Received: 23 October 2022 | Revised: 9 July 2023 | Accepted: 5 August 2023

DOI: 10.1002/spe.3257

RESEARCH ARTICLE

WILEY

Security-based code smell definition, detection, and impact quantification in Android

Yi Zhong[1,2,3] | Mengyu Shi[1,2,3] | Jiawei He[1,2,3] | Chunrong Fang[1,2,3] | Zhenyu Chen[1,2,3]

[1]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

[2]Shanghai Key Laboratory of Computer Software Evaluating and Testing, Shanghai, China

[3]Shenzhen Research Institute of Nanjing University, Shenzhen, China

Correspondence
Chunrong Fang and Zhenyu Chen, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China.
Email: fangchunrong@nju.edu.cn and zychen@nju.edu.cn

Funding information
National Natural Science Foundation of China, Grant/Award Numbers: 61932012, 62141215, 61832009; Science, Technology and Innovation Commission of Shenzhen Municipality, Grant/Award Number: CJGJZD20200617103001003; Shenzhen Fundamental Research Program, Grant/Award Number: 2021Szvup057

Abstract
Android's high market share and extensive functionality make its security a significant concern. Research reveals that many security issues are caused by insecure coding practices. As a poor design indicator, code smell threatens the safety and quality assurance of Android applications (apps). Although previous works revealed specific problems associated with code smells, the field still lacks research reflecting Android features. Moreover, the cost and time limit developers to repairing numerous smells timely. We conducted a study, including Definition, Detection, and Impact Quantification for Android code smell (DefDIQ): (1) define 15 novel code smells in Android from a security programming perspective and provide suggestions on how to eliminate or mitigate them; (2) implement DACS (Detect Android Code Smell) to automatically detect the custom code smells based on ASTs; (3) investigate the correlation between individual smells with DACS detection results, select suitable code smells to construct fault counting models, then quantify their impact on quality, and thereby generating code smell repair priorities. We conducted experiments on 4575 open-source apps, and the findings are: (i) Lin's CCC between DACS and manual detection results reaches 0.9994, verifying the validity; (ii) the fault counting model constructed by zero-inflated negative binomial is superior to negative binomial (AIC = 517.32, BIC = 522.12); some smells do indicate fault-proneness, and we identify such avoidable poor designs; (iii) different code smells have different levels of importance and the repair priorities constructed provide a practical guideline for researchers and inexperienced developers.

KEYWORDS
Android code smell, quality assurance, repair priority, security

1 | INTRODUCTION

The widespread popularity of Android smartphones and other mobile devices makes the app market dynamic and challenging. Go-anywhere applications support a wide range of financial, social, and enterprise services for any user with a

Abbreviations: AST, abstract syntax tree; Lin's CCC, Lin's concordance correlation coefficient; NB, negative binomial; ZINB, zero-inflated negative binomial.

Yi Zhong and Mengyu Shi contributed equally to this study.

cellular data plan. However, as Android evolved, security and quality concerns surrounding various software patterns and poor coding practices have increased. Any security issues in the apps can put the privacy and security of countless users at risk. Insecure code can corrupt data structures, generate or expose security vulnerabilities. For example, unencrypted or weakly encrypted data can be easily stolen by attackers, and some components may have default open permissions that will result in the abuse of some sensitive functions. The security and quality assurance of Android apps become crucial and vital to keep appealing and adapting to new devices. As a metric indicating the suboptimal design, smells are the main culprit.[1,2]

By statistics, Android holds 71.55% of the market share, about 2.5 times greater than IOS.[3] The high market share makes Android more vulnerable to attacks by external sources, and security risks from poor programming selections within the code will expose users, and the platform, to varying levels of risk.[4] In this context, code smell emerged as a research hit to identify potentially poor design.[5] Although many works revealed the specific issues related to code smell, research on the definition and identification of Android remains scarce.[6] Unlike traditional software, code smells are distributed differently in Android, and the smells that truly reflect Android characteristics have not been widely studied. Most researchers following the definitions of Fowler et al.,[7] and Brown et al.,[8] there is an absence of new literature that identifies and defines Android code smells. Simply defining these code smells without detection is often insufficient to help researchers capture actual problems. Constructing code smell detection can better research these irregularities and reduce the associated security risks by code cleaning. Consequently, it becomes challenging and necessary to define and detect more code smells related to security and quality assurance.

In practical software development, massive code smells occur attributed to lousy coding habits and poor design patterns. Some developers prefer to refactor the code when performance issues arise rather than anticipate and repair smells in advance.[9] The costs and time hinder developers from repairing all code smells before they threaten the app's security, quality, and hardware performance. Research finds that some code smells do indicate fault-prone code, different smells may potentially affect the apps differently, and arbitrary refactoring some smells may increase fault-proneness in some cases.[10] Unfortunately, refactoring code smells without guidance may be a waste of time as repairing the ones with lower importance first. The maintenance cost of apps will increase exponentially with the detection and repair time of smells. Especially after release, with Android's increasing complexity and scale, labor and costs are expected to rise substantially.

Motivated by these observations, we researched the definition, detection, and static analysis of the importance level for Androids code smell. Specifically, we defined 15 novel Android code smells, marked the parts of the code vulnerable to security risks, and provided suggestions for elimination or mitigation. No existing tool can identify the 15 smells, so we developed our own detection tool–DACS (Detect Android Code Smell) based on AST to identify code smells occurring in a Method body. Besides, we established a dataset of 4575 open-source apps and collected fault information from Github by version control and keywords match. According to what we informed, there is no work to (i) a large-scale survey to collect faults in Android apps, (ii) analyze the impact of Android code smells on the fault occurrence, and (iii) quantify the code smells impact degree and generate repair priority. To obtain the code smell importance level, we investigated the distribution of code smells and fault data in the apps. We constructed fault counting models based on negative binomial (NB) and zero-inflated negative binomial (ZINB), where the arguments are the occurrence of code smells, and the dependent variable is the fault counts in apps. We analyzed the impact of code smells on the magnitude of faults by regression models. Then we filtered smells with low-quality impact by a two-sided hypothesis test to determine the final arguments. By the regression coefficient of the model, we generate the repair priorities for the code smells.

Definition, Detection, and Impact Quantification for Android code smell (DefDIQ) is dedicated to reducing security risks, lowering technical debt, and enhancing code cleanliness. The fault-centric code smell repair priority provides insight into the quality and security assurance of Android apps. This work helps guide developers to identify common security structure issues and promulgate the impact of programming choices in creating secure apps. The main contributions are as follows.

- **Definition.** We defined 15 novel Android code smells from the perspective of coding specifications, programming practices, and code features related to Android security and found the relationship between smells and faults, which will assist developers in discovering more poor designs and obtaining high-quality, low-risk apps.
- **Tool.** We developed DACS to automatically identify targeted code representing the custom-defined smells by establishing smells rule library. The Lin's concordance correlation coefficient (Lin's CCC) between DACS and manual detection results reached 0.994, verifying its effectiveness.

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

- **Repair priority.** We constructed fault counting models to quantify the importance of code smells by taking faults as a carrier with NB and ZINB. First, we find ZINB is a better modeling technique for investigating the relationship between faults data and code smells (compared to NB). Third, we quantify 15 code smells and generate repair priority, which provides a practical guideline for inexperienced developers to remediate smells timely and enhance the benefits.

The remainder is organized as follows. Section 2 presents the background and related knowledge. Section 3 explains DefDIQ method. Section 4 describes the experimental setup, including the dataset and evaluation metrics. Section 5 discusses the evaluation and the results of the conducted experiment. Section 6 presents threats to validity, while Section 7 introduces the related works. The final section concludes the paper and provides directions for future work.

2 | BACKGROUND

2.1 | Code smell in Android

Code smell was originally proposed by Kent Bech on Wiki and popularized by Fowler.[7] Sobrinho et al.[11] investigated plentiful works related to code smell from 1990 to 2017 and found the ones defined by Fowler were extensively studied. Code smell is a poor, sub-optimal coding structure, which may hinder comprehension,[12] increase security risks and fault-proneness, and reduce maintainability.[13] When extending the app with new functions, the subtle latent errors may bring fatal consequences and hinder the project's progress. Some fields, such as aerospace, medical, self-driving, etc., will suffer huge losses and irreversible results upon software faults.[14]

Previous studies identified negative designs based on Java static code metrics for most Android apps, as they are typically developed in Java/C++.[15,16] In essence, these studies still work for Android apps, but Android features distribute smells differently,[17] and some specific code smells are more frequent.[18] The distinguishing features of Android app development, as compared to traditional software, are the highly integrated environment and heavy reliance on package usage. Therefore the constant permission acquisition and confusing API calls during usage pose a security risk. The studies revealed that security risks are often accompanied by irregular coding structures, and Reimann et al.[19] summarize a set of poor programming habits (e.g., a nonstatic internal class containing a reference to an external class), namely Android-specific code smells. The Android code smells may threaten the security, data integrity, and software quality of mobile apps.[2,20] Given the above research, this paper defines security-related code smells from the perspective of Android security, that is, a symptom of code predicting security risks.

2.2 | Counting model

In this paper, we utilize faults to comprehensively assess Android quality, including software defects such as performance, memory, and security. The importance of smells is quantified by examining the distribution relationship between faults and code smells, where faults and smells are obtained by counting. The counting model is the simplest way to fit such a series of integers, where the count type is the response variable. The Poisson regression model is the classical count model, but with the precondition: the mean = the variance. To permit the variance of the observed data over the mean, Greenwood et al.[21] extended the Poisson regression to a NB regression. The NB distribution consists of a connected compound Poisson distribution, with the Poisson mean obeying the γ distribution, and its probability distribution equation is:

$$P_r(Y=y)=\frac{\Gamma(y+\tau)}{y!\Gamma(\tau)}\left(\frac{\tau}{\lambda+\tau}\right)^{\tau}\left(\frac{\lambda}{\lambda+\tau}\right)^{y},\quad y=y=0,1,\ \dots\ ;\lambda,\tau>0, \tag{1}$$

where $\lambda=E(Y)$, and τ refers to the over-dispersion. Y denotes the counting type argument with nonnegative integers, and the variance is $\lambda+\lambda^2/\tau$. When $\tau\to\infty$, the variance equals the mean, and the data degenerates to Poisson distribution.

In some cases, the probability of a "zero" event is easily underestimated, making it impossible to fit the two data distributions satisfactorily. Johnson et al.[22] observed the "zero inflation" phenomenon and proposed the fence model to

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 1 Spearman rank correlation coefficient.

Correlation coefficient	Correlation level
0.0–0.1	No correlation
0.1–0.3	Weak correlation
0.3–0.5	Moderate correlation
0.5–0.7	Medium-high correlation
0.7–0.9	High correlation
0.9–1.0	Full correlation

overcome this shortcoming. Then, Greene et al.[23] applied the principle to the NB distribution and proposed ZINB regression, where the Berndt–Hall–Hall–Hausman (BHHH) method is utilized to calculate the SEs of the model parameters.[24] The ZINB regression model separates the data into two components. The variables above zero obey the NB distribution, while the zero variables obey the discrete distribution, with the following equation:

$$P_r(Y=y)=\begin{cases} p+(1-p)\left(1+\frac{\lambda}{\tau}\right)^{-\tau} & , \quad y=0 \\ (1-p)\frac{\Gamma(y+\tau)}{y!\Gamma(\tau)}\left(1+\frac{\lambda}{\tau}\right)^{-\tau}\left(1+\frac{\tau}{\lambda}\right)^{-y} & , \quad y=1,2,\dots \end{cases}, \tag{2}$$

where τ represents the overdispersion, and the mean and variance are: $E(Y)=(1-p)\lambda$ and $\mathrm{var}(Y)=(1-p)\lambda(1+p\lambda+\lambda/\tau)$. When $\tau\to\infty$ and $p\to 0$, the data obey ZINB and NB distributions, respectively.

