b
UNIVERSITÄT
BERN

Security in Android ICC

Bachelor Thesis

Patrick Frischknecht

from

Laupen BE, Switzerland

Faculty of Science
University of Bern

29 June 2018

Prof. Dr. Oscar Nierstrasz

Pascal Gadient

Software Composition Group
Institute of Computer Science
University of Bern, Switzerland

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 vulnerability 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 integrated development environment (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 manually inspect around 15% of these apps to assess the extent to which identifying such smells uncovers ICC security vulnerabilities.

Contents

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.

A recent survey on the StackOverflow website shows that about 65% of mobile developers work with Android.[1] This platform has captured over 80% of the smartphone market,[2] 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 [13], its platform [15], and end users [5]. Numerous security APIs, protocols, guidelines, and tools have been proposed. Nevertheless security concerns are often overridden by other concerns [1]. Many developers undermine their significant role in providing security [14]. As a result, security issues in mobile apps continue to proliferate unabated.[3]

Given this situation, previous work identified 28 security code smells *i.e.*, symptoms in the code that signal the prospect of security vulnerabilities [4]. The authors studied the prevalence of ten of such smells, and

[1]http://insights.stackoverflow.com/survey/2017
[2]http://www.gartner.com
[3]http://www.cvedetails.com

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 security vulnerabilities.

To promote the adoption of secure programming practices, we build on preceding 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 thesis we present these vulnerabilities and their corresponding smells in the code, and discuss how they could be eliminated or mitigated during development. Moreover, 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 integrated development environment (IDE) about the presence of such security smells in the code. 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 significant related work, especially that appearing in top-tier conferences and journals, and identified twelve avoidable ICC vulnerabilities and the 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 applied it to a repository of about 700 apps, mostly available on GitHub. We discovered that almost all apps suffer from at least one ICC security smell, but less than 10% suffer from more than two smells.
- **RQ**$_3$: *To which extent does identifying security smells facilitate detection of real vulnerabilities?* We manually inspected 100 apps, and compared our findings to the result of the tool. Our investigation showed that about half of the identified smells are in fact good indicators of security vulnerabilities.

1.1 Contributions

To summarize, this work represents an initial effort to spread awareness about the impact of programming choices in making apps secure, and to fundamentally reduce the attack surface 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 relevant context. Such feedback makes it easier to react to issues, and helps developers to learn from their mistakes [11]. This thesis extends earlier work [4] by (i) focusing specifically on ICC vulnerabilities, one of the most prevalent Android security issues, (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, and (iv) open-sourcing the Android Lint checks as well as the analyzed data.

1.2 Outline

The remainder of this thesis is organized as follows. In chapter 2 we show the current state of the art for recent analysis tools used to improve software security. We provide the necessary background about the Android OS and explain Android Lint in detail in chapter 3. We introduce ICC-related security code smells in chapter 4, followed by our empirical study in chapter 5. Finally, we conclude the thesis in chapter 6.

2
State of the Art

Numerous tools to assess the security of Android apps already exist, and online wikis that maintain lists referring to these tools have become quite popular.[1] As the tools often lack a proper integration into a productive environment, *e.g.*, they frequently provide no build process support and return unstructured results, we specifically focus in this chapter on the different levels of integration of popular tools.

2.1 Development Process Integration

The majority of security analysis tools provide feedback *on demand*. These tools have to be executed manually and commonly require the use of shell commands. Besides analysis on demand, other tools have emerged that are well integrated into an IDE and provide *just-in-time* feedback. However, most of these tools need extensive manual effort either to be integrated into any existing build process, or to provide just-in-time feedback. Only a few tools offer both features, none of them targeting specifically ICC security code smells.

[1]`https://mobilesecuritywiki.com/`[2]

2.1.1 On Demand Checks

In general, on demand tools provide a command line interface and usually take the `.apk` file as an argument amongst others. These tools typically return text output only upon successful analyses. Predestined for on-demand checks are resource intensive taint-based analyses that track data flows from security-sensitive *sources* to potentially insecure *sinks* by “tainting” each involved variable along the data flow. Common representatives of this category are Epicc [8] and IccTA [6]; both of them are taint analysis tools used to find ICC issues and they model the Android app life cycle to achieve high recall and precision. Other on demand tools that do not rely on command line interfaces can be run manually through UI elements such as buttons, menus, *etc.* These tools either run stand-alone or integrated as a plug-in to an IDE, for which the output is typically made accessible within the tool’s window. A tremendously popular example is the GUI-based plug-in *Find Security Bugs*, available for many popular IDEs, that supports developers on the identification of security issues in Java code.[3] This plug-in does not only include checks specific to Java, but also for Android and other frameworks.

2.1.2 Just-in-Time (JIT) Feedback

JIT tools serve developers instantaneously with feedback on their code while typing. These tools are generally integrated into IDEs and highlight problematic code sections, provide tooltips and, less commonly, offer quick fixes to automatically resolve issues. In order to provide meaningful feedback and to avoid any interferences with the user, these tools must efficiently review the code on every change. Consequently, JIT tools usually exploit simplified techniques compared to what is used for on demand analyses. A prime example of JIT analysis is Android Lint, an extensible framework integrated by default into recent releases of the Android platform’s default Android Studio IDE.[4] This extensible framework provides countless checks in different categories, *e.g.*, performance, correctness, and security. Thereupon, Peck *et al.* demonstrated how Android Lint can be extended with new security-related checks and verified the effectiveness on real world applications [9]. Their new checks have been added to the official Android Open Source Project (AOSP) and are now part of every Android Lint distribution. While Android Lint’s focus is set on Android Studio and its underlying JetBrains IntelliJ IDE, Do *et al.* engineered `CHEETAH`, an Eclipse-based IDE tool, that performs taint analysis on Android applications [3].

2.1.3 Build Process

Only very few analysis tools offer interfaces to application build processes. That said, Android Lint is applicable to any Android app build process, as it is well integrated into Gradle, *i.e.*, the default build automation system used in Android Studio. Therefore, Android Lint can be executed through its Gradle task after a successful build, and it is capable of generating HTML and XML reports. To further customize the

[3] `http://find-sec-bugs.github.io/`
[4] `https://developer.android.com/studio/write/lint.html`

analysis, *e.g.*, selection of checks or scope, the `build.gradle` file of an app project allows developers to set specific configuration parameters. The competitor *Find Security Bugs* offers even more functionality such as Maven / Ant integration and support for Continuous Integration (CI) services based on Jenkins and SonarQube, however, the tight Gradle integration is currently unavailable off the shelf.

3 Background

In this chapter we briefly explain Android OS-related terms and provide an introduction to Android Lint. For Android Lint we specifically present its architecture, the internal data structures and use, as well as the stored reports generated from successful inspections. We also explain how we can analyze both Java and Kotlin code in a unified way in Android Lint. Furthermore we present the additional tools we created to simplify batch analysis with Android Lint.

3.1 Android OS

Android is the most popular mobile operating system (OS) with a market share of more than 85%,[1] and it provides a rich set of ICC APIs for app developers to access and share app functionality. 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:

[1]`https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/`

- *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.
- *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.*, defined by a `Class` or `ComponentName` instance) the intent is sent. 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 run time. 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.

An Android app consists of an `.apk` package file containing the compiled byte code, any needed data, and resource files. Especially the `AndroidManifest.xml` file in the root folder of the package file is vital for app execution, as it specifies countless app-related parameters such as *used permissions*, component configurations, *etc.* 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.

3.2 Android Lint

A wide range of tools for analysis of Android projects exists as shown in chapter 2. In order to avoid reinventing the wheel, our goal was to augment an existing tool with new ICC security-related checks.

We found two tools worth considering, hence we compared in a first step *Android Lint (AL)* with *Find Security Bugs (FSB)*, the two very popular static analysis frameworks. We discovered that FSB suffers from severe limitations, such as:

- the inability to parse any `.xml` files (this is a requirement for many smells that emerge from the manifest)
- little assistance for Android-related checks as it focuses on traditional (web) applications, whereas Android Lint targets Android applications and covers a significantly wider range of Android-related issues

- support only for Java code, while Android Lint operates also on Kotlin, the emerging language used in Android apps
- its weak integration into Android Studio compared to Android Lint, which deeply prevents any reuse of existing GUI components for reasonable visualization
- the exclusive availability of on-demand code checks, whereas the checks of Android Lint are executed on-the-fly without any interruption of a developer's workflow

Along with these advantages, the tight integration of Android Lint into the build process of Android apps further eased our work. Consequently, Android Lint was our preferred choice for the foundation of our work.

The deployment of the extension is straightforward. Once the extension is built, the generated `.jar` archive has to be copied into the `~/.android/lint` folder. The new checks will then automatically run in the Android Studio IDE as well as during any subsequent Gradle builds of an Android project.

