

A STATIC ANALYSIS LANDSCAPE OF ANDROID APPLICATIONS: VULNERABILITY, PERMISSION, AND CODE QUALITY PATTERNS APPLICATIONS

Muhammad Akram Khan^{*1}, Dr. Muhammad Nadeem², Komal Zakir³, Tayyab Hussain⁴,
Dr. Jan Muhammad⁵, Misbah Rani⁶

^{*1,5}Department of Computer Engineering, FICT, BUITEM, Quetta

²Department of CIS HCT, Abu Dhabi UAE

^{3, 4, 6}MS (Computer Science), BUITEMS, Quetta

^{*1}writetoakramkhan@gmail.com, ²mnadeem@hct.ac.ae, ³komal.zakir11@gmail.com,

⁴hussain.tayyab@yahoo.com, ⁵jan.muhammad@buitms.edu.pk, ⁶misbahrani480@gmail.com

DOI: <https://doi.org/10.5281/zenodo.22201416>

Keywords

Android security, static analysis, SonarQube, code smells, technical debt, permission model, exported components, vulnerability analysis, software quality, and empirical software engineering.

Article History

Received: 02 April 2026

Accepted: 12 May 2026

Published: 31 May 2026

Copyright @Author

Corresponding Author: *

Muhammad Akram Khan

Abstract

The Open Distribution Model of Android coupled with the open source software quality test tools has forced the automated Static Analysis as the main scalable approach to characterise Application Level Security and Quality Risk. The Open Distribution Model of Android and the open source software quality test tools have compelled the automated Static Analysis as the main scalable approach to characterise Application Level Security and Quality Risk. This paper introduces an empirical corpus-wide study of 284 SonarQube static-analysis reports, from 263 different Android applications, containing 8,923 individual findings, both combined into two interconnected and audited relational tables and analysed by descriptive, correlational and comparative statistics. The majority of code smells (44.6%), followed by Security Hotspots (28.5%), Vulnerabilities (15.0%) and Bugs (11.9%). If we look at the rated ones, 85.7% of them have a mid or worst rating in SonarQube, with the most common rule in the corpus being “inadequately restricted exported Android component” (930 times), “dangerous permission declaration” (808 times), and “clear text network communication” (296 times). The highly significant correlation between dangerous permission count and the study's composite risk score (Spearman's $\rho = 0.83$, $p < 0.001$) suggests that the dangerous permission count is strongly related to the composite risk score; in contrast, the bug counts and security-relevant findings exhibit weak-to-moderate correlations with each other (Spearman's $\rho \sim 0.2-0.5$), suggesting that reliability and security are significantly distinct aspects of Android application quality in this corpus. The results bring the static analysis literature, which is dominated by work focused on open-source infrastructure, into a consumer-facing Android context, and indicate that permission-footprint auditing and exported-component review are reasonable triage priorities for static analysis-based organisations to address Android application risk on a large scale.

1. INTRODUCTION

1.1 The Growing Attack Surface of Mobile Software

Android is the predominant global mobile OS and its open distribution model is long known as a double edged sword, in that it makes software distribution easier, but also increases the attack surface, especially for the end user [1]. In fact, prior to the emergence of exploits in the wild, early large-scale analysis of the Android platform, including the horizontal study of 1100 popular free apps by [1] showed that static analysis of decompiled bytecode could uncover systemic vulnerabilities such as widespread abuse of device identifiers, deep penetration of third-party advertising libraries, and platform inconsistencies in adherence to the platform's security model. Further measurement research of the Google Play market place, including the PlayDrone crawler reported in [2] demonstrated that the size of the ecosystem (more than one million applications) makes it impossible to obtain all the applications' security through manual review, and thus reinforces automated static analysis as the only economically feasible first line of defence.

1.2 Static Analysis as a Software Quality and Security Discipline

Static Application Security Testing (SAST) tools analyze source code, bytecode or binaries, but do not run the program, and they identify constructs that are known to be insecure, unreliable, or poorly maintainable. One of the most popular static analysis platforms used in academia and industry is SonarQube [3, 4]. It distinguishes between four "canonical" categories of findings: Bugs, Vulnerabilities, Code Smells and Security Hotspots, and quantifies the total remediation effort as the technical debt metaphor proposed by Cunningham [5] for the cost of the shortcuts taken when designing software. SonarQube's rule catalogue covers Java/Kotlin logic defects, XML manifest misconfigurations, and embedded WebView/JavaScript logistical issues, allowing an analysis of a compiled Android application package (APK) to detect all of these types of potential defects in a single report [6].

1.3 The Android Permission Model and Its Empirical Scrutiny

Android's permission system is the most visible security mechanism that restricts an application's access to the sensitive hardware and personal data, as outlined by the platform as a collection of "normal", "dangerous", and "signature"-level protections that block access to specific APIs [7]. There have been many prior works, including those by Felt et al. [8] who were among the first to rigorously show that many applications request more privileges than they are needed to implement, an phenomenon they dubbed over-privilege, and established the now-familiar approach of mapping API calls to the permissions that guard them. Other research has focused on inter-component communication [9], advertising-library permission inheritance [10] and the long-term evolution of permission requests over application version updates. There is also a more limited, but complementary, sort of scrutiny: static analysers like SonarQube are not concerned whether a requested permission is required for the app's advertised functionality, but rather whether a dangerous-permission declaration exists, as this is an irreversible increase in the app's privilege footprint and thus its potential for harm if it is compromised.

1.4 Component Exposure and Inter-Process Communication Risk

The second big theme in the Android security research is exposure of application components such as Activities, Services, Broadcast Receivers, and Content Providers via Android's Intent-based inter-process communication (IPC) mechanism to other applications running in the same device. Explicitly exported by the android:exported attribute in the manifest or implicitly by an intent-filter, it becomes reachable by any other installed app or, in absence of an intent-filter permission check, by malicious ones too [11, 12, 13]. The incomplete reasoning of export decisions has been identified as a recurring and exploitable weakness in production application and software development kits (SDK) analyses [14-19] time and again. What this paper will demonstrate is that

this is exactly the class of weakness that is the most often violated by static analysis checking in the corpus examined in this paper: exported components without a restrictive permission.

1.5 Insecure Transport and Cryptographic Practice

In addition to permissions and component visibility, empirical Android security research has many times found that insecure network transmission and cryptographic misuse are common, easily detected issues. As shown by Fischer et al. [20], insecure code snippets found in public programming forums spread at scale to production Android apps; Egele et al. [21] discovered that many Android programs improperly use cryptographic APIs, which significantly undermines the confidentiality assurances. Static analysers can identify these types of defect by marking clear-text application-layer protocols (such as HTTP rather than HTTPS) as security-sensitive because a clear-text transport is easily intercepted on a common or hostile network.

