Computer Networks and Communications
http://ojs.wiserpub.com/index.php/CNC/

Research Article

A Review of Virtual Private Network Mobile Application Security

Suzanna Schmeelk[1*], James Dermezis[1], Andre Duchatellier[1], Charles Orbezo[1], Tomas Medina[1], Jared Reid[1], Denise Dragos[1], Lixin Tao[2]

[1] Department of Computer Science, Mathematics and Science, M.S. Cyber and Information Security Program, St. John's University, New York, USA
[2] Department of Computer Science, Pace University, New York, USA
E-mail: schmeels@stjohns.edu

Received: 24 February 2025; **Revised:** 4 June 2025; **Accepted:** 30 June 2025

Abstract: In an era of heightened online risks, the demand for Virtual Private Networks (VPNs) has surged. The VPN market has grown significantly, ranging from popular services like NordVPN, which holds a quarter of the market share, to applications with a small installation base. Studies show that as of 2024, 46% of Americans use at least one VPN application. Given VPNs' role in protecting sensitive data, questions have arisen regarding the security posture of VPN applications themselves. This study systematically reviewed 27 Android VPN applications selected from mobile app stores, following the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) guidelines to ensure methodological transparency and reproducibility. The analysis utilized application software review, the Mobile Security Framework (MobSF), and public-facing information to assess each application's vulnerabilities, required permissions, and data collection practices. The findings revealed significant variability across the applications and common issues, such as the use of insecure random number generators, excessive permission requests, exported components lacking proper access controls, and the logging of sensitive user information. Based on these results, the study highlights the need for improved secure coding practices to enhance the security posture of existing mobile VPN applications.

Keywords: Android applications, Virtual Private Networks, mobile application security, static analysis, mobile security framework

1. Introduction

The National Institute of Standards and Technology (NIST) defines a Virtual Private Network (VPN) as [1] "A restricted-use, logical (i.e., artificial or simulated) computer network that is constructed from the system resources of a relatively public, physical (i.e., real) network (such as the Internet), often using encryption (located at hosts or gateways), and often by tunneling links of the virtual network across the real network."

A VPN encrypts user data and masks the original IP address when connecting to a remote server through the service. Since the data is encrypted, it is difficult to intercept if implemented properly. A 2024 study [2] found that 46% of Americans use VPNs, a 7% increase from 2023. The study reported that a majority of VPN users are college educated, aged 45 to 60 years. The study also reported *NordVPN* as the most popular VPN, accounting for 27% of the market. Several other applications make up the remaining 73%, according to the study. The number of users who paid for VPNs was roughly equal to those who did not, with only a slight discrepancy [2].

Despite the various advantages, VPN applications can also incur privacy and security risks. A comprehensive study conducted by Ikram et al. [3] explores privacy and security risks posed by VPN applications that utilize the `BIND_VPN_SERVICE` permission. The researchers in this study found that although 67% of the VPN applications claim to enhance security and privacy, 75% of the applications utilize third-party tracking libraries, and 82% request excessive permissions, granting access to sensitive user information. In addition to these issues, the study focuses on various VPN privacy and security flaws, including implementing tunneling protocols without encryption, performing traffic manipulation and TLS interception, embedding malware, and failing to properly route DNS and IPv6 traffic. The researchers advise that the issues presented impact user confidentiality and anonymity [3].

The popularity of mobile VPN applications has led to the development of many such applications. With different VPN applications on mobile application stores, it can be challenging to determine trustworthy options that follow proper security and privacy standards. This study utilizes application software review, the Mobile Security Framework (MobSF) [4] and public-facing information to assess each application's vulnerabilities, required permissions, and data collection. This information can then be analyzed to determine whether an application is less-risky and whether it poses any threats on its own. The applications were selected through a systematic review process based on the PRISMA 2009 [5] guidelines to ensure transparency and reproducibility of the methodology. Following the selection process, MobSF performed static analysis and manual code inspection, highlighting several key security aspects, including permission requests, cryptographic implementations, logging behavior, data transmission practices, and misconfigurations of exported components.

2. Literature review

The literature review presents a comprehensive examination of relevant research topics identified through an analysis of peer-reviewed articles. The review is structured into three principal sections: static analysis, virtual private networks (VPNs), and mobile application security. This organization allows for a thorough exploration of each subject area.

2.1 *Static and dynamic analysis*

Static analysis is used to analyze source code without executing it. This can help identify poor coding practices and potential security flaws [6]. The automatically generated findings of static analysis tools must be manually evaluated to detect false positives and isolate important information. Static analysis is also capable of monitoring the flow of information in mobile applications [7]. However, this method is not without its share of drawbacks, leading to research on improved ways to analyze Android applications [8]. Klein et al. developed Dexpler, which converts strike/remove Dalvik bytecode into Jimple, which can be effectively processed. Research was done on AndroZoo to rebuild using an Android application's lineage to see how vulnerabilities evolve. AndroZoo has a repository of over three million applications analyzed for malware using tens of different Antivirus products. This research shows the challenges of static analysis of mobile applications, such as high amounts of false alarms and the modeling of Android applications. This is important in this field so other developers and researchers can find ways to provide solutions for these challenges [8].

In contrast, dynamic analysis detects vulnerabilities based on an application's run-time behavior. When used in conjunction with static analysis, dynamic analysis can detect vulnerabilities that would not be noticed otherwise. Analysis tools output a post-analysis report on the scanned application, including the APK information, permissions, and the various vulnerabilities. Dynamic analysis is usually more complex, requiring additional software that simulates an application and generates user input. Three types of dynamic analysis tests are fuzz testing, concolic testing, and search-based testing. Fuzz testing evaluates an application's reaction to unexpected input, monitoring crashes, failures, and possible memory leaks. Concolic testing uses both symbolic and concrete (literal) input variables to test a program's execution. Search-based testing is a technique used to automate a testing task. Such techniques are used in the currently deployed dynamic analysis tools such as AndroTest, Ares, and Troller [9]. A firmer grasp of the applications can be obtained using static and dynamic analysis with fewer false positives. These steps are essential to uncover possible vulnerabilities and the true significance of these findings.

Analyzing mobile applications using static and dynamic techniques can identify potential security risks, such as excessive permissions or logging of sensitive information. Logging sensitive information can create potential security risks that can compromise users' privacy, especially if they are unaware that it is occurring. Insecure logging practices can store sensitive information like hard-coded secrets, passwords, locations, or token information, which can be exploited and leveraged by malicious actors due to their lack of encryption and access controls. An examination can be conducted by using MobSF and/or manual code review to identify logging by keywords such as Log.d, android.util.log, or Logger [10]. When referring to permissions, it can be fairly difficult for users to differentiate when permission requests by a mobile application are being used for core functionality or for third-party libraries. Using third-party libraries in mobile applications has significant implications: they often request more permissions than needed and can leak sensitive personal information. Researchers from the Foundation for Research and Technology-Hellas (FORTH) have developed a dynamic analysis system, REAPER [11], to allow real-time tracking of permission requests used by third-party libraries.

Cryptographic Application Programming Interfaces (APIs) have often been used for data encryption, decryption, digital signatures, and other functions on Android devices. However, misuse can create vulnerabilities, and according to [12], nearly 80% of CVE vulnerabilities come from developers' lack of skill in mastering APIs. The research proposes a cryptographic misuse detection method for Android applications based on dynamic and static detection. Dynamic and static detection can cover their faults, utilizing static detection based on program slicing, with dynamic detection based on logging techniques to increase accuracy and minimize false positives [12]. This can provide developers with better security on their Android applications and lower the risk of private information being leaked by their Android applications. Suppose an application has incorrect cryptographic API calls. In that case, this can expose sensitive information, which will be detrimental, especially to mobile applications like VPNs. The takeaway from this is that Android application developers need to have a solid understanding of cryptographic basics and use the correct cryptographic API calls [3].

2.2 *Virtual Private Networks*

VPNs enhance security and privacy by encrypting internet traffic and masking the user's IP address. Although this makes VPNs appear attractive to end users and the general public, the security and trustworthiness of VPN applications have come under scrutiny, particularly on mobile platforms. In the context of VPN applications, the flexibility of the Android platform allows developers to request permissions and implement VPN protocols, cryptographic algorithms, and personalized features. This flexibility can lead to vulnerabilities and misuse, particularly when VPN applications fail to implement robust security measures or incorporate harmful practices such as insecure cryptographic or third-party library implementations. Notable studies on the analysis of Android VPN applications in search of harmful practices and abusive permissions have used various methods for code analysis, including static permission analysis, malware analysis [13], and static analysis paired with a convolutional neural network (CNN) [7].