To determine the suitable analysis technology, we test the distribution of our data. Since the faults and smells data studied in this paper do not obey normal distributions and nearly half of the data were zero, see Figure 3, we adopted the NB and ZINB models to investigate the distribution relationship between the data.

2.3 | Spearman rank correlation coefficient

There are two classical coefficients to measure the correlation between indicators: Spearman rank correlation coefficient and Pearson correlation coefficient.[25] The Pearson correlation coefficient has two restrictions: (1) the data obey the normal distribution; (2) the data units are consistent, and the zeros are relative, rather than absolute. If the measured metrics do not meet the Pearson conditions, it is necessary to consider the Spearman rank correlation coefficient. It tends to be applied for calculating the correlation between the sorted variables, so the initial data need to be transformed into ordered data. Assuming x_i and y_i are a pair of order data with a monotonically increasing trend, the Spearman rank correlation coefficient is calculated as follows:

$$\rho=\frac{\Sigma i(x_i-\overline{x})(y_i-\overline{y})}{\sqrt{\Sigma i(x_i-\overline{x})^2\Sigma i(y_i-\overline{y})^2}}. \tag{3}$$

Table 1 gives the correlation coefficients and the correlation levels. When the coefficient is 0, it represents that the two metrics are entirely unrelated; when the coefficient is 1, it denotes that the two are entirely related.

The Spearman correlation is not strict on data preconditions and ignores the overall data distribution and sample size. This paper uses Spearman's rank correlation coefficient to measure the correlation between code smells.

3 | DEFDIQ METHODOLOGY

3.1 | Overall framework

We propose DefDIQ, research on the definition, detection, and impact quantification for Android code smell. As shown in Figure 1, DefDIQ consists of three primary research components: (1) define 15 novel security-related Android code smells

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

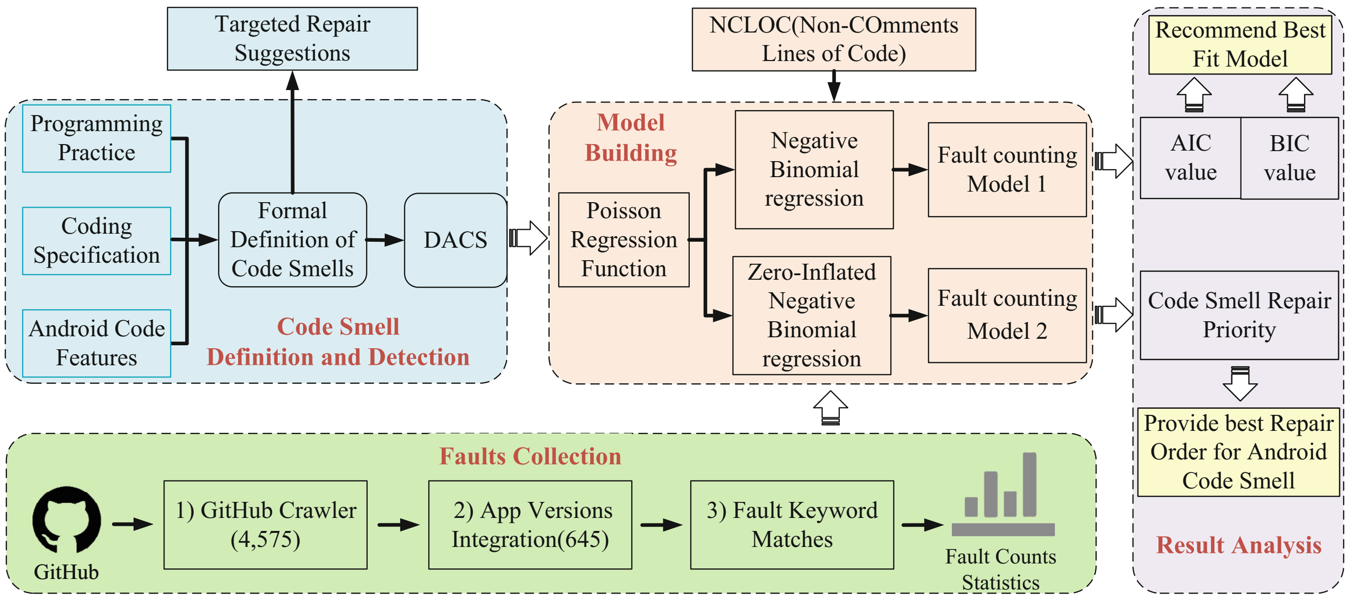

Targeted Repair Suggestions
Programming Practice
Coding Specification
Android Code Features
Formal Definition of Code Smells
DACS
Code Smell Definition and Detection
NCLOC(Non-COmments Lines of Code)
Model Building
Poisson Regression Function
Negative Binomial regression
Zero-Inflated Negative Binomial regression
Fault counting Model 1
Fault counting Model 2
Recommend Best Fit Model
AIC value
BIC value
Code Smell Repair Priority
Provide best Repair Order for Android Code Smell
Result Analysis
Faults Collection
GitHub
1) GitHub Crawler (4,575)
2) App Versions Integration(645)
3) Fault Keyword Matches
Fault Counts Statistics

FIGURE 1 DefDIQ flowchart.

and offer targeted repair suggestions, (2) realize DACS to detect targeted smells in Android apps source code automatically, and (3) construct fault counting models to investigate the distribution of custom smells and faults occurrences, quantify the impact of Android code smells and generate repair priorities.

3.1.1 | Define Android code smells

Code smells are mainly attributed to deviations from standard coding practices by developers, and are likely leading to security risks and faults. Also, with the specific structure, smells can be identified by relevant information. Previous research findings indicate that Android code smells perform better in security risk prediction.[26] Therefore, we analyzed the security structure in Android apps and defined 15 novel Android code smells by collecting programming practices, coding specifications, and Android code features related to Android security and defect prediction.

3.1.2 | DACS implementation

Code smells are subjective, so the community lacks the tools to identify the code smells we defined. Accordingly, we design a lightweight automatic detection tool–DACS for the 15 custom smells. DACS implements detection algorithms based on detailed definitions and generates a smell rule library. Firstly, DACS extracts semantics in the leaves and syntax in the nonleaves, effectively representing the semantic and structural information of the source code via AST.[27] Then search the AST with the detection rule library and identify the code smells. DACS implements a detection subsystem for each code smell and presents the final result as a *.csv* file or an interface. The source code of DACS is publicly available on GitHub at https://github.com/strongcat0325/DACS.git.

3.1.3 | Statistical analysis

We construct fault counting models with NB and ZINB to investigate the distribution between the occurrences of smells and faults, and statically analyze whether custom smells have an impact on faults. Based on the constructed models, we quantify the impact of code smells and generate repair priority, providing a practical guideline for inexperienced developers. Among them, the fault is a key indicator of app quality. We obtained faults information by the commit specification

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

from GitHub, and gathered faults by version control and keywords match. Then we adopt Android code smells as arguments, while the fault counts as response variables, and apply NB and ZINB regression to construct fault counting models. Ultimately, quantify the code smells' importance based on the parameters in the model and generate repair priorities for all smells by the correlation among smells.

3.2 | Android code smell definition

This paper summarizes and formally defines 15 novel Android code smells and provides targeted repair suggestions. The detailed definition lists are given below.

- **Weak Crypto Algorithm (WCA):** The weak crypto phenomenon usually occurs in Android apps, mainly for two reasons: using weak crypto algorithms; using strong crypto algorithms, but with vulnerabilities in the implementation.
  Implementation in software development: Specifically including: 1) Methods with weak crypto hash functions (such as MD2, MD4, MD5, SHA1, or RIPEMD) will trigger code smells; 2) Methods execute the DES algorithm or initialize in AES by ECB mode; 3) Method with RSA algorithm for the key length <512 bits; 4) Recommend to adopt `Cipher.getInstance(RSA/ECB/OAEPWithSHA256AndMGF1Padding)` for the crypto algorithm, or hackers will send malicious attacks by received packets.
  Repair suggestions: 1) Recommend developers to adopt SHA256 and SHA3 functions instead of weak crypto hash functions; 2) Recommend developers to adopt AES algorithm, specify CBC or CFB mode available with PKCS5Padding; 3) Recommend developers set the key length to 2048 bits in the RSA algorithm.
- **Improper certificate validation (ICV):** Android provides an embedded process to verify certificates signed by CA. When applying a self-signed certificate, the operating system will verify it through the apps. Developers often fail to implement proper certificate validation, modify certain methods but neglect the underlying logic and code implementation inside, leaving the communication channel over SSL/TLS vulnerable to man-in-the-middle attacks.
  Implementation in software development: 1) HTTPs communication via `Webview`, simply handling certificate errors with `proceed()` in the `onReceivedSslError()` method; 2) Developers customized to implement class `X509TrustManager`, rewrite method `checkClientTrusted()` and `checkServerTrusted()`, but there is no logical code for reviewing the certificate in the method (empty implementation), will generate code smell; 3) Developers adopt a custom implemented class `HostnameVerifier`, there is no check on the validity of the host name in the `verify()` (directly return true); 4) Method with unsafe HostnameVerifier:org.apache.http.conn.ssl.AllowAllHostnameVerifier, org.apache.http.conn.ssl.SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER, is equivalent to nonverification.
  Repair suggestions: 1) Recommend developers not to override `onReceivedSslError()` and not to leave the certificate problem unsolved to avoid the leakage of communication data; 2) For HTTPS websites with certificates signed by authorities, developers can adopt the certificate verification mechanism from Android instead of implementing it; 3) Implement HTTPS domain verification code in `HostnameVerifier`, that is, verify whether the domain connected to the HTTPS site and in the SSL certificate is the same; 4) Recommend developers to adopt the secure Android built-in `HostnameVerifier`: org.apache.http.conn.ssl.SSLSocketFactory.STRICT_HOSTNAME_VERIFIER).
- **Unconstrained intercomponent communication (UICC):** External apps can freely call components of an app, and there are no restrictions in the communication.
  Implementation in software development: Several apps reuse the third party components (Activity/ContentProvider/Service/BroadcastReceiver) by setting the attribute `exported` or `IntentFilter` in `AndroidManifest.xml`. For instance, when the `exported` is true, or the `exported` value is not set but set to the `IntentFilter`, the component is exposed to the risk of being invoked by malware.
  Repair suggestions: Recommend developers to set the exported value to false so that external apps are unable to invoke the component. If the developer expects a specific app to visit the component, then `exported` cannot be set to false, and the attribute `permission` needs to be set to a custom permission string.
- **Custom scheme channel (CSC):** When customizing the page jump protocol scheme in Android, there is a risk that external attackers can access app data through web pages.
  Implementation in software development: The `IntentFilter` (with three attributes: `action`, `category`, `data`) in `AndroidManifest.xml` is for parsing implicit `intent`. Including: 1) If `data android: scheme` is

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

specified, then start the `intent` class to complete communications among the components, and the attackers can visit the app components by web, causing code smell; 2) If the browser supports `Intent Scheme Uri`, and sets no filtering rules, then the attacker can visit the browser's files (whether private or public) through JS code, such as stealing cookies files.

Repair suggestions: 1) Recommend developers not to specify the attribute `data android:scheme`; 2) If invoking the function `Intent. parseUri`, then the `intent` must set strict filtering conditions with at least three modes: addCategory("android.intent.category."), setComponent(null), setSelector(null).

- **Headers attachment (HA):** When communicating with a server by HTTP requests, developers typically send sensitive data that relies on header transmission, such as platform ID, channel ID, system version, and other common information. Storing these sensitive data in header files causes insecurity in the communication process.

  Implementation in software development: Storing private data in header files is generally considered a code smell. There are three types of header attachments: set the http header parameters of `OkHttpClient`, `HttpURLConnection` and `HttpClient`.

  Repair suggestions: Before authenticating a third-party service, developers should not store sensitive data in the header.

