Empirical Software Engineering
https://doi.org/10.1007/s10664-018-9673-y

Security code smells in Android ICC

Pascal Gadient[1] · Mohammad Ghafari[1] ·
Patrick Frischknecht[1] · Oscar Nierstrasz[1]

Published online: 14 December 2018
© Springer Science+Business Media, LLC, part of Springer Nature 2018

Abstract
Android Inter-Component Communication (ICC) is complex, largely unconstrained, and hard for developers to understand. As a consequence, ICC is a common source of security vulnerabilities in Android apps. To promote secure programming practices, we have reviewed related research, and identified avoidable ICC vulnerabilities in Android-run devices and the security code smells that indicate their presence. We explain the vulnerabilities and their corresponding smells, and we discuss how they can be eliminated or mitigated during development. We present a lightweight static analysis tool on top of Android Lint that analyzes the code under development and provides just-in-time feedback within the IDE about the presence of such smells in the code. Moreover, with the help of this tool we study the prevalence of security code smells in more than 700 open-source apps, and manually inspect around 15% of the apps to assess the extent to which identifying such smells uncovers ICC security vulnerabilities.

Keywords Security code smells · Vulnerability · Static analysis · Android

1 Introduction

Smartphones and tablets provide powerful features once offered only by computers. However, the risk of security vulnerabilities on these devices is tremendous: smartphones are increasingly used for security-sensitive services like e-commerce, e-banking, and personal healthcare, which make these multi-purpose devices an irresistible target of attack for criminals.

Communicated by: Coen de Roover, David Lo and Jianjun Zhao

✉ Pascal Gadient
gadient@inf.unibe.ch

Mohammad Ghafari
ghafari@inf.unibe.ch

Patrick Frischknecht
patrick.frischknecht@students.unibe.ch

Oscar Nierstrasz
oscar@inf.unibe.ch

[1] Software Composition Group, University of Bern, Bern, Switzerland

A recent survey in the Stack Overflow website shows that about 65% of mobile developers work with Android (Stack Overflow developer survey results: http://insights.stackoverflow.com/survey/2017). This platform has captured over 80% of the smartphone market, and just its official app store contains more than 2.8 million apps. As a result, a security mistake in an in-house app may jeopardize the security and privacy of billions of users.

The security of smartphones has been studied from various perspectives such as the device manufacturer (Lei et al. 2013), its platform (Meng et al. 2016), and end users (Jones and Chin 2015). Numerous security APIs, protocols, guidelines, and tools have been proposed. Nevertheless security concerns are often overridden by other concerns (Balebako and Cranor 2014). Many developers undermine their significant role in providing security (Xie et al. 2011). As a result, security issues in mobile apps continue to proliferate unabated (The ultimate security vulnerability datasource: http://www.cvedetails.com).

Given this situation, in previous work we identified 28 security code smells, i.e., symptoms in the code that signal potential security vulnerabilities (Ghafari et al. 2017). We studied the prevalence of ten such smells, and realized that despite the diversity of apps in popularity, size, and release date, the majority suffer from at least three different security smells, and such smells are in fact good indicators of actual security vulnerabilities.

To promote the adoption of secure programming practices, we build on our previous work, and identify security smells related to Android Inter-Component Communication (ICC). Android ICC is complex, largely unconstrained, and hard for developers to understand, and it is consequently a common source of security vulnerabilities in Android apps.

We have reviewed state-of-the-art papers in security and existing benchmarks for Android vulnerabilities, and identified twelve security code smells pertinent to ICC vulnerabilities. In this paper we present these vulnerabilities and their corresponding smells in the code, and discuss how they could be eliminated or mitigated during development. We present a lightweight static analysis tool on top of Android Lint that analyzes the code under development, and provides just-in-time feedback within the IDE about the presence of such security smells in the code. Moreover, with the help of this tool we study the prevalence of security code smells in more than 700 open-source apps, and discuss the extent to which identifying these smells can uncover actual ICC security vulnerabilities. We address the following three research questions:

- **RQ$_1$**: *What are the known ICC security code smells?* We have reviewed related work, especially that appearing in top-tier conferences and journals, and identified twelve avoidable ICC vulnerabilities and the code smells that indicate their presence. We discuss each smell, the risk associated with it, and its mitigation during app development.
- **RQ$_2$**: *How prevalent are the smells in benign apps?* We have developed a tool that statically analyzes apps for the existence of ICC security smells, and we applied it to a repository of 732 apps, mostly available on GitHub. We discovered that almost all apps suffer from at least one category of ICC security smell, but fewer than 10% suffer from more than two categories of such smells. Interestingly, only small teams appear to be capable of consistently building software resistant to most security code smells. Furthermore, long-lived projects have more issues than recently created ones, and updates rarely have any impact on ICC security.
- **RQ$_3$**: *To which extent does identifying security smells facilitate detection of security vulnerabilities?* We inspected the identified smells in 100 apps, and verified whether they correspond to any security vulnerabilities. Our investigation showed that about half of the identified smells are in fact good indicators of security vulnerabilities.

To summarize, this work represents an effort to spread awareness about the impact of programming choices in making apps secure, and to fundamentally reduce the attack surface of ICC APIs in Android. We argue that this helps developers who develop security mechanisms to identify frequent problems, and also provides developers inexperienced in security with caveats about the prospect of security issues in their code. Existing analysis tools often overwhelm developers with too many identified issues at once. In contrast we provide feedback during app development where developers have the relevant context. Such feedback makes it easier to react to issues, and helps developers to learn from their mistakes (Tymchuk et al. 2018). This paper goes beyond our earlier work (Ghafari et al. 2017) by (i) providing a completely new study on ICC vulnerabilities, one of the most prevalent Android security issues, and identifying the corresponding security smells, (ii) providing more precise, while still lightweight, static analysis tool support to identify such smells, (iii) integrating our analysis into Android Lint, thus providing just-in-time feedback to developers, (iv) experimentation on a new dataset of open-source Android apps, and (v) open-sourcing the lint checks as well as the analyzed data.[1]

The remainder of this paper is organized as follows. We provide the necessary background about the Android OS and ICC risks from which Android apps suffer in Section 2. We introduce ICC-related security code smells in Section 3, followed by our empirical study in Section 4. We provide a brief overview of the related work in Section 5, before concluding the paper in Section 6.

2 Background

This section covers the necessary background in the Android platform, and briefly presents common security threats in the context of ICC scenarios.

2.1 Android Architecture

Android is the most popular operating system (OS) for smartphones and other types of mobile devices. It provides a rich set of APIs for app developers to access common features on mobile devices.

An Android app consists of an .apk file containing the compiled bytecode, any needed data, and resource files. The Android platform assigns a unique user identifier (UID) to each app at installation time, and runs it in a unique process within a sandbox so that every app runs in isolation from other apps. Moreover, access to sensitive APIs is protected by a set of permissions that the user can grant to an app. In general, these permissions are text strings that correlate to a specific access grant, e.g., `android.permission.CAMERA` for camera access.

Four types of components can exist in an app: activities, services, broadcast receivers, and content providers. In a nutshell:

- *Activities* build the user interface of an app, and allow users to interact with the app.
- *Services* run operations in the background, without a user interface.
- *Broadcast receivers* receive system-wide “intents”, i.e., descriptions of operations to be performed, sent to multiple apps. Broadcast receivers act in the background, and often relay messages to activities or services.

[1] We are collaborating with Google to officially integrate these checks into Android Studio.

- *Content providers* manage access to a repository of persistent data that could be used internally or shared between apps.

The OS and its apps, as well as components within the same or across multiple apps, communicate with each other via ICC APIs. These APIs take an *intent object* as a parameter. An intent is either *explicit* or *implicit*. In an explicit intent, the source component declares to which target component (i.e., Class or ComponentName instances) the intent is sent, whereas in an implicit intent, the source component only specifies a general action to be performed (i.e., represented by a text string), and the target component that will receive the intent is determined at runtime. Intents can optionally carry additional data also called *bundles*. Components declare their ability to receive implicit intents using “intent filters”, which allow developers to specify the kinds of actions a component supports. If an intent matches any intent filter, it can be delivered to that component.

2.2 ICC Threats

ICC not only significantly contributes to the development of collaborative apps, but it also poses a common attack surface. The ICC-related attacks that threaten Android apps are:

- **Denial of Service** Unchecked exceptions that are not caught will usually cause an app to crash. The risk is that a malicious app may exploit such programming errors, and perform an inter-process denial-of-service attack to drive the victim app into an unavailable state.
- **Intent Spoofing** In this scenario a malicious app sends forged intents to mislead a receiver app that would otherwise not expect intents from that app.
- **Intent Hijacking** This threat is similar to a man-in-the-middle attack where a malicious app, registered to receive intents, intercepts implicit intents before they reach the intended recipient, and without the knowledge of the intent’s sender and receiver.

Two major consequences of ICC attacks are as follows:

- **Privilege Escalation** The security model in Android does not by default prevent an app with fewer permissions (low privilege) from accessing components of another app with more permissions (high privilege). Therefore, a caller can escalate its permissions via other apps, and indirectly perform unauthorized actions through the callee.
- **Data Leak** A data leak occurs when private data leaves an app and is disclosed to an unauthorized recipient.

3 ICC Security Code Smells

In this section we present the guidelines we followed to derive the security code smells from previous research. Finally, we explain each security smell in detail.

3.1 Literature Review

Although Android security is a fairly new field, it is very active, and researchers in this area have published a large number of articles in the past few years. In order to answer the first research question (RQ_1), and to draw a comprehensive picture of recent ICC smells and their corresponding vulnerabilities, our study builds on two pillars, i.e., a literature review and a benchmark inspection.

We were essentially interested in any paper that matches our scope, i.e., explaining an ICC-related issue, and any countermeasures that involve ICC communication in Android.

For our analysis we considered multiple online repositories, such as IEEE Xplore and the ACM Digital Library, as well as the Google Scholar search engine. In each repository we formulated a search query comprising *Android*, *ICC*, *IPC* and any other security-related keywords such as *security*, *privacy*, *vulnerability*, *attack*, *exploit*, *breach*, *leak*, *threat*, *risk*, *compromise*, *malicious*, *adversary*, *defence*, or *protect*. In addition to increase our potential coverage on Android security, we also collected all related publications in recent editions of well-known software engineering venues like the *International Conference on Software Engineering (ICSE)*. This search led initially to 358 publications.

In order to retrieve only relevant information that lies within our scope, i.e., *Android application level ICC security*, we first read the title and abstract, and if the paper was relevant we continued reading other parts.

This process led to the inclusion of 47 papers in our study. We recursively checked both citations and cited papers until no new related papers were found. This added six new relevant papers in our list that in the end contained 53 relevant papers for an in-depth study, out a total of 430 papers. During the whole process, which was undertaken by two authors of this paper, we resolved any disagreement by discussions. The list of included papers in this study is available on the GitHub page of the project.[2]

We further studied the well-known DroidBench[3] and Ghera[4] benchmarks for our evaluation, both built with a focus on ICC. We relied on their technical implementation, or description where possible, to extract the desired information, i.e., issues under test, symptoms, and vulnerabilities. The inspection of these two benchmarks served two different purposes: on the one hand we wanted to ensure there are no smells that we might have missed to include in our list. On the other hand, we wanted to rely on some ground truth while explaining and examining the vulnerability capabilities of the smells.