Android Lint, and thus our extension, is limited in the scope of the analysis as it provides neither dynamic nor comprehensive taint analysis. Unfortunately, some of the issues in section 4.2 would require more sophisticated techniques for proper detection, however, those issues are still reported as the resulting false positives are further subjects to study. Nevertheless, the just-in-time feedback of Android Lint within the IDE provides essential context and thus developers should quickly be able to cope with inappropriate warnings [11].

An extension of Android Lint mainly consists of three elements: *Issues*, *Detectors* and *Issue Registries*.

- **Issue:** An `issue` represents a specific problem to find in the code, *e.g.*, the use of insecure implicit intents, and also includes all the related necessary information required by user dialogs. Therefore, each issue comprises several parameters, *i.e.*, an `id` as a short name describing the issue, a `brief-Description`, an `explanation`, followed by the corresponding `category` and `priority`, a `severity` value, and the actual `implementation`. In particular, the `briefDescription` and `explanation` both remain visible in the generated reports, but for the information in the IDE's tooltips only the explanation is used. Moreover, the `category` can be one out of eleven different predefined categories, *e.g.*, "Security", and the `priority` has to be a number in the range of one to ten that indicates the impact of the issue. Finally, the `severity` allows one to abort the build process for severe issues, and can be selected from five different values, *e.g.*, `Fatal` for immediate termination. The provided `Implementation` instance relies on two parameters, the `class` that refers to specific code of the issue detection, and the `scope` of the analysis which defines the required file origins (code, manifest or resources files). Additionally, supplementary material can be referenced with web links that appear in IDE tooltip messages for any detected security code smell. These links can be set up with `Implementation.addMoreInfo`. In order to avoid any conflicts between custom and existing issues Android Lint requires each issue `id` to be unique. Our 20 implemented issues altogether cover the 12 smells we thoroughly explain in chapter 4.

- **Detector:** Detectors find occurrences of an `Issue` in code and other project resources. A detector may implement different interfaces depending on the sources to analyze. For our analysis we used the `Detector.XmlScanner` (for XML resources) and the `Detector.UastScanner` interface (for Java and Kotlin resources). While some detectors need information from multiple sources and thus have to implement both interfaces, detectors that exclusively rely on one resource need to only implement either one of the interfaces. Further code reuse among detectors of very similar issues is ensured with the ability of a single detector to cover multiple issues. Currently, our implementation contains 16 detectors for 20 issues. Upon successful completion a detector files a report that includes for each detected smell the location, the XML or UAST node which defines the highlighted source code area, and a message, which is used within the tooltip. Depending on the analysis we set the locations accordingly, *i.e.*, (i) for intra-method analysis we always set the location to what we refer to as the *sink node*, (ii) for the PathPermission issue the location is set to the path permission node in the `AndroidManifest.xml`, and (iii) in all remaining scenarios we directly report the corresponding (sub)node location.

- **Issue Registry:** An `IssueRegistry` collects multiple `Issues` and promotes them to the Android Lint framework. For a successful Android Lint set-up, each `Issue` must be registered in an issue registry and each issue registry must be added to the manifest of the compiled `.jar` file; otherwise the framework will ignore them. We use only one `IssueRegistry` as multiple registries provide no benefits for our tool.

3.2.1 Abstract Syntax Trees

The *Universal Abstract Syntax Tree (UAST)* uses an abstract syntax initiated by JetBrains[2] to maintain both Java and Kotlin code in a unified representation. It describes a superset of elements that exist in Java and Kotlin. This syntax is used by Android Lint for the UAST parser to analyze the Java and Kotlin source code of a project. Figure 3.2 shows a simplified example of a class written in Kotlin and Java side by side with the UAST representation of it. The complete UAST implementation can be found open-sourced on GitHub.[3]

Previous versions of the detectors had to implement a *Program Structure Interface (PSI)* to traverse the PSI tree, and it became inevitable that the Java source code was represented by a PSI tree instead of a UAST tree. However, in Android Lint 2.4 JetBrains substituted PSI with UAST.[4] As a result, PSI and UAST had to be very similar to ensure backwards compatibility; even the documentation of the `Detector.UastScanner` states that UAST is only an augmentation of PSI and that it is not intended

[2]`https://www.jetbrains.com/`
[3]`https://github.com/JetBrains/intellij-community/tree/master/uast`
[4]`https://groups.google.com/forum/#!topic/lint-dev/7nLiXa04baM`

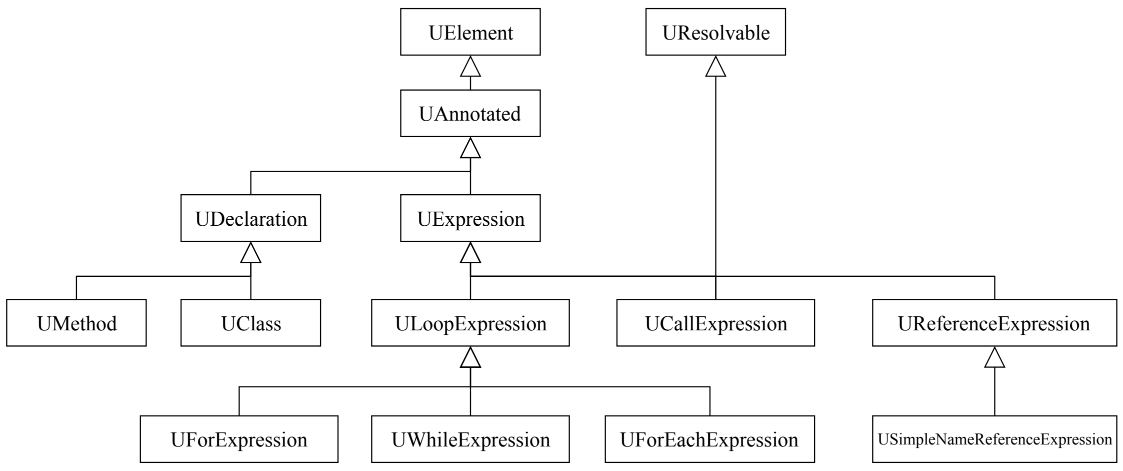


Figure 3.1: The hierarchy of relevant UAST interfaces

to completely replace it.[5] The main reason to prefer the UAST instead of the traditional PSI is that the UAST is able to represent Java and Kotlin code within the same tree structure, while a PSI representation may differ due to the different syntax.[6]

The different types of UAST elements are represented by a series of different interfaces. The actual nodes of the abstract syntax tree returned by the UAST parser are always represented by Java or Kotlin specific elements which implement the appropriate UAST interfaces. The root of the element hierarchy is by definition an element of the base type `UElement`, which all other UAST elements extend. A hierarchy of important UAST elements used in our detectors is shown in Figure 3.1, furthermore, Table 3.1 explains the purpose of essential UAST interfaces.

In order to maintain a tree, each `UElement`, with or without any children, maintains a reference to its parent which can be retrieved by the `containingElement()` method. To traverse the tree `UElement` offers the `accept()` method which accepts a visitor and passes it to all its children.