- **Exposed clipboard (EC):** In Android apps, the clipboard can store data temporarily in RAM leading to easy theft.

  Implementation in software development: The app invokes `settext()/getText()` or `setPrimaryClip()/getPrimaryClip()` of the `ClipboardManager` for writing/reading data to the clipboard. It is insecure to store private data (especially passwords) in the clipboard, which is readable by any app.

  Repair suggestions: Developers should avoid storing data in the clipboard.

- **Broken web view's sandbox (BWS):** In the Android SDK, `WebView` provides a browser allowing users to visit the web in the apps. Among them, the improper utilization of `WebView` will easily cause arbitrary execution of codes.

  Implementation in software development: This code smell occurs for three main reasons: 1) The `addJavascriptInterface` interface in the `WebView` generates a Java object, and JS invokes the object's methods to communicate with local apps. JS can do anything after getting the object, including reading sensitive information on the device's SD card; 2) The Js interface of `searchBoxJavaBridge` generates JS mapping objects by default, causing the execution of arbitrary code, triggering code smell; 3) Similarly, the Js interface of `accessibility` and `accessibilityTraversal` will also lead to the arbitrary execution of codes, generating code smell.

  Repair suggestions: 1) Before Android 4.2, the result to JS is handled by `prompt()` method; after Android 4.2, Google provides the annotation `@JavascriptInterface` to avoid vulnerability attacks; 2) Remove the `searchBoxJavaBridge` interface by invoking the `removeJavascriptInterface()` method; 3) Remove the `accessibility` and `accessibilityTraversal` interfaces.

- **Web view plain secret (WPS):** In Android apps, the `WebView` enables password saving by default and generate code smell.

  Implementation in software development: `WebView.setSave Password(true)`. When the user fills in and agrees to save the password, it will be stored in the database in plain-text.

  Repair suggestions: Suggest the user to close the password saving function: `WebView.setSavePassword(false)`.

- **Web view domain not strict (WDNS):** Android built-in method `setAllowFileAccess()` is for setting whether to allow the `WebView` to use the file protocol, the default setting is true, making it possible to visit some files.

  Implementation in software development: When the following setting is true: 1) `setAllowFileAccessFromFileURLs()`, for setting whether the JS code in the file path is available to visit the local files; 2) `setAllowUniversalAccessFromFileURLs()`, for setting whether the JS code in the file path is available to visit cross-domain sources, such as http domain; 3) `setJavaScriptEnabled()`, for setting whether to allow loading JavaScript.

  Repair suggestions: For general apps, developers should prohibit the file protocol, that is, `setAllowFileAccess(false)`; for apps requiring special functions with file protocol, developers should set the above attributes to the prohibited state.

- **Data back up any (DBA):** An incorrect setting of the allowBackup property in the Android configuration, which is used to manage data archiving for Android apps, can result in arbitrary copies of data.

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Implementation in software development: When the value of `allowBackup` is true, ADB debugging tool can copy and export data without super administrator authentication. Attackers will obtain private data by USB debugging, thus leading to user privacy leakage. We define this behavior as a smell.

Repair suggestions: Recommend developers set the attribute `allowBackup` in the `AndroidManifest.xml` to false.

- **Global file readable/writable (GFRW):** In Android apps, `Activity` exports data to a file by `openFileOutput(String globalfile_name, int operation_mode)`. Improper setting of "operation_mode" will cause malicious data reading.

  Implementation in software development: The details including: 1) When the operation mode is set to MODE_WORLD_READABLE, attackers can read sensitive information in the file. This behavior will produce smell; 2) When the operating mode is set to MODE_WORLD_WRITEABLE, attackers can write the file's contents freely and destroy the app's integrity, producing code smell.

  Repair suggestions: Developers should identify whether sensitive data is stored in the file. If sensitive data exists, they should: 1) Avoid setting the mode to MODE_WORLD_READABLE; 2) Avoid setting the mode to MODE_WORLD_WRITEABLE.

- **Configuration file readable/writable (CRW):** `SharedPreferences(String config_name, int operation_mode)` is a common data storage method in Android, improper setting of "operation_mode" will cause malicious data reading.

  Implementation in software development: Specifically, 1) When set mode to MODE_WORLD_READABLE, the third-party app can view the file, resulting in leakage of sensitive information; 2) When setting the mode to MODE_WORLD_WRITEABLE, the third-party app can delete and change the file, potentially introducing malicious code.

  Repair suggestions: Developers should set the mode of the `getSharedPreferences()` method to MODE_PRIVATE.

- **Malicious unzip (MU):** The Android app will take `ZipInputStream` and `ZipEntry` when decompressing/compressing zip files, while it will be considered malicious if the file name contains special symbols.

  Implementation in software development: The `ZipEntry` provides `getName()` to read the name of the zip file. If the file name has a special string like ../ when decompressing the file, it will produce a code smell.

  Repair suggestions: Recommend developers filter the upper directory string and check the file name when decompressing.

- **SD visit (SDV):** Android apps store public data in shared external storage by invoking `getExternalStorageDirectory()` without encrypting will be considered nonstandard.

  Implementation in software development: Developers store sensitive information on an external SD card without encryption.

  Repair suggestions: Recommend developers to prohibit `getExternalStorageDirectory()`, store sensitive information in the private directory of the app and encrypt the sensitive data.

- **Unsecure random (UR):** The `SecureRandom` is for obtaining random numbers in Android encryption algorithm. By default, the random number list is generated by dev/urandom, and the result is hard to predict. However, when setting the seed, the random number is generated based on a fixed algorithm and seed, so it can be easily predicted.

  Implementation in software development: When calling class `SecureRandom`, generating random seeds with the following method is considered a smell: `SecureRandom.SecureRandom(byte[] seed)`, `SecureRandom.setSeed(byte[] seed)` and `SecureRandom.setSeed(long seed)`.

  Repair suggestions: 1) Recommend developers not to generate random numbers by Random class; 2) Do not set a seed when using SecureRandom.

3.3 | Detect Android code smell

We developed DACS to detect 15 targeted Android smells in source code. Android code smells are fine-grained method-level code structures and must be identified by analyzing the source codes, so DACS completes the detection based on ASTs from source codes. DACS designs detection rules for each Android code smell and independently realizes the detection submodule to output the targeted smell results. As depicted in Figure 2, the detection method mainly

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

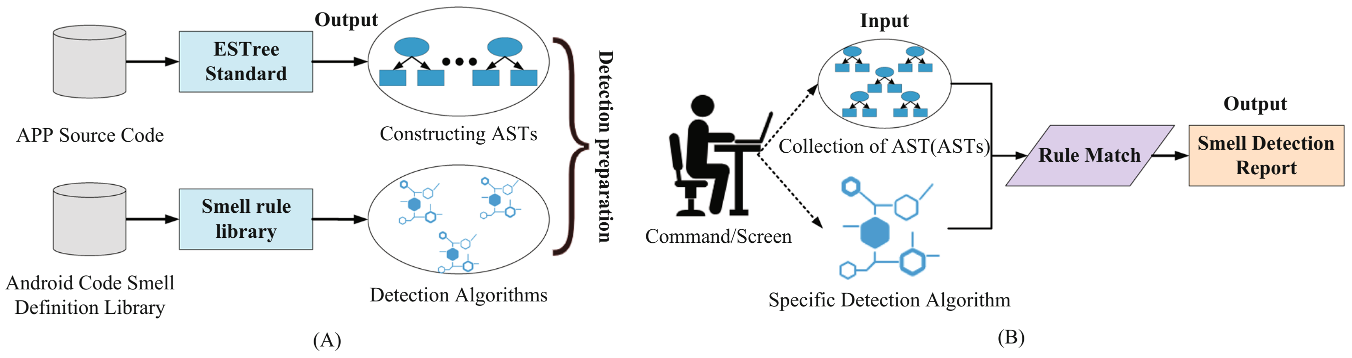


FIGURE 2 Detect android code smell (DACS) detection method. (A) Detection preparation module; (B) DACS's code smell-Specific detection.

contains two modules: (i) convert the Apps source code into ASTs under ESTree standard, and design smell library based on the definitions, then realize the specific identification algorithms, and (ii) collect the ASTs set and search the AST by specific Android code smell detection algorithm, and count the occurrences of each code smell based on the matching rules.

AST facilitates code smell detection in apps by abstracting the syntactic structure of source codes in the form of trees, with the relationship nodes representing lexical information and syntactic structure. Then, DACS traverses the AST following the detection algorithm and counts the number of identified code smells. Specifically, (1) depending on the specific definition, summarize the detection keywords; (2) taking the method level as the detection cell, match the keywords within each method body to determine whether the method contains the code smell; (3) calculate the total number of code smells present in the app.

The 15 code smells are detected by matching rules to complete the detection, and the implementation method is similar. We take a specific code smell–WCA as an example to show the detection process in detail, as shown in Algorithm 1. The algorithm's words roughly as follows: First, according to the definition in Section 3.2, extract the keywords for detecting WCA smells. Then, at the method level, match the keywords within each method body to determine whether there are code smells present. Finally, count the total number of WCA in the class. The input to the algorithm is a Java class, and the output is the number of WCA. We detect code smells in each .java file, accumulate the number of code smells, and ultimately output the number of each code smell present in each class.

3.4 | Statistical analysis of impact quantification

DfeDIQ proposes a fault counting model to quantify the relationship between Android code smells and the app's fault count (no association, positive association, negative association, or combined association), providing insights for repair priority. We verify whether code smells and their combinations matter to the fault counts by two-sided hypothesis testing, determine the rejection domain threshold value according to the set significance level, and then determine the model parameters. The discrete regression relationship is established to illustrate which code smells are of no concern to developers and which ones should be paid special attention to and repaired in time. Among them, the Android code smells are selected from Section 3.2; the fault information is extracted from the history submission data of practical open-source Android apps.

3.4.1 | Fault statistic

The software testing community has different definitions and conceptual descriptions of software problems. DefDIQ focuses on a broad definition of fault, and in this paper, the following issues are widely defined as fault, including (1) fault, describing the incorrect program design in software; (2) vulnerability, representing the static defects in software; (3) error,

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Algorithm 1. wca detection algorithm

```
Initialization:$cnt \leftarrow 0$
repeat
    method ← pClassBean.getMethods()
    content ← method.getTextContent()
    if content.contrains(“MD2”||“MD4”||“MD5”||“SHA-1”||“RIPEMD”||“DES”||“AES/ECB”) then
        $cnt \leftarrow cnt + 1$
    else if content.contrains(“RSA”) then
        if !content.contrains(“RAS/ECB/OAEPWithSHA256AndMGF1Padding”) then
            $cnt \leftarrow cnt + 1$
            $macher \leftarrow pattern.matcher(content)$
            while matcher.find() do
                if Integer.parseInt(matcher.group(2)) < 512 then
                    $cnt \leftarrow cnt + 1$
                end if
            end while
        end if
    end if
until pClassBean.getMethods() is null
Ensure: cnt //code smell number
```

describing the error results from running defective software; (4) failure, referring to the external software failure behavior observed by users; (5) bug, indicating the general description of software failures and errors; (6) defect, which can cause various bugs. These problems affect quality from different perspectives, including performance, security, memory, energy consumption, portability, and efficiency.

Fault data sources are a critical factor affecting fault counting models. Researchers use data warehouses to collect faults, and these data warehouses can be broadly classified into three categories: private/commercial warehouses, partial public/free warehouses, and public warehouses.[28] Among them, public repositories are adopted by most researchers as a rich and valuable source of fault data (e.g., Bugzilla, JIRA, PROMISE). However, Android apps rely on third-party libraries and update so quickly that no available public data repository has yet emerged. Due to the limitation of traditional manual time-consuming and labor-intensive, researchers mainly collect faults data in large datasets by the method of Zimmermann.[29]

Android apps on GitHub are diverse, with some in multiple development versions, some in only one version, and others even incomplete. To collect reasonable faults data, we focused the research on the same development version of the Android app and counted the faults by keyword matching. The specific rules for the app version selection are as follows:

- Incomplete version: search the first complete executable project from the original commit and record the faults between this commit and the latest commit.
- One version: record the faults between this version and the latest one.
- Version number ≥2: record the faults between the latest version and the last committed one.