3.2 List of Smells

We have identified twelve ICC security code smells that are listed in Table 1. For each smell we report the security *issue* at stake, the potential security *consequences* for users, the *symptom* in the code (i.e., the code smell), the *detection* strategy that has been implemented by our tool for identifying the code smell, any *limitations* of the detection strategy, and a recommended *mitigation* strategy of the issue, principally for developers.

We mined this information from numerous publications and benchmark suites, but only few of these resources provided detailed information about a given security issue. Therefore we put in a great manual effort to provide a comprehensive description for each smell, while consulting other resources such as the official Android documentation and external experts. For instance, authors who focused on vulnerability detection generally neglected the aspect of mitigation. This is very problematic, since it is very common for ICC-related issues to share strong similarities with only subtle differences, e.g., regular directed inter-app communication and broadcasts both rely on intents. Furthermore, manifold vulnerability terms that are used in the literature insufficiently reflect the symptoms as they do not name

[2]https://github.com/pgadient/AndroidLintSecurityChecks

[3]https://github.com/secure-software-engineering/DroidBench

[4]https://bitbucket.org/secure-it-i/android-app-vulnerability-benchmarks

Table 1 The identified ICC security code smells

ID	Security code smells	ID	Security code smells
SM01	Persisted dynamic permission	SM07	Broken service permission
SM02	Custom scheme channel	SM08	Insecure path permission
SM03	Incorrect protection level	SM09	Broken path permission precedence
SM04	Unauthorized intent	SM10	Unprotected broadcast receiver
SM05	Sticky broadcast	SM11	Implicit pending intent
SM06	Slack WebViewClient	SM12	Common task affinity

the involved component, e.g., “Confused Deputy” instead of “Unauthorized Intent”. Better naming conventions would greatly ease the understanding of security vulnerabilities.

– **SM01: Persisted Dynamic Permission** Android provides access to protected resources through a Uniform Resource Identifier (URI) to be granted at runtime.

  Issue: Such dynamic access is intended to be temporary, but if the developer forgets to revoke a permission, the access grant becomes more durable than intended.

  Consequently, the recipient of the granted access obtains long-term access to potentially sensitive data.

  Symptom: `Context.grantUriPermission()` is present in the code without a corresponding `Context.revokeUriPermission()` call.

  Detection: We report the smell when we detect a permission being dynamically granted without any revocations in the app.

  Limitation: Our implementation does not match a specific grant permission to its corresponding revocation. We may therefore fail to detect a missing revocation if another revocation is present somewhere in the code.

  Mitigation: Developers have to ensure that granted permissions are revoked when they are no longer needed. They can also attach sensitive data to the intent instead of providing its URI.

– **SM02: Custom Scheme Channel** A *custom scheme* allows a developer to register an app for custom URIs, e.g., URIs beginning with `myapp://`, throughout the operating system once the app is installed. For example, the app could register an activity to respond to the URI via an intent filter in the manifest. Therefore, users can access the associated activity by opening specific hyperlinks in a wide set of apps.

  Issue: Any app is potentially able to register and handle any custom schemes used by other apps.

  Consequently, malicious apps could access URIs containing access tokens or credentials, without any prospect for the caller to identify these leaks (Wang et al. 2013).

  Symptom: If an app provides custom schemes, then a scheme handler exists in the manifest file or in the Android code. If the app calls a custom scheme, there exists an intent containing a URI referring to a custom scheme.

  Detection: The `android:scheme` attribute exists in the `intent-filter` node of the manifest file, or `IntentFilter.addDataScheme()` exists in the source code.

  Limitation: We only check the symptoms related to receiving custom schemes.

  Mitigation: Never send sensitive data, e.g., access tokens via such URIs. Instead of custom schemes, use system schemes that offer restrictions on the intended recipients.

The Android OS could maintain a verified list of apps and the schemes that are matched when there is such a call.

- **SM03: Incorrect Protection Level** Android apps must request permission to access sensitive resources. In addition, custom permissions may be introduced by developers to limit the scope of access to specific features that they provide based on the protection level given to other apps. Depending on the feature, the system might grant the permission automatically without notifying the user, i.e., signature level, or after the user approval during the app installation, i.e., normal level, or may prompt the user to approve the permission at runtime, if the protection is at a dangerous level.

  Issue: An app declaring a new permission may neglect the selection of the right protection level, i.e., a level whose protection is appropriate with respect to the sensitivity of resources (Mitra and Ranganath 2017).

  Consequently, apps with inappropriate permissions can still use a protected feature.

  Symptom: Custom permissions are missing the right `android:protectionLevel` attribute in the manifest file.

  Detection: We report missing protection level declarations for custom permissions.

  Limitation: We cannot determine if the level specified for a protection level is in fact right.

  Mitigation: Developers should protect sensitive features with dangerous or signature protection levels.

- **SM04: Unauthorized Intent** Intents are popular as one way requests, e.g., sending a mail, or requests with return values, e.g., when requesting an image file from a photo library. Intent receivers can demand custom permissions that clients have to obtain before they are allowed to communicate. These intents and receivers are “protected”.

  Issue: Any app can send an unprotected intent without having the appropriate permission, or it can register itself to receive unprotected intents.

  Consequently, apps could escalate their privileges by sending unprotected intents to privileged targets, e.g., apps that provide elevated features such as camera access. Also, malicious apps registered to receive implicit unprotected intents may relay intents while leaking or manipulating their data (Chin et al. 2011).

  Symptom: The existence of an unprotected implicit intent. For intents requesting a return value, the lack of check for whether the sender has appropriate permissions to initiate an intent.

  Detection: The existence of several methods on the `Context` class for initiating an unprotected implicit intent like `startActivity`, `sendBroadcast`, `sendOrderedBroadcast`, `sendBroadcastAsUser`, and `sendOrderedBroadcastAsUser`.

  Limitation: We do not verify, for a given intent requesting a return value, if the sender enforces permission checks for the requested action.

  Mitigation: Use explicit intents to send sensitive data. When serving an intent, validate the input data from other components to ensure they are legitimate. Adding custom permissions to implicit intents may raise the level of protection by involving the user in the process.

- **SM05: Sticky Broadcast** A normal broadcast reaches the receivers it is intended for, then terminates. However, a “sticky” broadcast stays around so that it can immediately notify other apps if they need the same information.

  Issue: Any app can watch a broadcast, and particularly a sticky broadcast receiver can tamper with the broadcast (Mitra and Ranganath 2017).

Consequently, a manipulated broadcast may mislead future recipients.

Symptom: Broadcast calls that send a sticky broadcast appear in the code, and the related Android system permission exists in the manifest file.

Detection: We check for the existence of methods such as `sendStickyBroadcast`, `sendStickyBroadcastAsUser`, `sendStickyOrderedBroadcast`, `sendStickyOrderedBroadcastAsUser`, `removeStickyBroadcast`, and `removeStickyBroadcastAsUser` on the `Context` object in the code and the `android.permission.BROADCAST_STICKY` permission in the manifest file.

Limitation: We are not aware of any limitations.

Mitigation: Prohibit sticky broadcasts. Use a non-sticky broadcast to report that something has changed. Use another mechanism, e.g., an explicit intent, for apps to retrieve the current value whenever desired.

- **SM06: Slack WebViewClient** A `WebView` is a component to facilitate web browsing within Android apps. By default, a `WebView` will ask the Activity Manager to choose the proper handler for the URL. If a `WebViewClient` is provided to the `WebView`, the host application handles the URL.

  Issue: The default implementation of a `WebViewClient` does not restrict access to any web page (Mitra and Ranganath 2017).

  Consequently, it can be pointed to a malicious website that entails diverse attacks like phishing, cross-site scripting, etc.

  Symptom: The `WebView` responsible for URL handling does not perform adequate input validation. *Detection:* The `WebView.setWebViewClient()` exists in the code but the `WebViewClient` instance does not apply any access restrictions in `WebView.shouldOverrideUrlLoading()`, i.e., it returns `false` or calls `WebView.loadUrl()` right away. Also, we report a smell if the implementation of `WebView.shouldInterceptRequest()` returns `null`.

  Limitation: It is inherently difficult to evaluate the quality of an existing input validation.

  Mitigation: Use a white list of trusted websites for validation, and benefit from external services, e.g., SafetyNet API,[5] that provide information about the threat level of a website.

- **SM07: Broken Service Permission** Two different mechanisms exist to start a service: `onBind` and `onStartCommand`. Only the latter allows services to run indefinitely in the background, even when the client disconnects. An app that uses Android IPC to start a service may possess different permissions than the service provider itself.

  Issue: When the callee is in possession of the required permissions, the caller will also get access to the service.

  Consequently, a privilege escalation could occur (Mitra and Ranganath 2017).

  Symptom: The lack of appropriate permission checks to ensure that the caller has access right to the service.

  Detection: We report the smell when the caller uses `startService`, and then the callee uses `checkCallingOrSelfPermission`, `enforceCallingOrSelfPermission`, `checkCallingOrSelfUriPermission`, or `enforceCallingOrSelfUriPermission` to verify the permissions of the request. Calls on the `Context` object for permission check will then fail as the system mistakenly considers the callee's permission instead of the caller's. Furthermore, reported are

[5] https://developer.android.com/training/safetynet/safebrowsing.html

calls to `checkPermission`, `checkUriPermission`, `enforcePermission`, or `enforceUriPermission` methods on the `Context` object, when additional calls to `getCallingPid` or `getCallingUid` on the `Binder` object exist.

Limitation: We currently do not distinguish between checks executed in `Service.onBind` or `Service.onStartCommand`, and we do not verify custom permission checks based on the user id with `getCallingUid`.

Mitigation: Verify the caller's permissions every time before performing a privileged operation on its behalf using `Context.checkCallingPermission()` or `Context.checkCallingUriPermission()` checks. If possible, do not implement `Service.onStartCommand` in order to prevent clients from starting, instead of binding to, a service. Ensure that appropriate permissions to access the service have been set in the manifest.

– **SM08: Insecure Path Permission** Apps can access data provided by a content provider using path specifications of the form `/a/b/c`. A content provider may restrict access to certain data under a given path by specifying so called path permissions. For example, it may specify that other apps cannot access data located under `/data/secret`. The Android framework prohibits access to unauthorized apps only if the requested path strictly matches the protected path. For instance, `//data/secret` is different from `/data/secret`, and therefore the framework will not block access to it.

  Issue: Developers often use the `UriMatcher` for URI comparison in the `query` method of a content provider to access data, but this matcher, unlike the Android framework, evaluates paths with two slashes as being equal to paths with one slash.

  Consequently, access to presumably protected resources may be granted to unauthorized apps (Mitra and Ranganath 2017).

  Symptom: A `UriMatcher.match()` is used for URI validation.

  Detection: We look for `path-permission` attributes in the manifest file, and `UriMatcher.match()` methods in the code.

  Limitation: We are not aware of any limitation.

  Mitigation: As long as the bug exists in the Android framework, use your own URI matcher.

– **SM09: Broken Path Permission Precedence** In a content provider, more fine-grained path permissions e.g., on `/data/secret` take precedence over those with a larger scope e.g., on `/data`.