A 2016 research study [3] performed a comprehensive analysis of 283 Android applications that request the `BIND_VPN_SERVICE`, or VPN permission on Android devices, and addressed the potential security risks. The authors of the research paper express concerns about this permission, as it is a powerful Android feature that allows the specific application to manipulate, intercept, and forward all web traffic on the device. This paper will further investigate this concern by examining the permission requests of an original dataset, including the `BIND_VPN_SERVICE` permission, to identify potential security risks and assess the prevalence of overly permissive Android permissions. The 2016 study concluded that 38% of the VPN applications analyzed contained malware despite having several hundred thousand installations and positive ratings on the Google Play Store. The malware signatures found by analyzing these specified Android VPN applications correspond to various trojans, adware, "malvertising", and spyware. In addition to finding the presence of malware on over a third of the tested applications, 18% of the applications implement tunneling protocols without encryption, which calls into question claims of increased security and anonymity [3]. Overall, the data indicates that VPN applications mislead consumers with the presence of malware and tracking services, despite promises of increased security and online anonymity. Another research study [13] proposes MVDroid, an optimized detection system for malicious Android VPN applications based on a convolutional neural network (CNN). The study trained the model using a dataset of 1,300 VPN applications, applying static analysis methods with an emphasis on application permissions and behavior to identify malicious behavior. Similarly to research presented by [3], the study also selected applications based on the `BIND_VPN_SERVICE` permission and discovered that developers may be potentially misleading about requesting this permission by not disclosing the permission in the profile found on the Google Play Store [3].

A security assessment from Abbas et al. [14] discusses several security considerations about VPNs and their potential impact. The security assessment found that many VPNs rely on default configurations that expose the applications to threats and concerns, such as man-in-the-middle (MITM) attacks or DNS leaks. These misconfigurations, whether intentional or accidental, can result in privacy exposure, traffic in unencrypted form, and unnecessary logging of data. Traffic in an unencrypted form can cause traffic to be vulnerable to surveillance and man-in-the-middle attacks, which VPNs are designed to protect against. The HTTP protocol and PPTP/L2TP support are a common source of vulnerability due to being outdated. Without careful consideration, users can expose their sensitive data, such as passwords, leading to data privacy attacks. Another problem occurs when DNS requests are wrongfully disclosed, destroying the anonymity VPNs are designed to protect. Furthermore, VPNs are capable of logging data without consent and then selling the information to third parties.

VPN applications are also subject to compliance and regulatory frameworks, including the GDPR [15] and FISMA [16]. Compliance and regulatory frameworks force VPN applications to have the proper security protocols for handling sensitive data with secure encryption and the right for users to remove their data. GDPR also requires logging information, which an organization must be aware of and verify what information is being logged. [15]. VPNs that fail to adhere to compliance and regulatory frameworks risk being exposed to attack vectors. If sensitive information is logged, an attacker could potentially exfiltrate data directly from the VPN services themselves.

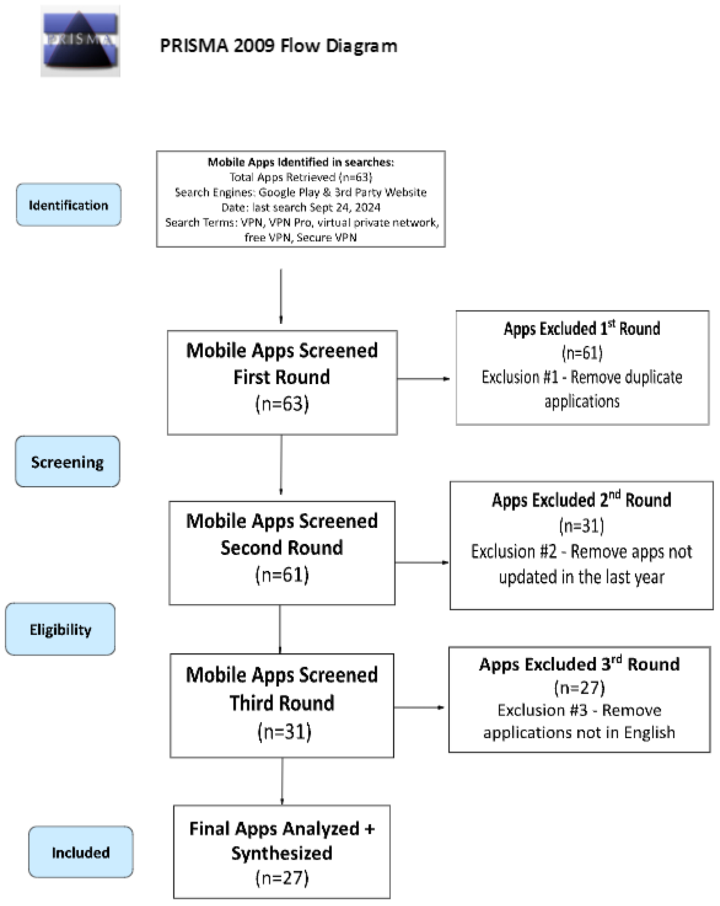
2.3 *Mobile application security*

Mobile application security has become increasingly critical as mobile devices proliferate. Mobile devices contain a substantial amount of user personal information, heightening the importance of safeguarding these devices' security [17]. Consequently, individuals must consider the security posture of any applications installed on their devices. A privacy protection framework could mitigate this issue. Such a measure would prevent applications from stealing user data by restricting access from unnecessary permissions. It interacts with the application and modifies the data permission inside to protect the user data while keeping the functionality. The last portion is significant, as some applications may cease functioning when restricting unnecessary permissions. Consumers should be aware that data is being used or collected for reasons that may not be deemed necessary. This will provide consumers greater control over their devices and information security [18].

3. Methodology

To begin the research, the methodology involves analyzing various Android VPNs available on the Google Play Store and hosted by a 3rd-party. A review process based on the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) [5] guidelines was used to obtain applications for a controlled dataset. The PRISMA review process, as shown in the flow diagram (Figure 1), was chosen to communicate research effectively, while allowing transparent reporting of why the systematic review was done, the methods used, and what was found in the results. Widely adopted in over 60,000 reports and endorsed by almost 200 journals, PRISMA serves as a vital tool to standardize the process while increasing the quality of meta-analysis reporting in trials [5]. These characteristics enhance reproducibility, allowing future researchers to perform a similar process and validating the credibility of the data.

Following the usage of the PRISMA methodology, the identification process began with selecting Android applications from the mobile playstore [19] related to VPNs. The initial dataset consisted of 63 applications (n = 63), retrieved by using keyword searches designed to capture a broad range of VPN-related applications while limiting the scope of the search. The search terms used were as follows: *VPN, virtual private network, VPN Pro, Free VPN,* and *Secure VPN.*



PRISMA 2009 Flow Diagram
Identification
Mobile Apps Identified in searches: Total Apps Retrieved (n=63) Search Engines: Google Play & 3rd Party Website Date: last search Sept 24, 2024 Search Terms: VPN, VPN Pro, virtual private network, free VPN, Secure VPN
Mobile Apps Screened First Round (n=63)
Apps Excluded 1st Round (n=61) Exclusion #1 - Remove duplicate applications
Screening
Mobile Apps Screened Second Round (n=61)
Apps Excluded 2nd Round (n=31) Exclusion #2 - Remove apps not updated in the last year
Eligibility
Mobile Apps Screened Third Round (n=31)
Apps Excluded 3rd Round (n=27) Exclusion #3 - Remove applications not in English
Included
Final Apps Analyzed + Synthesized (n=27)

Figure 1. Mobile Application PRISMA Search Criteria.

These applications were screened for eligibility through predetermined exclusion criteria to ensure relevance and consistency. In the initial screening, the primary criterion was the removal of duplicate applications, which reduced the number from 63 to 61. This step helped limit redundancy that could impact the results. The subsequent round of screening filtered out applications that had not been updated within the past 12 months, bringing the total down to 31. The goal of this step was to focus the analysis on applications that were actively maintained, as applications with limited support may not be representative of the current VPN ecosystem. Finally, applications not available in English were excluded, resulting in a final dataset of 27 applications for further analysis.

After obtaining the APK files for each application in the dataset, a static analysis tool was employed to examine the files and generate detailed reports. This requirement was fulfilled by the open-source platform known as the Mobile Security Framework (MobSF) [4]. MobSF provided the tools necessary to properly analyze the APK files and generate structured reports on security issues detected. Each of the 27 applications was uploaded to the MobSF platform and statically analyzed, and the results were compiled into a metadata spreadsheet. The spreadsheet included details shown through the analysis reports, including, but not limited to, the package name, the MobSF report rating and score, origin country, developer information, version, ratings, and update timestamps. MobSF classifies risk based on 'High', 'Medium', 'Informational', and 'Secure' severity ratings [4], and utilizes this information to calculate an application's overall security score. Additionally, MobSF assigns a letter grade (A to F) for each application based on the final security score after MobSF statically analyzes a VPN application [4]. These scores, along with detailed findings, served as the basis for comparative analysis in this study. The formula used by MobSF to calculate risk scores was as follows:

$$\text{Security Score} = 100 - \left(\frac{(\text{High} \times 1.0) + (\text{Medium} \times 0.5) - (\text{Secure} \times 0.2)}{\text{Total Findings}} \times 100 \right) \tag{1}$$

Building on the results of the analysis, areas of recurring or high severity risks were observed through the analysis report and became the focal point of this research paper and the findings. To further investigate these issues, a manual code review was performed on the source code and the manifest files to confirm the accuracy of the findings generated by MobSF. The manual code review provided a method to uncover and validate findings proposed in the static analysis reports and eliminate false positives [20]. These issues seen across the applications in the dataset varied in severity, ranging from informational to high risk. The specific categories of risk analyzed were as follows:

3.1 *Category #1: Cleartext traffic*

The first security issue observed was cleartext traffic enabled in various applications. According to MobSF, the application seeks to use cleartext network traffic via unencrypted protocols and services such as HTTP, FTP stacks, DownloadManager, and MediaPlayer. These protocols can put users at risk as they expose data to interception or tampering by network attackers. This is detrimental as it could occur with benign data or possibly sensitive data such as passwords or any other personal information sent on the VPN, violating users' expectations of secure browsing. This leaves the information vulnerable to attacks that exploit this vulnerability, such as ARP, DNS poisoning, or MITM attacks [14].