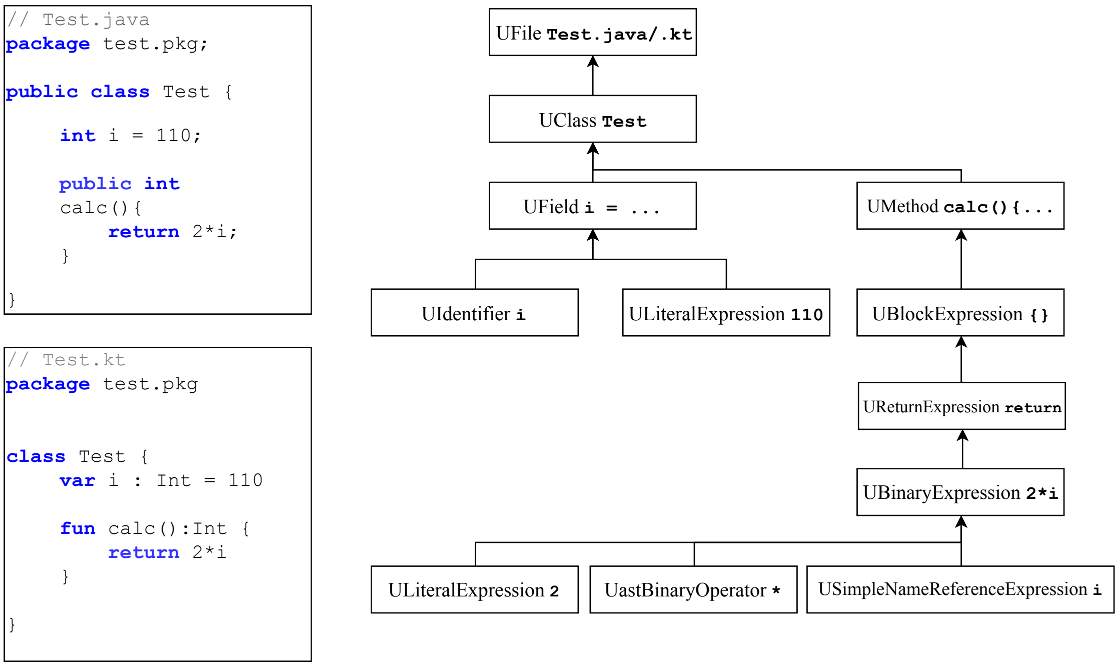
A wide range of `UElement` types exist that reference other source code elements, such as a `UCall-Expression` that references the `UMethod` it calls. It is common to resolve expressions as it allows, for example, to get the declaration of a local variable based on a later reference to it in the method, or to get the declaration of a parameter in a method. Although the UAST interface is the successor of PSI, the resolution of fields still requires the use of PSI methods. Consequently, to resolve a reference in a `UElement`, *e.g.*, to get the declaration a reference points to, manipulations on the underlying PSI element, like `PsiType` or `PsiClass`, are required. This feature is quite important as a significant amount of each AST comprises references that point to a declaration such as class identifiers and variables. The majority of all `UElement` entities returned by the UAST parser are backed as well by a PSI element, which can be retrieved by

[5]`https://android.googlesource.com/platform/tools/base/+/studio-master-dev/lint/libs/lint-api/src/main/java/com/android/tools/lint/detector/api/SourceCodeScanner.kt`

[6]`https://groups.google.com/forum/#!topic/lint-dev/7nLiXa04baM`

Table 3.1: Overview of essential UAST interfaces

UAST Interface	**Purpose**
`UElement`	Basic interface for all UAST elements.
`UResolvable`	Represents a `UElement` that references another element in the code, like a method call or a used variable which both reference their declaration. Provides the `resolve` method to retrieve the referenced PSI element.
`UDeclaration`	Generic interface for all declarations: Classes, methods and variables.
`UClass`	Declaration of a class. Provides access to all further declarations, like methods and variables, within the class.
`UMethod`	Declaration of a method. Provides access to method parameters and body.
`UExpression`	Represents anything that can be evaluated like expressions and statements.
`UCallExpression`	Represents any method, constructor, or initializer call, and provides methods to gain access to the receiver, the arguments, and the called method.
`ULoopExpression`	Generic interfaces for all `for` and `while` loops.
`UForExpression`	Represents a `for` loop.
`UWhileExpression`	Represents a `while` loop.
`UForEachExpression`	Represents a `for` loop used to iterate over a collection of objects with the colon (`:`) notation.
`UReferenceExpression`	Represents any type of reference.
`USimpleNameReferenceExpression`	Represents plain, non-qualified identifiers.



```java
// Test.java
package test.pkg;

public class Test {

    int i = 110;

    public int
    calc(){
        return 2*i;
    }

}
```

```kotlin
// Test.kt
package test.pkg


class Test {
    var i : Int = 110

    fun calc():Int {
        return 2*i
    }

}
```

Figure 3.2: Representation of a Java and Kotlin class in UAST

the `getPsi()` method of the `UElement` interface. Consequently, all `UElement` entities, which are resolvable, must implement the `UResolvable` interface. Note that `getPsi()` might return `null` in case the `UElement` does not contain any PSI element. In general, UAST is used for everything within methods and for field initializers, whereas PSI is used on the "outer level" for declaration and signatures of methods, classes and packages.[7] In other words, when we resolve declarative information, *e.g.*, the receiver type of a `UCallExpression` or the class containing a `this` reference, the manipulation of PSI elements is still a necessity.

3.2.2 JUnit Tests

Android Lint provides a rich testing facility that is based on the well-known Java unit test framework JUnit 4.[8] The class `BaseLintDetectorTest` extends a plain JUnit `TestCase` and is the parent of the abstract class for all detector unit tests `LintDetectorTest`, which provides numerous methods that facilitate test initialization, configuration, execution, and verification of results.

Each detector test case has to override `getDetector` to return the detector class under test. In particular, `getIssues` must return a list of all issues to take care of, as one detector could implement multiple issues, however, not all of them may be suitable for any test. We built any `TestLintTask` by first calling

[7] https://android.googlesource.com/platform/tools/base/+/studio-master-dev/lint/libs/lint-api/src/main/java/com/android/tools/lint/detector/api/SourceCodeScanner.kt

[8] https://junit.org/junit4/

`LintDetectorTest.lint`, followed by the inclusion of files with the `TestLintTask.files` method. This `files` method expects each file's content represented as `string` value encapsulated in a `TestFile` object, and several static utility functions on the class `LintDetectorTest` provide support in building them. While all `java` and `kotlin` related methods in the aforementioned utility class describe source code files, the `xml` method describes XML files and it requires a string constant to distinguish between the different types of XML files used throughout Android projects. For our tool we exclusively rely on the `FN_ANDROID_MANIFEST_XML` XML type used in Android app's manifest.

After successful creation of a test case it can be executed by the `run` method. The Android Lint framework will then initiate the issue detection process and return a `TestLintResult`. The `TestLintResult` provides multiple hooks, such as `expectCount` and `expectMatches`, to verify the results of the issue analysis.

In our project we introduced a one to many relationship between detectors and `LintDetectorTest` classes as a result of some detectors that cover multiple issues and therefore require multiple test classes. Based on the complexity of the issue we implemented up to ten test cases. While the elementary issues found in the manifest file tend to only require few simple test cases, the opposite is true for the more complex issues in the source code. During our evaluation we encountered several unhandled exceptions raised within our detectors due to unexpected code statements used in some apps. As a consequence, we enhanced our test case set with code that caused the issue.

3.2.3 Analysis of Java & Kotlin Code

Android Lint provides multiple helper and utility classes to simplify the use of the UAST.

The relevant classes are namely:

- **JavaEvaluator:** The `JavaEvaluator` provides a wide range of methods to get more contextual information about PSI elements such as `PsiClass` or `PsiMethod`. Compatible with this helper class are all classes that implement at least a PSI interface. This also includes classes that implement both, PSI and UAST interfaces. Unlike the name suggests, this evaluator can be used for `JavaElements` and `KotlinElements` returned by UAST parsers. Crucial methods are `isMemberInClass` and `isMemberInSubClassOf` which allow reasoning about a method's origin. The `JavaEvaluator` can be received by calling `getEvaluator()` on a `JavaContext`.
- **ConstantEvaluator:** The `ConstantEvaluator` evaluates constant properties of ordinary `U-Elements`. The `evaluate()` method, for example, returns an `Object` that either precisely matches the value of the `UElement` or its referenced value. The evaluation is rather complex as the majority of the `UElement` subclasses demand for disparate treatment. For instance, while all types of literals can be converted effortlessly into corresponding Java objects, expressions need to be evaluated beforehand. Furthermore, `JavaEvaluator` is capable of determining the

last assigned value for local variables and fields. `UElements` must evaluate to primitives and strings, both of them possess literal expressions in Java and Kotlin, in order to foster correct resolving by the `ConstantEvaluator`. If the value of a `UElement` cannot be evaluated the `ConstantEvaluator` returns `null`.

- **UastUtils:** The `UastUtils` class contains multiple means to extract additional information from a `UElement`. More precisely, it provides methods such as `getContainingUMethod()` and `getContainingUClass()` to unwrap embedded `UMethod` and `UClass` objects of a `UElement`. Furthermore, these utilities also support `tryResolve()` for any resolvable `UElement` together with other resolving methods that return PSI elements.

Aside from the aforementioned classes there exist several extra such as `TypeEvaluator` and `ResourceEvaluator`, moreover, a wide range of helper and utility classes for the PSI are also present in current releases of Android Lint.

The implementation of the `Detector.UastScanner` interface is the key factor for the successful creation of a custom detector. For each `UastScanner` instance the framework traverses the UAST and calls the detector's `visit` implementation on previously registered element types. These implementations distinguish `UElement` objects of type method (`visitMethod`), constructor (`visitConstructor`), reference (`visitReference`), resource reference (`visitResourceReference`), and class (`visitClass`). Besides these visit methods, the `UastScanner` also provides measures to further restrict the visited elements by name. For example, the methods visited by the `visitMethod` can be constrained to the names in the list from `getApplicableMethodNames`. If this method returns `null`, no name restrictions will apply and all elements will be visited, which is the default behavior.