  Issue: A path permission never takes precedence over a permission on the whole content provider due to a bug that exists in the `ContentProvider.enforceReadPermissionInner()` method. For example, if a content provider has a permission for general use, as well as a path permission to protect `/data/secret` from untrusted apps, then the general use permission takes precedence.

  Consequently, content providers may mistakenly grant untrusted apps access to presumably protected paths.

  Symptom: A content provider is protected by path-specific permissions.

  Detection: We look for a `path-permission` in the definition of a content provider in the manifest file.

  Limitation: We are not aware of any limitation.

  Mitigation: As long as the bug exists in Android, instead of path permissions use a distinct content provider with a dedicated permission for each path.

– **SM10: Unprotected Broadcast Receiver** Static broadcast receivers are registered in the manifest file, and start even if an app is not currently running. Dynamic broadcast receivers are registered at run time in Android code, and execute only if the app is running.

Issue: Any app can register itself to receive a broadcast, which exposes the app to any other app able to initiate the broadcast.

Consequently, if there is no permission check, the receiver may respond to a spoofed intent yielding unintended behavior or data leaks (Mitra and Ranganath 2017).

Symptom: The `Context.registerReceiver()` call without any argument for permission exists in the code *Detection:* We report cases where the permission argument is missing or is `null`.

Limitation: We are not aware of the permissions' appropriateness.

Mitigation: Register broadcast receivers with sound permissions.

– **SM11: Implicit Pending Intent** A `PendingIntent` is an intent that executes the specified action of an app in the future and on behalf of the app, i.e., with the identity and permissions of the app that sends the intent, regardless of whether the app is running or not.

  Issue: Any app can intercept an implicit pending intent (Mitra and Ranganath 2017) and use the pending intent's `send` method to submit arbitrary intents on behalf of the initial sender.

  Consequently, a malicious app can tamper with the intent's data and perform custom actions with the permissions of the originator. Relaying of pending intents could be used for intent spoofing attacks.

  Symptom: The initiation of an implicit `PendingIntent` in the code.

  Detection: We report a smell if methods such as `getActivity`, `getBroadcast`, `getService`, and `getForegroundService` on the `PendingIntent` object are called, without specifying the target component

  Limitation: Arrays of pending intents are not yet supported in our analysis.

  Mitigation: Use explicit pending intents, as recommended by the official documentation.[6]

– **SM12: Common Task Affinity** A *task* is a collection of activities that users interact with when carrying out a certain job.[7] A task affinity, defined in the manifest file, can be set to an individual activity or at the application level.

  Issue: Apps with identical task affinities can overlap each others' activities, e.g., to fade in a voice record button on top of the phone call activity. The default value does not protect the application against highjacking of UI components.

  Consequently, malicious apps may hijack an app's activity paving the way for various kinds of spoofing attacks (Ren et al. 2015).

  Symptom: The task affinity is not empty.

  Detection: We report a smell if the value of a task affinity is not empty.

  Limitation: We are not aware of any limitation.

  Mitigation: If a task affinity remains unused, it should always be set to an empty string on the application level. Otherwise set the task affinity only for specific activities that are safe to share with others. We suggest that Android set the default value for a task affinity to empty. It may also add the possibility of setting a permission for a task affinity.

In summary, each security smell introduces a different set of vulnerabilities. We established a close relationship between the smells and the security risks with the purpose of providing accessible and actionable information to developers, as shown in Table 2.

[6] https://developer.android.com/reference/android/app/PendingIntent.html

[7] https://developer.android.com/guide/components/activities/tasks-and-back-stack.html

Table 2 The relationship between vulnerabilities and security code smells

Vulnerabilities	Security code smells
Denial of service	SM01, SM02, SM03, SM04, SM06, SM07, SM10, SM12
Intent spoofing	SM02, SM03, SM04, SM05, SM07, SM08, SM09, SM10, SM11
Intent hijacking	SM02, SM03, SM04, SM05, SM10, SM11

4 Empirical Study

In this section we first present the Lint-based tool with which we detect security code smells, and introduce a dataset of more than 700 open-source Android projects that are mostly hosted on GitHub. We then present the results of our investigation into RQ_2 and RQ_3 by analyzing the prevalence of security smells in our dataset, and by discussing the performance of our tool, respectively.

The results in Section 4.3 suggest that although fewer than 10% of apps suffer from more than two categories of ICC security smells, only small teams are capable of consistently building software resistant to most security code smells. With respect to app volatility, we discovered that updates rarely have any impact on ICC security, however, in case they do, they often correspond to new app features. On the other hand, we found that long-lived projects have more issues than recently created ones, except for apps that receive frequent updates, where the opposite is true. Moreover, the findings of Android Lint's security checks correlate to our detected security smells.

In Section 4.4, our manual evaluation confirms that our tool successfully finds many different ICC security code smells, and that about 43.8% of the smells in fact represent vulnerabilities. We consequently hypothesize that the tool can offer valuable support in security audits, but this remains to be explored in our future work.

We performed analyses similar to our previous work, e.g., exploring the relation between star rating and smells, or the distribution of smells in app categories, and we did not observe major differences with our past findings (Ghafari et al. 2017). Our results are therefore in line with our prior research that did not consider ICC smells, and found that the majority of apps suffer from security smells, despite the diversity of apps in popularity, size, and release date.

4.1 Linting Tool

Our Linting tool is built using Android Lint, a static analysis framework from the official Android Studio IDE[8] for analyzing Android apps. Android Lint provides various rich interfaces for analyzing XML, Java, and Class files in Android. Using these interfaces, one can implement a so-called "detector" that is responsible for scanning code, detecting issues, and reporting them. More specifically, each detector is represented by a Java class that implements Android Lint interfaces to access Android Lint's abstract syntax trees (ASTs) of the app built from XML, source, or bytecode. In order to ease the AST traversal, Android Lint provides an implementation of the visitor design pattern with additional helper methods to support further interaction with the tree. The majority of methods use idiomatic names that

[8]https://sites.google.com/a/android.com/tools/tips/lint

closely resemble the developer's intention, e.g., `UastUtils.tryResolve()` to resolve a variable, or the class `ConstantEvaluator` to evaluate constants. The latest Android Lint provides more than 300 different detectors to check several categories of issues such as, e.g., Accessibility, Usability, Security, etc.

We extended Android Lint by developing twelve new detectors. These detectors implement `UastScanner`[9] and `XmlScanner` interfaces to check the presence of security code smells in source code and manifest files, respectively. We implemented the detection strategies that we introduced for each security smell in Section 3. The complexity of our detectors varies; the average size of a detector is 115 lines of code.

Android Lint brings analysis support directly into the Android Studio IDE. Developers can therefore receive just-in-time feedback during app development about the presence of security code smells in their code. For this purpose, the .jar file that contains our detectors should be copied into the Lint directory. These detectors will then be run automatically during programming in the latest Android Studio IDE (i.e., the Canary build), and notify developers about the security code smells once they appear in the code under development. Each notification includes an explanation of the smell, mitigation or elimination strategies, as well as a link to some references.

Linting in batch mode is also possible through the command line interface, given the availability of the successfully built projects. In our experience, a successful build often entails changing build paths, and updating Gradle and its project configurations to a version that is compatible with the current release of Android Lint. We created a script to automate most of this non-trivial process. After a successful build of each project, another script runs the executable of Android Lint, and collects the analysis results in XML files.

The tool is publicly available for download from a GitHub repository.[10]

4.2 Dataset

We collected all open-source apps from the F-Droid[11] repository as well as several other apps directly from GitHub.[12] In total we collected 3 471 apps, of which we could successfully build 1 487 (42%). For replication of our results we explicitly provide the package names of all successfully analyzed apps,[13] instead of a binary compilation, because of the dataset's storage space requirements of more than 27 GBytes. In order to reduce the influence of individual projects, in case there existed more than one release of a project, we only considered the latest one. Finally, we were left with 732 apps (21%) in our dataset. The median project size in our dataset is about 1.2 MB, while the median number of data files per project is 108.

4.3 Batch Analysis

This section presents the results of applying our tool to all the apps in our dataset.

[9]The `UastScanner` is the successor of the `JavaScanner`, and, in addition to Java, also supports Kotlin, a new programming language used in the Android platform.

[10]https://github.com/pgadient/AndroidLintSecurityChecks

[11]https://f-droid.org/

[12]https://github.com/pcqpcq/open-source-android-apps

[13]https://github.com/pgadient/AndroidLintSecurityChecks/blob/master/dataset/analyzed_apps.csv

4.3.1 Prevalence of Security Smells

Figure 1 shows how prevalent the smells are in our dataset. Almost all apps suffer from *Common Task Affinity* issues (99%) followed by the much less prevalent *Unauthorized Intent* smell (11%). The default value of task affinity configurations does not protect the application against highjacking of UI components, and only few developers appear to be aware of the issue and set the property accordingly. *Custom Scheme Channel* and *Implicit Pending Intent* each contribute about 8% of the smells. Furthermore, *WebViewClient* is in line with our observation that apps increasingly rely on web components for their UI. At the other end of the spectrum, *Sticky Broadcast*, *Incorrect Protection Level*, *Broken Service Permission*, and *Persisted Dynamic Permission* cause less than 2% of all issues. The threat of path permissions is not very common, as no apps suffered from SM08 or SM09.

We were also interested in the relative prevalence of different security smells in the apps (see Fig. 2). Less than 1% did not suffer from any security smell at all, whereas the majority of apps, i.e., over 90%, suffered from one or two different smells. 9% of all apps were affected by three or more smells. No apps, fortunately, suffered from more than seven different types of smells. It is important to recall that the more issues that are present in a benign app, the more likely it is that a malign app can exploit it, e.g., with denial of service, intent spoofing, or intent hijacking attacks.