DefDIQ categorizes the apps by version, locates the commits that contain all fault keywords in the “open” and “closed” problems, and counts the faults. The method is summarized as follows:

- Find commits with the “fail,” “vulnerability,” “error,” “failure,” “defect,” “bug,” “mistake,” “fix(ed),” and “update(d)” in the committed and resolved problems. In particular, ignore the case of the keyword while matching.
- Due to the irregularity by the developers, many commits with faulty keywords are not purely faulting repairs. Therefore, it is essential to manually review the commits that successfully match the keywords. Due to massive faults, random sampling detection may be a great choice.

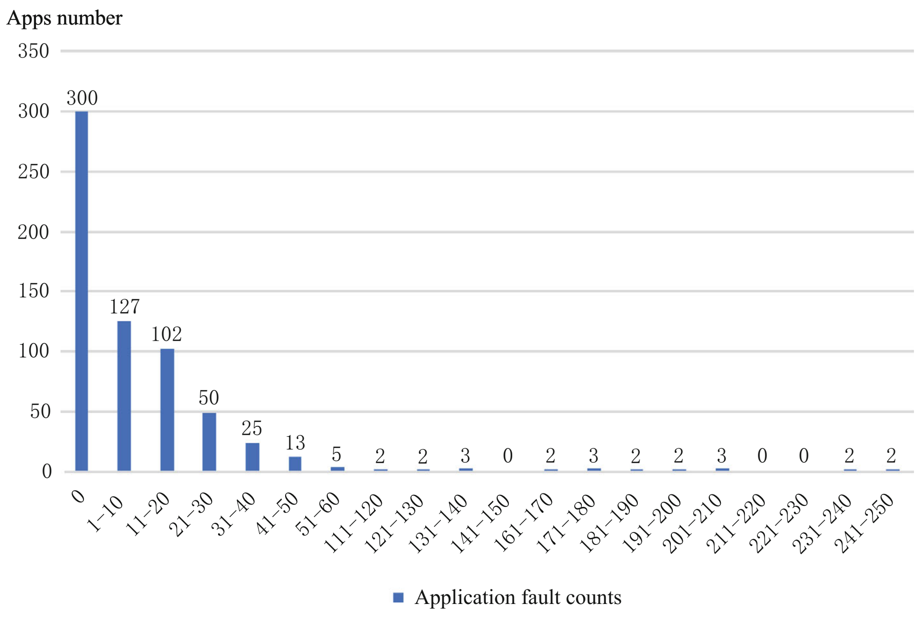


FIGURE 3 Application fault counts statistics.

We obtained the statistical information about the app fault through the above steps, and all of the raw fault data is available at https://github.com/strongcat0325/Fault-Data.git, as presented in Figure 3. The original fault counts do not obey the normal distribution with a significant degree of dispersion, and most apps' fault counts are 0.

3.4.2 | Construct fault counting model

By establishing a fault counting model, we can further identify and repair the high-impact defect structures to reduce code review/testing costs. The counting model achieves a linear fitting for discrete data, and the classical discrete counting models are Poisson regression models and zero-inflated Poisson models. Since Poisson distribution has constrained preconditions, the intra-group correlation cannot be negligible for some interdependent data, so other similar models should be considered.

For determining the suitable counting model, we observed the original data distribution and found that (1) the faults are distributed discretely; (2) the majority of apps do not contain faults, with statistical results showing that the majority have zero faults; (3) The faults dataset does not obey normal distribution, the variance > the mean, and exists zero-inflated phenomenon. The main reasons for (1) and (2) are that the Android app development cycle is short and evolves rapidly, while the development is not standardized, and testing is insufficient for small and medium-sized projects on GitHub. For (3), we conducted a dispersion test on the faults data by the `dispersiontest()` in R (to verify whether the variance is equal to the mean). The findings indicate that there is a small probability ($p < 0.001$) for the case of data variance = the mean, thus rejecting the original hypothesis. Therefore, NB and ZINB regression models are more suitable for fitting the data than the Poisson regression model.

We design the following scheme to address the discrete data.

- To minimize the dispersion of fault counts, we take the Density of Faults (i.e., the number of faults per thousand lines of code [NCLOC] in the file) as the response variable, and the measurement unit is faults/KLOC. The processed data are depicted in Figure 4, but the Density of Faults distribution still does not follow a normal distribution. As fault data with a normal distribution is absent, many counting models, such as Poisson distribution, failed to work.
- Considering the presence of excessive 0 in the faults data, DefDIQ adopts the NB regression model and ZINB regression model to analyze the relationship between the faults and Android code smells. And the two regression models are inclusive of over-discrete response variables, that is, allowing the variance > the mean. Nearly half of the apps have no faults, such data are subject to significant bias in the traditional Poisson regression model, while ZINB model provides

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

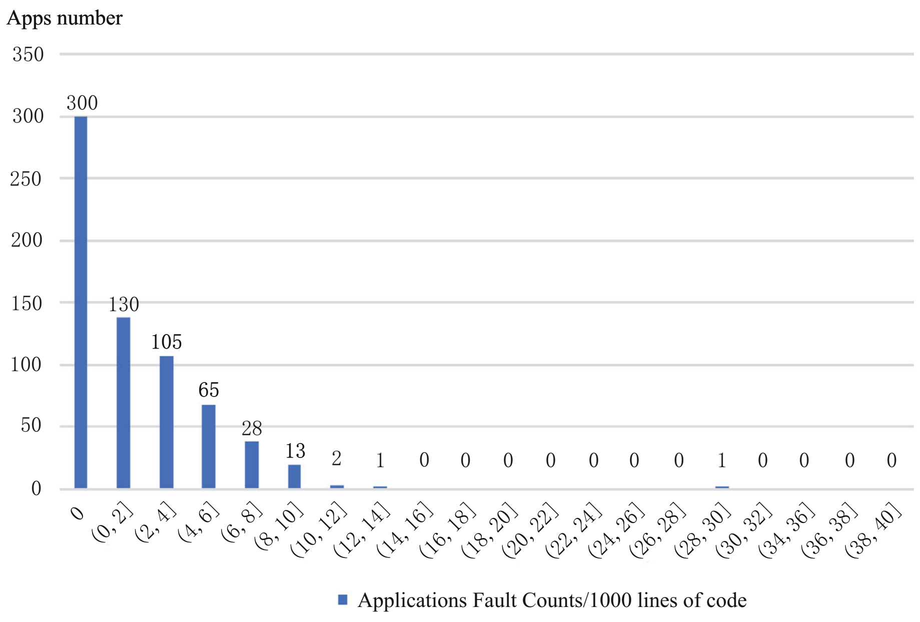


FIGURE 4 Application fault counts density statistics.

insight to address this issue. In DefDIQ, we apply NB and ZINB to build the counting models and evaluate the model fitness by Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC) metrics. Where the fault counts are dependent variables and some custom Android code smells are selected as arguments.

4 | EXPERIMENTAL SETUP

To investigate and evaluate the performance of DefDIQ in automatically detecting Android code smell and constructing fault counting models, we designed the following research questions (RQs):

- **RQ1:** How effective is DACS detecting custom Android code smells?
- **RQ2:** What is the correlation between Android code smells and the faults in the fault counting model?
- **RQ3:** What is the fitting degree of the fault counting model constructed by NB and ZINB, and to what extent do they affect the quality of the apps?

RQ1 investigates the detection performance of DACS and explores whether the DACS can replace manual detection. Since the code smells detected are novel and customizable, there is no mature tool for comparison. To obtain a reliable reference for the detection results, we invited researchers to accomplish the code smell detection of the apps and take it as a baseline to verify the effectiveness.

RQ2 intends to investigate the distribution between five selected code smells and faults by constructing fault counting models, and observe the correlation between code smells and faults to direct developers and researchers to smells with more significant impacts on security and quality.

RQ3 investigates which regression model is more appropriate for the experimental dataset and quantifies the impact of security code smells on quality based on the best model. This helps to determine the relative importance of code smells on quality (fault causation), assisting in the generation of repair priorities and providing a practical guideline for inexperienced developers.

4.1 | DACS experiment

In this section, we investigate the DACS performance and verify its effectiveness by measuring the agreement with manual detection results. Specifically, note that, manual detection and DACS only share the same code smell definitions while not sharing any other detection strategies.

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 2 Apps for detect Android code smell effectiveness evaluation.

App ID	Name	Category	Version	App ID	Name	Category	Version
1	NewPipe	Video & Audio	v0.14.2	11	Cupboard	Life	v1.0
2	FxcnBeta	Reading	3.2	12	RITA	Tool	v1.0
3	RGBTool	Tool	v1.4.2	13	Desserter	learning	v1.0
4	EasyBudget	Life	1.6.2	14	TorchLight	Shopping	v2.4
5	PodTube	Video	1.3.2	15	VINCLES	Social	3.12.0
6	Typesetter	Tool	v1.0.1	16	intra42	Learning	v0.6.2
7	Snapdroid	Video and audio	v0.15.0	17	Natibo	Learning	v0.2.4
8	LAY	Health	v0.1.3	18	Lexica	Game	v0.11.4
9	Habitat	Tool	v1.0.9	19	Timber	Video and audio	v1.6
10	WhuHelper	Learning	v1.0	20	MateSolver	Game	v2.6

4.1.1 | Dataset

The experiment was deployed on 20 open-source Android apps from small- and medium-sized projects with high star ratings on GitHub. The 20 apps originate from distinct categories involving learning, video and audio, reading, social, game, etc., with different sizes. Table 2 summarizes the release and package links associated with the apps.

4.1.2 | Experimental settings

We use DACS to label the custom code smells within Java files in corresponding apps and obtain detection reports. Then we arranged two experienced Java programmers to manually check the files based on the detailed definitions and count the smells. To minimize the errors caused by individual subjective judgment, the second programmer was arranged to detect again after the first one completed the detection work. The second review focused on whether the first programmer misidentified the smells. The two programmers worked independently and without communication during the detection. The results showed only 11 code segments detected by the first programmer were categorized as misidentified by the second. After discussion, six code segments were finally identified as misidentified, and then revised the results. Additionally, to avoid bias, the programmers did not know the experimental details and the specific rules of DACS. By this point, a relatively unbiased and accurate code smell detection result was generated, and we regard it as a comparison baseline.