1.6 Standardised Vulnerability Taxonomies

The security community uses standardized weakness taxonomies to enable inter-tool, inter-language and inter-organisation comparisons of static-analysis results. Managed by the MITRE corporation, the Common Weakness Enumeration (CWE) supplies a hierarchical list of more than six hundred generic software and hardware weakness types, and is regularly summarized into a "Top 25 Most Dangerous Software Weaknesses" list of weakness categories that are most often found to be associated with real-world exploitation [22, 23, 24]. OWASP (Open Worldwide Application Security Project) has released a free, mobile-focused risk enumeration—OWASP Mobile Top 10—which was most recently updated in 2024 to account for mobile-specific threats, including inadequate cryptography, improper credential usage and inadequate supply-chain security [25]. Each finding in the dataset studied in this paper is passed through from CWE metadata described in

the rules used by SonarQube, so that individual findings can be linked back to the more general, tool-independent taxonomies.

1.7 Code Quality, Technical Debt, and the Maintainability Dimension

Static analysis is not only defined by the security axis; there is another that can be reduced to the code smell and technical debt metaphors, and it is internal quality of software. Fowler and Beck's "catalogue of bad smells" [26] and Brown et al.'s "AntiPatterns" catalogue [27] laid the groundwork for vocabulary that has been translated into a multitude of rules to be automatically detected by the modern static analysers (e.g., SonarQube). Large scale empirical studies of code smells in open-source repositories, such as those by Palomba et al. [28] on the diffuseness and maintainability impact of code smells, Taibi et al. on developer perceptions of code smells [29], Saarimaki et al. on the diffuseness of technical debt items [30], Lenarduzzi et al. on the fault- and change-proneness of SonarQube issues [31] and Digkas et al. on how Apache developers repay technical debt [32] and Nocera et al. on how open-source projects configure SonarQube Cloud in practice [33] all find that the number of code smells is many times higher than the number of bugs or vulnerabilities in absolute terms, and that the relationship between an issue's SonarQube assigned type and its actual empirical fault-proneness is weaker and more inconsistent than suggested by the tool's labelling. This body of work directly influences one of the analytical questions, which is explored in this paper, namely, whether bug counts and security risk are correlated within a single corpus of Android software, or whether, as the open-source literature would suggest, they are largely independent dimensions of software quality.

1.8 Positioning of the Present Study

Existing prior empirical studies on Android security have focused on either a large-N analysis of hundreds of thousands of Google play applications, such as in [2], [10], or a small-N, manual analysis of a few apps, such as in [12]. Comparatively little published work applies a full-

spectrum, multi-category static analysis engine, like SonarQube, that reports bugs, vulnerabilities, security hotspots, code smells, and permission usage in a single, internally-consistent scoring system to a collection of real, consumer-facing Android applications with a broad spectrum of app categories and code base sizes. This paper fills this void on the basis of a carefully selected dataset that contains 284 analysis runs on 263 different Android apps and 8,923 individual findings, reporting descriptive, correlational and comparative patterns along the dimensions of the vulnerability, permission and code-quality of the corpus.

2. LITERATURE REVIEW

2.1 Foundational Studies of Android Application Security

Android application security was first examined as an independent research field in 2011, when Enck et al. performed a horizontal analysis of 1,100 applications using the dex2jar decompiler to demonstrate the feasibility of static bytecode analysis, a scalable way for characterising the security of the entire Android ecosystem without the need to obtain source code [1]. Their side work, TaintDroid, pushed this research into dynamic information-flow tracking and shows that static analysis (which the present study applies) is commonly reporting potential, not realised, risk [34] as the static findings (declared permissions) and dynamic findings (actual sensitive-data flows) often differ. Viennot et al.'s PlayDrone measurement study then extended this approach to more than 1 million Google Play applications to uncover patterns in library reuse, application duplication, and mismanagement of authentication-keys, which are not apparent on the scale of single application audits [2].

2.2 The Permission Model: Over-Privilege and Risk Signalling

Felt et al. [8] have conducted the most cited empirical research on Android permission over-privilege stating that around one-third of a sample of 940 applications requested permissions that their code never actually used, and that much of

this is due to inadequate or ambiguous platform API documentation, rather than malicious intent. Chin et al. took this research further to inter-application communication by demonstrating that intention-based component interaction needs to be considered when using permission checks to characterise exposure [9]. Chia et al. conducted a study of how the number of application permissions and community-sourced risk signals correlate with expert-assessed risk for a large set of applications in the Android Market, a study that confirms that neither the number of permissions an application is requesting nor the user rating is reliable factors that will predict expert-assessed risk, and thus, further research is needed to discover more structured, tool-derived indicators of risk that can be used in this area, such as the dangerous-permission rule that SonarQube provides [10]. Together, this literature provides a foundation that a raw permission count is a coarse but non-trivial proxy for risk, which is directly tested in Section 4 with the composite of risk scores from the present data set.

2.3 Exported Components and Inter-Process Communication Vulnerabilities

There is also a parallel and growing body of work looking at the security implications of IPC in Android using Intent. In summary, the three patterns that have been documented in industry security research, which occur when a component receives and passes attacker-influenced data without proper validation [12, 18], are intent redirection, intent spoofing and intent interception. The exploits can be traced back to manifest-level misconfigurations (an implicit intent-filter without an explicit `android:exported="false"`, overly permissive `protectionLevel` values on custom permissions) and are documented in SecureFlag's vulnerability knowledge base. The real-world, supply-chain aspect of this risk class was exemplified by a disclosure made by Microsoft Security Response Centre (SRC) in 2026, when a third party software development kit (SDK) was found to be used in many production apps that inadvertently contained an exported activity, making them

vulnerable to an intent-redirection attack in every app that used the SDK (including several mobile wallet apps) [14]. The present finding matches exactly the top finding in the 8,923 finding set, namely "Exported component access should be restricted with appropriate permissions."

2.4 Insecure Transport, Cryptography, and Code-Reuse Risk

Empirical studies of the security problems found on Android devices have shown that many security problems come from the unsuspecting reuse of publicly available code by developers rather than intentional mistakes. A few recent large-scale studies have investigated how code snippets found in the Stack Overflow repository relate to their usage in real-world Android apps. For instance, in the study by Fischer et al. [20] it was observed that a significant percentage of security-related code snippets that are copy-pasted from the repository are also vulnerable to security issues, and that these vulnerabilities are then carried over unchanged in the shipped applications. Although correct and standard-library implementations are available, the complementary study by Egele et al. of API usage in a large Android corpus revealed that the APIs were widely misused, including the use of constant encryption keys, non-random IVs, and weak or deprecated ciphers [21]. The present dataset has 296 instances of the clear-text-protocol rule, which is a part of the overall problem.

2.5 Standardised Weakness and Risk Taxonomies (CWE, CVSS-Adjacent Scoring, and OWASP)

For comparability between static analysis tools, vulnerability databases and organisations, there is a need for shared classification schemes. The Common Weakness Enumeration, under the auspices of the U.S. Cybersecurity and Infrastructure Security Agency (CISA) and operated by the MITRE Corporation, is the most prevalent hierarchical organization of generic types of software weaknesses, distinction being made between a weakness (a class of flaw, like CWE-89 SQL Injection) and a vulnerability (a particular, instantiated occurrence of that flaw in

a particular product) [22], [23]. Most of the Security Hotspot and Vulnerability findings in SonarQube are tagged with one or more CWE identifiers and the tags field of the current dataset is derived from this mapping. Some of the more common tag combinations are "android", "android, cwe", and "android, cwe, privacy". The OWASP Mobile Top 10 2024 is a free, risk prioritized (not only taxonomic) list of top ten security issues that are mobile specific and covers most of the following categories: credential handling, supply-chain integrity, authentication, input validation, communication security, privacy controls, binary protection, configuration, data storage, and cryptography [25]. The most common OWASP categories found in the current corpus are mostly directly mapped to the three rule categories that are most common to trigger in the present corpus: OWASP M6: Inadequate Privacy Controls, OWASP M5: Insecure Communication, and OWASP M8: Security Misconfiguration.