4.3.2 Contributor Affiliation

Figure 3 shows the relationship between the number of contributors participating in a project and the mean number of security smell categories apps suffer from. For example, the second

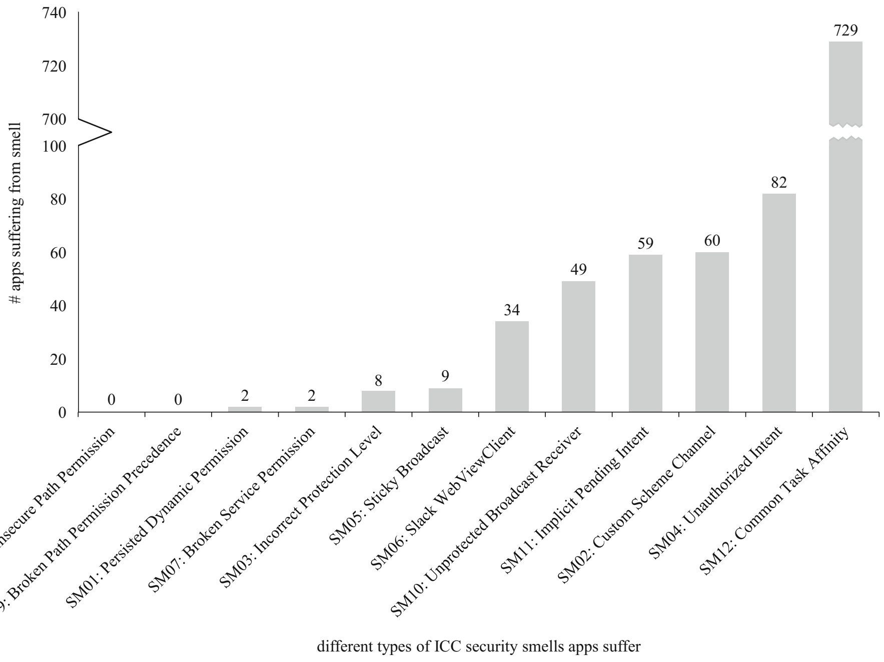


Fig. 1 Distribution of security smells in the apps

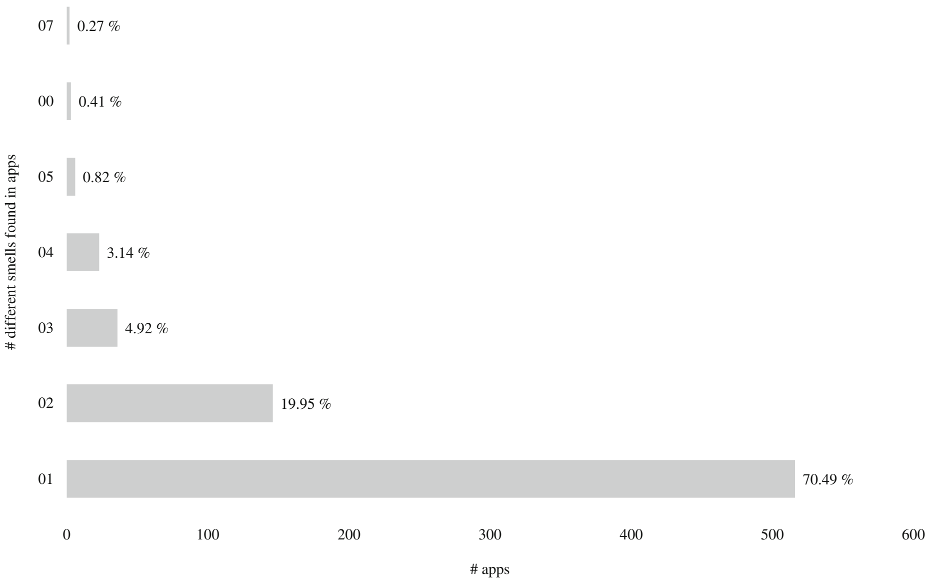


Fig. 2 Prevalence of different security smells in apps

last bar represents the number of all projects maintained by 41 to 60 participants, while the line chart shows that projects with this many participants suffer on average from 2.5 security smell categories. We see that most apps are maintained by two contributors, followed by projects developed by individuals. A trend exists that projects with many participants

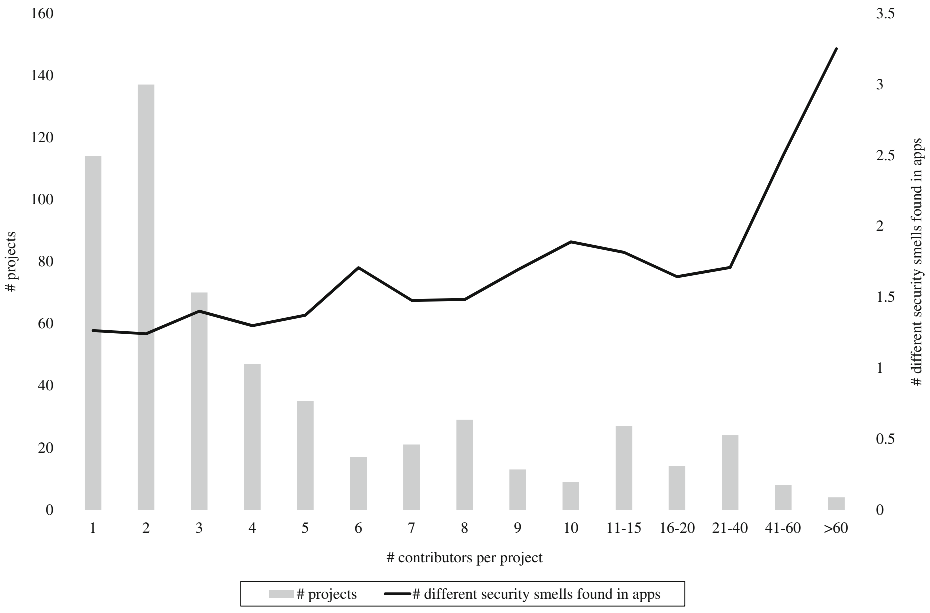


Fig. 3 Relation between number of a project's participants, its prevalence, and the average number of different security smells found

are less common than projects with only a few contributors. The more people are involved in a project the more the security decreases, especially for large teams. More precisely, we found statistical evidence that only small teams of up to five people are capable of consistently building projects resistant to most security code smells, by using the nonparametric Mann-Whitney U test that does not require the assumption of normal distributions for the dataset. The mean different smell occurrences in the groups “projects with one contributor” and “projects with six contributors” were 1.263 and 1.705; the distributions in the two groups differed significantly (Mann-Whitney $U = -2.086$, $n1 < n2 = 0$, $P < 0.05$ two-tailed). Similarly, we found that the distributions in the two groups “projects with six to forty contributors” and “projects with more than forty contributorn were diverse (Mann-Whitney $U = -2.204$, $n1 < n2 = 0$, $P < 0.05$ two-tailed) with mean different smell occurrences of 1.655 and 2.750, respectively.

4.3.3 App Updates

We investigated the smell occurrences in subsequent app releases. Of the 732 projects, 33 (4%) of them released updates that either resolved or introduced issues. By inspection of source code we noticed that many of the updates targeted new functionality, e.g., addition of new implicit intents to share data with other apps, implementation of new notification mechanisms for receiving events from other apps using implicit pending intents, or registration of new custom schemes to provide further integration of app related web content into the Android system. We believe this is due to developers focusing on new features instead of security.

For the majority of the app updates that introduced new security smells, we found that the dominant cause for decreased security is the accommodation of social interactions and data sharing features in the apps updates. Hence, developers should be particularly cautious when integrating new functionality into an app.

4.3.4 Evolution

Every new Android version introduces changes that strengthen security. The targeting of outdated Android releases will not only limit the supported feature set to the respective release, but also introduce potential security issues as security fixes are continuously integrated into the OS with each update.

Figure 4 shows the evolution of security smells across different Android releases. For those apps that had more than one release in our dataset, we only considered the latest release. The horizontal axis shows the different Android releases apps are targeting in their configuration, whereas the vertical axis shows the contribution of a specific smell to the total amount of smells detected. As in previous work (Ghafari et al. 2017), we see changes in some of the security smells apps suffer from. We believe that the positive trend in *Unauthorized Intent* within apps is the consequence of built-in sharing functionalities to external services. The relative growth of *Implicit Pending Intent* could correlate to the introduction of a new storage access framework in Android release 19, which heavily relies on intents, and allows developers to browse and open documents, images, and other files with ease. Google’s efforts to raise the developer’s awareness of web-related security issues appears to be working: the occurrences of *Slack WebView Client* have decreased in more recent releases. Despite the lack of comprehensive data on API levels 10 and 11 due to the relatively few apps available for study, the occurrences of the majority of smells remain constant as a result of the early feature availability since API level 1.

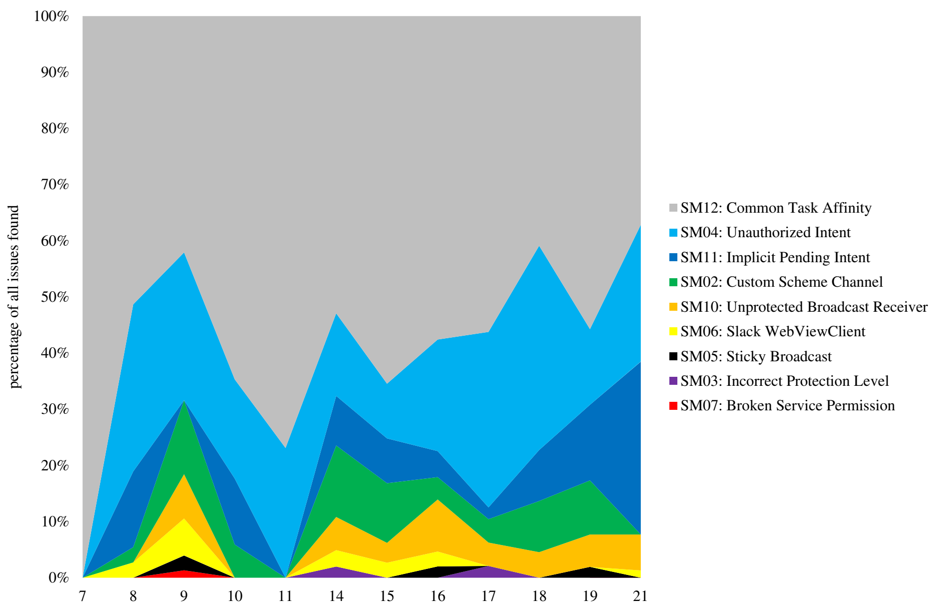


Fig. 4 Evolution of security code smells in different Android releases

4.3.5 Comparison to Existing Android Lint Checks

In order to compare our findings with other issues in the apps, we correlated the results from the existing Android Lint framework with security code smells. We wanted to explore whether frequent reports of specific Android Lint issue categories were also indicative of security issues, or in other words, if security checks by the Android Lint framework agree with our security smells and whether other quality aspects of an app could relate to its security level. We collected all available issue reports for each app and then extracted the occurrences of each detected issue.

We applied the Pearson product-moment correlation coefficient algorithm for each ICC security smell category combination according to the following formula:

$$Pearson(x, y) = \frac{\sum(x-\bar{x})(y-\bar{y})}{\sqrt{\sum(x-\bar{x})^2 \sum(y-\bar{y})^2}}, \text{ where}$$

x is the array of all apps issue occurrences in category *ICC security code smells*,

y is the array of all apps issue occurrences in the respective Android Lint category, and

$\bar{x}$, $\bar{y}$ represent the corresponding sample means.