To further support this concern, the National Institute of Standards and Technology (NIST) and, more specifically, NIST SP 800-82r3 [1] describe cleartext as unencrypted information. NIST warns that such usage makes the data vulnerable to compromise, unless mitigated through proper policies. Misconfiguration, whether on purpose or by accident, could compromise the information on portable devices, which are widely used for many use cases [1]. This risk was identified by analyzing the APK files and observing if it was enabled. Upon reviewing the reports, the application could be identified as having used cleartext. Verifying the usage of cleartext was done by examining the source code and the Android manifest file, giving greater insight into determining if this setting was set to true. With manual code review, MobSF reports could be validated.

3.2 *Category #2: Sensitive information logging*

The second category of focus within the methodology was the logging of sensitive information, as personal or sensitive information should never be stored. This issue was rated as "information" severity and can be seen through the code analysis section on MobSF. When checking through the files in MobSF, not every file is a log storing sensitive information, so manual code review was conducted to identify what logs are storing sensitive information. It is important to identify what information is being stored and what would be considered sensitive data that would be valuable for attackers, such as passwords, credit card information, and Personally Identifiable Information (PII). If an attacker or adversary retrieves this information, social engineering attacks can be carried out, personal accounts can be hijacked, and abuse information can be obtained. Information can be stored in log files. Logging is important for keeping track of crashes, errors, and usage statistics; however, logging sensitive data can expose it to attackers and violate user confidentiality. Log files are also required for certain laws, such as GDPR (General Data Protection Regulation) [15] in the European Union and CCPA (California Consumer Privacy Act) [21] in the United States.

The MobSF report references standards of CWE-532: Insertion of Sensitive Information into Log File and Open Worldwide Application Security Project (OWASP) MASVS: MSTG-STORAGE-3. CWE-532 states the scope is confidentiality, and the impact is that full path names, system information, etc., can provide attackers with a path to acquiring information [10]. OWASP has conducted multiple tests to check the stored sensitive information. In this case, sensitive data in the logs is of great interest. Tests are conducted by analyzing source code for logging-related code, which would be done during the testing phase, checking the application data directory for log files, gathering system messages and logs, and analyzing any sensitive data [10]. In this case, it is interesting for VPNs to store sensitive data in log files since the primary purpose of VPNs is to safeguard user's privacy. It's important for consumers to be careful of VPNs storing their information, as they can potentially sell it without them knowing. Further testing would be done on each of the applications to determine the number of files potentially leaking sensitive information and what information is being potentially leaked. The ultimate goal is to understand if there is a legitimate reason for storing sensitive information.

3.3 *Category #3: Insecure RNGs*

The third category of focus within the methodology was the usage of insecure random number generators (RNGs). Utilizing insecure random number generators in Android applications, particularly with virtual private networks, can introduce significant security vulnerabilities. Overall, insecure random number generators often have low entropy and produce predictable random numbers, effectively compromising cryptographic operations designed to protect the confidentiality, integrity, and availability of user information. As a result, insecure RNGs may be exploited to infer cryptographic keys, bypass authentication mechanisms, or expose sensitive user information if used for these operations [3].

According to the MobSF static analysis report, applications that are flagged under this specific vulnerability correspond to CWE-330: Use of Insufficiently Random Values [22], which highlights the use of RNGs that lack sufficient entropy or predictability. According to the Common Weakness Enumeration [22], poor random number generators used in cryptographic functions can be exploited by an attacker or adversary to potentially predict the sequence of generated random numbers, compromising potential protection mechanisms or allowing for unauthorized access.

According to the guidelines presented by the Open Worldwide Application Security Project (OWASP) [23], when testing for random number generation, all custom or well-known insecure implementations should be identified and replaced with an algorithm proven to be resilient against prediction attacks and pass statistical randomness tests. For example, further guidance [24] in this area suggests that, in Android applications it is not recommended to use the 'java.util.Random' class, as it does not provide sufficient randomness. Instead, OWASP guidelines [25] recommend implementing the 'java.security.SecureRandom' class, as it produces non-deterministic output and sufficiently generates random values.

The process of identifying instances of insecure RNGs began with analyzing Android Application Package (APK) files using MobSF static analysis. After reviewing the static analysis reports and flagged usage, a deeper inspection of the source code was used to determine whether the MobSF static analysis identified a false positive. The process involved extracting and decompiling the Dalvik Executable (DEX) files into Java source code. Manual code review was then performed to determine if the flagged uses were false positives, and to understand whether the source of the flagged usage was the main application code or an integrated third-party library.

3.4 *Category #4: Permissions*

The fourth category involved the permissions required by each VPN application. There are several things to consider when reviewing applications, including whether permission is necessary or even beneficial to the functionality of an application. Furthermore, each permission is inherently a vulnerability. If an application is compromised by a threat actor, the attacker can abuse its access. Therefore, permissions should not be granted liberally. OWASP defines the "principle of least permission" as only allowing any party the necessary permissions to function and no privileges beyond their mandate [26]. Therefore, each VPN application should only use permissions that help in its operation. For this analysis, permissions involving monetization, such as payment collection and advertising, will also be considered necessary, provided that they are implemented lawfully. Android permissions are divided into two major categories: install-time and runtime. Install-time permissions are granted upon installation. Users inherently accept these permissions by installing an application. Runtime permissions are requested while using an application. These permissions are only granted if the user manually accepts them or unlocks them from their device's settings menu [27]. These two categories are further divided into subcategories, which are elaborated upon below:

MobSF identifies the permissions required by an application and rates them based on their severity using Android's official protection level categories [28]. Permissions that do not pose a threat to security are labeled as "normal," while permissions that create a potential vulnerability are labeled as "dangerous." However, it is worth noting that normal permissions do extend beyond the application's sandbox. Permissions that MobSF does not recognize are classified as "unknown." Finally, "signature" permissions are unique in that they automatically grant system access to another application without notifying the user, but only if both applications have matching certificates. Other types of permissions exist, but are not pertinent to this analysis [28]. The number of each type of permission will be tabulated using a graph.

Android outlines three best practices for developers to follow when incorporating permissions into applications. First, users should have as much control over data they share as possible. Second, developers should be transparent, so that the user understands what the application is accessing and the reason for the necessary access. Finally, applications should use as little data as possible. [28] This analysis uses MobSF reports for data collection, so limited insight is provided as far as these best practices are concerned. These reports do not show the user interface and explain the functionality of permissions, not their implementation.

MobSF allows users to view the exact files within the source code associated with a permission. As a rule, permissions should only be requested by code using first-party (Android) libraries or trusted third parties. This study observed each file within the source code linked to a potentially dangerous permissions to determine the uses, which could not be determined from the nature of the permissions alone. This allows for the elimination of false positives.

3.5 *Category #5: Exported components*

In mobile application security, exported components, such as activities, services, and broadcast receivers, introduce serious risks when they are configured incorrectly. An exported component is one that is accessible to other applications; if it is not properly protected, then it becomes a vector of unauthorized access or malicious exploitation. According to the OWASP, components are declared exported either by explicitly setting *android:exported="true"* or by defining an intent-filter which, if absent, implicitly sets the component as exported. These will lead, if proper permissions are not implemented, to sensitive operations via malicious applications that invoke these components, which can finally result in a security breach [29]. MobSF was then used to perform a static analysis on the mobile application dataset to assess these vulnerabilities.

This tool decompiles application packages (APKs) and examines the *AndroidManifest.xml* file to identify exported components and their associated permissions. Exported components present substantial risk not simply because of their accessibility, but due to the operations they expose in the absence of adequate safeguards. Activities exported without permission requirements or input validation can be triggered by any app on the device, which enables threats such as session hijacking, credential theft, or unauthorized VPN activation. Broadcast receivers that are exported and listen for system or custom events may also be exploited to leak sensitive data, perform unwanted operations, or even trigger denial-of-service, especially where permissions or intent restrictions are missing.

Each component's type (activity, service, receiver), exported status, and protection level was cataloged, evaluating not just the presence but the actual security posture—whether permission checks were enforced, input was validated, and sensitive actions were restricted. Analysis consistently found that the greatest risk exists when exported components are both accessible and under-protected, especially when app state or sensitive user data can be altered without robust access control. This approach aligns with NIST guidelines, which call for careful vetting to ensure mobile apps are free from vulnerabilities that can be exploited to steal data or control a user's device [30].