2.6 Code Smells, Technical Debt, and the Maintainability Literature

The technical debt metaphor, coined by Ward Cunningham in a 1992 experience report describing the WyCash portfolio management system, frames expedient but structurally compromising design and coding decisions as a form of financial obligation that accrues interest ~ in the form of increased future maintenance cost - until it is explicitly repaid through refactoring [5]. Fowler and Beck's subsequent catalogue of named, refactorable code smells, and the earlier AntiPatterns catalogue of Brown et al., together supplied the taxonomic vocabulary that contemporary static analysers such as SonarQube encode as automatically detectable maintainability rules [26], [27]. A substantial empirical literature has since interrogated whether SonarQube-flagged issues actually predict the outcomes the technical-debt metaphor implies. Palomba et al.'s large-scale investigation of code-smell diffuseness found that smells are pervasive across open-source Java systems and are associated with, though not perfectly predictive of, reduced maintainability

[28]. Taibi et al.'s replicated developer-perception study found that developers do not uniformly regard all tool-flagged smells as harmful, with perceived severity varying substantially by smell type [29]. Saarimaki et al. and Lenarduzzi et al., analysing 33 Apache Java projects, found that SonarQube-labelled "bugs" are only weakly associated with actual fault-proneness, while a disproportionate share of genuinely fault-associated violations are labelled as code smells rather than bugs ~ a mislabelling that complicates any at-face-value interpretation of SonarQube's own categorical taxonomy [30], [31]. Digkas et al.'s study of technical-debt repayment behaviour across the Apache ecosystem found that developers preferentially fix issues that are cheap to resolve rather than those with the highest estimated debt-interest, a finding with direct implications for how the composite risk score in Section 3.6 of the present paper should be interpreted as a prioritisation signal rather than a ground-truth severity measure [32]. Most recently, Nocera et al.'s mining study of GitHub projects integrated with SonarQube Cloud found that a majority of projects rely on default quality-gate configurations rather than customising thresholds to their own risk tolerance, suggesting that the rating distributions observed in tool-generated datasets such as the present one partly reflect default tool configuration as much as genuine, deliberately calibrated project-specific quality targets [33].

2.7 Static Analysis Tool Adoption and Practitioner Perception

A separate line of research looks at the usage of ~ or lack of ~ static analysers by the practitioners. In an interview study of 20 professional software developers, Johnson et al. discovered that although all respondents believed static analysis to be useful in the abstract, the most prevalent obstacles to persistent use were false positive and issues of the context in which the warning is presented [35]. A similar study by Vassallo et al. revealed that the developers' desire to fix a warning from the static analysis tool is much more dependent upon the other information contained

in the context of the development process (such as whether the code under the warning is on a critical path) than on the nominal severity of the warning itself [3]. This practitioner-centred view is echoed by the work of Sadowski et al. on static analysis tooling at Google, where they distinguish the usefulness of tools, not only based on their raw detection accuracy, but also on how they are integrated and presented [4]. Overall, these results serve as a warning that raw numbers of raw SonarQube findings, collected as in the current study, should not be interpreted as a straightforward, unmediated assessment of an application's security posture, but as an organized signal which needs to be further processed and translated into remediated risk for the application in question, requiring further triage, contextualisation tasks, and, in many organizations, active developer interaction.

2.8 Synthesis and Identified Gap

The literature reviewed provides three points of consensus that directly motivate the design of the present study. First, permission-based and component-exposure risk-indicators are individually coarse, but the actual, and repeatedly replicated empirical signals from these indicators are valid and are about application risk [8, 10, 12]. Second, this paper's own correlation analysis (in Section 4.3) contradicts the more prevalent paradigm of a single language, open-source-infrastructure corpus of projects (chiefly Apache Java projects) to consider code smells and reliability defects (bugs) as a single underlying "badness" factor, but examines a cross category (Android), code specific corpus. Third, standardised taxonomies (CWE, OWASP Mobile Top 10) offer the vocabulary to connect tool-specific rule identifiers with more general, cross-tool security conversation, as this paper explicitly uses the CWE tags that were already provided in the source dataset to make that connection [22, 25]. This paper's research gap is empirical and dataset-level in that the literatures on permissions use and code-smell and transport-security findings are mature, but few published studies are available that combine all three findings in a single, multi-

category static analysis corpus of real, consumer Android applications such as those assembled here.

3. METHODOLOGY

3.1 Overview

The study adopts an observational, retrospective, quantitative research design that is suitable for secondary analysis of an existing measurement

corpus, in line with the software engineering empirical research guidance [36, 37]. The data used in this paper was not collected, but rather, a set of 284 Android application analysis runs from a previous data collection effort with SonarQube were consolidated, audited, and statistically analyzed. As shown in Figure 1, there are eight stages in the methodological pipeline described in Sections 3.2–3.7.



Figure 1. The consolidated Android static-analysis dataset was built and analyzed using 8-stage methodology pipeline.

3.2 Corpus Acquisition

The underlying corpus comes from a collection of 284 Android application packages subjected to static analysis with SonarQube and analyzed and exported as structured PDF analysis reports. As a result of its strong presence in the open-source and industrial static-analysis research, its multi-language rule coverage (Java, Kotlin, XML, JavaScript, CSS, and HTML are all included in the

corpus under analysis), and its uniform classification of findings into Bugs, Vulnerabilities, Code Smells and Security Hotspots [3], [4], SonarQube was chosen as the analysis engine of the source corpus. Note that there are multiple analysis runs for the same application (app_id), each using a different branch, version, or product variant (e.g., there are nine separate runs of analysis for the product

named "Video", corresponding to different distinct builds of Player, Editor, and Maker), so the source_file is the reliable one-to-one join key between summary-level and finding-level records of the dataset documentation, not app_id.

3.3 PDF Report Parsing and Structured Extraction

Two types of information were extracted from each of the SonarQube PDF reports: (i) summary-level metrics ~ lines of code, size rating, quality gate outcome, reliability/security/maintainability ratings, and aggregate counts of bugs, vulnerabilities, code smells, and security hotspots; and (ii) finding-level records ~ one row per individual bug, vulnerability, code smell, and security hotspot, including its type, rule identifier and name, the associated permission (if any), source language, CWE-derived tags, file path, line number, and descriptive message. This extraction resulted in 5 intermediate tables (summary.csv, master_bugs.csv, master_issues.csv, master_permissions.csv, and master_vulnerabilities.csv) that will serve as the raw data for the consolidation step described in the next step.