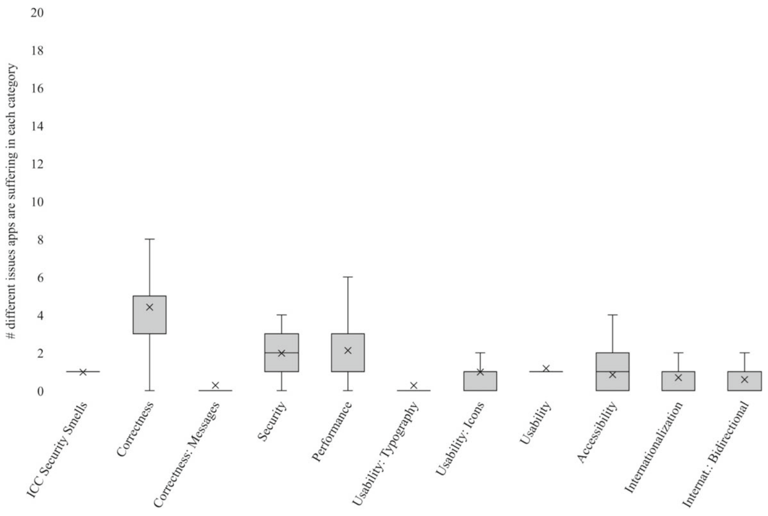
It provides a linear correlation between two vectors represented as a value in the range of -1 (total negative linear correlation) and $+1$ (total positive linear correlation). The correlation of the Android Lint categories and our ICC smell category in Table 3 reveals several interesting findings: (1) Our ICC security category strongly correlates with the Android Lint security category (+0.72), which contains checks for a variety of security-related issues such as the use of user names and passwords in strings, improper cryptography parameters, and bypassed certificate checks in WebView components. (2) Another discovery is the minor correlation between the ICC security smells and the Android Lint correctness

Table 3 Correlation of ICC security smells with Android Lint issue categories

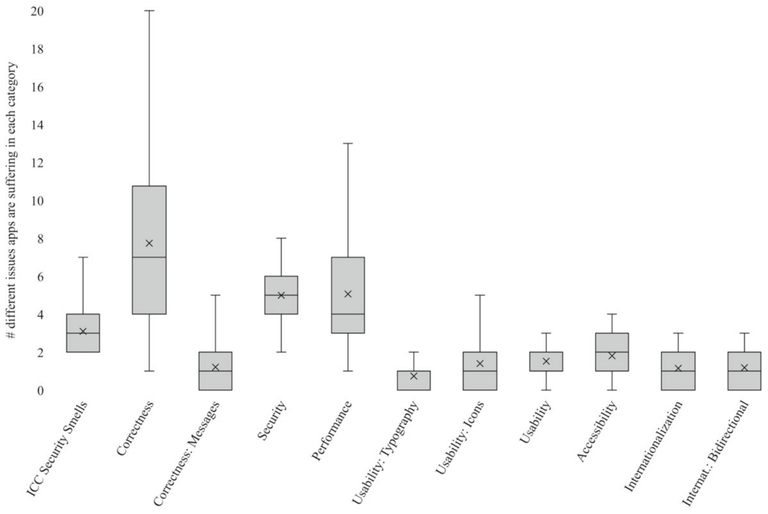
Android Lint category	Correlation with ICC security smells
Security	0.72
Correctness	0.29
Correctness: Messages	0.27
Accessibility	0.25
Performance	0.25
Usability: Typography	0.21
Internationalization	0.13
Internationalization: Bidirectional	0.11
Usability: Icons	0.11
Usability	0.07

category (+0.29). This category includes checks for erroneously configured project build parameters, incomplete view layout definitions, and usages of deprecated resources. (3) Furthermore, we assume that usability does not impede security (+0.07), because issues in usability are closely related to UI mechanics. (4) Finally, minor correlations are shown for performance, accessibility, and internationalization. These three categories have in common that they rely heavily on UI controls and configurations.

To further assess how our tool performs on real world apps against the Android Lint detections, we take the 100 apps with the most and least prevalent ICC security smells and compare them to Android Lint's analysis results. We expect to see significant similarities in the increase of issues detected as our security smells correlate to Android Lint's security checks, i.e., the least vulnerable apps should suffer less in both, the Android Lint checks and our security smell detectors. Figure 5 illustrates two plots, each presenting our analysis results for the 100 apps suffering the most and the least from ICC security smells, respectively. The vertical axis represents the condensed mean number of found issues, that is, we conflated all detected ICC security smell issues, regardless of their smell categories, into "ICC Security Smells". The remaining Android Lint categories on the x-axis are treated accordingly. The crosses represent the mean value of the number of different issues apps are suffering from in each category, and, as we hid any outliers to increase readability, these values can exceed the first quartiles. The least and most affected apps clearly correspond in terms of issue frequency among specific categories, that is, the mean number of issues found in *each* category is between 29% and 332% higher on behalf of the 100 most vulnerable apps. Besides the ICC Security Smells category with an increase of 219% in issues found, the Android Lint security category experienced an increase of 152%. The Correctness: Message and the Usability: Typography categories of Android Lint achieved, unexpectedly, an increase in issues found of about 332% and 174%, respectively. After manual verification, we discovered that these gains were mostly caused by flawed language dictionary entries used for internationalization, such as missing or misunderstood language dependent string declarations, spelling mistakes, and the use of strings containing three dots instead of the ellipsis character. While the 100 most vulnerable apps appear to prominently incorporate translations for several different languages, the 100 least vulnerable apps rarely make use of these features, hence, they suffer from much fewer issues. The remaining categories encountered an increase of less than 139%.



(a) 100 least vulnerable apps



(b) 100 most vulnerable apps

Fig. 5 Prevalence of Android Lint issues in the 100 most and least vulnerable apps

Interestingly, the internationalization category does not encounter a noticeable increase in issues due to its limited scope, i.e., it only covers five specific flaws regarding insufficient language adaption, and the use of uncommon characters or encodings. We propose that some of these issue detections should be reallocated to other categories, e.g., spelling mistakes should be assigned to internationalization, and vice versa the issue *SetTextI18n* in the category internationalization that reports any use of methods that potentially fail with number conversions.

4.3.6 Influence of Project Age and Activity

To explore the effect of recent updates, which we believed would improve app security, we evaluated our ICC category as well as the Android Lint security and correctness categories according to time since the last commit. More precisely, we were interested in the question: Do recent updates improve app security? A related question arises from the age of a project, i.e., are mature projects more secure than recent creations? We investigated these two questions based on available GitHub metadata, and brought the dates into perspective with the reported issues.

Figure 6 shows the mean number of detected issues per app on the vertical axis, either for the ICC security smells, or the Android Lint security category. The black dots reveal the app's project creation dates, whereas red dots indicate the most recent commit dates of projects, hence every app is represented by one black and one red dot in each plot. The creation date for the majority of apps dates back to less than 6.5 years. We can clearly see in every plot a correlation between both the creation date, and the date of the last commit to the overall issue count, based on the pictured linear trends (dotted lines). These trends, which are very similar in terms of elevation, are a further indicator for the close relationship between our tool and the Android Lint checks. Moreover, the Lint security category shows strong evidence that mature projects have more security issues than recent ones. We assume that this is caused by the less comprehensive checks that older IDEs performed on the source code. Similarly, apps that frequently introduce changes, i.e., receive updates, are prone to have more issues.

4.3.7 Influence of Code Size

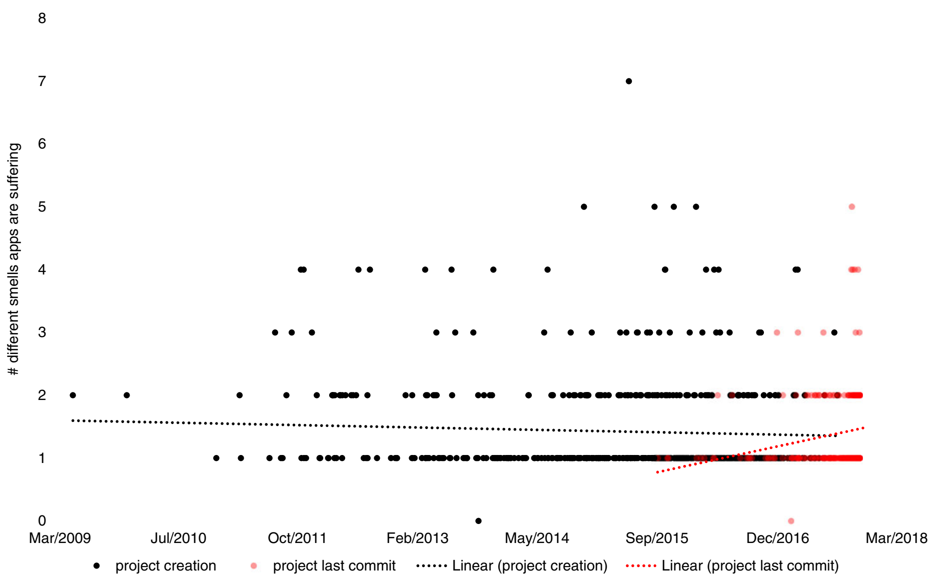
Another popular indicator used in software analysis is the code size, which we measured in thousands of lines of code (kLOC) with the open-source tool cloc.[14] As Android projects consist aside from source code of different configuration, resource and other utility files, we first ran the analysis of adopted software languages (e.g., Java, Kotlin, XML) that required each of those items, before we excluded all elements except the Java code in the main Java source folders for the kLOC measurements. We conjectured that we would see a trend of small teams developing small apps that are less likely to have problems. In contrast, we expect that aging projects are more likely to have smells as they are larger than more recent ones. Figure 7 illustrates the relation between the kLOC and other relevant properties.

In Fig. 7a we categorized projects according to their size on the x-axis, while the left y-axis displays contributors per project, and the right y-axis the number of different categories of security smells found in apps, and the number of different languages used. We see a trend that larger projects rely on more contributors with a minor exception at 20–24 kLOC.

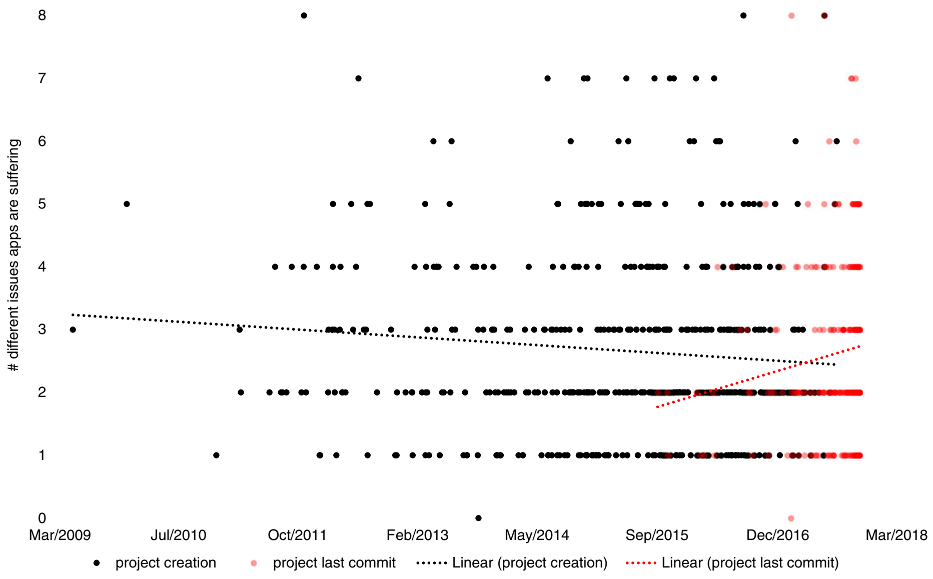
Furthermore, it is interesting to see that projects of up to 10 kLOC are maintained by five or fewer developers. In addition, we see that larger projects tend to suffer from more smells, and those projects are also using more languages. After a manual inspection of apps exploiting different languages we discovered that those apps are rather collections of frameworks, e.g., for network penetration tests using a plethora of different tools written in different languages.