4.1.3 | Evaluation method

Code smell is a subjectively defined code structure that possibly negatively influences the app's performance. However, there is a lack of open-source, detailed code smell detection methods in the industry, which makes it infeasible to adopt precision or recall as evaluation metrics without a baseline of reliable detection data. Consequently, we measure the agreement with manual detection as DACS evaluation metrics. Cohen's Kappa statistic is a common metric for agreement testing, but it is suitable for categorical ratings while having some limitations in count-based models.[30] For continuous integer experimental data, Lin's CCC will be better for measuring the agreement of two objects.[31] Assuming two random variables x and y, the formula for CCC ρ_C is given in Equation (4).

$$\rho_C = \frac{2\rho\sigma_x\sigma_y}{(\mu_x - \mu_y)^2 + \sigma_x^2 + \sigma_y^2}, \tag{4}$$

where μ_x and μ_y are the means of the two variables, σ_x^2 and σ_y^2 are the corresponding variances, and ρ is the correlation coefficient. When the dataset length is N, that is, $(x_n, y_n), n = 1, \ldots, N$. Then the CCC is calculated in Equation (5).

$$r_c = \frac{2s_{xy}}{s_x^2 + s_y^2 + (\bar{x} - \bar{y})^2}. \tag{5}$$

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Among them, the mean is obtained by Equation (6), the variance s_x^2 and covariance s_{xy} are expressed by Equation (7).

$$\bar{x} = \frac{1}{N}\sum_{n=1}^{N} x_n. \tag{6}$$

$$s_x^2 = \frac{1}{N}\sum_{n=1}^{N}(x_n - \bar{x})^2, \quad s_{xy} = \frac{1}{N}\sum_{n=1}^{N}(x_n - \bar{x})(y_n - \bar{y}). \tag{7}$$

DefDIQ relied on the CCC evaluation indicator as discrimination for DACS effectiveness, here $N = 15$, representing the 15 custom smells. The data pairs denote each code smell i, the occurrences for manual detection x_i and DACS y_i. Similar to the correlation coefficient, $-1 \le \rho_C \le 1$ and $-1 \le r_C \le 1$, when the metric approaches +1, it indicates strong agreement between x and y, and as the metric gets closer to −1, it indicates strong disagreement between x and y. Another accepted method is to interpret Lin's CCC metric as Spearman rank correlation coefficient (e.g., <0.20 denotes poor agreement, while >0.80 indicates good agreement). In this paper, we follow this interpretation to analyze the agreement strength of the DACS and manual detection results, and the details are shown in Table 1. We rely on the CCC function in the DescTools library in R to obtain Lin's CCC with the confidence interval $1 - \alpha$, $(\alpha = 0.05)$.

4.2 | Fault counting model experiment

4.2.1 | Data collection

In this paper, we collect, 575 practical Android apps from GitHub and count the faults based on keywords (see Section 3.4.1 for the details). To guarantee the correspondence between the faults and apps, DefDIQ records the faults within one development version, so the 4575 original apps were reduced to 645. We randomly selected 516 apps as training data and the rest as prediction data. The obtained original data is shown in Figure 3.

Two main factors were considered in determining the model parameters:

a i.The model is vulnerable to over-fitting with excessive parameters or insufficient sample data. Through investigation, we found that most studies constructed models with ≤ 6 parameters,[10,32-34] so we set around five parameters for the fault counting model.

b ii. The correlation between the parameters. The higher correlation between code smells implies that some smells are redundant and multicollinear, which leads to (1) an unnecessary increase in the feature dimensions, (2) a high tendency to over-fit the model, and (3) interaction between features will reduce model credibility.

To determine suitable parameters to complete the model construction, we detected 15 smells by DACS and counted the occurrences of each smell in 4575 apps, and the percentages are shown in Figure 5. To reduce model errors caused by insufficient data, we selected smells with higher occurrences for model construction. Besides, we measured Spearman rank correlation to discover the relationship among 15 custom Android code smells based on DACS detection results. Figure 6 demonstrates the confusion matrix of Spearman correlation coefficients for code smells, which can visualize the magnitude of the correlation. We found that the correlation among the code smells was quite low (<0.30), except for the strong correlation for WCA with CSC, MU, SDV, MU with SDV, and HA with MU, SDV. These code smells are independent and will not affect each other, meeting the prerequisites of some regression models while avoiding the experiment's randomness and ensuring the experiment design's rationality. Therefore, we take into account the occurrences of the 15 code smells as well as their correlations to select appropriate ones as parameters for fault counting model construction.

Finally, we selected WCA, HA, DBA, MU, and SDV as the arguments to fit the faults counting model. In addition, we take the code size into account (NCLOC) as an additional factor to investigate whether file size affects faults. Figure 7 shows a matrix scatter matrices of the five smells, provides the faults distribution in the apps, and visualizes the changing relationship between the smells and faults. Furthermore, Figure 7 illustrates the difference in the density and distribution of code smells in apps, where the horizontal and vertical coordinates are two different code smells.

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

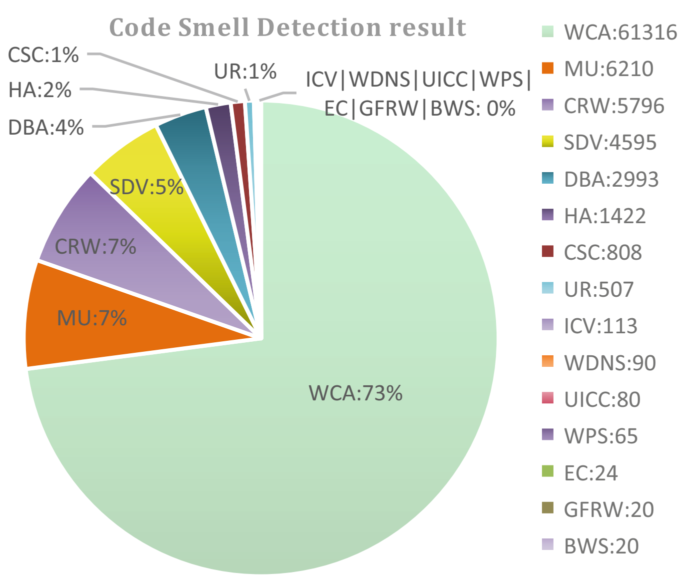
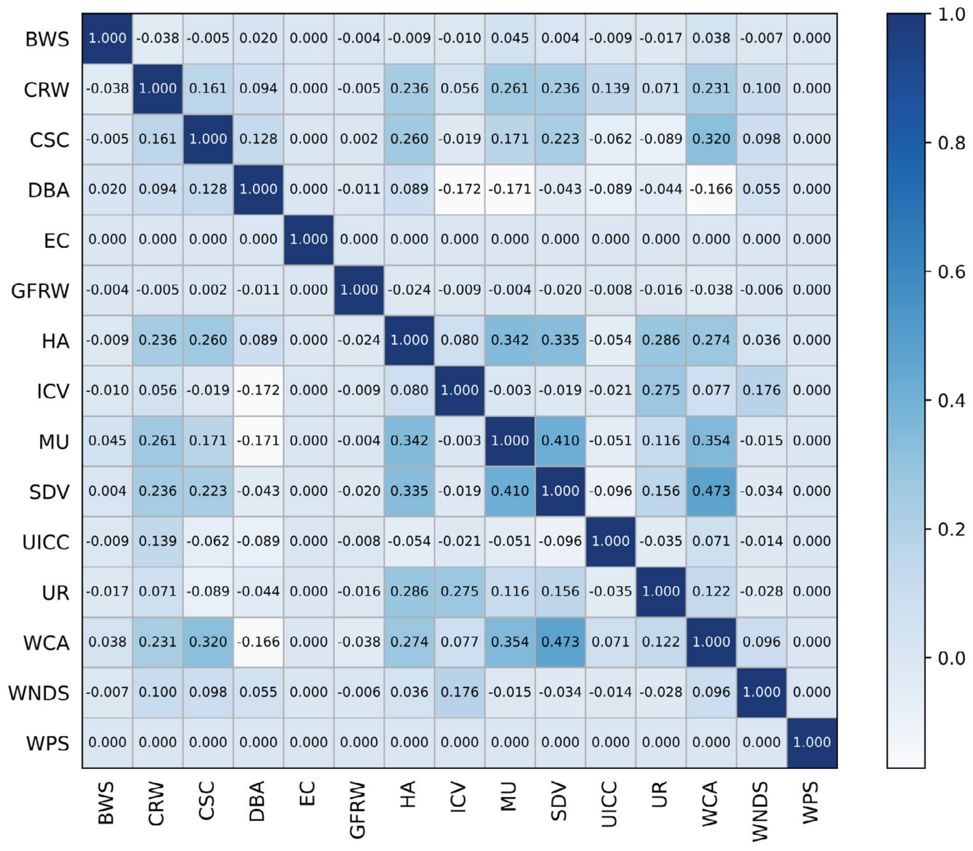


FIGURE 5 Code smell detection results statistics.



	BWS	CRW	CSC	DBA	EC	GFRW	HA	ICV	MU	SDV	UICC	UR	WCA	WNDS	WPS
BWS	1.000	-0.038	-0.005	0.020	0.000	-0.004	-0.009	-0.010	0.045	0.004	-0.009	-0.017	0.038	-0.007	0.000
CRW	-0.038	1.000	0.161	0.094	0.000	-0.005	0.236	0.056	0.261	0.236	0.139	0.071	0.231	0.100	0.000
CSC	-0.005	0.161	1.000	0.128	0.000	0.002	0.260	-0.019	0.171	0.223	-0.062	-0.089	0.320	0.098	0.000
DBA	0.020	0.094	0.128	1.000	0.000	-0.011	0.089	-0.172	-0.171	-0.043	-0.089	-0.044	-0.166	0.055	0.000
EC	0.000	0.000	0.000	0.000	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
GFRW	-0.004	-0.005	0.002	-0.011	0.000	1.000	-0.024	-0.009	-0.004	-0.020	-0.008	-0.016	-0.038	-0.006	0.000
HA	-0.009	0.236	0.260	0.089	0.000	-0.024	1.000	0.080	0.342	0.335	-0.054	0.286	0.274	0.036	0.000
ICV	-0.010	0.056	-0.019	-0.172	0.000	-0.009	0.080	1.000	-0.003	-0.019	-0.021	0.275	0.077	0.176	0.000
MU	0.045	0.261	0.171	-0.171	0.000	-0.004	0.342	-0.003	1.000	0.410	-0.051	0.116	0.354	-0.015	0.000
SDV	0.004	0.236	0.223	-0.043	0.000	-0.020	0.335	-0.019	0.410	1.000	-0.096	0.156	0.473	-0.034	0.000
UICC	-0.009	0.139	-0.062	-0.089	0.000	-0.008	-0.054	-0.021	-0.051	-0.096	1.000	-0.035	0.071	-0.014	0.000
UR	-0.017	0.071	-0.089	-0.044	0.000	-0.016	0.286	0.275	0.116	0.156	-0.035	1.000	0.122	-0.028	0.000
WCA	0.038	0.231	0.320	-0.166	0.000	-0.038	0.274	0.077	0.354	0.473	0.071	0.122	1.000	0.096	0.000
WNDS	-0.007	0.100	0.098	0.055	0.000	-0.006	0.036	0.176	-0.015	-0.034	-0.014	-0.028	0.096	1.000	0.000
WPS	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000

FIGURE 6 Correlation between Android code smells.