3.4 Subset Reconciliation

Before consolidation, the four tables at the finding level were each confirmed to be a 100% match on all shared columns with the master table, e.g., master_issues.csv contained all of the rows from master_bugs.csv plus all of the rows from master_vulnerabilities.csv and master_permissions.csv. This verification step was similar in spirit to the data cleaning and validation practice recommended in a guide to systematic software engineering data collection [37] which meant that the four tables could be safely stored in a single canonical findings table (Findings_Master) rather than the four partially-overlapping tables that would lead to potential double-counting in subsequent statistical analysis.

3.5 Schema Consolidation

These two tables were merged into a single reconciled data table. App_Master (284 rows, 34

columns): One row per analysis run per app, with all the original summary.csv fields included, as well as counts recomputed right from the finding-level data as a cross-check with the finding-level data in the source PDF. Findings_Master (8,923 rows, 14 columns) contains one row for each finding, with an additional column for a finding that is associated with a dangerous-permission declaration, called permission_identifier. The two tables have source_file as their join key, allowing analyses to seamlessly switch between the application level and the finding level, as needed.

3.6 Data-Quality Auditing and Feature Derivation

In order to be consistent with recommended practice for transparent secondary data analysis [37] and [36] three data quality problems were identified during construction of the dataset and explicitly identified in the accompanying documentation. In the first place three rows in the original summary.csv were discovered to be exact full row copies of other rows, and these were deleted. Second, 87 of 284 application records lack reliability, security and maintainability ratings in the original summary data, and 74 lack bug/vulnerability/smell counts in the original summary data; these are coded as "missing (not available)" instead of imputing a zero. Third, the number of bug-type rows that can be extracted from the source PDF differs significantly from the number of bugs declared in the summary (for instance, in one application, there were 758 bugs declared in the summary, but only 43 bugs can be extracted from the source PDF); this is due to the way the source PDF is being paginated, not an error in the consolidated dataset, and is indicated through a boolean declared_vs_extracted_bug_mismatch indicator. For each application record, these three audited features were combined with two derived ones: dangerous_permission_count, which was a straight count of the distinct dangerous Android permissions that had been declared in the application; and composite_risk_score, which was a linear combination of the three features,

weighted by the relative importance of each feature, defined as

$$\text{composite_risk_score} = 3 \times \text{vulnerabilities} + 2 \times \text{security_hotspots} + 1 \times \text{bugs} + 0.25 \times \text{code_smells}$$

It is a rhetorical, heuristic rather than validated, industry-standard scoring model (such as the Common Vulnerability Scoring System), and it is meant to be a single ordinal ranking signal for comparison across different applications, not an absolute measure of how easy an exploit would be to create; so the behaviour of this score is described in terms of the relative, rank-based behaviour (Sections 4.4-4.6) rather than its absolute value.

3.7 Statistical Analysis Procedure

All statistical analysis was done in Python 3 with the pandas, NumPy, and SciPy scientific computing libraries. All fields were included in both tables: frequency counts, medians, and interquartile ranges were calculated for each. Since the continuous measures in this dataset (lines of code, number of findings, number of permissions, and composite risk score) are all right-skewed, with a small number of very large applications dominating the upper tail, association between pairs of continuous measures was quantified using the Spearman's rank correlation coefficient, which has been used in the software engineering literature as the standard measure for non-normally distributed measures [38]. Results were presented as medians and interquartile ranges in addition to means, so as to provide group-level comparisons (such as composite risk score by size-rating strata), which are robust against the influence of outlying, very large applications. Unless otherwise indicated, all findings were reported at $p < 0.001$ (two-tailed).

3.8 Threats to Validity

There are three limitations which directly impact the interpretation of the results that follow. First, external validity is limited by the application sample used in the application, which is not a

random or stratified sample of the catalogue of the Google Play store, and may over- or under-represent certain application categories; the results are therefore not representative of the Android ecosystem as a whole, but should be read as representative of this application sample. As mentioned, similar large scale measurement studies of the Android ecosystem have been performed in the past [2] and [1]. Second, the construct validity of the results reported here are limited to the extent that the SonarQube rule design, which identifies code for manual review as a potential risk, will not, by the rule's own definition, confirm any vulnerability [39]; throughout this analysis, therefore, hotspot counts are used as a risk-indicative signal rather than a risk-confirming signal. Third, the eight reports with reported and extracted bug counts that differ have a documented non-random undercount of bug counts for the specified application(s); all reported rows are distinct and are explicitly referenced wherever they materially impact upon an aggregate statistic reported below.

4. RESULTS AND DISCUSSION

4.1 Corpus Composition

This is a consolidated corpus of 284 application-level analysis records, representing 263 different application identifiers and 8,923 individual findings. Out of 284 analysis runs, 94 (33.1%) contain no findings in Findings_Master, which is due to either a clean scan or the lack of a PDF to be parsed into row-level findings, again a distinction that cannot be made retroactively and is shown transparently in the findings_detail_available Boolean: 94 of 284 reports have that value set to False. Lines of code data were available for 247 applications, and the application size is heavily skewed to the right, with a median application size of 52,499 lines of code, a mean of 88,596 lines of code, and a maximum of 1,376,765 lines of code – the right skew is typical of consumer application codebases due to the documented heterogeneity of the codebase.

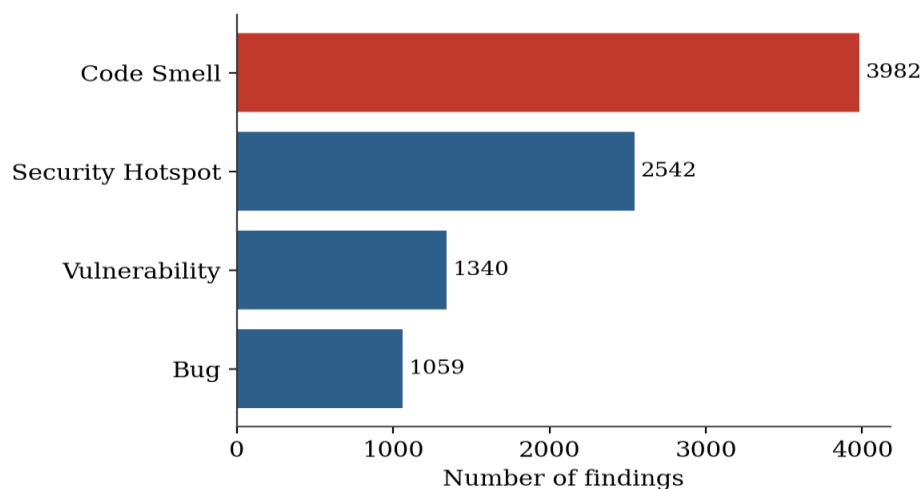


Figure 2. The numbers of the 8,923 findings by their four canonical types in SonarQube (Code Smell, Security Hotspot, Vulnerability, Bug).