Figure 7b uses the same feature for the x-axis, but presents on the left y-axis the number of projects, and on the right y-axis the number of days since project creation, and the

[14]https://github.com/AlDanial/cloc



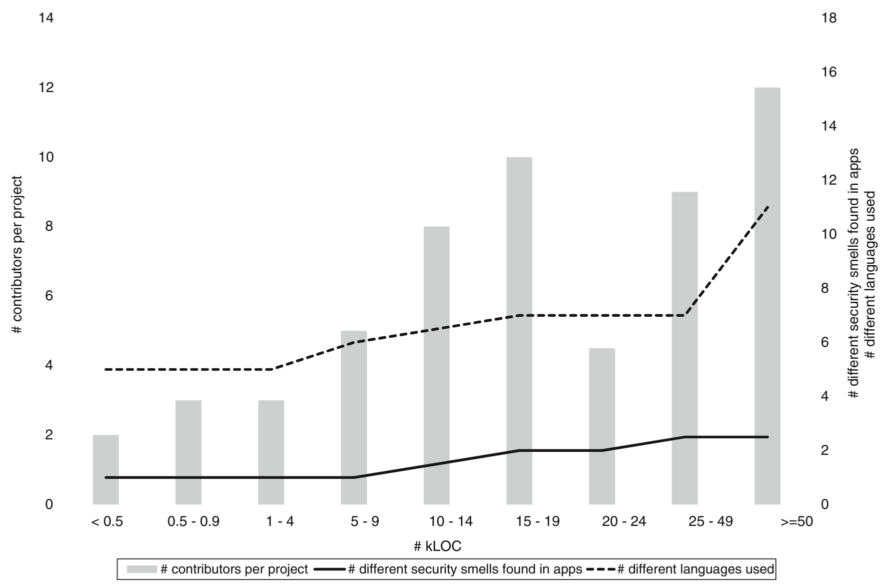
(a) Relation of dates to our ICC security smells



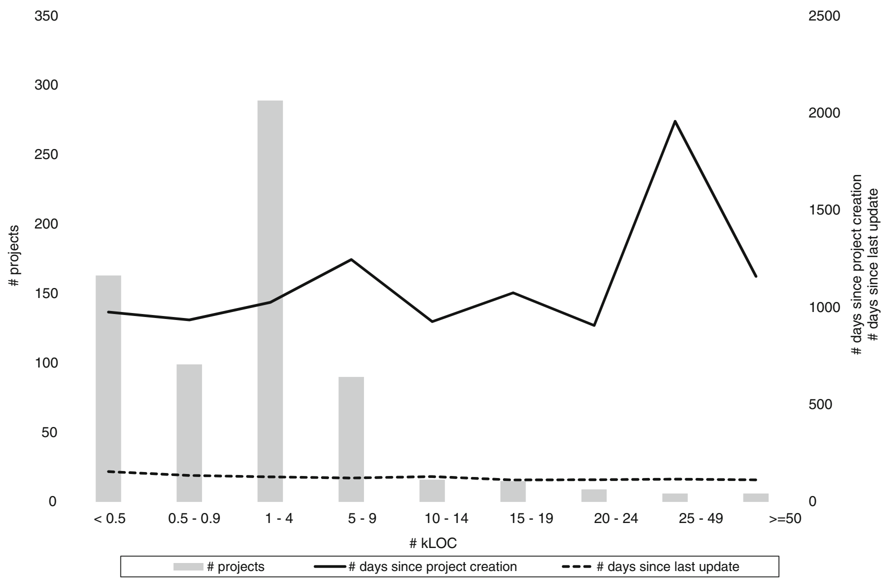
(b) Relation of dates to Lint security

Fig. 6 GitHub project creation and last commit date in relation to each project's issue count

number of days since last update. The majority of projects in our dataset consist of less than 10 kLOC, and especially projects with 1–4 kLOC have been very prevalent, followed by apps that are less than 500 LOC. Only six projects contained more than 50 kLOC.



(a) Relation of kLOC to contributors, ICC security smells, and used languages



(b) Relation of kLOC to number of projects, days since project creation, and days since last update

Fig. 7 Different project properties in relation to kLOC

Interestingly, we cannot derive clearly any major trend regarding the age of projects and LOC, although projects of 25–49 kLOC evidently are older than the others. On the contrary, we can see a minor trend regarding the time since last update. It appears that smaller apps are updated less frequently than larger apps. We expect that the larger an application becomes, the more maintenance work is required due to library updates, obsolete external references, and content changes.

4.4 Manual Analysis

To assess the performance of our tool and show how reliable these findings are to detect security vulnerabilities, we manually analyzed 100 apps. We invited two participants to independently evaluate the precision and recall of our tool. Participant A is a senior developer with more than 5 years of professional experience in development and security of mobile apps. Participant B is a junior developer with less than two years of experience in Java and C# software development. We provided both participants an introduction to Android security, and individually explained every smell in detail. We subsequently selected the top 100 apps, that is more than 13% of the whole corpus, with most smells in accordance with our ICC security smell list, for which we can say with 95% confidence that the population's mean smell occurrences of the top 100 apps are between 3.04 and 3.48, while they are between 1.38 and 1.50 for the whole data set of about 732 apps. Then we provided the participants with our tool, the sources of the top 100 apps, and a spreadsheet to record their observations. Each participant was asked to import the sources of each app in Android Studio, which had been prepared to run a customized version of our analysis plug-in, to verify each reported smell according to the symptoms of any smell described in Section 3. We were also interested in vulnerability detection capability of security smells, [15] thus the participants were asked to investigate if a security smell indicates the presence of a security vulnerability based on the vulnerability information available in the benchmarks.

4.4.1 Tool Evaluation

While the assessment of true positives (TPs, reported code that is a smell) and false positives (FPs, reported code that is not a smell) requires participants to manually check only the tool's results, the extraction of true negatives (TNs, unreported code that is not a smell) and false negatives (FNs, unreported code that is a smell) is resource intensive and error prone. Therefore to avoid an exhaustive code inspection, we developed a relaxed analysis that shows ICC-related APIs in the code to support the participants.

We obtained relatively high smell detection rates, especially for SM02, SM04, SM10, SM11 and SM12, as indicated by the TPs in Fig. 8. The reason is that these smells occur frequently and are straightforward to detect, mostly relying on some very specific method calls and permissions.

We encountered above average FPs in SM12 due to the intended use of task affinity features in apps that try to separate activities with empty task affinities. This smell would require additional semantical, architectural, and UI information for proper assessment. While some of the exposed activities are non-interactive, and thus supposedly secure, some of them are interactive and could be misused in combination with other spoofing techniques, like clickjacking, in which an adversary unexpectedly shows the exposed activity to trick users into providing unintended inputs. In particular, call recorders and various client-server apps for chat, video streaming, home automation, and other network services have been affected by this issue.

[15] We define a vulnerability capability as the possibility a security issue can compromise a user's security and privacy.

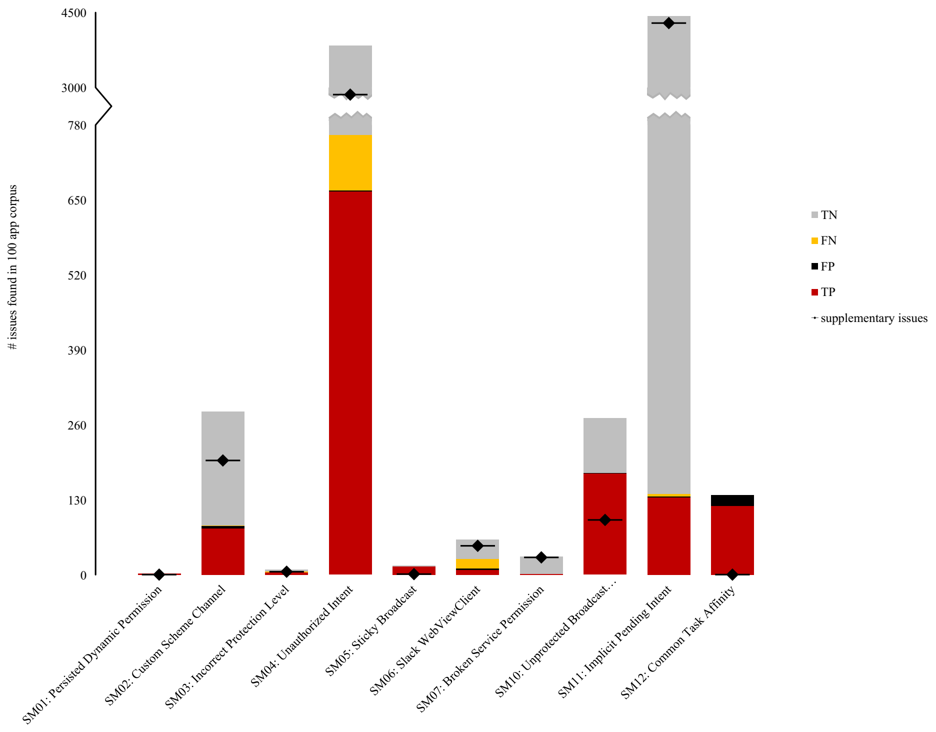


Fig. 8 Tool evaluation results

Our participant had to check 7 241 locations in the code to examine the TNs and FNs in 100 apps. In more than 98.36% of cases participants confirmed that there are no security smells beyond what the tool could identify; we consider this very low proportion of FNs, i.e., 1.7%, encouraging.

We are surprised to see only a few FNs in SM04 as we expected much more to appear due to the countless ways that intents can be created in Android. A substantial number of FNs were missed because of complex chained executions and calls initiated from sophisticated UI related classes containing URIs. For SM06, we discovered that the FNs have been frequently caused by lack of context, e.g., unawareness of data sensitivity, or custom logic that does not mitigate the vulnerability. For example, our tool was unable to verify the correctness of custom web page white-listing implementations for `WebView` browser components, which would actually reduce security if implemented incorrectly.

We did not encounter any instances of the two smells SM08 and SM09, that is, we retrieved zero reports on both of them for our 100 app dataset, hence, we excluded them for all subsequent plots and discussions in this subsection.

We could find common security smells while reviewing the feedback from the two participants, for example, that some apps were using `shouldOverrideUrlLoading` without URL white-listing to send implicit intents to open the device's default browser, rather than using their own web view for white-listed pages, thus fostering the risk of data leaks. Another discovery was the use of regular broadcasts for intra-app communication. For these scenarios, developers should solely rely on the `LocalBroadcastManager` to prevent accidental data leaks. The same applies for intents that are explicitly used for communication within the app, but do not include an explicit target, which would similarly mitigate the risk of data leaks. Moreover, unused code represents a severe threat. Several apps requested specific permissions without using them, increasing the impact of potential privilege escalation attacks.

4.4.2 Tool Performance

Figure 9 presents the tool's performance based on the precision, recall, and lastly the F-measure for existing smells in 100 apps. All smells except SM03 and SM06 show outstanding results, nonetheless, some of them could be biased as a result of their low occurrences which is true for SM01 and SM07. We performed a follow up manual investigation of SM03 and SM06. Apparently, the detection of SM03 suffers from the difficulty to discern data sensitivity and the need to approximate the required protection level. Besides that, SM06 is heavily affected by custom web API implementations that (mis)use security features, which are, in fact, not secure.