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

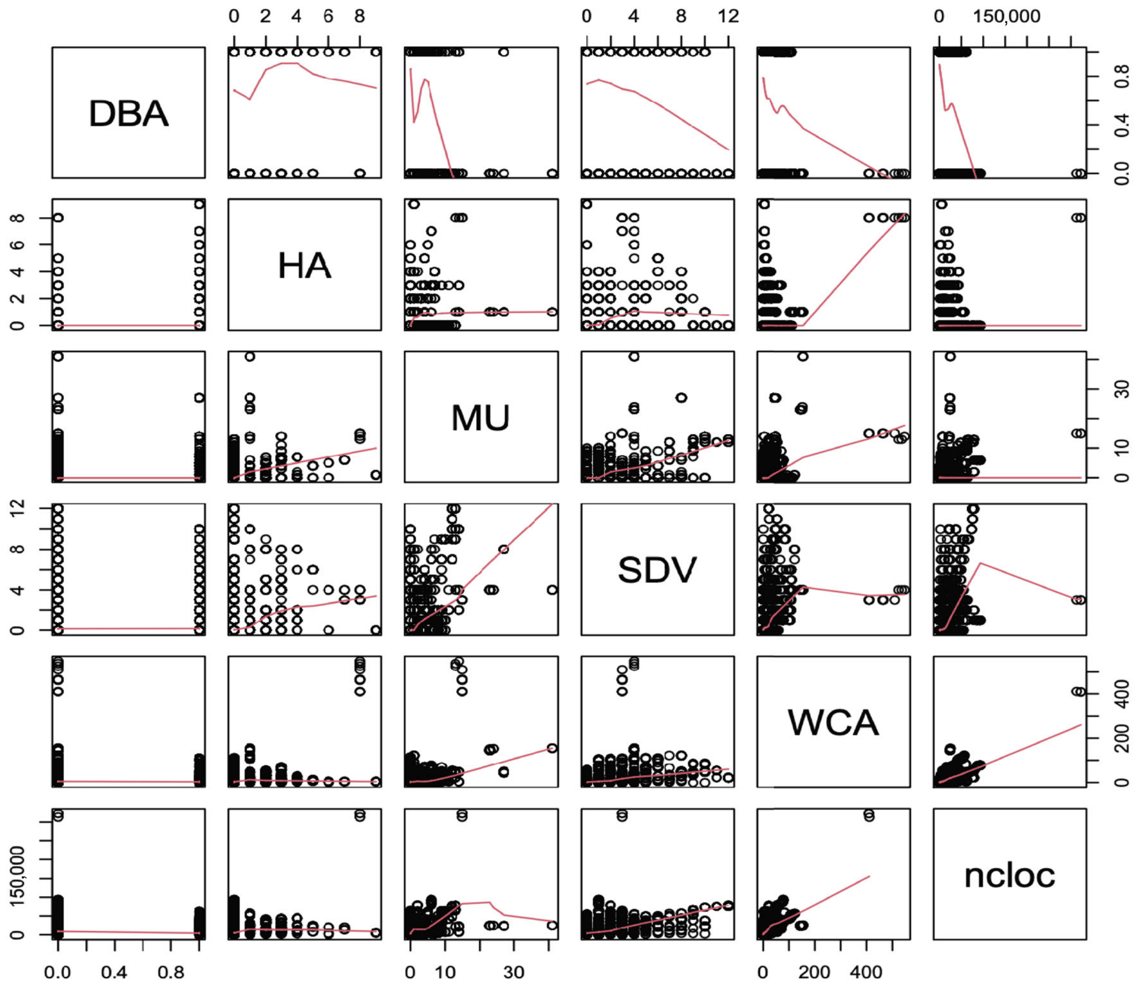


FIGURE 7 A scatter plot matrix of code smells.

TABLE 3 p-Value explanation.

p-Value	Occasionality	Original hypothesis	Statistical significance
>0.05	Probability of random error occurring occasionally >5%	Can not reject the original hypothesis	Code smell has no significant impact on the application fault counts
<0.05	Probability of random error occurring occasionally <5%	Can reject the original hypothesis	Code smell has significant impact on the application fault counts
<0.01	Probability of random error occurring occasionally <1%	Can reject the original hypothesis	Code smell has a strong significant impacton the application fault counts

4.2.2 | Experimental hypothesis

This part aims to investigate whether the five code smells targeted are more likely to be associated with fault-prone files. We adopted a two-sided hypothesis test with a significance level of $\alpha = 0.05$ to determine the model parameters by p-value. p-Value is an indicator for judging whether to accept the original hypothesis, as explained in Table 3. Specifically, we test the following hypothesis at the significance level of $p < 0.05$.

Hypothesis 1. *WCA has no effect on the fault counts in apps (either alone or in combination).*

Hypothesis 2. *HA has no effect on the faults counts in apps (either alone or in combination).*

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Hypothesis 3. *DBA has no effect on the faults counts in apps (either alone or in combination).*

Hypothesis 4. *MU has no effect on the faults counts in apps (either alone or in combination).*

Hypothesis 5. *SDV has no effect on the faults counts in apps (either alone or in combination).*

Hypothesis 6. *NCLOC has no effect on the faults counts in apps (either alone or in combination).*

4.2.3 | Experimental settings

DefDIQ constructs a simple and suitable fault counting regression model with minimal parameters under the condition of highly fitted. Hence, we apply the NB and ZINB to model the relationship between fault counts and code smells, respectively. p-Values overrode partial hypotheses to filter out some code smells and their combinations affecting faults. After repeated model evaluation and selection, they finally determined the free parameters (code smells). Specifically, the gradual regression steps for model construction are as follows.

- The first-order model contains all independent arguments, that is, five custom code smells, NCLOC, and combinations. It separates and analyzes the effect of each argument on the faults.
- Select arguments from the first-order model with significant correlation to the fault counts (p-value < 0.05) and build a simpler data model.
- Repeat the previous task, gradually constructing simpler and better-fitting models for the data.

4.2.4 | Evaluation method

To compare the fitness of the NB and ZINB regression model on the relationship between faults and code smells, we adopted the AIC and BIC values of the fault counting models under both algorithms to select the counting model with the higher fitting degree. AIC is a weighted function of fitting accuracy and parameters number, and the contribution is to find the accurate model containing the least free parameters based on information entropy. And the AIC compares the deviation of the fitted values of the fault counting model constructed by the regression function with the true values to assess the model's goodness. AIC is obtained by Equation (8).

$$AIC = -2ln(L) + 2\alpha, \tag{8}$$

where L denotes the likelihood function, and α represents the number of free parameters that can be estimated. AIC is inversely related to whether the model can better reveal a significant relationship between the response variables and arguments. A smaller AIC value indicates a better fit between the model and the true values, suggesting a higher degree of fitting for the fault counting model.

BIC is an evaluation metric in Bayesian statistics for choosing among two or more alternative models. Although the model selection criteria of AIC and BIC are distinct, there are many similarities, such as the similar interpretation of penalty terms. As demonstrated in Equations (8) and (9), when the likelihood value is larger (more accurate model), and the number of free parameters in the model is larger (simpler model), the AIC and BIC values will be smaller, indicating better model performance. For larger data volume, BIC sets a penalty parameter $ln(n)$ with a larger weight than AIC to determine the optimal parameters and alleviate the overfitting of the model. In addition, BIC also considers the sample number. When there are excessive data samples, the evaluation metric can balance the model in terms of accuracy and complexity.

$$\text{BIC} = -2ln(L) + \alpha ln(n), \tag{9}$$

where L denotes the likelihood function, α is the parameter estimated by the model, and n is the sample size (the observations or data points number). The smaller the value of the BIC, the better the model performs.

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 4 Code smell detection results.

Code smell	WCA	ICA	UCE	ISU	HA	EC	WRCE	WSP	WDNS	DBA	GFRW	PRW	MU	SDV	UR
Manual check result	158	2	1	2	12	2	4	3	6	17	15	8	5	13	57
DACS check result	163(+5)	2	2(+1)	2	8(−4)	3(+1)	5(+1)	5(+2)	8(+2)	17	16(+1)	11(+3)	6(+1)	9(−4)	61(+4)

TABLE 5 Lin's concordance of manual and detect Android code smell detection results.

est	lwr.ci	upr.ci
0.9994	0.9976	0.9996

5 | EXPERIMENTAL RESULTS

5.1 | Answer to RQ1

In this experiment, we took 15 custom Android code smells as detection objects and statistically recorded the code instances of each code smell (at the method level). DACS identified a total of 322 code smell instances, and the most common ones are WCA (163 instances), UR (61 instances), and DBA (17 instances). Statistically, the 20 open-source apps contain 2232 java files, with 283 files containing the defined code smells.

As shown in Table 4, DACS and manual detection are consistent in detecting ICV, ISU, and DBA, and the results are closer in detecting other code smells. However, DACS tended to detect more instances of code smells than manual detection. The main reason is that manual detection will combine the context-sensitive code to determine whether the developers deal with some dangerous behaviors. If there is a corresponding handling method, our programmer will not categorize this code structure as a smell, even if the code contains vulnerable fragments. DACS relies purely on matching rules for detection, which is not very flexible, so there is a certain amount of over-reporting and false judgments.

For the 15 custom Android code smells, DACS can identify almost all code smell instances in Android apps correctly. As shown in Table 5, Lin's correlation coefficient r_c reaches 0.9994 with a confidence interval (99.5%, 99.9%), indicating a high agreement degree for DACS and manual detection results. Only in two cases did the results not meet the desired level, and there will be missed detection, namely malicious unzipping code smell (MU) and header file code smell (HA).

We analyzed the unidentified smell instances by DACS and investigated why they were missed. Eventually, we discovered that the DACS missed some smells detection due to the inconsistent compression libraries used for classes affected by the MU and classes considered in the detection rules. For example, in the VINCLES project (id = 15), `com.iceteck.silicompressorr` package defines class `SiliCompressor` for video, audio, and other files (Gallerypresenter, ChatPresenter, FileUtils) for compression. However, DACS has not yet extended the SiliCompressor library, so DACS will not identify such code smells. Meanwhile, the reason for ignoring HA smells is roughly the same. While Android apps request communication with the server, except for the common request libraries HttpClient and OKhttp provided in the Smell rule library, there are also higher-level encapsulation libraries such as Feign, Retrofit, and Volley, so DACS can hardly identify some complex smells correctly.

RQ1 provides a potential method to improve the detector's performance. In the future, we will consider adding various libraries to DACS for Android developers. With the support of more third-party libraries, DACS performance will be higher.

Main findings for RQ1: Except for the omission of MU and HA, for most code smells, the detection result of DACS strongly agrees with the manual detection. Lin's concordance correlation coefficient–r_c reaches 0.99, demonstrating that DACS can effectively detect 15 custom Android code smells and capture developers' sub-optimal coding practices. Overall, the results confirm the accuracy of DACS in detecting code smells.

5.2 | Answer to RQ2

5.2.1 | Results analysis of the NB fitting model

NB is applied to analyze the relationship between fault counts and code smells. Table 6 presents the first-order interaction model, where the *Intercept* row summarizes the average effect of some not explicitly mentioned factors on faults. The remaining rows represent code smells, NCLOC, and combinations associated with the fault counts. Multiplicative interaction terms test the combinations, that is, if there are two code smells in the test file, then the multiplicative interaction result is 1 (1×1). And the interactive combinations neglect the existence of any other code smells in the files.

As shown in Table 6, the constructed first-order interaction model's residual sum of squared (RSS) over 535 Degrees of Freedom (DF) is 529.8. Since the RSS < DF, the model meets the requirements. However, Table 6 shows that NCLOC, WCA, MU, SDV, NCLOC:WCA, NCLOC:MU, NCLOC:SDV, HA:DBA, and HA:MU are the terms significantly associated with the faults (p-value < 0.05). Therefore, we select the above indicators to create a simpler model, shown in Table 7. This model has a better fit compared with the first-order interaction model, with RSS = 530.1 and DF = 547. NCLOC, WCA, MU, SDV, NCLOC:WCA, and NCLOC:MU significantly correlate with the fault counts (p-value < 0.05).

The six valid parameters discussed above were used to fit the simplest fault counting model, and the details are shown in Table 8. The model has RSS = 526.8 over DF = 550. There is a significant association (p-value < 0.05) between the faults and six code smells, that is, a significant fault-proneness to the app quality.

TABLE 6 First-order fault counting model based on negative binomial.

(Intercept)	Estimate	SE	z Value	$P_r(> \|z\|)$
	−0.2615	0.0785	−2.24	0.0138*
NCLOC	0.0013	0.0003	3.58	0.0001***
WCA	−0.5612	0.1843	−2.28	0.0019**
HA	−0.3851	0.3254	−1.08	0.1995
DBA	−0.3021	0.2134	−1.72	0.1123
MU	0.0949	0.0411	2.30	0.0116*
SDV	−0.0996	0.0384	−2.18	0.0199*
NCLOC:WCA	0.0022	0.0013	2.67	0.0035**
NCLOC:HA	0.0007	0.0006	−1.07	0.2810
NCLOC:DBA	−0.0022	0.0020	−1.61	0.1641
NCLOC:MU	−0.0001	0.0002	−2.48	0.0079**
NCLOC:SDV	0.0003	0.0001	2.39	0.0049**
WCA:HA	−0.4969	0.5088	−0.93	0.3321
WCA:DBA	0.2849	0.3899	0.81	0.4392
WCA:MU	0.0193	0.0310	0.49	0.5939
WCA:SDV	−0.0291	0.0864	−0.36	0.6895
HA:DBA	0.8248	0.4201	2.01	0.0498*
HA:MU	0.1811	0.0568	3.09	0.0018**
HA:SDV	−0.1722	0.1298	−1.24	0.2014
DBA:MU	0.0149	0.0309	0.52	0.6278
DBA:SDV	0.0187	0.1019	0.20	0.8455
MU:SDV	−0.0157	0.0101	−1.59	0.1017

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 7 Second-order fault counting model based on negative binomial.