4.2 Distribution of Finding Types

Across the 8,923 findings in the corpus, Code Smells are the most numerous category (3,982 findings, 44.6% of the total), followed by Security Hotspots (2,542; 28.5%), Vulnerabilities (1,340; 15.0%), and Bugs (1,059; 11.9%). This ordering is similar to the overall open-source SonarQube literature, which shows that, across different corpora, maintainability oriented Code Smell findings consistently outnumber reliability oriented Bug findings by a large margin [30], [31], [28] ~ a pattern reproduced by this Android-specific, cross-language collection, but one for which there is a significant variation in application domain and source language mix compared to the Apache-Java corpora that dominate the previous literature. Interestingly, the relatively high percentage of the Security Hotspot findings (28.5% of all findings) is a feature particular to the Android context as opposed to the general open source studies, and is significantly explained by two families of rules that are specific to Android and which are known as Security Hotspot by SonarQube and are not confirmed Vulnerabilities pending manual review [39] as mentioned in Section 4.5: exported-component exposure, and dangerous-permission declaration. A non-trivial proportion of Code Smell findings are instead

from web-content accessibility rules ~ rules whose tag field value is "accessibility, wcag2-a" 425 times in the corpus, showing that a significant proportion of the WebView-hosted or hybrid Web content in this corpus is not written against current accessibility or markup standards.

4.3 Security Rating Distribution

The distribution of the security ratings assigned by SonarQube to 197 application records shows that 85 applications (43.1%) had the lowest rating (E), 84 (42.6%) had a C rating, and 28 (14.2%) had the highest rating (A). Because of how SonarQube rates applications, there was no application in the corpus with a B (very good) or D (good) security rating; SonarQube's security rating is based on the number of open vulnerabilities rather than a continuous scale. The figure of 85.7% of all rated applications that are both worst rated (E) and mid-rated (C) suggests that an application with a security problem that has not been resolved is more common than is an application with an exceptional security problem; that is, in this corpus, the non-exceptional existence of applications with systemic security issues is more common than the exceptional security issues of individual applications.

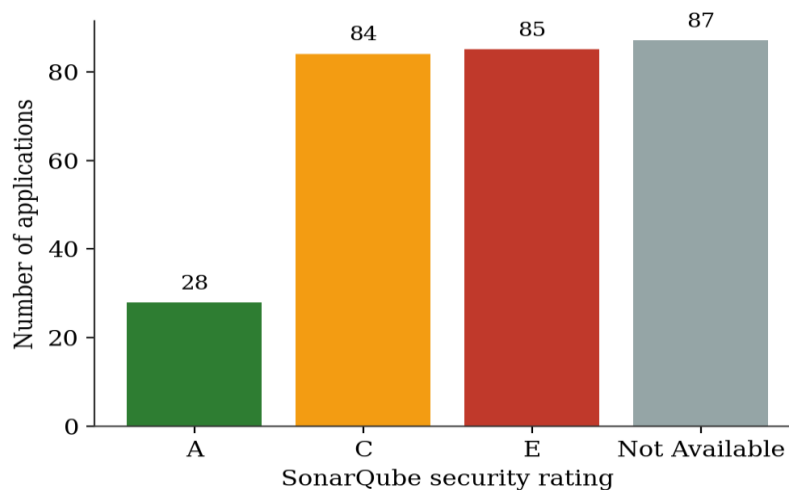


Figure 3. Distribution of SonarQube security ratings across the 284 application analysis records (87 records have no assigned rating).

By contrast, all 197 non-missing records have maintainability_rating A, with 284 records have coverage_pct 0.0, which is a well-known fact in the design of unit test coverage: it is almost never instrumented into a post-hoc, black-box static analysis of compiled Android application packages.

4.4 The Most Frequently Triggered Rules

For the entire corpus, Figure 4 shows the top 10 rules triggered by SonarQube. The top three findings are "Exported component access should be restricted with appropriate permissions" (930), "Requesting dangerous Android permissions is security-sensitive" (808), "Variables should be declared with 'let' or 'const'" (416, a JavaScript/ES6 modernisation rule [41]), "Receiving intents is security-sensitive" (309), and "Using clear-text protocols is security-sensitive" (296). The prevalence of the first, second, and fourth rules, which are all directly related to Android's Intent-based inter-process communication and

permission model, validates the industry security literature's focus on exported-component exposure as the most widely found, tool detectable weakness class in production Android applications [12][13][14][15]. The "Allowing application backups is security-sensitive" (191 occurrences) also indicates a very well known but often underreported Android manifest misconfiguration: `android:allowBackup="true"` that can lead to an application's private data directory being exposed to extraction on unpatched or rooted devices through Android Debug Bridge backup. Other than the top ten, the most common (64) use of "Cognitive Complexity of functions should not be too high" performs a complexity computation that uses a control-flow-graph-based complexity measure that is derived from McCabe's original cyclomatic complexity function [42] to identify functions with a branching structure that could make them hard to understand and modify safely.

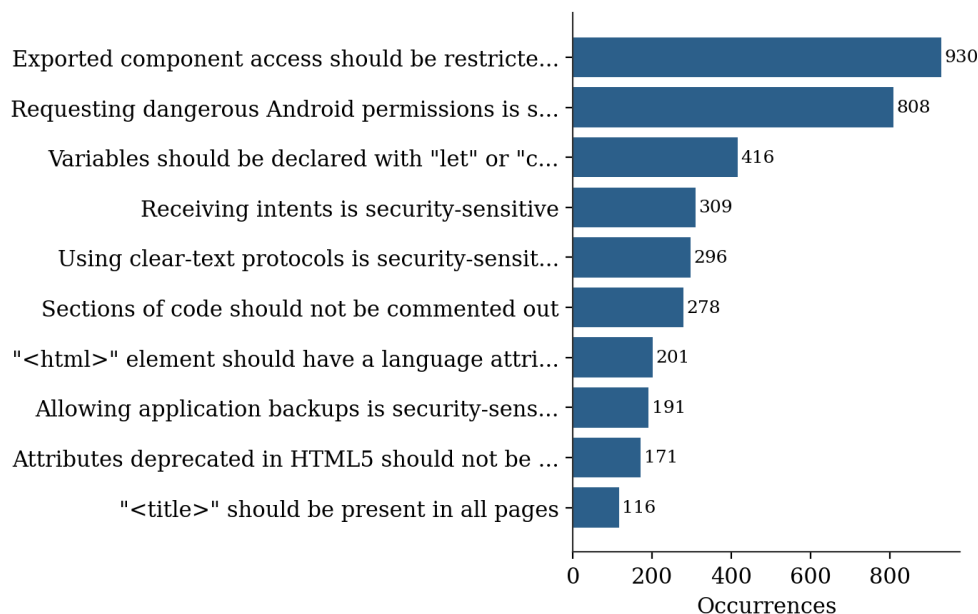


Figure 4. Top 10 rules across the 8,923 finding corpus that are triggered most frequently.