4.4.3 Smells and Their Vulnerability Capability

Figure 10 shows the vulnerability capability perception of both participants against the reported smells. For each smell we show two grouped columns: the left column reports the results from the more experienced participant A (PA), and the right column reports the results from the less experienced participant B (PB). Each column consists of three different segments *yes*, *uncertain*, and *no*. The category *yes* is used for all reported smells that introduce critical risks, such as plain-text exposure of user passwords through network sockets. The *uncertain* category is used for risks that potentially exist, and are challenging to inspect manually, for instance, vulnerabilities that require prerequisites for successful exploitation such as potentially dangerous user-defined schemes. Finally, all smells assigned to the *no*

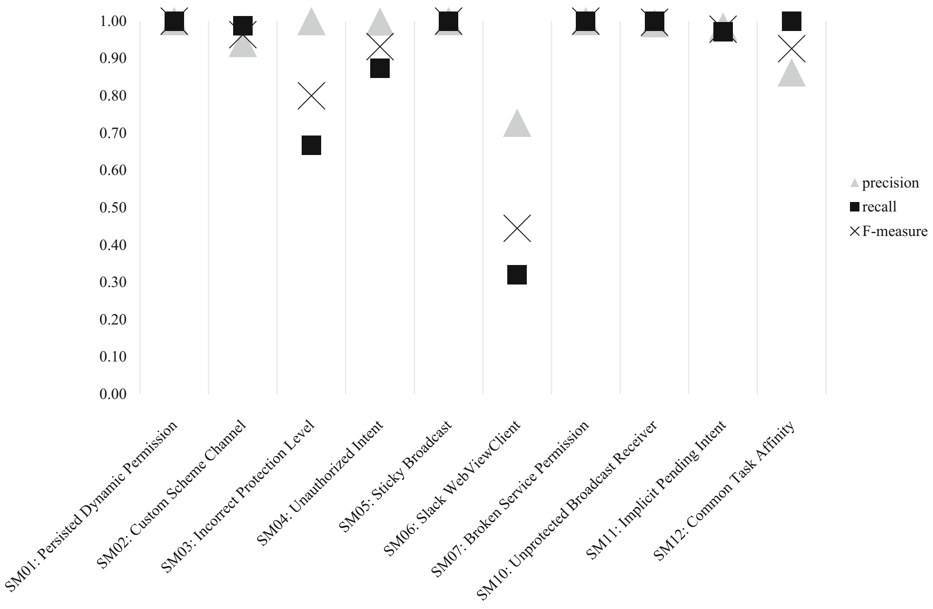


Fig. 9 Tool performance

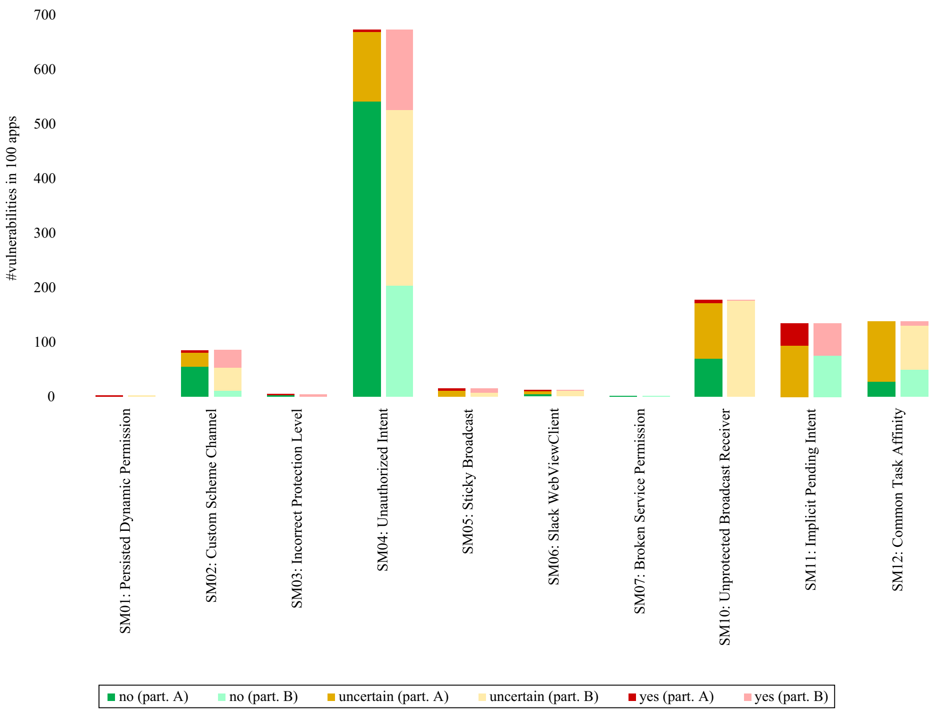


Fig. 10 Vulnerability capability of detected issues

category are not vulnerable to any attacks, either because they do not contain any user information, or because they are sufficiently secure with respect to the participant's opinion. Apps that send static non-sensitive information commonly match this category. For all our considerations the participants were told to treat any user data as sensitive, since they could potentially contain sensitive information at run time.

According to the reports by PA, 38.5% of smells represent potential threats, i.e., *uncertain* category, and only 5.3% of smells represent critical threats, i.e., *yes* category. In other words, only about 44% of security smells could lead to security vulnerabilities.

A further comparison of the reports between the two participants shows that they expect somewhat similar risks for the smell categories SM05, SM07, and SM12, whereas the participants tended to interpret diversely the threat caused by *Custom Scheme Channel*, *Unauthorized Intent*, and *Slack WebViewClient* smells. We reviewed the feedback of the participants and discovered that for *Custom Scheme Channel* predefined system schemes are considered less harmful for PA (category *no*), while PB assigns them to the category *uncertain*. For *Unauthorized Intent* PB assessed the risks similar to PA, however, PB encountered difficulties to predict adequately the threat capability of many intent instances, thus PB assigned them to section *uncertain*. For the smell *Slack WebViewClient* PB performed a conservative risk assessment by not assigning any custom security feature implementations to *no*, instead PB assigned them to *uncertain*, unlike PA who concluded many of them as secure. An example thereof is an app with a network security penetration test suite that requires opening insecure web pages for security validation purposes.

It is interesting to observe that PB, in contrast to PA, does consider fewer instances as harmful for SM11 and SM12. For the first smell *SM11: Implicit Pending Intent* PB considers intents with assigned actions frequently as secure, while PA considers them as potential risk, which is more accurate. For the second smell *SM12: Common Task Affinity* PB considers most apps that used empty task affinity properties as secure, while PA performed a more thorough analysis of the UI and considered additionally the misuse capability of such exposed views, which resulted in many assignments to the category *uncertain*. We conclude that the very complex and flexible ICC implementation provided by Android overwhelms inexperienced developers, even worse, it could mislead those developers to create insecure code due to their misunderstanding.

Overall, most of the vulnerabilities seem to emerge from SM10 and SM11 that collectively contribute to more than 72% of all detected critical issues. On the other hand, SM04 on its own provides with 77% the largest proportion of false alarms regarding vulnerability capabilities.

4.5 Threats to Validity

One important threat to validity is the completeness of this study, i.e., whether we could identify and study all related papers in the literature. Although we could not review all publications, we strived to explore top-tier software engineering and security journals and conferences as well as highly-cited work in the field. For each relevant paper we also recursively looked at both citing and cited papers. Moreover, to ensure that we did not miss any important paper, for each identified issue we further constructed more specific queries and looked for any new paper on GoogleScholar.

We were only interested in studying benign apps as in malicious ones it is unlikely that developers will spend any effort to accommodate security concerns. Thus, we merely collected apps that were available on GitHub and the F-Droid repository. However, our dataset may still have malicious apps that evaded the security checks of the community or the marketplace.

We analyzed the existence of security smells in the source code of an app, whereas third-party libraries could also introduce smells.

Our analysis is intra-procedural and suffers from inherent limitations of static analysis. Moreover, many security smells actually constitute security risks only if they deal with sensitive data, but our analysis cannot determine such sensitivity.

The Android Lint tool we used for the analysis is prone to errors that could lead to FNs, for example, when Android Lint crashes due to file parsing issues, an immediate termination of the inspection occurs which could cause some misses.

Finally, the fact that the results of our analysis tool are validated against manual analysis performed by the authors is a threat to construct validity through potential bias in experimenter expectancy. We mitigated this threat by including an external participant in the process in addition to the co-author who simultaneously played the senior developer's role.

5 Related Work

Reaves et al. studied Android-specific challenges to program analysis, and assessed existing Android application analysis tools. They found that these tools mainly suffer from lack of maintenance, and are often unable to produce functional output for applications with known vulnerabilities (Reaves et al. 2016). Li et al. studied the state-of-the-art work that statically analyses Android apps (Li et al. 2017). They found that much of this work supports detection of private data leaks and vulnerabilities, a moderate amount of research is dedicated to permission checking, and only three studies deal with cryptography issues. Unfortunately, much state-of-the-art work does not publicly share the concerned artifacts. Linares-Vasquez et al. mine 660 Android vulnerabilities available in the official Android bulletins and their CVE details,[16] and present a taxonomy of the types of vulnerabilities (Linares-Vásquez et al. 2017). They report on the presence of those vulnerabilities affecting the Android OS, and acknowledge that most of them can be avoided by relying on secure coding practices. Finally, Sadeghi et al. review 300 research papers related to Android security, and provide a taxonomy to classify and characterize the state-of-the-art research in this area (Sadeghi et al. 2016). They find that 26% of existing research is dedicated to vulnerability detection, but each study is usually concerned with specific types of security vulnerabilities. Our work expands on such studies to provide practitioners with an overview of the security issues that are inherent in insecure programming choices.

Some research is devoted to educating developers in secure programming. Xie et al. interviewed 15 professional developers about their software security knowledge, and realized that many of them have reasonable knowledge but do not apply it as they believe it is not their responsibility (Xie et al. 2011). Weir et al. conducted open-ended interviews with a dozen app security experts, and determined that app developers should learn analysis, communication, dialectics, feedback, and upgrading in the context of security (Weir et al. 2016). Witschey et al. surveyed developers about their reasons for adopting or not adopting security tools (Witschey et al. 2015). Interestingly, they found the perceived prestige of security tool users and the frequency of interaction with security experts to be important for promoting security tool adoption. Acar et al. suggest a high-level research agenda to achieve usable security for developers. They propose several research questions to elicit developers' attitudes, needs and priorities in the area of security (Acar et al. 2016). Our work is complementary to these studies in the sense that we provide an initial assessment of developers' security knowledge, and we highlight the significant role of developers in making apps more secure.

Numerous researchers have dedicated their work to detecting common ICC vulnerabilities. Despite the fact that their expression has changed over time, the vulnerability classes have remained largely the same. Chin et al. discuss the ICC implementation of Android and examine closely the interaction between sent and received ICC messages (Chin et al. 2011). Despite the fact that their work is based on a small corpus containing only 20 apps, they were able to detect various denial-of-service issues in numerous application components, and conclude that the message-passing system in Android enables rich applications, and encourages component reuse, while leaving a large potential for misuse when developers do not take any precautions.

Felt et al. discovered that permission re-delegation, also known as confused deputy or privilege escalation attack, is a common threat, and they pose OS level mitigations conceptually similar to the same origin policy in web browsers (Felt et al. 2011). The community aimed on the one hand for preciseness, as countless tools to detect these flaws in ICC have been released, notably Epicc (Octeau et al. 2013) and IccTA (Li et al. 2015) with a significantly improved precision. On the other hand, the app coverage began to play a major role, as in the work of Bosu et al. who recently discovered with their tool inadequate security measures, including privilege escalation vulnerabilities, among inter-app data-flows from 110 000 real-world apps (Bosu et al. 2017).