Mitigation strategies supported by OWASP and Android security guidance stress the need to integrate strict permission checks, validate all incoming data, and minimize exportation to only components required for essential app functionality. Recent documentation from Android highlights that proper evaluation should move beyond simply labeling all exported components as risky and must consider the context, data, and actions exposed [31].

4. Findings/Discussion

The 27 VPN applications were analyzed using MobSF and the public meta-data about each of the applications were reviewed. Figure 2 depicts the aggregated data from the MobSF static analysis reports:

- The average MobSF security grade was B, with an average security score of 47.52. Out of the analyzed dataset,

- n = 0 received a grade of A
- n = 26 received a grade of B
- n = 1 received a grade of C
- n = 0 received a failing grade of F (score below 28)

• MobSF identified an average of 22.4 medium and 3.9 high vulnerabilities per application across the dataset. Out of the analyzed dataset,

- n = 3 had 15 or fewer medium vulnerabilities.
- n = 4 had 30 or more medium vulnerabilities.
- n = 9 had 2 or fewer high vulnerabilities.
- n = 11 had 5 or more high vulnerabilities.

• MobSF identified an average of 5.22 user/device trackers per application across the dataset. Out of the analyzed dataset,

- n = 3 integrated no user/device trackers.
- n = 13 integrated between 1 and 5 user/device trackers.
- n = 5 integrated 10 or more user/device trackers.

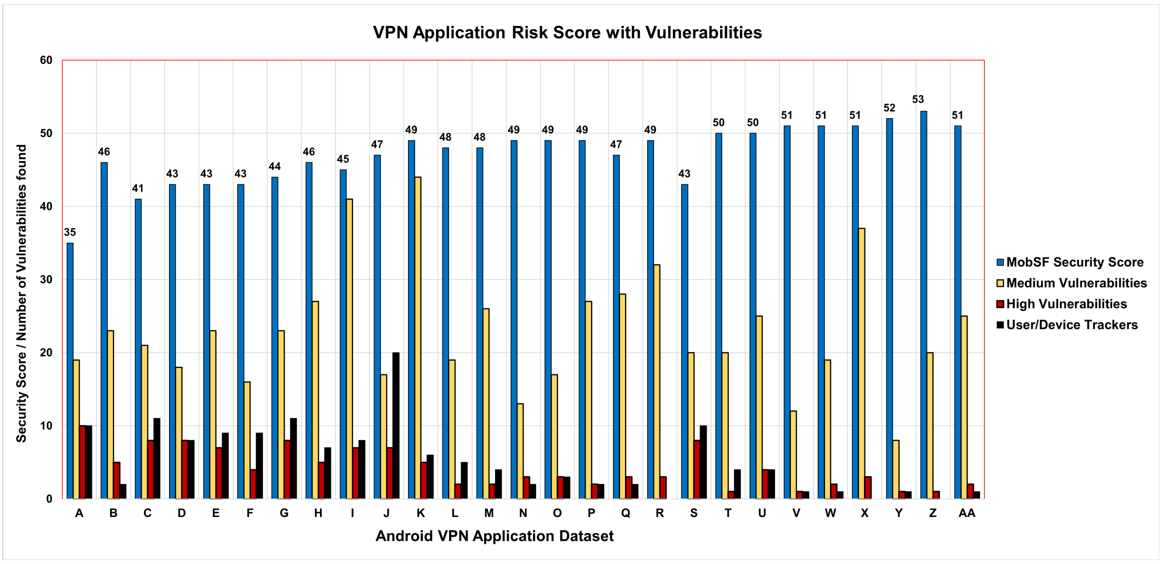


Figure 2. MobSF Risk Scores and Vulnerabilities

4.1 *Category #1: Cleartext traffic*

An analysis was conducted on all selected applications, focusing on code related to cleartext traffic. Of the 27 reports generated by MobSf, several applications were found to allow cleartext communication, a practice considered insecure. According to Figure 3, approximately 70.37% (19 applications) had cleartext disabled. However, 29.63% (8 applications) enabled it, which is surprising given the security risks. As this feature allows for unencrypted use, it is clear how such a feature could be an issue for user privacy [32]. Exploitation would allow data to be captured, logged, or freely modified, leading to possible MITM attacks [14]. Furthermore, such exposure can result in DNS poisoning, where an attacker spoofs DNS responses and changes a DNS cache entry to redirect users to fraudulent or proxy servers [33].

In Figure 4, a sample from the Android manifest of an application shows the permissions, features, and configuration settings, making it a core file in every application. This figure shows the configurations in the application that enable cleartext traffic via the *android:usesCleartextTraffic=true*. What is unique about this is how it relates to API levels in the Android Platform. According to [32], the guidelines are meant for Android applications running Android 8.1 (API level 27) and earlier; they do not enforce restrictions on cleartext traffic. However, in Android 9 (API level 28) and later, cleartext is disabled by default through HTTPS. Although this setting is left up to the developers to disable, there could be reasons for it to remain in the files. One reason could be that it is a forgotten feature for testing purposes of the application. Another reason is legacy compatibility with older systems, where disabling it could cause instability in interactions with certain applications. This is the case with libraries that support HTTP, such as Ktor [32]. Thus, this setting becomes vital for certain applications while becoming a possible vulnerability.

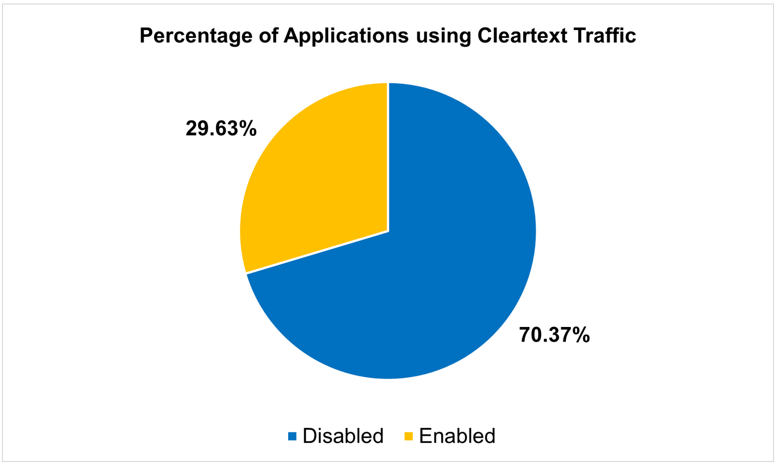
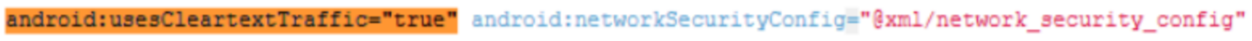


Figure 3. Percentage of Applications using Cleartext Traffic

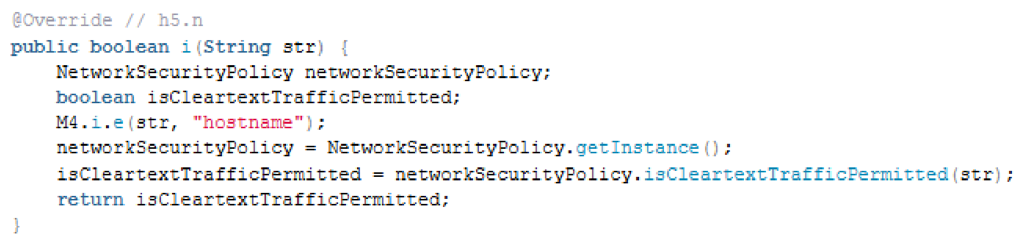
To further verify the usage of cleartext, Figure 5 shows some of the application's inner workings obtained through manual code review. This code demonstrates the use of the *NetworkSecurityPolicy* class concerning cleartext traffic. Specifically, the line `isCleartextTrafficPermitted=networkSecurityPolicy.isCleartextTraffcPermitted(str);` checks whether cleartext is allowed for a "hostname." It would store the true or false values from it; if true, it would allow HTTP traffic for the given host [34]. The code was analyzed according to trends observed in applications that had cleartext enabled, with each one deviating slightly based on the developer's coding styles or what is desired. *NetworkSecurityPolicy* is a common trend seen in all applications, making it possible to control the relevant aspects of network security behaviors.



```
android:usesCleartextTraffic="true" android:networkSecurityConfig="@xml/network_security_config"
```

Figure 4. Android manifest file showcasing cleartext is permitted

To implement mitigations by [32], it is recommended that the data be transported over encrypted communication channels. The usage of insecure protocols should be retired, and a secure alternative should be installed. General guidance is to use the `NetworkSecurityConfig.xml` feature to opt out of cleartext traffic. The only exception would be for debugging purposes. Using protocols such as HTTPS and SFTP is recommended, as they follow the following best practices: authentication through secure means, authorization of intended resources, and ensuring that secure ciphers are used. It is also recommended to use well-maintained third-party libraries or OS libraries if developers use well-known protocols similar to HTTPS [32].



```
@Override // h5.n
public boolean i(String str) {
    NetworkSecurityPolicy networkSecurityPolicy;
    boolean isCleartextTrafficPermitted;
    M4.i.e(str, "hostname");
    networkSecurityPolicy = NetworkSecurityPolicy.getInstance();
    isCleartextTrafficPermitted = networkSecurityPolicy.isCleartextTrafficPermitted(str);
    return isCleartextTrafficPermitted;
}
```