4.5 Dangerous Permission Landscape

The results for each permission_identifier are focused primarily on storage and on permissions related to sensors. The top two most requested dangerous permissions are WRITE_EXTERNAL_STORAGE (149), and READ_EXTERNAL_STORAGE (136), followed by CAMERA (103), RECORD_AUDIO (81), ACCESS_COARSE_LOCATION (60), and ACCESS_FINE_LOCATION (58). The 12 most common permissions are visualised in figure 5. The distribution of the number of different dangerous permissions per application is right-

skewed with a mean of 2.81 (median 2, IQR 0-5, max 19) dangerous permissions per application. The pattern is identical to the one used in the Android ecosystem by Felt et al., which is a long tail of heavily over-provisioned applications, with a smaller, modal number of conservatively over-provisioned applications at the head, though the present dataset measures the number of declared dangerous permissions, not the number of API calls used to trigger over-privilege as in Stowaway, and the two are not necessarily equivalent operationalisations of the same underlying construct.

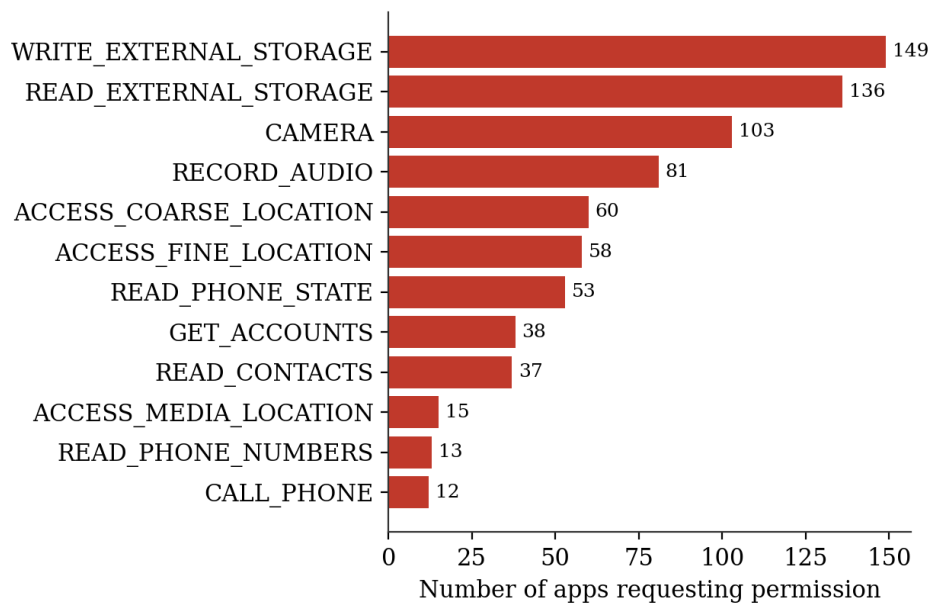


Figure 5. The top 12 most commonly granted Android permissions for the corpus of apps.

4.6 Relationships Among Risk-Relevant Metrics

Spearman correlation among the main static analysis metrics for applications are shown in Table I and Figure 6. The composite risk score is more highly correlated with the other risk measures than with raw bugs or lines_of_code; in particular, it is highly correlated with dangerous_permission_count ($\rho = 0.83$), security_hotspots ($\rho = 0.77$), and vulnerabilities ($\rho = 0.70$), and less correlated with raw bugs ($\rho = 0.49$) and lines_of_code ($\rho = 0.20$). Not surprisingly, composite_risk_score is, by design, an increasing linear function of vulnerabilities and

security_hotspots; the interesting finding is rather the high degree of correlation between the score and dangerous_permission_count ($\rho = 0.83$), a variable not used in the formula of the composite risk score. This suggests that the permission scope variable, as it was used in [8] and [10] as a theoretical argument for the dangerous-permission footprint is a valid, non-tautological proxy for overall application risk within this corpus, including the theoretical motivations, if not the exact empirical approaches, of the over-privilege framework of Felt et al. [8] and the risk-signal study of Chia et al. [10].

TABLE I. SPEARMAN RANK CORRELATIONS WITH COMPOSITE RISK SCORE (C.R.S.)

Metric	Spearman's rho	p-value
Dangerous permission count	0.826	< 0.001
Security hotspots	0.772	< 0.001
Vulnerabilities	0.703	< 0.001
Bugs	0.494	< 0.001
Code smells	0.462	< 0.001

Metric	Spearman's rho	p-value
Lines of code	0.202	0.001

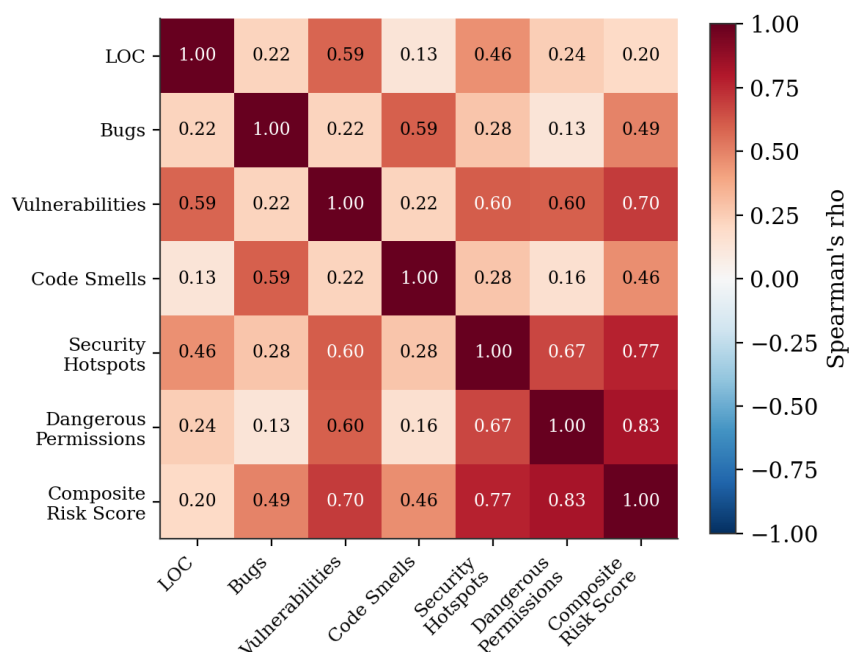


Figure 6. Spearman rank correlation matrix for some of the primary app-level static-analysis metrics.

Importantly, there was only a moderate correlation between bugs and code_smells ($\rho = 0.59$), and only weak-to-moderate correlations between bugs and security-relevant measurements (e.g. vulnerabilities, $\rho = 0.22$) as well as between code_smells and the same measurements (e.g. vulnerabilities, $\rho = 0.22$). The near-independence of the reliability-oriented findings (bugs) and the security-oriented findings (vulnerabilities, security hotspots, dangerous permissions) directly reproduces in an Android-specific corpus the central finding from the open-source literature on SonarQube that the two types of findings are not just the same underlying "badness" factor, but are in fact quite different: code quality and security risk can be considered as separate dimensions by themselves [30], [31]. Realistically, this means that a clean bug profile

does not imply a low security risk and vice versa ~ these two aspects must be treated separately.