(Intercept)	Estimate	SE	z Value	$P_r(>\|z\|)$
	−0.2499	0.0810	−3.30	0.0021**
NCLOC	0.0019	0.0003	4.68	0.0001***
WCA	−0.5838	0.2047	−2.79	0.0039**
MU	0.0841	0.0219	3.88	0.0001***
SDV	−0.1219	0.0459	−2.58	0.0082**
NCLOC:WCA	0.0019	0.0005	3.48	0.0004**
NCLOC:MU	−0.0003	0.0001	−3.29	0.0011**
NCLOC:SDV	0.0003	0.0002	1.41	0.1579
HA:DBA	−0.0479	0.2588	−0.23	0.8358
HA:MU	0.0272	0.0331	0.84	0.4158

TABLE 8 Third-order fault counting model based on negative binomial.

(Intercept)	Estimate	SE	z Value	$P_r(>\|z\|)$
	−0.2958	0.0759	−3.92	0.0001***
NCLOC	0.0028	0.0003	6.31	0.0000***
WCA	−0.5993	0.2065	−2.99	0.0035**
MU	0.0677	0.0198	3.31	0.0010***
SDV	−0.0806	0.0321	−2.66	0.0121**
NCLOC:WCA	0.0020	0.0006	3.49	0.0003**
NCLOC:MU	−0.0002	0.0001	−2.89	0.0031**

TABLE 9 Fault counting model based on zero-inflated negative binomial.

(Intercept)	Estimate	SE	z Value	$P_r(>\|z\|)$
	−0.2768	0.0821	−4.15	0.0001***
NCLOC	0.0031	0.0002	6.25	0.0000***
WCA	−0.5980	0.2001	−2.75	0.0025**
MU	0.0665	0.0172	3.28	0.0008***
SDV	−0.0146	0.0100	−4.83	0.0031**
NCLOC:WCA	0.0019	0.0005	3.38	0.0002**
NCLOC:MU	−0.0001	0.0001	−2.78	0.0028**

5.2.2 | Results analysis of the ZINB fitting model

We apply the ZINB model to investigate the distribution relationship between faults and code smells, and the modeling process is similar to NB. After obtaining the significantly correlated parameter individuals, we continue constructing the fault counting regression model until all the parameters in the model are significantly correlated. The final model results are presented in Table 9, and we omit listing all detailed model information.

As displayed in Tables 8 and 9, the results remain consistent, showing a significant correlation between six code smells and the faults in apps. However, this correlation is not always positive, some code smells (WCA, SDV, and NCLOC:MU) are negatively correlated with the fault counts. The remaining code smells (NCLOC, MU, and NCLOC:WCA) positively correlate with the fault counts. The results show that several code smell combinations increase the fault-proneness, while

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

some code smells decrease the faults. Nonetheless, we should note that the limited availability of experimental data might have contributed to this phenomenon. Many projects on GitHub are experimental and not intended for commercial purposes, lacking adequate testing, maintenance, and standardization. As a result, we may have collected limited information about both code smells and faults. Developers need to repair and refactor the code smells with positive fault correlation in time and carefully refactor the code smells with negative fault correlation.

Main findings for RQ2: The two-sided hypothesis test eliminated the code smells and combinations with no significant effect on the faults at the significance level $\alpha = 0.05$, which ultimately determined six free arguments in the model construction. The research revealed the following findings among the five code smells, (1) distinct code smells and combinations have different importance, some code smells do indicate fault-proneness, such as NCLOC, MU, and NCLOC:WCA; (2) Some smells are not directly related to faults, e.g., HA and DBA had no significant effect on the fault-proneness.

5.3 | Answer to RQ3

We evaluate the fitness for fault counts and code smells by AIC and BIC values of the fault counting model under the same dataset, and judge the performance of the regression models constructed by NB and ZINB. When the model becomes more complex (k rises, i.e., the feature number grows), the likelihood function also increases, thus reducing the AIC and BIC values. However, if the data is too heavy, this can cause overfitting issues. In the equations $AIC = 2k - 2ln(L)$, $BIC = ln(n) * k - 2ln(L)$, L represents the maximum likelihood, k refers to the free parameters for estimation (6), and n is the experimental data volume (516). The AIC and BIC values of the final NB fault counting model are 552.68 and 557.48, respectively, while for the ZINB model, the values are 517.32 and 522.12, respectively, obviously indicating that the ZINB model fits the experimental data better than the NB (both AIC_ZINB and BIC_ZINB values are lower than NB). Overall, the fault counting model constructed by ZINB performed better than the one based on NB.

For a more intuitive view of the extent to which code smell affects fault counts, we construct the repair priority level according to the best-fitting fault counting model on ZINB. Specifically, considering the direct correlation of the coefficients of the arguments with the faults in the model, we take them as the influence factors of the arguments and map the coefficients to the interval [0,1] by the Sigmoid function. The Sigmoid function is represented by 10:

$$f(x) = \frac{1}{1 + e^{-x}}. \tag{10}$$

According to the ZINB fault counting model, we obtained coefficients representing the importance levels of three independent code smells (WCA, MU, SDV), and then calculated the importance value by S-function. The importance values for the remaining smells were derived by the Spearman correlation coefficient, where correlations below 0.1 were considered as no correlation. Therefore, we calculated the importance-value of HA, CSC, CRW, UR, DBA, ICV, UICC, and WNDS based on the spearman part >0.1. Importance values for EC, WPS, BWS, and GFRW could not be obtained due to a lack of research data and low correlation. We assigned these smells the lowest repair level considering their occurrence frequency in the files. The repair priority levels of the 15 code smells are presented in Table 10.

Main findings for RQ3: Among the two fault counting models constructed by NB and ZINB regression algorithms, the ZINB-based fault counting model is more suitable for the dataset under work (AIC=517.32, BIC=522.12). We generate fault repair priority levels based on the ZINB fault counting model by quantifying the code smells impact on the faults. This assists developers and researchers focus on fault-prone code smells. The priority list drives the building of high-quality and low-debt apps by refactoring poor designs with high risk first, thereby increasing the repair benefits.

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 10 Repair priority level on zero-inflated negative binomial fault counting model.

Code smell	Importance value	Repair priority level
MU	0.5116	I
CRW	0.5069	II
SDV	0.4964	III
HA	0.4405	IV
CSC	0.4273	V
WCA	0.3548	VI
UR	0.3065	VII
ICV	0.0842	VIII
UICC	0.0706	IX
DBA	0.0549	X
WNDS	0.0148	XI
EC, WPS, BWS, GFRW	empty	XII

6 | THREATS TO VALIDITY

6.1 | Threats to internal validity

Code smell is an abstract definition of poor design by researchers from different practice results, and the definition is subjective and lacks strict criteria. Due to varying opinions on what constitutes an Android code smell, practitioners often rely on their own experiences to define them. To achieve broad acceptance, we carefully investigate technical blogs and conferences related to Android security and code specification, provide the Code Smell Definition, and targeted repair suggestions based on programming practices.

6.2 | Threats to external validity

The experimental apps selected for this study are obtained from GitHub, and the validity of the experimental findings for other communities is debatable. Android apps depend on third-party libraries and are developed on a small scale with rapid iterations, and there is no public database for faults. When obtaining fault data, irregular commits by developers lead to the less reliable classification of commits with matching fault keywords as faults. Besides, some projects on GitHub are immature, and most do not follow the complete software lifecycle steps. The project development is not standardized with inadequate testing and maintenance, resulting in the scare fault information and smell data. And the limited dataset under test resulted in a single-digit number of identified code smells. However, given that our apps are from different domains and sizes, we believe the findings still provide valuable information for code smell and app quality studies.

7 | RELATED WORKS

7.1 | Code smell research and detection

Code smell detection is a technique for analyzing and understanding the defined poor design in apps by extracting code metrics from the source code to explore fault-prone design issues. Thus, detection methods, as well as tools, have been continuously adopted to detect code smells in apps.[35-37] Traditional Java static code metric detection is unable to precisely focus on poor design and implementation in Android apps for Android features. With the maturity of Android technology, researchers started to put more effort into detecting Android code smells. Huang et al.[38] proposed four mapping metrics

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

and implemented a static detection tool, HBSniff, to detect 14 code smells in object-oriented programming. Mario et al.[39] implemented DECOR to detect several object-oriented code smells in apps. Hecht et al. developed PAPRIKA to identify four Android-specific code smells based on manually defined detection rules.[40] Palomba et al.[41] implemented aDoctor to detect 15 Android-specific code smells with the abstract syntax of the source code. Fatima et al. extended custom rules based on the AL to detect and correct 12 Android code smells, which ultimately outperformed PAPRIKA.[42] Ghafari et al. investigated 28 Android code smells indicating security and discussed relevant elimination methods.[2]

7.2 | Effects of smell on faults and security

To quantify the influence of Android code smell on app quality, fault counts are often used as an indicator. The fault prediction models have been researched to locate faulty code, thereby improving software quality and using resources better. Hall et al.[43] analyzed 36 fault-related works systematically and found that source code-based spam filtering techniques performed well in prediction.[44,45] Additionally, there are empirical studies involving bug triggering, based on TensorFlow analysis of bug types, distribution, etc.[46] While static code metrics have also contributed to fault prediction, their individual predictive performance is relatively poor. Zhou et al. applied various complexity metrics to logistic regression models and found that NCLOC outperformed other metrics.[47] Furthermore, fault prediction models using only the NCLOC perform better than other source code metrics, and LOC positively correlates with fault counts. Ostrand et al.[48] and Hongyu et al.[49] both reported that NCLOC is a generic, simple and effective fault prediction metric. Although some researches indicate that the predictive performance of NCLOC is inferior to other metrics, overall, NCLOC has significant advantages in fault prediction models.[50,51] Tang et al.[52] studied the security threats posed by link hijacking attacks based on the weaknesses of the existing app link mechanism in apps, and provided corresponding detection and defense methods. Sazzadur et al.[53] also focused on vulnerabilities in Java projects, researching the misuse of encryption APIs and vulnerable certificate verification. Security-oriented code smell is a code feature that can be captured at the code structure level and may lead to security issues. They serve as a predictive measure for security risks. Currently, more research focuses on faults and safety issues, so we concentrate our studies on how to avoid security problems caused by nonstandard implementations from a code structure perspective.

8 | CONCLUSIONS

Software faces multiple security and quality threats within the continuous development of attack and counter-attack techniques. The security of Android apps with a high market share is even more critical, given the vast amount of sensitive data involved. To reduce the vulnerable code practice, we promote security design practices. In this paper, we customize 15 novel Android code smells and provide targeted suggestions for eliminating or mitigating them. Then we developed a lightweight tool to complete automatic detection and assess their prevalence. We investigated the distribution between code smells and faults by constructing fault counting models, then generated repair priority levels based on the experimental findings. The majority of the app datasets collected contained at least four security smells, and we believe that the identified code smells are a valuable indicator of security and quality issues. The experimental results show that code smells do have an impact on the faults in apps, and the code smell repair priority generated can serve as a practical baseline for researchers and inexperienced developers.

In the future research works, we are committed to a more comprehensive study involving DACS performance repair, a more extensive Android apps faults repository, and the validation of repair suggestions, realizing more mature research on the topic.