Figure 5. Android VPN code showcasing NetworkSecurityPolicy

4.2 *Category #2: Sensitive information logging*

This category focuses on the number of files per VPN application that potentially leak sensitive information, according to MobSF static analysis reports. It was interesting to notice the number of files that MobSF suspects of potentially leaking sensitive information, with applications ranging from 2 to 561 files. In addition, MobSF highlights specific sections in the files source code that it detects as leaking sensitive information. After performing extensive code analysis, it does not seem every file listed is leaking sensitive information, and lists files containing logging. However, developers should review the issue warning on MobSF static analysis reports and the suspected files to ensure that sensitive information such as usernames, passwords, API keys, and other PII information is not leaked. Logging information can be helpful, especially in tracking crashes, and can be mandatory for General Data Protection Regulation (GDPR) [15], California Consumer Privacy Act (CCPA) [21], etc. Logging sensitive information can be dangerous and even violate user confidentiality. In the case of VPN applications, if the sensitive information is being logged without user knowledge, they may not be aware if it is being sold or if an adversary has it, which will defeat the purpose of having a VPN in the first place. Figure 6 depicts data on the number of files with sensitive information logging for each application in the collected dataset.

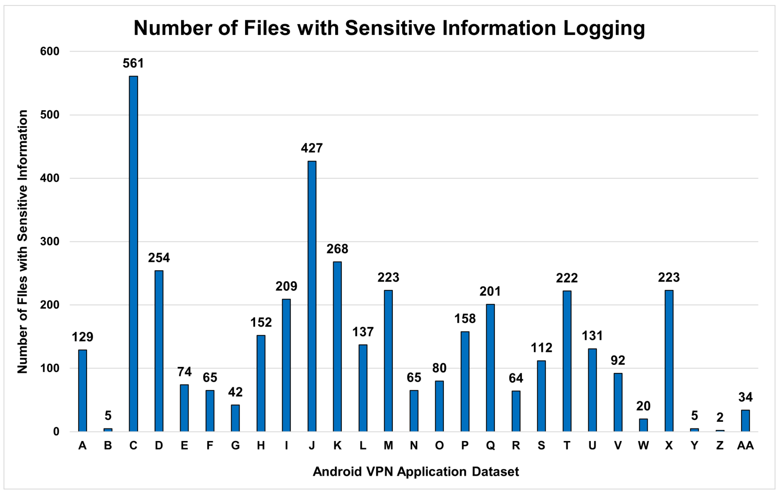
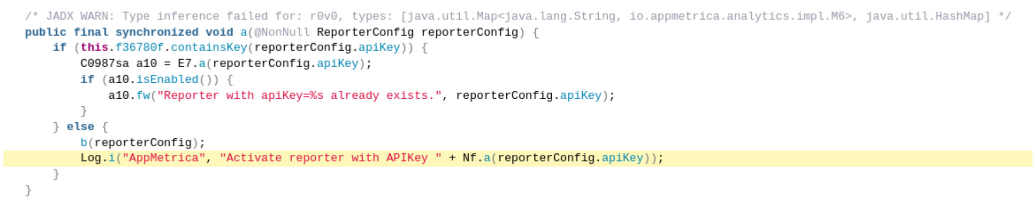


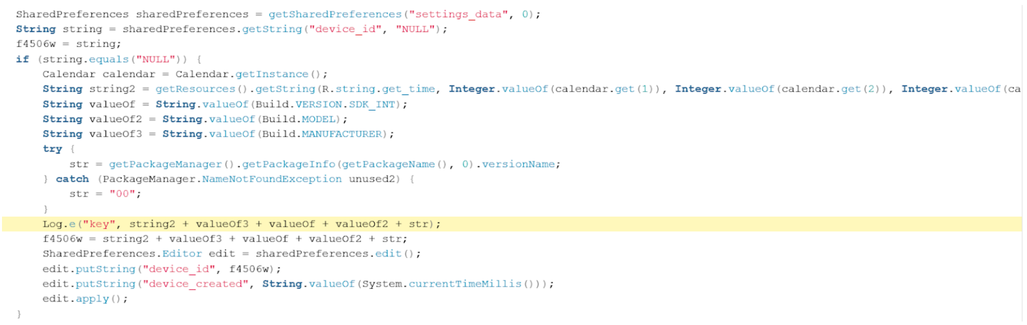
Figure 6. Number of Files with Sensitive Information Logging

Through the use of MobSF, under the code analysis section, the issue is labeled "The App logs information. Sensitive information should never be logged." Code obfuscation presents a challenge to detect sensitive information being logged, simply because PII could be renamed, or the difficulty to determine what variable is being logged. Despite the challenges, the screenshots below show confirmed examples of sensitive information being logged. The code snippet, shown in Figure 7, logs the APIKey, which is considered sensitive information. This inherently violates CWE-532 and MASVS MSTG-STORAGE-3. Additionally, the code snippet, shown in Figure 8 was obtained from MobSF, logs, build version, model, manufacturer, and device ID with no encryption. If an attacker obtains this information, they can potentially identify a device or user session.



```
/* JADX WARN: Type inference failed for: r0v0, types: [java.util.Map<java.lang.String, io.appmetrica.analytics.impl.M6>, java.util.HashMap] */
public final synchronized void a(@NonNull ReporterConfig reporterConfig) {
    if (this.f36780f.containsKey(reporterConfig.apiKey)) {
        C0987sa a10 = E7.a(reporterConfig.apiKey);
        if (a10.isEnabled()) {
            a10.fw("Reporter with apiKey=%s already exists.", reporterConfig.apiKey);
        }
    } else {
        b(reporterConfig);
        Log.i("AppMetrica", "Activate reporter with APIKey " + Nf.a(reporterConfig.apiKey));
    }
}
```

Figure 7. Code snippet obtained showing the logging of APIKeys



```
SharedPreferences sharedPreferences = getSharedPreferences("settings_data", 0);
String string = sharedPreferences.getString("device_id", "NULL");
f4506w = string;
if (string.equals("NULL")) {
    Calendar calendar = Calendar.getInstance();
    String string2 = getResources().getString(R.string.get_time, Integer.valueOf(calendar.get(1)), Integer.valueOf(calendar.get(2)), Integer.valueOf(ca
    String valueOf = String.valueOf(Build.VERSION.SDK_INT);
    String valueOf2 = String.valueOf(Build.MODEL);
    String valueOf3 = String.valueOf(Build.MANUFACTURER);
    try {
        str = getPackageManager().getPackageInfo(getPackageName(), 0).versionName;
    } catch (PackageManager.NameNotFoundException unused2) {
        str = "00";
    }
    Log.e("key", string2 + valueOf3 + valueOf + valueOf2 + str);
    f4506w = string2 + valueOf3 + valueOf + valueOf2 + str;
    SharedPreferences.Editor edit = sharedPreferences.edit();
    edit.putString("device_id", f4506w);
    edit.putString("device_created", String.valueOf(System.currentTimeMillis()));
    edit.apply();
}
```

Figure 8. Code snippet obtained from MobSF showing logs, build version, model, manufacturer, and device ID with no encryption

4.3 *Category #3: Insecure RNGs*

After performing the preliminary analysis with MobSF and manual code analysis outlined in the Methodology, of the 27 selected Android VPN applications, it was found that approximately 89% (24 of 27 applications) were identified as implementing insecure RNGs. In addition, several of the applications contained more than one implementation of an insecure random number generator. Figure 9 represents the percentages of applications flagged as implementing insecure RNGs. The pi-graph shows that of the n = 27 applications, n = 24 applications (approximately 89%) were identified as implementing insecure RNGs. Manual code analysis confirmed that the most common implementation of an insecure random number generator was the usage of the `java.util.Random` class, which is presented by OWASP [24] as vulnerable and does not provide sufficient randomness for security-critical operations, such as cryptography and key generation. A deeper investigation of the source code revealed that the majority of the insecure RNG implementations were found present in third-party libraries that facilitated log monitoring, advertisements, analytics, and other services unrelated to the primary functionality of the VPN application. One notable example was a third-party library that utilized the `java.util.Random` class in components handling DNS resolution. However, notably, insecure RNGs were also discovered within the core application logic, including network request handling, VPN tunnel logic, and in one case, within the `SESSION_KEY_ENCRYPTION` functionality. Additionally, several instances of insecure RNGs were used for non-security-critical purposes, particularly in third-party libraries, for internal logic. The findings of the manual code analysis highlight the importance of investigating integrated third-party libraries, in addition to the main application code.