Each `visit` method retrieves the `JavaContext` as a parameter which is primarily used to determine the location of a node, but also to report an issue and to access the `JavaEvaluator`. Contrary to its name, the `JavaContext` works for Kotlin as well. The `visitMethod` is a core piece of all our detectors, and unlike the name implies it visits only method calls (`UCallExpression`), but no method declarations. Although `getApplicableMethodNames` restricts the visited methods by name, the `JavaEvaluator.isMemberInClass` or `JavaEvaluator.isMemberInSubClassOf` have still to be used to assure that the visited method does not conflict with another method of a different class. For some specific cases, *i.e.*, methods that only differ in parameters, the parameter type checks are necessary to assure a method matches the expected entity.

For elements that lack a corresponding visit method in the `UastScanner` interface a `UElementHandler` can be used instead. Such `UElementHandler` instances, returned by `createUastHandler`, only visit UAST elements of types returned by overridden `getApplicableUastTypes()` methods. In contrast to ordinary visitors, a `UElementHandler` is explicitly intended to check specific UAST elements, but not for generic tree traversal. With the aim of visiting any (partial) tree the `AbstractUastVisitor` must be implemented instead. Therefore, `AbstractUastVisitor` instances are used in our extension to evaluate all statements within a method, whereas the actual visiting mechanism is initiated by passing that visitor to the `UMethod.accept` method. To retrieve the surrounding `UMethod` of a `UElement` we used the `UastUtils.getContainingUMethod()` helper method.

More complex security issues, *e.g.*, issues that require multiple methods on a specific object to be called in order, that is, issues which involve multiple separated nodes in the UAST, do not align well with the `Detector.UastScanner` interface, as it is primarily intended to find individual nodes. A prime example of a complex security issue is the ICC security code smell *Unauthorized Intent* that depends on specific method calls executed on the intent object to distinguish between an implicit or an explicit intent. *Unauthorized Intent* requires exact knowledge of (i) the source of the intent, *i.e.*, the location of the object creation or retrieval, (ii) any state-changing modifications to the intent, *i.e.*, methods executed on the intent, and finally, (iii) the method that sends the intent. In general, these complex issues would yield better results with taint analysis, which tracks data flows from sensitive sources to insecure sinks, or dynamic analysis techniques, because some of the properties are unavailable in source code, *e.g.*, the properties of intents that are retrieved from external apps during run time.

Based on the limitations of Android Lint, we focus in our work on the evaluation within particular methods, also known as intra-method analysis. Due to our intra-method approach we first search for a method call which could lead to an issue based on its arguments. We term this method call *sink* and the method containing the sink a *base method*. Next, within the base method, we start to resolve the relevant method arguments to find their *sources*, *i.e.*, a constructor or the previous assignment of these variables. Finally, we perform further evaluations on argument objects of the method calls. Because our search is constrained to the base method we are unable to find sources of arguments that originate from classes or methods other than the base method. For that reason we are unable to detect changes applied to argument objects which are accessible outside of the base method. If an argument is considered problematic due to its source or state changes, the detector will report the issue in the IDE at the location of the sink. Figure 3.3 exemplifies our intra-method analysis for the *Unauthorized Intent* issue.

3.2.4 Analysis of Manifest

In addition to plain Java and Kotlin code, Android Lint supports the analysis of eXtensible Markup Language (XML) data. Android uses XML for various configuration files, such as resource files that include static content required by the application to build its views, localize string tables, and assign variable values, *e.g.*, strings, numbers, and colors. A crucial XML file of any Android application is the `AndroidManifest.xml`, which contains the essential configuration of the application including numerous security relevant parameters such as availability of public interfaces, or custom permissions defined to protect specific components. Each Android Lint issue must define its related `scope`, *i.e.*, the type of files to consider for its analysis. The evaluation of the `AndroidManifest.xml`, for example, requires `Scope.MANIFEST_SCOPE`. Besides the source code scopes we solely rely on the XML-based scope `Scope.MANIFEST_SCOPE` in our tool, as we never analyze any further resource files.

```
public class MainActivity extends Activity {

     @Override
     protected void onCreate(Bundle instanceState) {
          Intent i = new Intent("test.pkg.ACTION");
          i.setClassName("test.pkg", "Receiver");
          sendBroadcast(i);
     }
}
```

Locate the "sink", here sendBroadcast(i)

```
public class MainActivity extends Activity {

     @Override
     protected void onCreate(Bundle instanceState) {
          Intent i = new Intent("test.pkg.ACTION");
          i.setClassName("test.pkg", "Receiver");
          sendBroadcast(i);
     }
}
```

Locate the "source" of the intent argument i within the sourrounding method. It is an intent constructor without an explicit target, just an action thus we still consider our intent implicit.

```
public class MainActivity extends Activity {

     @Override
     protected void onCreate(Bundle instanceState) {
          Intent i = new Intent("test.pkg.ACTION");
          i.setClassName("test.pkg", "Receiver");
          sendBroadcast(i);
     }
}
```

Try to find further modifications of i. We find `i.setClassName("test.pkg", "Receiver")`, `setClassName()` sets an explicit target for an intent therefore it is no longer implicit and we can be sure it reaches only the intended target. Thus we do not report anything.

Figure 3.3: Intra-method analysis example of an *Unauthorized Intent* issue within the `Activity.onCreate` method

As expected, the `Detector.XmlScanner` interface is very similar to the `Detector.UastScanner` interface. The interface allows a detector to restrict the access to certain types of XML elements by overriding `getApplicableElements` which returns a list of element names (*i.e.*, strings) the detector has to visit, or `null` if the detector should visit all elements which is again the default behavior. During the visiting process `visitElement` is called on any matching element. The same applies for attributes. `visit-Attribute` either visits every attribute, or only those returned by `getApplicableAttributes` in case they are not `null`. The `XmlContext` class provides measures to report an issue including the affected element's location in the code. Therefore, all `visit` methods receive the `XmlContext` object as parameter. Each of our detectors usually identifies specific attributes of a certain element type, hence our tool is able to merely visit the interesting elements by implementing `getApplicableElements` accordingly. Thereafter, in the method `visitElement`, we retrieve each element's attributes of importance and verify its values. Six detectors implement the `Detector.XmlScanner` interface in our tool.

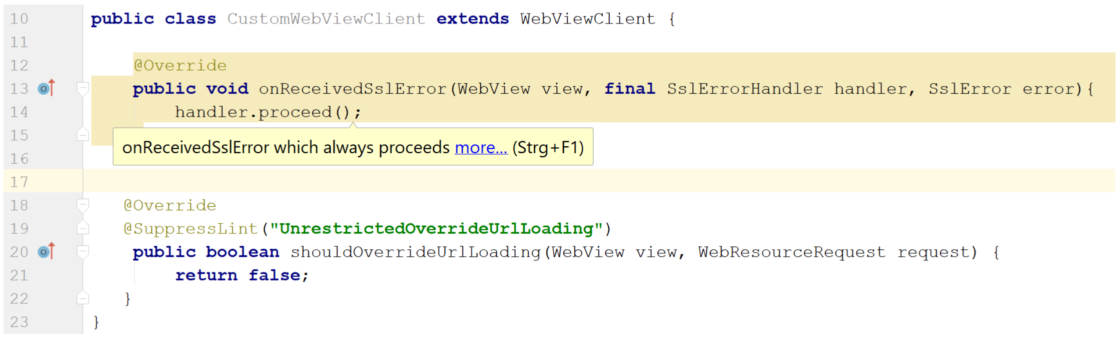
3.2.5 Collective Reports

Issues are either reported just-in-time within the IDE through markers and hints, or exported into collective HTML and XML reports. Each generated collective report contains a definite collection of issue occurrences in an app, and includes for each a specific message, the reported issue, and the location of the issue in the code. We used this export feature on a large set of open source apps to further evaluate our tool and a plethora of apps. A detector can report an issue by calling `report` on the context it receives within the different visit methods.

The context method `report` takes up to five arguments:

- **issue**: The `issue` instance the detector reports.
- **scope**: The Android Lint framework will check for suppressive directives, that would cancel the report if necessary, with respect to the element itself or elements enclosing it. A `scope` can be any `UElement` if the issue remains in the UAST, or be any `Node` if the issue resides in the manifest file.
- **location**: A `location` object contains the actual file, lines and columns of the content to be highlighted, furthermore, the `location` is exported to all HTML and XML reports. The current `location` can be retrieved with the context's `getLocation` method within any `visit` method.
- **message**: A string that describes the problem in short.
- **quickfixData**: Defines a quick fix which can be applied in the IDE to resolve the issue without any further user interaction. Quick fixes are not yet implemented in our tool.