4.7 Risk Score by Application Size Class

There is a distribution of composite risk score by the 5 size rating strata of SonarQube (XS, S, M, L, XL), as illustrated in figure 7. As one might expect, there is a general trend of increasing median risk score with the size class of the codebase, ranging from 7.0 (XS, $n = 1$), through 14.5 (S, $n = 22$), 24.6 (M, $n = 176$), 67.0 (L, $n = 40$), up to 229.0 (XL, $n = 7$). However, the size-rating variable in SonarQube is analysed lines of code and doesn't directly reflect the level of security relevant findings as well as the purely stylistic findings, and as shown in Figure 7, the positions of two codebases of the same size in this corpus, but

different sizes on the underlying distribution, can differ significantly.

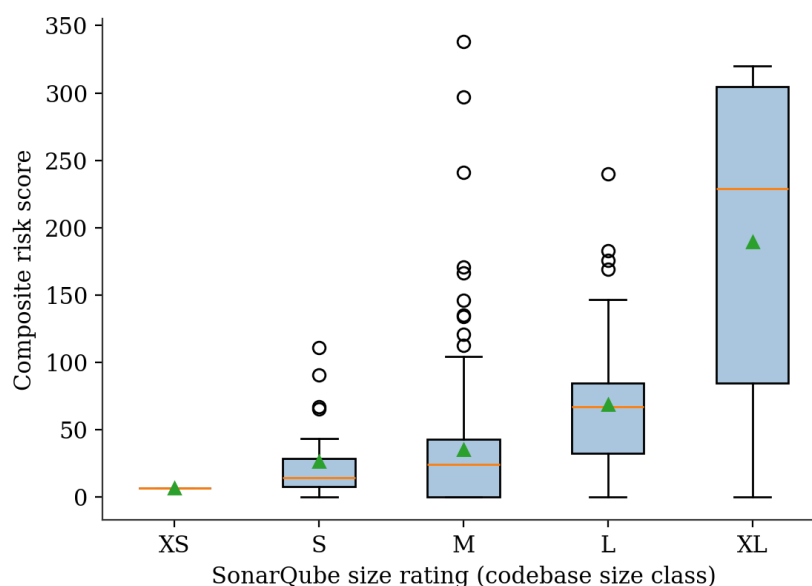


Figure 7. Distribution of risk score by SonarQube size-rating strata (XS, S, M, L, XL).

4.8 Permission Count Versus Composite Risk Score

Figure 8 displays a scatter plot of `dangerous_permission_count` vs. `composite_risk_score`, including a linear regression trendline. The relationship is clearly monotonic and, as described in Section 4.6, statistically significant ($\rho = 0.83$, $p < 0.001$), but also shows some meaningful scatter for applications that have a high number of

permissions, and a still relatively small number of identified vulnerabilities or hotspots. This scatter is aligned with the practitioner-adoption literature's warnings about considering any individual static-analysis-derived metric as a fully determinate risk predictor [35, 3]: The number of permissions does not have a good to perfect correlation with the composite risk ranking of the corpus.

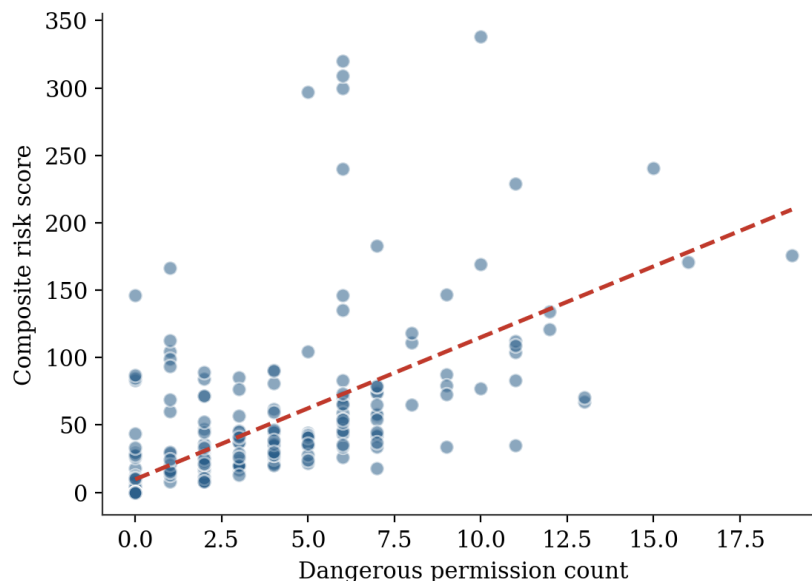


Figure 8. Overlay of a linear trend (Spearman's $\rho = 0.83$, $p < 0.001$) on top of the dangerous permission count versus the composite risk score.

4.9 Discussion: Synthesis Across Vulnerability, Permission and Quality

The findings above highlight three general themes. At this granularity of analysis, the findings that are most likely to trigger are Android-specific inter-process-communication issues (exported components, dangerous permissions, intent handling), supporting an empirical argument that these aspects of Android are areas of risk that have hitherto been reported primarily in the context of industry security research into specific applications or SDKs, and not through a corpus-wide quantitative analysis (as in this work). Second, at the metric level, dangerous permission count serves as a strong non-tautological proxy to overall composite risk, and thus continues to provide the security teams at scale with a quantitative, corpus-level aid to the qualitative over-privilege concerns raised more than ten years ago by Felt et al. [8] that have been revisited since by Chia et al. [10] and others, and that continue to make it a viable triage heuristic for security teams at scale. Thirdly, at the software-quality level, the significant independence between reliability findings (bugs) and security findings

(vulnerabilities, hotspots) reported on this Android corpus extends a previous report that found this separation within single-language open-source repositories of infrastructure software [30], [31], [28] to a different application domain and software language mix, thereby confirming the broader methodological observation that code-quality review and security review are complementary, non-substitutable static-analysis tasks that should be resourced and prioritised separately (and not as the signs of a single favourable outcome).

5. CONCLUSION

This paper showed a landscape of 284 reports of Android application analysis, 263 of which were distinct applications and 8,923 findings, drawn from the SonarQube output and summarized by two interconnected and audited tables and analysed using descriptive, correlational and comparative statistics. The majority of findings were Code Smells (44.6%), followed by the Security Hotspots (28.5%), Vulnerabilities (15.0%), and Bugs (11.9%). Overall, over 85% of rated applications (85.7%) received a mid-to-low

grade rating on security from the SonarQube tool, and the most common rule at the corpus level was about insufficiently restricted exported Android components, which is consistent with the results of individual security scans in the industry. The main results of the study showed that dangerous permission count was a strong, but not tautological, correlate of the study's composite risk score (Spearman's $\rho = 0.83$), while the counts of bugs and security-relevant findings revealed only weak-to-moderate correlations with each other, suggesting that reliability and security are rather distinct aspects of Android application quality in this dataset. Together these results argue for the adoption of permission-footprint auditing and exported-component review as two sensible, high-reward priorities to consider when using static analysis output to triage the risk of Android applications, and warn against assuming that means a clean security profile as well.