Figure 10 depicts an example of an application importing and using this class in the main application source code. The usage of insecure random number generators in mobile VPN applications is concerning, as they are designed for and

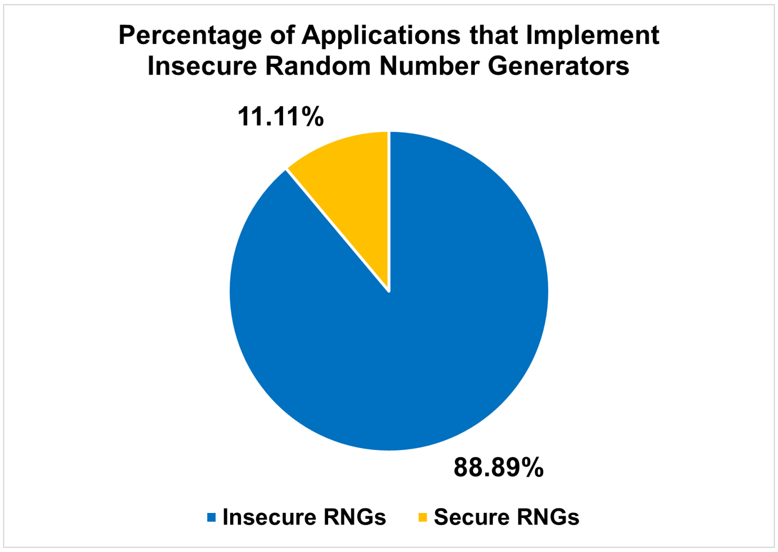
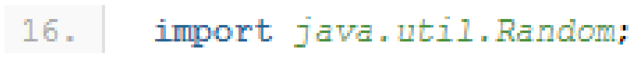


Figure 9. Percentage of Applications that Implement Insecure Random Number Generators (RNGs)

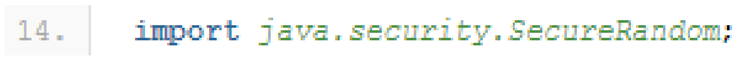
advertise strong encryption and security mechanisms to protect sensitive user information. In addition, this can potentially create an attack surface which may be exploited to intercept traffic, infer cryptographic keys, bypass authentication mechanisms, or expose sensitive user information [35].



```
import java.util.Random;
```

Figure 10. Example of an application importing the insecure ‘java.util.Random’ class within the main application source code

In contrast, the remaining 11% of applications followed the secure coding guidelines properly, as presented by OWASP [24], and implemented secure random number generators. Manual code analysis confirmed that the most common implementation of secure random number generators was the `java.security.SecureRandom` class, initialized by the default constructor with no arguments. Figure 11 depicts an example of an application importing and using this class in the main application source code.



```
import java.security.SecureRandom;
```

Figure 11. Example of an application importing the secure ‘java.security.SecureRandom’ class within the main application source code

To mitigate potential risk associated with predictable randomness and insecure RNGs, developers should avoid utilizing insecure classes, such as `java.util.random`, in any cryptographic implementations. Instead, all cryptographic constructions, involving key or token generation, should employ secure classes or libraries, such as `java.security.SecureRandom`. In accordance with OWASP guidelines [24], developers must ensure that RNGs not specifically documented as cryptographically secure are not used where unpredictable randomness is required. In addition, developers should exercise caution when integrating third-party libraries that employ insecure random number generators, even if they are not used for security-critical purposes.

4.4 *Category #4: Permissions*

As shown in Figure 12, nearly all VPN applications that were analyzed contained some dangerous permissions, although these were always outnumbered by “normal” permissions. Most applications also contained a small amount of “unknown” permissions. Signature permissions were the least common, being present in only four applications. In all four cases, the signature permission in question was used to determine where the application had been downloaded from.

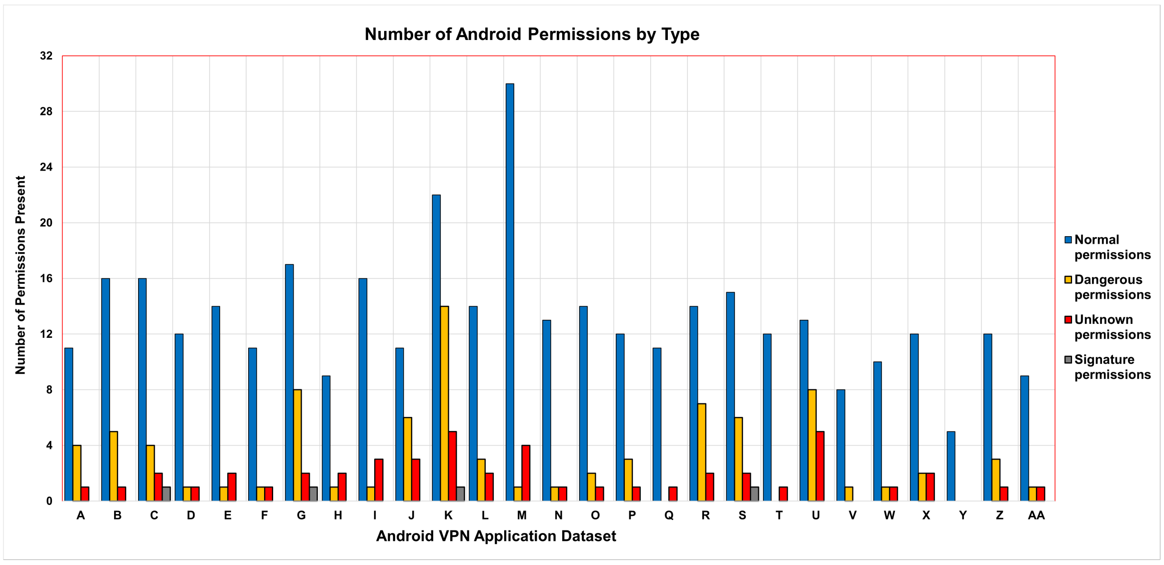


Figure 12. Number of Application Permissions by Type

At the bare minimum, a VPN application would only need network-related permissions. However, many VPN applications used different types of location access, which is considered a dangerous permission [26]. Another dangerous permission was the ability to read and write files, which should not be necessary, since VPNs are not meant to interact with stored data. Several common permissions were unrelated to functionality, but instead to marketing. These included permissions relating to advertising, billing, and notifications. Since developers need to monetize their products, these permissions are a necessary evil, although some methods of monetization are more ethical than others. Ideally, developers should aim to be transparent and frugal in the use of device permissions [28].

Examining the source code files associated with potentially hazardous permissions revealed that the majority of VPN applications required access to device storage, notifications, and location. All source code files linked to these permissions collect information that serves to profile users for targeted advertisements. These do not contribute to the core functionality of the application and exist for the sole purpose of monetization. Therefore, these permissions are non-essential as the application operates as intended without them.

4.5 *Category #5: Exported components*

Although the security configuration analysis of the exported components in VPN applications provided a robust framework, the identification of potential vulnerabilities was ensured. Components were categorized as Activities, Services, Receivers, and Providers by systematically extracting data from MobSF reports, with detailed documentation of their ‘android:exported=true’ status and protection levels. This standardized approach allowed for consistent comparison across VPN applications, highlighting differences in their security configurations. The explicit focus on protection mechanisms, or lack thereof, helped point out vulnerabilities that can expose applications to unauthorized access or abuse.

For example, as demonstrated in the example manifest, Figure 13, activities such as `InstrumentationActivity Invoker$BootstrapActivity` and receivers such as `AccountChangedReceiver` are marked as exported, either explicitly or through intent-filters, and some services such as `AuthenticatorService` are also exported. These highlights visually illustrate how exported components can be systematically identified and evaluated for their security posture. The analysis of these components revealed that the greatest risk exists when exported components are both accessible and under-protected, especially when app state or sensitive user data can be altered without robust access control. Without

```
<meta-data android:name="androidx.profileinstaller.ProfileInstallerInitializer"
android:value="androidx.startup" />
</provider>
<activity android:theme="@android:style/Theme"
android:name="androidx.test.core.app.InstrumentationActivityInvoker$BootstrapActivity"
android:exported="true">
<intent-filter>
<action android:name="android.intent.action.MAIN" />
</intent-filter>
</activity>
<activity android:theme="@android:style/Theme"
android:name="androidx.test.core.app.InstrumentationActivityInvoker$EmptyActivity"
android:exported="true">
<intent-filter>
<action android:name="android.intent.action.MAIN" />
</intent-filter>
</activity>
<activity android:theme="@android:style/Theme.Dialog"
android:name="androidx.test.core.app.InstrumentationActivityInvoker$EmptyFloatingActivity"
android:exported="true">
<intent-filter>
<action android:name="android.intent.action.MAIN" />
</intent-filter>
</activity>
<meta-data android:name="com.facebook.sdk.AutoLogAppEventsEnabled" android:value="false" /:
<meta-data android:name="com.facebook.sdk.AutoInitEnabled" android:value="false" />
<receiver android:name="com.avast.android.account.internal.account.AccountChangedReceiver"
android:permission="com.avast.android.account.avg.ACCOUNT_PERMISSION_V2"
android:exported="true">
<intent-filter>
<action android:name="com.avast.android.account.ACCOUNTS_CHANGED_V2" />
<action android:name="android.accounts.LOGIN_ACCOUNTS_CHANGED" />
</intent-filter>
</receiver>
<service android:name="com.avast.android.account.internal.account.authenticator.AuthenticatorSe
android:exported="true">
<intent-filter>
<action android:name="android.accounts.AccountAuthenticator" />
</intent-filter>
<meta-data android:name="android.accounts.AccountAuthenticator"
android:resource="@xml/authenticator" />
</service>
```

Figure 13. Example Android Manifest file

adequate permission checks or input validation, as evidenced in the manifest example, such components may be susceptible to session hijacking, credential theft, unauthorized VPN activation, or even denial-of-service attacks if malicious apps exploit exported activities or receivers. This approach aligns with NIST guidelines [30], which call for careful vetting to ensure that mobile apps are free from vulnerabilities that can be exploited to steal data or control a user's device. The manifest analysis, visually supported by the highlighted sections, underscores the critical need for restricting exportation, enforcing permissions, and validating external input to minimize risk.

