:10.1145/2906388.2906392.

[83] A. Ruggia, D. Nisi, S. Dambra, A. Merlo, D. Balzarotti, and S. Aonzo. Unmasking the veiled: A comprehensive analysis of Android evasive malware. In *Proceedings of the 19th ACM ASIA Conference on Computer and Communications Security (ASIACCS)*. (July 2024). doi:10.1145/3634737.3637658.

[84] A. Saha, T. Denning, V. Srikumar, and S. K. Kasera. Secrets in Source Code: Reducing False Positives Using Machine Learning. In *Proceedings of the International Conference on COMmunication Systems & NETworkS (COMSNETS)*. (Jan. 2020). doi:10.1109/COMSNETS48256.2020.9027350.

[85] E. F. D. Santos. Dependency Management in iOS Development: A Developer Survey Perspective. In *Proceedings of the 10th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESOFT)*. (Apr. 2024). doi:10.1145/3647632.3647992.

[86] D. Schmidt, A. Ponticello, M. Steinböck, K. Krombholz, and M. Lindorfer. Analyzing the iOS Local Network Permission from a Technical and User Perspective. In *Proceedings of the 46th IEEE Symposium on Security & Privacy (S&P)*. (May 2025). doi:10.1109/SP61157.2025.00045.

[87] D. Schmidt, S. Schrittwieser, and E. Weippl. GitHub – Leaky Apps: Full Comparison of Valid Credentials Table. https://github.com/CDL-AsTra/leaky_apps/blob/main/tables/comparison.md.

[88] D. Schmidt, S. Schrittwieser, and E. Weippl. GitHub – Leaky Apps: Full Dependency Management Table. https://github.com/CDL-AsTra/leaky_apps/blob/main/tables/dependencies.md.

[89] D. Schmidt, S. Schrittwieser, and E. Weippl. GitHub – Leaky Apps: Full List of Services. https://github.com/CDL-AsTra/leaky_apps/blob/main/analysis/rules.md.

[90] D. Schmidt, S. Schrittwieser, and E. Weippl. GitHub – Leaky Apps: Full Valid Credential Table. https://github.com/CDL-AsTra/leaky_apps/blob/main/tables/service_table.md.

[91] S. Schrittwieser, P. Frühwirt, P. Kieseberg, M. Leithner, M. Mulazzani, M. Huber, and E. Weippl. Guess Who's Texting You? Evaluating the Security of Smartphone Messaging Applications. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*. (Feb. 2012).

[92] V. S. Sinha, D. Saha, P. Dhoolia, R. Padhye, and S. Mani. Detecting and Mitigating Secret-Key Leaks in Source Code Repositories. In *Proceedings of the 12th International Conference on Mining Software Repositories (MSR)*. (May 2014). doi:10.1109/MSR.2015.48.

[93] Smilegate Holdings, Inc. App Store – Epic Seven. Archived at https://archive.ph/ZBZl2. https://apps.apple.com/us/app/epic-seven/id1322399438.

[94] M. Steinböck, J. Bleier, M. Rainer, T. Urban, C. Utz, and M. Lindorfer. Comparing Apples to Androids: Discovery, Retrieval, and Matching of iOS and Android Apps for Cross-Platform Analyses. In *Proceedings of the 21st International Conference on Mining Software Repositories (MSR)*. (Apr. 2024). doi:10.1145/3643991.3644896.

[95] Swift.org. Swift.org – Package Manager. Archived at https://archive.ph/4i8F5. https://www.swift.org/documentation/package-manager/.

[96] Trufflesecurity. GitHub – TruffleHog. Version: 3.84.0. https://github.com/trufflesecurity/trufflehog/.

[97] United Command International Ltd. App Store – Jelly Blast. Archived at https://archive.ph/ydPsK. https://apps.apple.com/us/app/jelly-blast/id372948897.

[98] Unity. Unity Real-Time Development Platform. Archived at https://archive.ph/qwkgW. https://unity.com/.

[99] T. Watanabe et al. Understanding the Origins of Mobile App Vulnerabilities: A Large-Scale Measurement Study of Free and Paid Apps. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*. (May 2017). doi:10.1109/MSR.2017.23.

[100] Wattpad Corp. App Store – Wattpad - Read & Write Stories. Archived at https://archive.ph/k1ukh. https://apps.apple.com/us/app/wattpad-read-write-stories/id306310789.

[101] E. Wen, J. Wang, and J. Dietrich. SecretHunter: A Large-scale Secret Scanner for Public Git Repositories. In *Proceedings of the 21st IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. (Nov. 2022). doi:10.1109/TrustCom56396.2022.00028.

[102] H. Wen, J. Li, Y. Zhang, and D. Gu. An Empirical Study of SDK Credential Misuse in iOS Apps. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC)*. (Dec. 2018). doi:10.1109/APSEC.2018.00040.

[103] D. Yeke, M. Ibrahim, G. S. Tuncay, H. Farrukh, A. Imran, A. Bianchi, and Z. B. Celik. Wear's my Data? Understanding the Cross-Device Runtime Permission Model in Wearables. In *Proceedings of the 45th IEEE Symposium on Security & Privacy (S&P)*. (May 2024). doi:10.1109/SP54263.2024.00077.

[104] Y. Zhang, Y. Yang, and Z. Lin. Don't Leak Your Keys: Understanding, Measuring, and Exploiting the AppSecret Leaks in Mini-Programs. In *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. (Nov. 2023). doi:10.1145/3576915.3616591.

[105] Y. Zhou, L. Wu, Z. Wang, and X. Jiang. Harvesting Developer Credentials in Android Apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WISEC)*. (June 2015). doi:10.1145/2766498.2766499.

[106] O. Zungur, A. Bianchi, G. Stringhini, and M. Egele. AppJitsu: Investigating the Resiliency of Android Applications. In *Proceedings of the 6th IEEE European Symposium on Security & Privacy (EuroS&P)*. (Mar. 2021). doi:10.1109/EuroSP51992.2021.00038.

[107] C. Zuo, Z. Lin, and Y. Zhang. Why Does Your Data Leak? Uncovering the Data Leakage in Cloud from Mobile Apps. In *Proceedings of the 40th IEEE Symposium on Security & Privacy (S&P)*. (May 2019). doi:10.1109/SP.2019.00009.

A Appendix

```
Dear {Developer_Name},

We are security researchers from the University of Vienna studying
↪  Android and iOS apps. During our research, we discovered in
↪ your {platform} app ({app_name}) the following finding:

**Hardcoded Credentials** We found hardcoded AWS credentials (
↪ access and secret key) {CREDENTIALS} (removed parts of the key
↪ from the mail) in the following file {FILE}. The credentials
↪ can give attackers unauthorized AWS cloud resource access and
↪ might lead to data breaches on AWS. We recommend revoking the
↪ keys and using Amazon Cognito instead.

Despite best efforts, some findings might be false positives or
↪ already fixed in the latest version.
If you have any further questions or comments, please contact us.

Sincerely, David Schmidt
```