In Android Studio each reported issue is highlighted in the source file representation according to its `location`. In addition, a short tooltip with the provided `message` is displayed as soon as the user

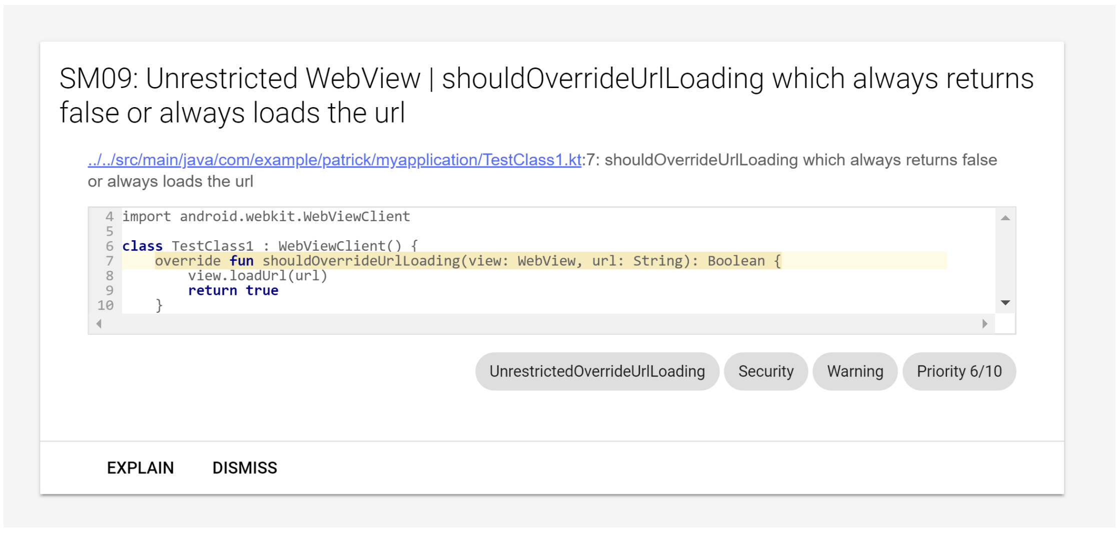


```
public class CustomWebViewClient extends WebViewClient {

    @Override
    public void onReceivedSslError(WebView view, final SslErrorHandler handler, SslError error){
        handler.proceed();
 onReceivedSslError which always proceeds more... (Strg+F1)


   @Override
   @SuppressLint("UnrestrictedOverrideUrlLoading")
    public boolean shouldOverrideUrlLoading(WebView view, WebResourceRequest request) {
        return false;
    }
}
```

Figure 3.4: A reported issue in the Android Studio. The tooltip shows a brief description of the problem. An issue in the `shouldOverrideUrlLoading` method is suppressed by the `@SuppressLint` annotation.



SM09: Unrestricted WebView | shouldOverrideUrlLoading which always returns false or always loads the url

../../src/main/java/com/example/patrick/myapplication/TestClass1.kt:7: shouldOverrideUrlLoading which always returns false or always loads the url

```
import android.webkit.WebViewClient

class TestClass1 : WebViewClient() {
    override fun shouldOverrideUrlLoading(view: WebView, url: String): Boolean {
        view.loadUrl(url)
        return true
    }
```

Figure 3.5: One reported issue out of a collective HTML report.

hovers over the highlighted area, and when users click on “more” they will see a more comprehensive `explanation` of the issue. An example report in Android Studio is shown in Figure 3.4.

A single reported issue out of a collective HTML report looks similar to a reported issue in the IDE. Nevertheless, the HTML report also provides additional features, such as an overview with hyperlinks to each issue description. For collective reports each individual report is grouped based on issue categories, and each report presents a hyperlink to the affected file and `message` to the user. If a `location` has been provided, a code snippet including the highlighted `location`, is further shown to the user. When users click on the “explain” button they will see the `explanation` of the issue.

An example issue of a collective HTML report can be seen in Figure 3.5.

To turn off a single report in Java or Kotlin code a `@SuppressLint` annotation must be added to the enclosing method, or class, as a suppressive directive. The argument of the annotation can either be the `id` of a specific issue, or “all” to suppress any issue in the subsequent code section. Figure 3.4 shows the use of `@SuppressLint`. Different to the suppression of issues in Java or Kotlin code, XML code requires the addition of the `tools:ignore` attribute to the (parent) element. The value of `tools:ignore` should contain the `id` of each issue that should be ignored, or similarly, “all” if all issues should be ignored in any child element.

3.3 Additional Tools

Android Lint is already integrated into the build process of any Android Gradle project. This enables the use of Android Lint in batch mode on Gradle projects: provided the projects build successfully, any number of projects can be analyzed by an initiation of their regular build process. To further streamline the automated evaluation, we created two tools, one to run the analysis and another to collect the results. However, we encountered several difficulties. For example, the Gradle project configurations have to be compatible with recent releases of the `Android Build Tools` and the Gradle, especially since an upgrade of these build tools to at least version 3.2.0 is mandatory for UAST support in Android Lint. Moreover, any prior configuration modifications that disable Android Lint’s HTML or XML reports must be removed. To facilitate this task we implemented a script that patches all disadvantageous settings before it initiates a Gradle build. After successful analyses, the results of the generated XML reports are backed up and aggregated by additional scripts. These aggregating scripts collect the number of reports per issue for each project and write the result to a `.csv` file that eases further evaluation with other applications.

4
App-level Security

In this chapter we describe Inter Component Communication (ICC) threats in Android and present a list of common ICC Security Smells that are detected by our Android Lint extension.

4.1 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 the 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.

4.2 ICC Security Smells

In order to answer *"What are the known ICC security code smells?"*, 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.

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. We were essentially interested in any paper explaining an ICC-related issue, and any countermeasures that involve ICC communication in Android. We used a keyword search over the title and abstract of papers in IEEE Xplore and the ACM Digital Library, as well as those indexed by the Google Scholar search engine. 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*. We read the title and, if necessary, skimmed the abstract of each paper, and included all security-related ones. We then read the introduction of these papers, and excluded those that were not primarily concerned with app security. In order to extend the search, for each included paper we also recursively explored both citing and cited papers until no new related papers were found. Finally, we carefully reviewed all remaining papers. During the whole process, we resolved any disagreement by discussion.

We further studied the well-known DroidBench[1] and the Ghera[2] benchmarks for our evaluation, both built with a focus on ICC. We collected the symptoms, the smells, and the corresponding vulnerabilities these benchmark suites revealed to further refine our smell list.

We identified twelve ICC security code smells. 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.

SM01: Persisted Dynamic Permission. Resources provided by Android applications can be obtained through Uniform Resource Identifiers (URIs), if access has been granted during installation or run time.

[1]`https://github.com/secure-software-engineering/DroidBench`
[2]`https://bitbucket.org/secure-it-i/android-app-vulnerability-benchmarks`

Issue: Access granted at run time is often intended to be temporary, but if the developer forgets to revoke the access grant, it 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 [12].
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 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. Depending on a permission's protection level, the system might grant the permission automatically without notifying the user if the applications is created by the same developer (*signature* protection level), or after the user approval during the app installation (*normal* protection level), or may prompt the user to approve the permission at run time (*dangerous* protection 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 [7].
Consequently, apps that were not intended to retrieve the permission may still declare its use and access

protected features.
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 as 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. As a result, any call that is able to submit intents supports the declaration of a permission that a potential receiver must match in order to receive the intent. These intents and receivers are "protected".
Issue: Any app can send an unprotected intent, or it can register itself to receive unprotected intents.
Consequently, apps could escalate their privileges by sending intents to unprotected 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 [2].
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`, `sendBroadcast-AsUser`, 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 wherever possible. 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 [7].
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 the existence of methods such as `sendStickyBroadcast`, `sendSticky-BroadcastAsUser`, `sendStickyOrderedBroadcast`, `sendStickyOrderedBroadcast-AsUser`, `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 [7]. *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,[3] that provide information about the threat level of a website.

SM07: Broken Service Permission. Two different mechanisms exist to implement 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. The problem is caused by flawed permission checks that will verify the callee's permissions instead of the caller's permissions.
Consequently, a privilege escalation could occur [7].
Symptom: The lack of appropriate permission checks to ensure that the caller has access right to request the service by calling `startService`.
Detection: We report the smell when the caller uses `startService`, and then the callee uses `check-CallingOrSelfPermission`, `enforceCallingOrSelfPermission`, `checkCallingOr-SelfUriPermission`, 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 calls to `check-Permission`, `checkUriPermission`, `enforcePermission`, or `enforceUriPermission` methods on the `Context` object, when additional calls to `getCallingPid` or `getCallingUid` on

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

the `Binder` object exist.
Limitation: We currently do not distinguish between checks executed in `Service.onBind` or `Service.onStartCommand`, and we do not verify other custom permission checks.
Mitigation: Wherever feasible set permissions to protect a service in the manifest and avoid custom run time checks within the service implementation. Otherwise, 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.