The analysis, as visualized in Figure 14, revealed a distinct and concerning pattern: many VPN applications explicitly export multiple components—most notably activities that handle main user interfaces, deep links, or auxiliary features—yet these often lack robust security mechanisms such as permission gating or thorough input validation. Android Developer documentation underscores that exported components, if not tightly protected, permit untrusted applications to interact with privileged features or initiate unintended operations, leading to elevated risk of exploitation [31]. Particularly, exported activities and receivers demonstrate susceptibility to attacks such as intent spoofing, where a malicious app can send crafted intents to trigger sensitive operations or leak data.

The results, depicted in Figure 14, show a worrying tendency: although many components are explicitly exported, a large part of them lack proper protection mechanisms. Exported Activities and Receivers often lack or have poor protection. Even though some Services are protected by permissions such as `BIND_VPN_SERVICE` android permission, the configuration lapses are evident in some applications. Quantitative review of the dataset revealed that approximately 60% of components to be low-risk (non-exported or permission-protected), while 25% were medium-risk (exported without consistently handling sensitive data), and 15% were high-risk (exported and capable of processing external input or user credentials without proper access controls). Notably, VPN tunnel services, which are critical for app security, were seldom exported, reflecting best practices recommended by the Android security model.

Basic quantitative data, such as the number of unprotected components, combined with qualitative insights, such as the actual vulnerabilities in applications, gives significant reason to prior configuration settings for security in development. This type of analysis not only identifies areas of concern but also provides actionable recommendations to developers and stakeholders on how to create more resilient and secure applications against exploitation [30]. In summary, security risk is not inherent to component exportation alone, but emerges when these interfaces lack rigorous permission enforcement and input validation. Adhering to platform recommendations limiting exported components to those necessary for intended inter-app communication, enforcing permissions, and validating all incoming data remains essential for reducing attack surface and aligning with both empirical findings and Android's official privacy and security guidance [31].

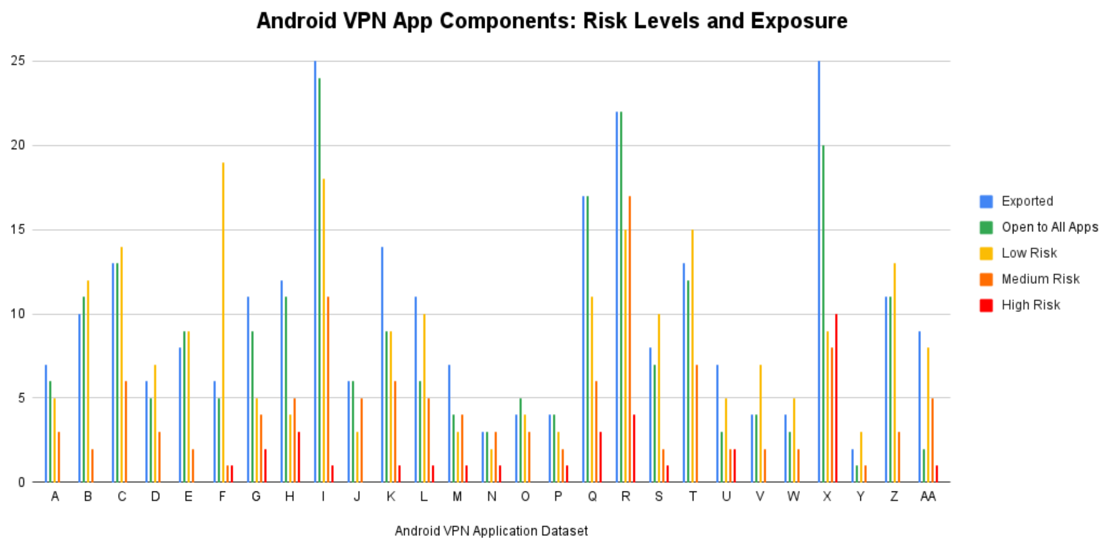


Figure 14. Number of Exported Activities, Services, and Receivers by Risk Type

5. Future work & limitations

The scope of this study was exclusively limited to static analysis and manual code review. Although this method produces meaningful data, it is unable to obtain a holistic view of an application's security posture. Static analysis views code without running it and must therefore make assumptions as to how the code will be compiled. These assumptions, which are based on known components of the code, are limited by human knowledge and what is programmed in the tool, showing how MobSF is limited to rules that detect known vulnerabilities. In addition, static code analysis and manual code review are significantly hindered by code obfuscation and anti-debugging techniques, which conceal internal logic and impede reverse engineering techniques. These techniques make it more difficult for researchers to understand relationships between classes and files within the source code.

In order to differentiate a theoretical vulnerability from a legitimate danger, one must employ hybrid analysis, which uses both static and dynamic analysis. In short, this means viewing both an application's code and operation. For example, an application using insecure number generation to simulate a coin flip or dice roll would have no impact on security. However, a static analysis tool may flag the use of an insecure random number algorithm as inherently dangerous, highlighting an additional limitation of relying solely on static analysis tools.

Future research should incorporate dynamic analysis techniques to determine the impact of perceived vulnerabilities in practice, rather than the more abstract evaluation performed in this research. For example, tools such as REAPER [11] can be used to monitor and log permissions requests in real-time, providing additional insight into whether permissions are involved by the application's core functionality or by third-party libraries. In addition, Wireshark [36] and mitmproxy [37] can be used for further traffic analysis to identify cleartext transmission of sensitive information. Finally, Drozer [38] can facilitate the inspection of exported components and the use of insecure cryptographic practices, such as insecure or weak random number generators. Incorporating these tools into future research studies would provide a more comprehensive evaluation of vulnerabilities identified through static analysis tools.

Overall, static analysis provides a particularly limited scope of security risks posed by user permissions. MobSF collects data from an application's list of requested permissions and calculates risk based on the amount of data that each permission could potentially access. This is a worst-case scenario that fails to recognize the extent to which each permission is used, whether the permissions are optional, the application's transparency, and the necessity of each permission with regard to the application's functionality. This model is inclined toward false positives. For example, an application can harass a user with surplus notifications, but most reputable applications exhibit restraint. Therefore, the danger of notification permissions that MobSF identifies is seldom realized in practice. Recognizing this limitation, this study focused on permissions that did not contribute to an application's functionality or (legal and transparent) monetization. Future research should also incorporate user experience and usability studies to promote transparency. This could include evaluating application permissions with how clearly they are explained to users and how privacy policies align with application behavior.

However, the scope of our study was still constrained by the limits of static analysis. When developing software, compliance with legal and ethical standards is as indispensable as functionality. At its core, a VPN only requires access to a device's network settings. When performing static analysis, it can be unclear whether a permission is an act of intrusion or a mechanism for compliance. For example, several VPN applications were given access to the host device's storage and web activity. This could be used to violate user privacy or as an emergency measure during a legal investigation. To determine whether either of these extremes applied, one would need to examine the application's functionality and behavior.

6. Conclusion

Across the VPN applications analyzed, there are varying levels of security, with each having areas that could be improved. It is important to note that the results and conclusions made in this study do not reflect the entire range of VPN applications available. On average, the observed applications received a satisfactory 'B' rating from MobSF. None received an exemplary 'A' rating. Although some vulnerabilities are necessary imperfections, others are avoidable. Multiple applications logged sensitive information, which is not advisable. Furthermore, almost 90% of the applications used insecure number generators. Almost all the applications observed used sensitive permissions that were not related to the functionality of the VPN. This included the ability to read and sometimes write to private files as well as the use of cameras and peripherals. Finally, exports were given inadequate protection. Although no applications were observed as exemplifying extremely low risk, these vulnerabilities were shown through static analysis. It is concluded that developers should observe best practices when creating mobile applications and patching vulnerabilities.

Confilict of Interest

The authors declare no conflicts of interest.