AUTHOR CONTRIBUTIONS

Yi Zhong facilitated the acquisition of financial support for the project leading to this publication, and participated in reviewing and editing the manuscript, polishing the paper, as well as responding to review comments. Mengyu Shi was responsible for designing the methodology, crafting the orginal drafts (including substantial translation work), and polising the paper. This encompassed conceptualizing experimental ideas, visualizing experimental results, designing experiments, and engaging in the manuscript's writing, reviewing, and editing process. Jiawei He conducted research and

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

investigations, with specific focus or experiment execution and data/evidence collection. Additionally, he also carried out data analysis and the validation of experiments.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their insightful comments.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon reasonable request.

ORCID

Mengyu Shi https://orcid.org/0000-0002-3720-6075
Chunrong Fang https://orcid.org/0000-0002-9930-7111
Zhenyu Chen https://orcid.org/0000-0002-9592-7022

REFERENCES

1. Lewowski T, Madeyski L. How far are we from reproducible research on code smell detection? A systematic literature review. *Inform Softw Technol*. 2022;144:106783. doi:10.1016/j.infsof.2021.106783
2. Ghafari M, Gadient P, Nierstrasz O. Security smells in android. Paper presented at: 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM); 2017; Shanghai, China:121-130.
3. StatCounter. Mobile operating system market share worldwide; 2022. https://gs.statcounter.com/os-market-share/mobile/worldwide. Accessed October 6, 2022.
4. Senanayake J, Kalutarage H, Al-Kadri MO, Petrovski A, Piras L. Developing secured android applications by mitigating code vulnerabilities with machine learning. Paper presented at: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security (ASIA CCS'22), Nagasaki, Japan; 2022:1255-1257.
5. Alkandari MA, Kelkawi A, Elish MO. An empirical investigation on the effect of code smells on resource usage of android Mobile applications. *IEEE Access*. 2021;9:61853-61863. doi:10.1109/ACCESS.2021.3075040
6. Boutaib S, Bechikh S, Palomba F, Elarbi M, Makhlouf M, Said LB. Code smell detection and identification in imbalanced environments. *Expert Syst Applic*. 2021;166:114076. doi:10.1016/j.eswa.2020.114076
7. Fowler M. Refactoring: improving the design of Existing Code. Paper presented at: Extreme Programming and Agile Methods - XP/Agile Universe 2002, Managua, Nicaragua; 2002:256.
8. Brown WH, Malveau RC, McCormick HWS, Mowbray TJ. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc.; 1998.
9. Habchi S, Moha N, Rouvoy R. Android code smells: from introduction to refactoring. *J Syst Softw*. 2021;177:110964. doi:10.1016/j.jss.2021.110964
10. Hall T, Zhang M, Bowes D, Sun Y. Some code smells have a significant but small effect on faults. *ACM Trans Softw Eng Methodol*. 2014;23(4):1-39. doi:10.1145/2629648
11. Sobrinho EVP, De Lucia A, Maia MA. A systematic literature review on bad smells–5 W's: which, when, what, who, where. *IEEE Trans Softw Eng*. 2021;47(1):17-66. doi:10.1109/TSE.2018.2880977
12. Walter B, Alkhaeir T. The relationship between design patterns and code smells: an exploratory study. *Inform Softw Technol*. 2016;74:127-142. doi:10.1016/j.infsof.2016.02.003
13. Fenske W, Schulze S. Code smells revisited: a variability perspective. *Proceedings of the 9th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'15)*. Association for Computing Machinery; 2015:3-10.
14. Xian Zhang JZ. A slice-granularity defect prediction method based on code naturality. *Ruan Jian Xue Bao/J Softw*. 2021;32(7):2219-2241.
15. Viega J, McGraw G, Mutdosch T, Felten EW. Statically scanning Java code: finding security vulnerabilities. *IEEE Softw*. 2000;17(5):68-74. doi:10.1109/52.877869
16. Lu Z, Mukhopadhyay S. Model-based static source code analysis of java programs with applications to android security. Paper presented at: 2012 IEEE 36th Annual Computer Software and Applications Conference; 2012; Izmir, Turkey:322-327.
17. Mannan UA, Ahmed I, Almurshed RAM, Dig D, Jensen C. Understanding code smells in android applications. Paper presented at: Proceedings of the International Conference on Mobile Software Engineering and Systems (MOBILESoft'16); 2016; New York, NY:225-234.
18. Hecht G, Rouvoy R, Moha N, Duchien L. Detecting antipatterns in android apps. Paper presented at: 2nd ACM International Conference on Mobile Software Engineering and Systems; 2015; Piscataway, NJ:148-149.
19. Jan Reimann UA. A tool-supported quality smell catalogue for android developers. *Softw Trends*. 2014;34(2):1-2.
20. Hecht G, Moha N, Rouvoy R. An empirical study of the performance impacts of android code smells. Paper presented at: Proceedings of the International Conference on Mobile Software Engineering and Systems (MOBILESoft'16); 2016:59-69.
21. Greenwood M, Yule GU. An inquiry into the nature of frequency distributions representative of multiple happenings with particular reference to the occurrence of multiple attacks of disease or of repeated accidents. *J R Stat Soc*. 1920;83(2):255-279.
22. Evertt B. *Distributions in Statistics: Discrete Distributions*. Wiley Online Library; 1970;482-483.

23. Greene WH. Accounting for Excess Zeros and Sample Selection in Poisson and Negative Binomial Regression Models. Technical Report. New York University. 1994.
24. Caudill SB. Estimating the circle closest to a set of points by maximum likelihood using the BHHH algorithm. *Eur J Oper Res*. 2006;172(1):120-126. doi:10.1016/j.ejor.2004.09.031
25. Fieller EC, Hartley HO, Pearson ES. Tests for rank correlation coefficients. *I Biometrika*. 1957;44(3/4):470-481.
26. Gong A, Zhong Y, Zou W, Shi Y, Fang C. Incorporating android code smells into Java static code metrics for security risk prediction of android applications. Paper presented at: 2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS), Vilnius, Lithuania; 2020:30-40.
27. Fang C, Liu Z, Shi Y, Huang J, Shi Q. Functional code clone detection with syntax and semantics fusion learning. Paper presented at: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020), Virtual Event; 2020:516-527.
28. Rathore SS, Kumar S. A study on software fault prediction techniques. *Artif Intell Rev*. 2019;51(2):255-327. doi:10.1007/s10462-017-9563-5
29. Kim S, Zimmermann T, Whitehead EJ, Zeller A. Predicting faults from cached history. Paper presented at: 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN; 2007:489-498.
30. Fleiss JL, Cohen J, Everitt BS. Large sample standard errors of kappa and weighted kappa. *Psychol Bull*. 1969;72(5):323.
31. Lawrence I, Lin K. A concordance correlation coefficient to evaluate reproducibility. *Biometrics*. 1989;45(1989):255-268. doi:10.2307/2532051
32. Muse BA, Rahman MM, Nagy C, Cleve A, Khomh F, Antoniol G. On the prevalence, impact, and evolution of SQL code smells in data-intensive systems. CoRR. 2022; abs/2201.02215. https://arxiv.org/abs/2201.02215
33. Li W, Shatnawi R. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *J Syst Softw*. 2007;80(7):1120-1128. doi:10.1016/j.jss.2006.10.018
34. Olbrich S, Cruzes DS, Basili V, Zazworka N. The evolution and impact of code smells: a case study of two open source systems. Paper presented at: 3rd International Symposium on Empirical Software Engineering and Measurement; 2009:390-400.
35. Moha N, Guéhéneuc YG, Duchien L, Le Meur AF. Decor: a method for the specification and detection of code and design smells. *IEEE Trans Softw Eng*. 2009;36(1):20-36. doi:10.1109/TSE.2009.50
36. Patnaik A, Padhy N. A hybrid approach to identify code smell using machine learning algorithms. *Int J Open Source Softw Process*. 2021;12(2):21-35. doi:10.4018/IJOSSP.2021040102
37. Sharma T, Efstathiou V, Louridas P, Spinellis D. Code smell detection by deep direct-learning and transfer-learning. *J Syst Softw*. 2021;176:110936. doi:10.1016/j.jss.2021.110936
38. Huang Z, Shao Z, Fan G, Yu H, Yang K, Zhou Z. HBSniff: a static analysis tool for Java hibernate object-relational mapping code smell detection. *Sci Comput Program*. 2022;217:102778. doi:10.1016/j.scico.2022.102778
39. Linares-Vásquez M, Klock S, McMillan C, Sabané A, Poshyvanyk D, Guéhéneuc YG. Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in Java Mobile apps. Paper presented at: Proceedings of the 22nd International Conference on Program Comprehension (ICPC 2014), Hyderabad, India; 2014:232-243.
40. Hecht G, Benomar O, Rouvoy R, Moha N, Duchien L. Tracking the software quality of android applications along their evolution (t). Paper presented at: 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, Nebraska; 2015:236-247.
41. Palomba F, Di Nucci D, Panichella A, Zaidman A, De Lucia A. Lightweight detection of android-specific code smells: the adoctor project. Paper presented at: 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Klagenfurt, Austria; 2017:487-491.
42. Fatima I, Anwar H, Pfahl D, Qamar U. Detection and correction of android-specific code smells and energy bugs: an android lint extension. Vol. 2020. APSEC, Singapore; 2020:71-78.
43. Hall T, Beecham S, Bowes D, Gray D, Counsell S. A systematic literature review on fault prediction performance in software engineering. *IEEE Trans Softw Eng*. 2011;38(6):1276-1304. doi:10.1109/TSE.2011.103
44. Mizuno O, Ikami S, Nakaichi S, Kikuno T. Spam filter based approach for finding fault-prone software modules. Paper presented at: MSR'07: ICSE, Minneapolis, MN; 2007:4.
45. Mizuno O, Kikuno T. Training on errors experiment to detect fault-prone software modules by spam filter. Paper presented at: Esec-Fse'07. The organization, New York, NY; 2007:405-414.
46. Du X, Xiao G, Sui Y. *Fault Triggers in the TensorFlow Framework: An Experience Report*. ISSRE; 2020:1-12.
47. Zhou Y, Xu B, Leung H. On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. *J Syst Softw*. 2010;83(4):660-674. doi:10.1016/j.jss.2009.11.704
48. Ostrand TJ, Weyuker EJ, Bell RM. Where the bugs are. *ACM SIGSOFT Softw Eng Notes*. 2004;29(4):86-96. doi:10.1145/1013886.1007524
49. Zhang H. An investigation of the relationships between lines of code and defects. Paper presented at: 2009 IEEE International Conference on Software Maintenance, Alberta, Canada; 2009:274-283.
50. Bell RM, Ostrand TJ, Weyuker EJ. Looking for bugs in all the right places. Paper presented at: Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA'06), Portland, Maine; 2006:61-72.
51. Olague HM, Etzkorn LH, Messimer SL, Delugach HS. An empirical validation of object-oriented class complexity metrics and their ability to predict error-prone classes in highly iterative, or agile, software: a case study. *J Softw Maintenance Evol Res Pract*. 2008;20(3):171-197. doi:10.1002/smr.366

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

52. Tang Y, Sui Y, Wang H, Luo X, Zhou H, Xu Z. All your app links are belong to us: understanding the threats of instant apps based attacks. Paper presented at: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020), California; 2020:914-926.
53. Rahaman S, Xiao Y, Afrose S, et al. CryptoGuard: high precision detection of cryptographic vulnerabilities in massive-sized Java projects. Paper presented at: 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK; 2019:2455-2472.

How to cite this article: Zhong Y, Shi M, He J, Fang C, Chen Z. Security-based code smell definition, detection, and impact quantification in Android. *Softw Pract Exper*. 2023;53(11):2296-2321. doi: 10.1002/spe.3257

1097024x, 2023, 11, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.3257 by <Shibboleth>-member@javeriana.edu.co, Wiley Online Library on [11/08/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License