6. FUTURE WORK

There are a number of directions in which the current analysis could be extended. First, the underlying corpus is a snapshot of the applications' code in a single point in time, so cannot address changes in the patterns of vulnerability, permission, and code-smell over successive application versions; a longitudinal extension of this corpus, similar to the technical-debt repayment study conducted by Digkas et al. [32] would answer the question of whether the risk pattern revealed here is a stable characteristic of an individual application, or a transient artefact of a particular release. Second, the composite risk score used in the current study has not been validated against real-world usage (e.g., applications with any publicly disclosed vulnerability history), and future efforts might involve rating other composite schemes against known ground truth deployments to see if the current ranking is similar to the actual experience of the attackers, bringing the scoring literature from the CVSS realm into the Android static-analysis sphere. Third, the research results of the source dataset would be provided by a single static analysis tool, so cross validating a subset of the

corpus against a second independent static tool, or against dynamic/hybrid analysis techniques like the TaintDroid-style information-flow tracking [34], would help identify application risk versus tool-specific rule idiosyncrasies. Lastly, 94 of 284 analysis runs within the current corpus produced no row-level findings from the PDF parsing, but only because the scans were not confirmed clean, thus future data-engineering efforts to recover or re-analyse the underlying source reports would greatly enhance the completeness and, in turn, the statistical power of any subsequent analysis conducted on this dataset.

REFERENCES

- [1] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A Study of Android Application Security," in Proc. 20th USENIX Security Symposium, San Francisco, CA, USA, 2011.
- [2] N. Viennot, E. Garcia, and J. Nieh, "A Measurement Study of Google Play," in Proc. ACM SIGMETRICS, 2014, pp. 221-233.
- [3] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall, "Context Is King: The Developer Perspective on the Usage of Static Analysis Tools," in Proc. 25th IEEE Int. Conf. on Software Analysis, Evolution and Reengineering (SANER), 2018, pp. 38-49.
- [4] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from Building Static Analysis Tools at Google," Communications of the ACM, vol. 61, no. 4, pp. 58-66, 2018.
- [5] W. Cunningham, "The WyCash Portfolio Management System," ACM SIGPLAN OOPS Messenger, vol. 4, no. 2, pp. 29-30, 1992.
- [6] SonarSource S.A., "Rules for Android (Java, Kotlin, XML)," SonarQube / SonarSource Rules Repository, 2024.
- [7] Google Android Developers, "Permissions on Android," Android Developer Documentation, 2024.

- [8] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android Permissions Demystified," in Proc. 18th ACM Conf. on Computer and Communications Security (CCS), 2011, pp. 627-638.
- [9] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing Inter-Application Communication in Android," in Proc. 9th Int. Conf. on Mobile Systems, Applications, and Services (MobiSys), 2011.
- [10] A. Chia, Y. Yamamoto, and N. Asokan, "Is This App Safe? A Large-Scale Study on Application Permissions and Risk Signals," in Proc. 21st Int. Conf. on World Wide Web (WWW), 2012.
- [11] Google Android Developers, "App Manifest Overview – the android:exported Attribute," Android Developer Documentation, 2024.
- [12] Snyk Labs, "Exploring Intent-Based Android Security Vulnerabilities on Google Play," Snyk Labs Research Blog, 2021.
- [13] SecureFlag, "Exported Components Vulnerability in Android," SecureFlag Security Knowledge Base, 2026.
- [14] Microsoft Security Response Center, "Intent Redirection Vulnerability in Third-Party SDK Exposed Millions of Android Wallets to Potential Risk," Microsoft Security Blog, Apr. 2026.
- [15] Payatu, "Penetrate the Protected Component in Android – Part 2," Payatu Security Research Blog, 2026.
- [16] Zensar Technologies, "Understanding Attacks on Exported Components in Android Applications," Zensar Technology Insights, 2026.
- [17] Appknox, "A Pentester's Guide to Android Component & Configuration Security," Appknox Blog, 2026.
- [18] AppSecure Security, "Securing Android Intents: Redirection, Spoofing & Interception," AppSecure Security Blog, 2025.
- [19] bithowl, "Exported Android Components: A Silent Security Risk," Medium, 2026.
- [20] F. Fischer, K. Bottinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl, "Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security," in Proc. 38th IEEE Symposium on Security and Privacy (SP), 2017, pp. 121-136.
- [21] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An Empirical Study of Cryptographic Misuse in Android Applications," in Proc. 2013 ACM SIGSAC Conf. on Computer & Communications Security, 2013, pp. 73-84.
- [22] MITRE Corporation, "Common Weakness Enumeration (CWE)," cwe.mitre.org, 2024.
- [23] Bugcrowd, "Common Weakness Enumeration (CWE)," Bugcrowd Glossary, 2022.
- [24] MITRE Corporation, "CWE Top 25 Most Dangerous Software Weaknesses," cwe.mitre.org, 2025.
- [25] OWASP Foundation, "OWASP Mobile Top 10 – 2024," Open Web Application Security Project, 2024.
- [26] M. Fowler and K. Beck, Refactoring: Improving the Design of Existing Code. Boston, MA, USA: Addison-Wesley Longman, 1999.
- [27] W. J. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. New York, NY, USA: John Wiley & Sons, 1998.
- [28] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the Diffuseness and the Impact on Maintainability of Code Smells: A Large-Scale Empirical Investigation," Empirical Software Engineering, vol. 23, pp. 1188-1221, 2018.

- [29] D. Taibi, A. Janes, and V. Lenarduzzi, "How Developers Perceive Smells in Source Code: A Replicated Study," *Information and Software Technology*, vol. 92, pp. 223-235, 2017.
- [30] N. Saarimaki, V. Lenarduzzi, and D. Taibi, "On the Diffuseness of Code Technical Debt in Open Source Projects," in *Proc. Int. Conf. on Technical Debt (TechDebt)*, 2019.
- [31] V. Lenarduzzi, N. Saarimaki, and D. Taibi, "Some SonarQube Issues Have a Significant but Small Effect on Faults and Changes: A Large-Scale Empirical Study," *Journal of Systems and Software*, vol. 170, art. 110750, 2020.
- [32] G. Digkas, M. Lungu, P. Avgeriou, and A. Chatzigeorgiou, "How Do Developers Fix Issues and Pay Back Technical Debt in the Apache Ecosystem?," in *Proc. IEEE Int. Conf. on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 153-163.
- [33] S. Nocera, D. Fucci, and G. Scanniello, "Dealing with SonarQube Cloud: Initial Results from a Mining Software Repository Study," *arXiv preprint arXiv:2508.18816*, 2025.
- [34] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.
- [35] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?," in *Proc. 35th Int. Conf. on Software Engineering (ICSE)*, 2013, pp. 672-681.
- [36] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering*. Berlin, Germany: Springer, 2012.
- [37] B. Kitchenham and S. Charters, "Guidelines for Performing Systematic Literature Reviews in Software Engineering," Technical Report EBSE 2007-001, Keele University and Durham University, 2007.
- [38] S. M. Ross, *Introductory Statistics*, 4th ed. Amsterdam, Netherlands: Academic Press, 2017.
- [39] SonarSource S.A., "Security Hotspots," SonarQube Documentation, 2024.
- [40] World Wide Web Consortium (W3C), "Web Content Accessibility Guidelines (WCAG) 2.1," W3C Recommendation, 2018.
- [41] Ecma International, "ECMAScript 2015 Language Specification (ES6)," 6th ed., 2015.
- [42] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308-320, 1976.