References

[1] K. Stouffer, V. Pillitteri, S. Lightman, M. Abrams, and A. Hahn, "Guide to operational technology (OT) security," NIST Special Publication 800-82, Revision 3, 2023. [Online]. Available: https://csrc.nist.gov/pubs/sp/800/82/r3/final [Accessed Mar. 10, 2025]

[2] B. Cruz, "2024 VPN trends, statistics, and consumer opinions," Security.org, 2024. [Online]. Available: https://www.security.org/resources/vpn-consumer-report-annual/ [Accessed Mar. 10, 2025]

[3] M. Ikram, N. Vallina-Rodriguez, S. Seneviratne, M. A. Kaafar, and V. Paxson, "An analysis of the privacy and security risks of android VPN permission-enabled apps," In Proc. 2016 ACM Internet Measurement Conference, Santa Monica, CA, USA, Nov. 14–16, 2016, pp. 355-368.

[4] A. Abraham, "MobSF/Mobile-Security-Framework-MobSF GitHub," 2024. [Online]. Available: https://github.com/MobSF/Mobile-Security-Framework-MobSF [Accessed Mar. 10, 2025]

[5] PRISMA, "Preferred reporting items for systematic reviews and meta-analyses (PRISMA) website," 2020. [Online]. Available: https://www.prisma-statement.org/ [Accessed Mar. 10, 2025]

[6] L. Qiu, Y. Wang, and J. Rubin, "Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe," In Proc. 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, Amsterdam, Netherlands, Jul. 16-21, 2018, pp. 176-186.

[7] CC Editor, "Static code analyzer - glossary," NIST, 2022. [Online]. Available: https://csrc.nist.gov/glossary/term/static_code_analyzer [Accessed Mar. 10, 2025]

[8] J. Klein, "A journey through android app analysis: solutions and open challenges," In Proc. 2021 International Symposium on Advanced Security on Software and Systems, Hong Kong, China, Jun. 7, 2021, pp. 1-8.

[9] T. Sutter, T. Kehrer, M. Rennhard, B. Tellenbach, and J. Klein, "Dynamic security analysis on android: a systematic literature review," *IEEE Access*, vol. 12, pp. 1-1, 2024, https://doi.org/10.1109/access.2024.3390612.

[10] OWASP Foundation, "MASTG-TEST-0003: testing logs for sensitive data," 2024. [Online]. Available: https://mas.owasp.org/MASTG/tests/android/MASVS-STORAGE/MASTG-TEST-0003/ [Accessed Mar. 10, 2025]

[11] M. Diamantaris, E. P. Papadopoulos, E. P. Markatos, S. Ioannidis, and J. Polakis, "REAPER: real-time app analysis for augmenting the android permission system," In Proc. Ninth ACM Conference on Data and Application Security and Privacy, Richardson, TX, USA, Mar. 13, 2019, pp. 37-48.

[12] D. Lazar, H. Chen, X. Wang, and N. Zeldovich, "Why does cryptographic software fail? A case study and open problems," In Proc. 5th Asia-Pacific Workshop on Systems, Beijing, China, Jun. 25-26, 2014, pp. 1-7.

[13] S. Seraj, S. Khodambashi, M. Pavlidis, and N. Polatidis, "MVDroid: an android malicious VPN detector using neural networks," *Neural Computing and Applications*, vol. 35, no. 28, pp. 1-15, 2023, https://doi.org/10.1007/s00521-023-08512-1.

[14] H. Abbas, M. Asim, N. Tariq, T. Baker, and S. Abbas, "Security assessment and evaluation of VPNs: a comprehensive survey," *ACM Computing Surveys*, vol. 55, no. 13s, pp. 1-47, 2023, https://doi.org/10.1145/3579162.

[15] B. Wolford, "What is GDPR, the EU's new data protection law?" GDPR.eu, 2024. [Online]. Available: https://gdpr.eu/what-is-gdpr/ [Accessed Mar. 10, 2025]

[16] International Business Machines Corporation, “IBM QRadar security intelligence platform,” 2025. [Online]. Available: https://www.ibm.com/docs/en/qradar-common?topic=extensions-federal-information-security-modernization-act-fisma [Accessed Mar. 10, 2025]

[17] T. Speed, D. Misev, J. Rittinghouse, and B. Hancock, *Mobile Security: How to Secure, Privatize, and Recover Your Devices: Keep Your Data Secure on the Go*. Birmingham, UK: Packt Publishing Limited, 2013, pp. 25-28.

[18] B. Mishra, A. Jaiswal, N. Kumar, and S. Agrawal, “Privacy protection framework for android,” *IEEE Access*, vol. 10, pp. 7973-7988, 2022, https://doi.org/10.1109/access.2022.3142345

[19] Google, “Android apps on Google Play,” 2024. [Online]. Available: https://play.google.com/store/games?hl=en [Accessed Mar. 10, 2025]

[20] W. Charoenwet, P. Thongtanunam, V. T. Pham, and C. Treude, “An empirical study of static analysis tools for secure code review,” *Software Engineering*, pp. 691-703, 2024, https://doi.org/10.1145/3650212.3680313

[21] State of California Department of Justice, “California consumer privacy act (CCPA),” 2025. [Online]. Available: https://oag.ca.gov/privacy/ccpa [Accessed Mar. 10, 2025]

[22] The MITRE Corporation, “CWE - CWE-330: use of insufficiently random values (4.3).” [Online]. Available: https://cwe.mitre.org/data/definitions/330.html [Accessed Mar. 10, 2025]

[23] OWASP Foundation, “Mobile app cryptography,” 2024. [Online]. Available: https://mas.owasp.org/MASTG/0x04g-Testing-Cryptography/ [Accessed Mar. 10, 2025]

[24] OWASP Foundation, “MASTG-TEST-0016: testing random number generation,” 2024. [Online]. Available: https://mas.owasp.org/MASTG/tests/android/MASVS-CRYPTO/MASTG-TEST-0016 [Accessed Mar. 10, 2025]

[25] OWASP Foundation, “MASTG-TEST-0001: use secure random number generator APIs,” 2024. [Online]. Available: https://mas.owasp.org/MASTG/best-practices/MASTG-BEST-0001 [Accessed Mar. 10, 2025]

[26] OWASP Foundation, “Access control for software security,” 2020. [Online]. Available: https://owasp.org/www-community/Access_Control [Accessed Mar. 10, 2025]

[27] OWASP Foundation, “MASTG-TEST-0024: testing for app permissions,” 2024. [Online]. Available: https://mas.owasp.org/MASTG/tests/android/MASVS-PLATFORM/MASTG-TEST-0024/ [Accessed Jan. 10, 2025]

[28] Google, “Permissions overview,” Android Developer, 2024. [Online]. Available: https://developer.android.com/guide/topics/permissions/overview [Accessed Jan. 10, 2025]

[29] OWASP Foundation, “MASTG-TEST-0029: testing for sensitive functionality exposure through IPC,” 2024. [Online]. Available: https://mas.owasp.org/MASTG/tests/android/MASVS-PLATFORM/MASTG-TEST-0029/ [Accessed Mar. 10, 2025]

[30] M. Ogata, J. Franklin, J. Voas, V. Sritapan, and S. Quirolgico, “Vetting the security of mobile applications,” *National Institute of Standards and Technology*, Special Publication 800-163, Revision 1, 2019, https://doi.org/10.6028/NIST.SP.800-163r1

[31] Android Developers, “Permission-based access control to exported components,” 2024. [Online]. Available: https://developer.android.com/privacy-and-security/risks/access-control-to-exported-components [Accessed Mar. 10, 2025]

[32] Google, “Cleartext communications,” 2024. [Online]. Available: https://developer.android.com/privacy-and-security/risks/cleartext-communications [Accessed Mar. 10, 2025]

[33] D. Dagon, M. Antonakakis, P. Vixie, T. Jinmei, and W. Lee, “Increased DNS forgery resistance through 0x20-bit encoding: security via leet queries,” In Proc. 15th ACM Conference on Computer and Communications Security, Alexandria, VA, USA, Oct. 27-31, 2008, pp. 1-15.

[34] Android Developers, “NetworkSecurityPolicy|API reference,” 2025. [Online]. Available: https://developer.android.com/reference/android/security/NetworkSecurityPolicy [Accessed Mar. 10, 2025]

[35] OWASP, “A02 cryptographic failures - OWASP top 10: 2021,” 2021. [Online]. Available: https://owasp.org/Top10/A02_2021-Cryptographic_Failures/ [Accessed Mar. 10, 2025]

[36] Wireshark, “Wireshark documentation,” 2019. [Online]. Available: https://www.wireshark.org/docs/ [Accessed Mar. 10, 2025]

[37] A. Cortesi, M. Hils, and T. Kriechbaumer, “GitHub - mitmproxy/mitmproxy: an interactive TLS-capable intercepting HTTP proxy for penetration testers and software developers.” [Online]. Available: https://github.com/mitmproxy/mitmproxy [Accessed Mar. 10, 2025]

[38] WithSecure, “Drozer.” [Online]. Available: https://labs.withsecure.com/tools/drozer [Accessed Mar. 10, 2025]