SM08: Insecure Path Permission. When sharing data with other apps, besides regular permissions that apply to the whole of a content provider, it is possible to set path-specific permissions that are more fine-grained.
Issue: The path-permission check in the manifest file differentiates between paths containing double slashes and paths with one slash. Hence, if there is a mismatch the permission only on the whole content provider is considered. However, the `UriMatcher` provided by the Android framework, which is recommended for URI comparison in the `query` method of a content provider, considers such paths to be identical, and will forward the request to the initially intended resource.
Consequently, access to presumably protected resources may be granted to unauthorized apps [7].
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 permissions should take precedence over those with larger scope.
Issue: A path permission does not take precedence over permission on the whole provider due to a bug that we identified in the `ContentProvider.enforceReadPermissionInner()` method in recent releases of the Android framework.[4] Therefore, any access grant to a provider discloses all protected subpaths.
Consequently, content providers may mistakenly grant access to other apps.
Symptom: The 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.

[4]The bug can be found at line 574. The class is publicly available at `https://android.googlesource.com/platform/frameworks/base/+/oreo-r6-release/core/java/android/content/ContentProvider.java`

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 [7].
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 [7] 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 call.
Limitation: Arrays of pending intents are not yet supported in our analysis.
Mitigation: Use explicit pending intents, as recommended by the official documentation.[5]

SM12: Common Task Affinity. A *task* is a collection of activities that users interact with when carrying out a certain job.[6] 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.
Consequently, malicious apps may hijack an app's activity paving the way for various kinds of spoofing attacks [10].
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.

[5] `https://developer.android.com/reference/android/app/PendingIntent.html`
[6] `https://developer.android.com/guide/components/activities/tasks-and-back-stack.html`

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 4.1.

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

Table 4.1: The relationship between vulnerabilities and security code smells

5 Empirical Study

In this section we first introduce a dataset of more than 700 open-source Android projects that are mostly hosted on GitHub, and we present the results of our investigation into RQ_2 and RQ_3 by analyzing the prevalence of security smells in our dataset. We finish this chapter with a discussion of the manual evaluation of the tool's results.

The results in section 5.2 suggest that fewer than 10% of apps suffer from more than two ICC security smells. With respect to app volatility, we discovered that updates rarely have any impact on ICC security, however, in case they have, they often correspond to new app features. Moreover, the findings of Android Lint's security checks correlate to our detected security smells.

According to our manual investigation in section 5.3, we confirm that our tool successfully finds many different ICC security code smells, and about 48% of them in fact represent vulnerabilities. The tool can consequently offer valuable support in security audits.

We performed analyses similar to prior 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 [4]. Our results are therefore in line with 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.

5.1 Dataset

We collected all open-source apps from the F-Droid[1] repository as well as several other apps directly from GitHub.[2] In total we collected 3 471 apps, of which we could successfully build 1 487 (42%). 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, corresponding to 108 files.

5.2 Batch Analysis

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

5.2.1 Prevalence of Security Smells

Figure 5.1 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%). *Custom Scheme Channel* and *Implicit Pending Intent* each contribute about 8% of the smells. 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 Figure 5.2). 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.

5.2.2 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. 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.

[1]`https://f-droid.org/`
[2]`https://github.com/pcqpcq/open-source-android-apps`