[16]http://cve.mitre.org — Common Vulnerabilities and Exposures, a public list of known cyber-security vulnerabilities.

Along with passive analysis, active countermeasures and attacks have emerged in the scientific community. Garcia et al. crafted a state-of-the-art tool to automatically detect and exploit vulnerable ICC interfaces to provoke denial-of-service attacks amongst others (Garcia et al. 2017). They identified exploits for more than 21% of all apps appraised as vulnerable. Xie et al. presented a bytecode patching framework that incorporates additional self-contained permission checks avoiding privilege issues during runtime, generating a remarkably low computational overhead (Xie et al. 2017). Ren et al. successfully investigated design glitches in the multitasking implementation of Android, uncovering task hijacking attacks that affected every OS release and were potentially duping user perception (Ren et al. 2015). They considered in particular the taskAffinity and taskParentReparenting attributes of the manifest file that allow views to be dynamically overlaid on other apps, and provided proof-of-concept attacks. Wang et al. assessed the threat of data leakage on Apple iOS and Android mobile platforms and show serious attacks facilitated by the lack of origin-based protection on ICC channels (Wang et al. 2013). Interestingly, they found effective attacks against apps from such major publishers as Facebook and Dropbox, and more importantly, indicate the existence of cross-platform ICC threats. Researchers have found interest in reinforcing the Android ICC core framework. Khadiranaikar et al. propose a certificate-based intent system relying on key stores that guarantee integrity during message exchanges (Khadiranaikar et al. 2017). In addition to securing the ICC-based communication, Shekhar et al. proposed a separation of concerns to reduce the susceptibility for manipulation of Android apps, by explicitly restricting advertising frameworks (Shekhar et al. 2012). Ahmad et al. elaborated on problematic ICC design decisions on Android, and found that missing consistent message types and conformance checking, unpredictable message interactions, and a lack of coherent versioning could break inter-app communication and pose a severe risk (Ahmad et al. 2016). They recommend a centralized message-type repository that immediately provides feedback to developers through the IDE.

In summary, existing studies have often dealt with a specific issue, whereas we cover a broader range of issues, making the results more actionable for practitioners. Moreover, previous work often overwhelms developers with many identified issues at once, whereas we provide feedback during app development where developers have the relevant context. Such feedback makes it easier to react to issues, and helps developers to learn from their mistakes (Tymchuk et al. 2018).

6 Conclusion

We have reviewed ICC security code smells that threaten Android apps, and implemented a linting plug-in for Android Studio that spots such smells, by linting affected code parts, and providing just-in-time feedback about the presence of security code smells.

We applied our analysis to a corpus of more than 700 open-source apps. We observed that only small teams are capable of consistently building software resistant to most security code smells, and fewer than 10% of apps suffer from more than two ICC security smells. We discovered that updates rarely have any impact on ICC security, however, in case they do, they often correspond to new app features. Thus developers have to be very careful about integration of new functionality into their apps. Moreover, we found that long-lived projects suffer from more issues than recently created ones, except for apps that are updated frequently, for which that effect is reversed. We advise developers of long-lived projects to continuously update their IDEs, as old IDEs have only limited support for security issue reports, and therefore countless security issues could be missed.

A manual investigation of 100 apps shows that our tool successfully finds many different ICC security code smells, and about 43.8% of them in fact represent vulnerabilities, thus it constitutes a reasonable measure to improve the overall development efficiency and software quality.

We recommend security aspects such as secure default values and permission systems, to be considered in the initial design of a new API, since this would effectively mitigate many issues like the very prevalent Common Task Affinity smell. We plan to explore the extent to which APIs can be made secure by design. While we analyzed the existence of ICC security smells in apps, studying their absence, i.e., secure ICC uses, could offer different insights that we plan to pursue in future. Moreover, we are interested in evaluating the usefulness of our tool during a security audit process, as well as in an app development session.

Acknowledgements We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Analysis" (SNSF project No. 200020-162352, Jan 1, 2016 - Dec. 30, 2018). We also thank Astrid Ytrehorn for her contribution to the empirical study.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

References

Acar Y, Fahl S, Mazurek M (2016) You are not your developer, either: a research agenda for usable security and privacy research beyond end users. In: IEEE SecDev 2016

Ahmad W, Kästner C, Sunshine J, Aldrich J (2016) Inter-app communication in Android developer challenges. In: 2016 IEEE/ACM 13th working conference on mining software repositories (MSR). IEEE, pp 177–188

Balebako R, Cranor L (2014) Improving app privacy: nudging app developers to protect user privacy. IEEE Secur Priv 12(4):55–58

Bosu A, Liu F, Yao DD, Wang G (2017) Collusive data leak and more: large-scale threat analysis of inter-app communications. In: Proceedings of the 2017 ACM on Asia conference on computer and communications security. ACM, pp 71–85

Chin E, Felt AP, Greenwood K, Wagner D (2011) Analyzing inter-application communication in Android. In: Proceedings of the 9th international conference on mobile systems, applications, and services, MobiSys '11. ACM, New York, pp 239–252

Felt AP, Wang HJ, Moshchuk A, Hanna S, Chin E (2011) Permission re-delegation: attacks and defenses. In: USENIX security symposium, vol 30, p 88

Garcia J, Hammad M, Ghorbani N, Malek S (2017) Automatic generation of inter-component communication exploits for Android applications. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering. ACM, pp 661–671

Ghafari M, Gadient P, Nierstrasz O (2017) Security smells in Android. In: 2017 IEEE 17Th international working conference on source code analysis and manipulation (SCAM), pp 121–130

Jones BH, Chin AG (2015) On the efficacy of smartphone security: a critical analysis of modifications in business students' practices over time. Int J Inf Manag 35(5):561–571

Khadiranaikar B, Zavarsky P, Malik Y (2017) Improving Android application security for intent based attacks. In: 2017 8th IEEE annual information technology, electronics and mobile communication conference (IEMCON). IEEE, pp 62–67

Li L, Bartel A, Bissyandé TF, Klein J, Traon YL, Arzt S, Rasthofer S, Bodden E, Octeau D, McDaniel PM (2015) Iccta: Detecting inter-component privacy leaks in Android apps. In: Proceedings of the 37th international conference on software engineering - volume 1, ICSE '15. IEEE Press, Piscataway, pp 280–291

Li L, Bissyandé TF, Papadakis M, Rasthofer S, Bartel A, Octeau D, Klein J, Traon Le (2017) Static analysis of Android apps: a systematic literature review. Inf Softw Technol 88:67–95

Linares-Vásquez M, Bavota G, Escobar-Velásquez C (2017) An empirical study on Android-related vulnerabilities. In: Proceedings of the 14th international conference on mining software repositories, MSR '17. IEEE Press, Piscataway, pp 2–13

Mitra J, Ranganath V-P (2017) Ghera: a repository of Android app vulnerability benchmarks. In: Proceedings of the 13th international conference on predictive models and data analytics in software engineering. ACM, pp 43–52

Octeau D, McDaniel P, Jha S, Bartel A, Bodden E, Klein J, Traon YL (2013) Effective inter-component communication mapping in Android with Epicc: an essential step towards holistic security analysis. In: Presented as part of the 22nd USENIX security symposium (USENIX security 13). USENIX, pp 543–558

Reaves B, Bowers J, Gorski III SA, Anise O, Bobhate R, Cho R, Das H, Hussain S, Karachiwala H, Scaife N, Wright B, Butler K, Enck W, Patrick T (2016) *Droid: assessment and evaluation of Android application analysis tools. ACM Comput Surv 49(55):1–55, 30

Ren C, Zhang Y, Xue H, Wei T, Liu P (2015) Towards discovering and understanding task hijacking in Android. In: USENIX security symposium, pp 945–959

Sadeghi A, Bagheri H, Garcia J, Malek S (2016) A taxonomy and qualitative comparison of program analysis techniques for security assessment of Android software. IEEE Trans Softw Eng PP(99):1–1

Shekhar S, Dietz M, Wallach DS (2012) Adsplit: Separating smartphone advertising from applications. In: USENIX security symposium

Tymchuk Y, Ghafari M, Nierstrasz O (2018) JIT Feedback — what experienced developers like about static analysis. In: Proceedings of the 26th IEEE international conference on program comprehension (ICPC'18)

Wang R, Xing L, Wang X, Chen S (2013) Unauthorized origin crossing on mobile platforms threats and mitigation. In: ACM conference on computer and communications security

Weir C, Rashid A, Noble J (2016) Reaching the masses: a new subdiscipline of app programmer education. In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering, FSE 2016. ACM, pp 936–939

Witschey J, Zielinska O, Welk A, Murphy-Hill E, Mayhorn C, Zimmermann T (2015) Quantifying developers' adoption of security tools. In: Proceedings of the 2015 10th joint meeting on foundations of software engineering, ESEC/FSE 2015. ACM, pp 260–271

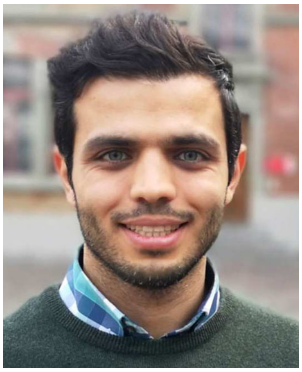
Lei W, Grace M, Zhou Y, Chiachih W, Jiang X (2013) The impact of vendor customizations on Android security. In: Proceedings of the 2013 ACM SIGSAC conference on computer & communications security, CCS '13. ACM, New York, pp 623–634

Xie J, Lipford HR, Chu B (2011) Why do programmers make security errors? In: 2011 IEEE symposium on visual languages and human-centric computing (VL/HCC), pp 161–164

Xie J, Xiao F, Xiaojiang D, Luo B, Guizani M (2017) Autopatchdroid: a framework for patching inter-app vulnerabilities in Android application. In: 2017 IEEE international conference on communications (ICC). IEEE, pp 1–6

Meng X, Song C, Ji Y, Shih M-W, Lu K, Zheng C, Duan R, Jang Y, Lee B, Qian C, et al. (2016) Toward engineering a secure Android ecosystem: a survey of existing techniques. ACM Comput Surv (CSUR) 49(2):38

Pascal Gadient is a PhD candidate in the Software Composition Group of the University of Bern (Switzerland). His research interests are security in mobile apps and mining software repositories.



Mohammad Ghafari is a senior research assistant in the Software Composition Group, and a lecturer in the Computer Science Institute of the University of Bern. He obtained a doctoral degree in Software Engineering from Politecnico di Milano (Italy) in 2015, and has been affiliated with the University of Bern since 2016. His research interest is at the confluence of program analysis, mining software repositories, and secure software engineering.

Patrick Frischknecht is a software developer and a student of Computer Science at the University of Bern.

Oscar Nierstrasz is a professor at the Institute of Computer Science of the University of Bern, where he founded the Software Composition Group in 1994. He is co-author of over 300 publications in diverse topics related to object-oriented software development and software evolution. He is co-author of the open-source books “Object-Oriented Reengineering Patterns” and “Pharo by Example”.