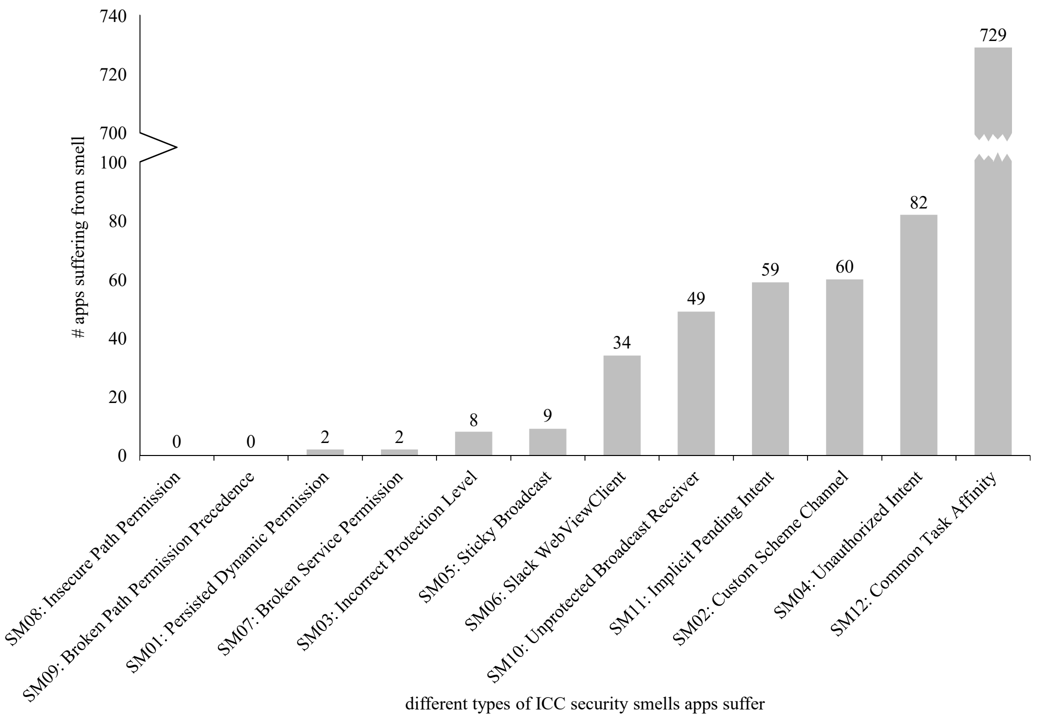


Figure 5.1: Distribution of security smells in the apps

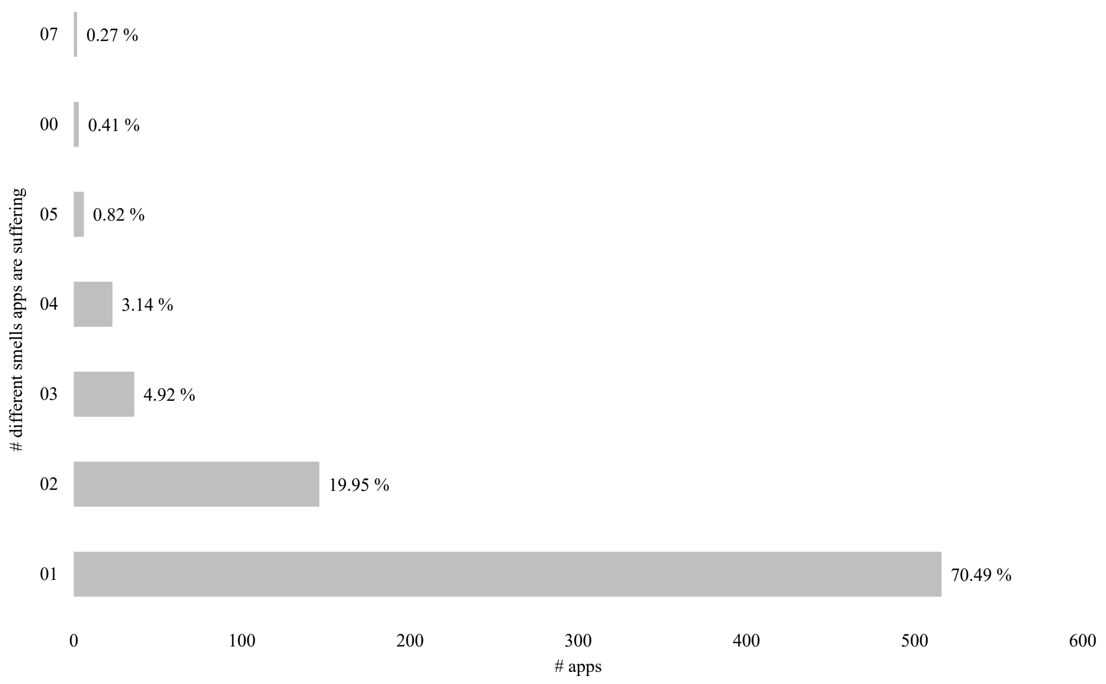


Figure 5.2: Partitioning apps by number of security smells

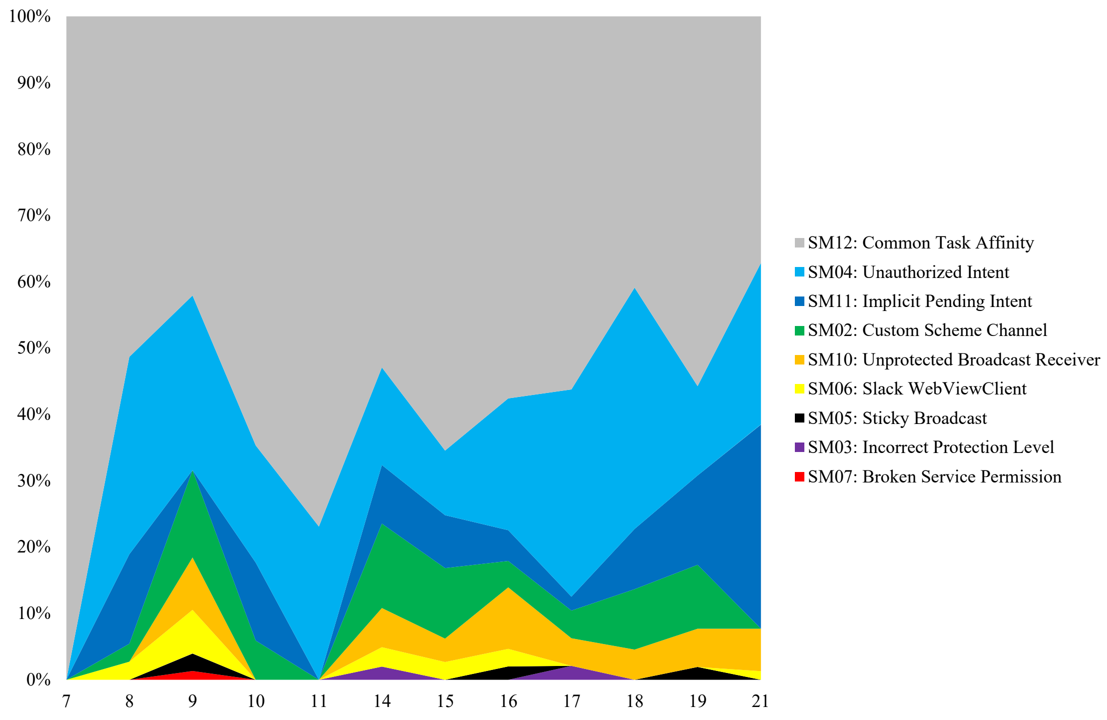


Figure 5.3: Evolution of security code smells in different Android releases

For the majority of the app updates that introduced new security smells, we found that the dominant cause for decreased security is the implementation of new ICC functionality, *i.e.*, social interactions or data sharing. Hence, developers should be particularly cautious when integrating new functionality into an app.

5.2.3 Evolution

Figure 5.3 shows the evolution of security smells across Android releases 7 through 21. As in previous work, we see changes in some of the security smells apps suffer. 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 work: 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.

	ICC Security Smells	Correctness	Correctness: Messages	Security	Performance	Usability: Typography	Usability: Icons	Usability	Accessibility	Internationalization	Internat.: Bidirectional
ICC Security Smells		0.29	0.27	0.72	0.25	0.21	0.11	0.07	0.25	0.13	0.11
Correctness			0.42	0.45	0.73	0.50	0.49	0.52	0.57	0.53	0.57
Correctness: Messages				0.27	0.42	0.37	0.20	0.27	0.41	0.25	0.32
Security					0.38	0.27	0.26	0.26	0.31	0.20	0.26
Performance						0.49	0.47	0.50	0.61	0.57	0.60
Usability: Typography							0.27	0.34	0.38	0.32	0.36
Usability: Icons								0.29	0.37	0.35	0.37
Usability									0.45	0.40	0.40
Accessibility										0.48	0.49
Internationalization											0.51
Internat.: Bidirectional											

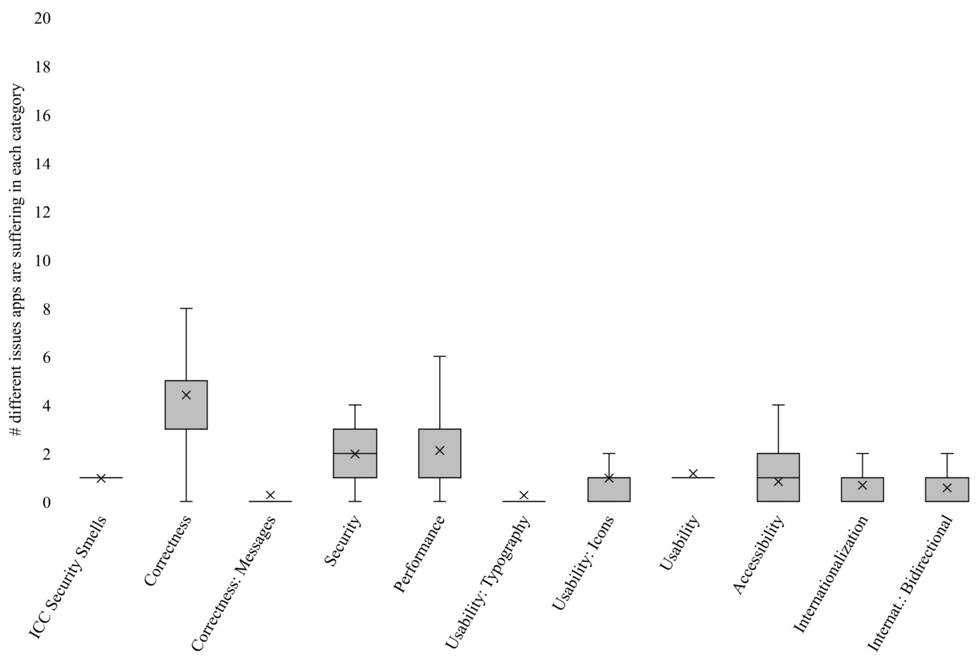
Figure 5.4: Correlation matrix of the different Android Lint issue categories

5.2.4 Comparison to Existing Android Lint Checks

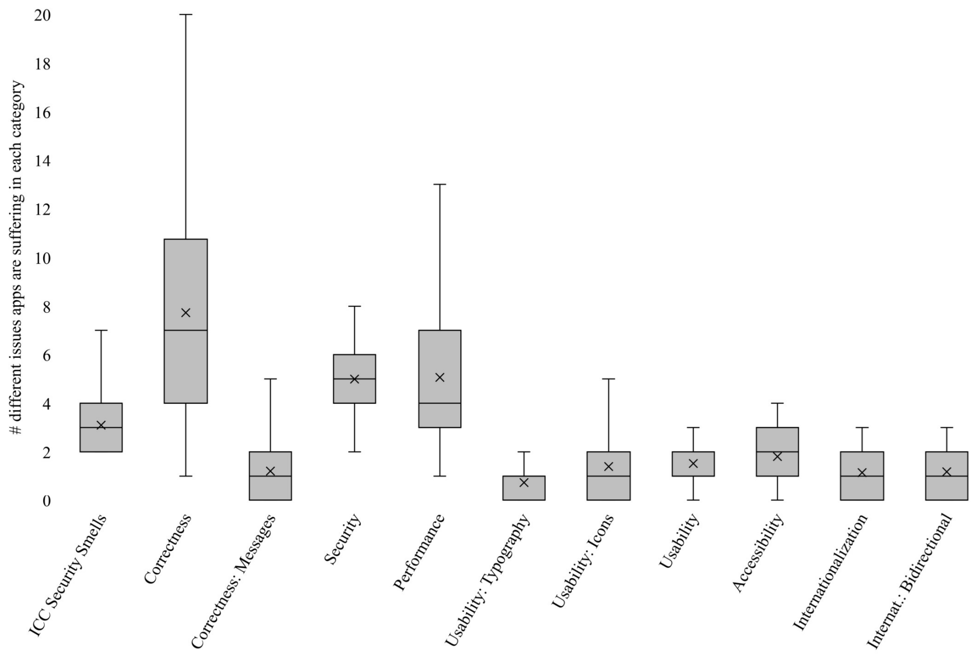
In order to compare our findings with other issues in the apps, we correlated the results from the existing Android Lint framework with security code smells. We wanted to explore whether frequent reports of specific Android Lint issue categories were also indicative of security issues. We collected all available issue reports for each app and then extracted the occurrences of each detected issue. Android Lint categorizes more than 300 issues into 11 different categories; issues in the category “Correctness: Chrome”, which includes checks that ensure compatibility to Google’s Chrome OS, were almost never detected, and thus it has been removed from the dataset.

We apply the Pearson product-moment correlation coefficient algorithm. 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 Figure 5.4 reveals several interesting findings: (i) Our ICC security category strongly correlates to 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 web browsers. (ii) Another discovery is the minor correlation between the ICC security smells and the Android Lint correctness category (+0.29). This category includes checks about erroneously configured project build parameters, incomplete view layout definitions, and usages of deprecated resources. (iii) Furthermore, we assume that usability does not impede security (+0.07), because issues in usability are closely related to UI mechanics. (iv) Finally, minor correlations are shown for usability, 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. In Figure 5.5 the least and most affected apps clearly correspond in terms of issue frequency among specific categories. The crosses represent the mean value of the number of different issues apps are suffering in each category, and, as we hid any outliers to increase readability, these values can exceed the first quartiles.



(a) 100 least vulnerable apps



(b) 100 most vulnerable apps

Figure 5.5: Prevalence of Android Lint issues in the 100 most and least vulnerable apps

5.3 Manual Analysis

To assess how reliable these findings are to detect security vulnerabilities, we manually analyzed 100 apps. Therefore, we invited two participants independently, to evaluate our tool and report their findings. Participant A is a junior developer with less than two years of experience, and participant B is a senior developer with several years of experience in 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 with most smells in accordance with our ICC security smell list, and provided the participants with our tool and a spreadsheet to record their observations. They were asked not only to evaluate the smells reported by our tool, but also to investigate all occurrences of the described symptoms of any smell in chapter 4. Moreover, they evaluated the vulnerability potential for each security smell based on the vulnerability information available in the benchmarks.

5.3.1 Tool Performance

To evaluate the performance of the tool, we selected the tool's proposal of smells (true and false positives), and the proposals from both study participants (true positives). Both participants evaluated all smells detected by our tool, and additionally searched for false negatives according to the ICC security smell list. We consider the ground truth to be the union of the evaluation results of participants A and B. We obtained quite high success rates, especially for SM02, SM03, and SM05, as shown in Figure 5.6. We observed that participants tended to interpret diversely the threat caused by the *Unauthorized Intent* smell. We assume that this is caused by the very complex and flexible implementation that has been provided by Android. As expected, we were able to find false positives, however, only a few false negatives remained as we continuously improved our tool. Some of the false negatives, however, were caused by Android Lint encountering errors in file parsing. Despite the Lint failures, false positives were frequently caused by the lack of context, *e.g.*, unawareness of data sensitivity, or custom logic that mitigates the vulnerability. For example, our tool was unable to verify custom web page white-listing implementations for `WebView` browser components, which would actually improve security.

5.3.2 Common Security Smells

Here too we made some interesting observations. A major discovery was the inappropriate use of regular broadcasts for intra-app communication. For these scenarios, developers should solely rely on the `Local-BroadcastManager` 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.

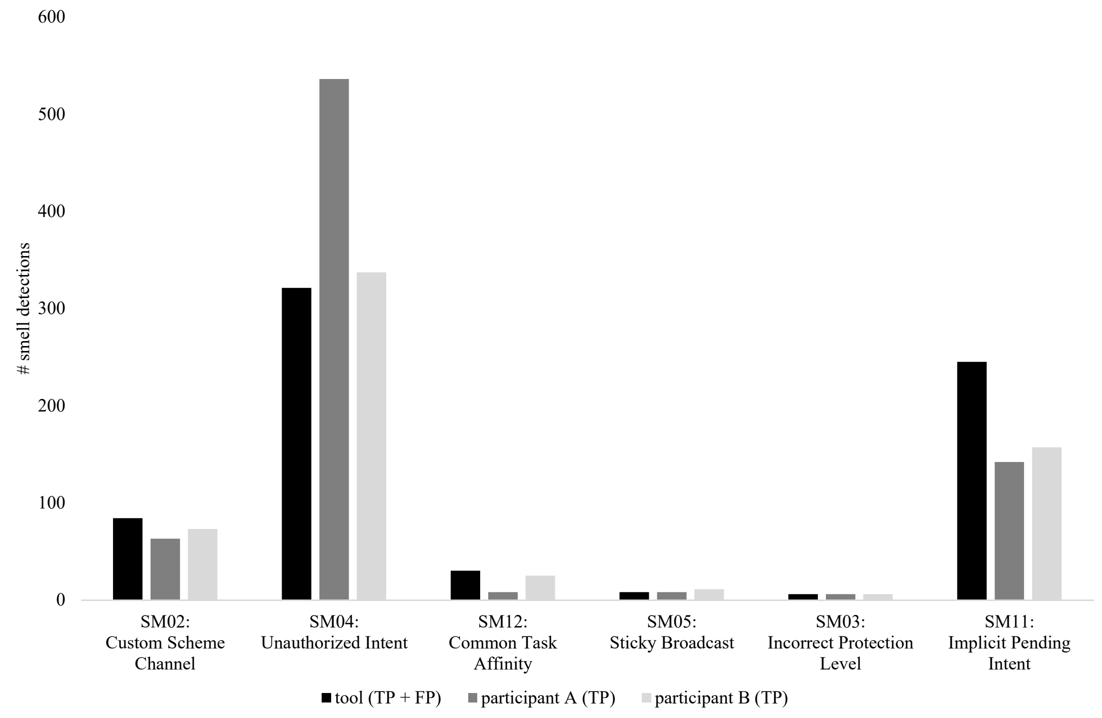


Figure 5.6: Tool performance

In conclusion to the remaining reports of the two reviewers, our tool was able to correctly detect the security risk in 48% of cases, which is mainly due to the fact that discerning data sensitivity is non-trivial.

5.4 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. We could not review all the publications, but 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 citations 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 market.

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 are in fact true smells only if they deal with sensitive data, but our analysis cannot determine such sensitivity.

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 reviewer in the process.

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 fewer than 10% of apps suffer from more than two ICC security smells, and discovered that updates rarely have any impact on ICC security, however, in case they have, they often correspond to new app features. Thus developers have to be very careful about integration of new functionality into their apps.

A manual investigation of 100 apps shows that our tool successfully finds many different ICC security code smells, and about 48% 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.

Bibliography

[1] R. Balebako and L. Cranor. Improving app privacy: Nudging app developers to protect user privacy. *IEEE Security Privacy*, 12(4):55–58, July 2014.

[2] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[3] L. N. Q. Do, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. Murphy-Hill. Cheetah: Just-in-time taint analysis for Android apps. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 39–42, Piscataway, NJ, USA, 2017. IEEE Press.

[4] M. Ghafari, P. Gadient, and O. Nierstrasz. Security smells in Android. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 121–130, Sept 2017.

[5] B. H. Jones and A. G. Chin. On the efficacy of smartphone security: A critical analysis of modifications in business students practices over time. *International Journal of Information Management*, 35(5):561 – 571, 2015.

[6] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in Android apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 280–291, Piscataway, NJ, USA, 2015. IEEE Press.

[7] J. Mitra and V.-P. Ranganath. Ghera: A repository of Android app vulnerability benchmarks. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 43–52. ACM, 2017.

[8] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. 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)*, pages 543–558, Washington, D.C., 2013. USENIX.

[9] M. Peck, K. Gananand, and A. Pyles. Android security analysis final report. *MITRE Technical Papers*, Apr. 2016.

[10] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu. Towards discovering and understanding task hijacking in Android. In *USENIX Security Symposium*, pages 945–959, 2015.

[11] Y. Tymchuk, M. Ghafari, and O. Nierstrasz. JIT feedback — what experienced developers like about static analysis. In *Proceedings of the 26th IEEE International Conference on Program Comprehension (ICPC'18)*, 2018.

[12] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *ACM Conference on Computer and Communications Security*, 2013.

[13] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on Android security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 623–634, New York, NY, USA, 2013. ACM.

[14] J. Xie, H. R. Lipford, and B. Chu. Why do programmers make security errors? In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 161–164, Sept 2011.

[15] M. Xu, C. Song, Y. Ji, M.-W. Shih, K. Lu, C. Zheng, R. Duan, Y. Jang, B. Lee, C. Qian, et al. Toward engineering a secure Android ecosystem: A survey of existing techniques. *ACM Computing Surveys (CSUR)*, 49(2):38, 2016.

A
Anleitung zum wissenschaftlichen Arbeiten

The Anleitung consists of the journal paper “Security Code Smells in Android ICC”.[1]

P. Gadient, M. Ghafari, P. Frischknecht, and O. Nierstrasz. Security code smells in Android ICC. Submitted to *Empirical Software Engineering Special Issue: SCAM 2017*, 2018.

[1] `http://scg.unibe.ch/download/supplements/Security-Smells-in-Android-ICC-(Submission).pdf`