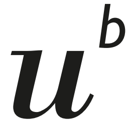

b
UNIVERSITÄT
BERN

Developer Tool Support for Security Code Smells

Bachelor Thesis

Dominik Briner
from
Densbüren AG, Switzerland

Faculty of Science
University of Bern

31 July 2021

Prof. Dr. Oscar Nierstrasz
Pascal Gadient

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

Abstract

Mobile apps tend to gain new features with every release, and startups continue to develop services without thorough experience. Unfortunately, such an environment is a breeding ground for security code smells, and the automatic mitigation of those smells is complex and requires knowledge about the context. Hence, they still remain in many apps.

In this work, we explore the possibility of quick fixes for security code smells, *i.e.*, we provide a plug-in that seamlessly integrates with Android Studio and supports the detection of six different security code smells. We performed a brief manual evaluation of our tool on 25 random apps from the *F-Droid* app store as well as some additional apps from the previous study, and checked whether the reported security smells are correct, and if they can be automatically mitigated with a quick fix. The results are promising. According to our manual validation, the tool identified 92 security code smells, of which it was able to fix more than 91%, many of them without any additional intervention of the developer.

Contents

1 Introduction **1**

2 Background **3**
2.1 Features . . . 3
2.1.1 Inspection . . . 3
2.1.2 Quick fix . . . 4
2.2 Code Abstraction . . . 4
2.2.1 AST . . . 5
2.2.2 PSI . . . 5
2.2.3 UAST . . . 7

3 Implementation **9**
3.1 Persisted Dynamic Permission . . . 11
3.2 Missing Protection Level . . . 13
3.3 Unprotected Implicit Intent . . . 14
3.4 Sticky Broadcast . . . 16
3.5 Implicit Pending Intent . . . 16
3.6 Common Task Affinity . . . 17

4 Framework Limitations **19**
4.1 IntelliJ Framework . . . 19
4.1.1 Lack of Context . . . 19
4.1.2 Lack of Continuity . . . 20
4.1.3 Lack of Documentation . . . 20
4.1.4 Lack of Performance . . . 20
4.1.5 Lack of User Characteristics . . . 21
4.1.6 Lack of User Feedback . . . 21
4.1.7 Lack of UI Integration . . . 21
4.2 Unsupported Security Code Smells . . . 22
4.2.1 Custom Scheme Channel . . . 22
4.2.2 Slack WebViewClient . . . 22

4.2.3 Broken Service Permission 22
4.2.4 Insecure Path Permission 23
4.2.5 Broken Path Permission Precedence 23
4.2.6 Unprotected Broadcast Receiver 23
4.3 Discussion 24

5 Evaluation **25**
5.1 Methodology 25
5.2 Results 26
5.2.1 Persisted Dynamic Permission 26
5.2.2 Missing Protection Level 26
5.2.3 Unprotected Implicit Intent 27
5.2.4 Sticky Broadcast 27
5.2.5 Implicit Pending Intent 27
5.2.6 Common Task Affinity 28
5.3 Discussion 28

6 Threats to Validity **30**

7 Related Work **32**
7.1 Tool Support 32
7.2 Tool Usage 34
7.3 Security Smells 34

8 Conclusion **36**

A Anleitung zum wissenschaftlichen Arbeiten **40**
A.1 Implementation 40
A.1.1 Inspection 40
A.1.2 Quick Fix 43
A.1.3 UI Controls 46
A.2 Execution 47
A.3 Packaging 48

1 Introduction

Security code smells, *i.e.*, symptoms in the code that signal the prospect of a security vulnerability, are an emerging threat in the Android mobile operating system, as shown in previous research [11]. Researchers found that despite the diversity of apps in popularity, size, and release date, the majority suffer from at least three different security code smells, and that inter-component communication (ICC)-related smells are the most prevalent. As a result, researchers began to study ICC-related security code smells in more detail, including how they could be eliminated or mitigated during development [8]. However, tool support is missing to easily mitigate such smells.

In this thesis, we build on the previous work and implement tool support for six of the twelve reported ICC security code smells for Google's official IDE for Android developers, *i.e.*, Android Studio. In particular, we designed interactive quick fixes that can resolve ICC security code smells in the favor of the developer without compromising the functionality of the existing code. In order to leverage such functionality, a developer has to install and enable our Android Studio plug-in.

The difficulty in designing quick fixes for security smells is, besides the technical complexity and the high update cadence of Android Studio, the lack of context to create reasonable fixes for a particular smell instance. For example, there exist multiple different solutions if a developer forgot to add a permission to an implicit intent. The suitability of each solution depends on the future needs of the developer, and cannot be accurately predicted. Therefore, we interactively ask the developer for additional

context whenever we encounter a lack of information.

The ultimate goal of this work is to investigate how quick fixes can help developers to avoid potential issues before they could lead to vulnerabilities, and the limitations of these fixes. In essence, this thesis addresses the following research question: *Can security code smells be mitigated with the help of quick fixes, and if yes how successful are they?*

Advancing the previous work, we implemented quick fixes for six of the twelve reported Android ICC security code smells. With respect to their implementation, we comprehensively discuss the technical difficulties and trade-offs, the resulting solution strategies, and the impact on app development. We manually inspected 25 apps, and compared our findings to the result of the linting tool from previous work. We found that the quick fix plug-in accurately detects smell instances, and is able to fix them in 77% of all cases.

In summary, this work continues the journey of preventing security code smells in Android mobile apps. The ease of use combined with the continuous guidance of the user let especially inexperienced Android developers highly benefit from this tool.

The remainder of this thesis is organized as follows. We provide the required background in chapter 2, we discuss the quick fix tool architecture and the implemented remediation strategies in chapter 3, and we review the limitations of the tool in chapter 4. We present the results of our evaluation in chapter 5. Finally, we recap the threats to validity in chapter 6, and we summarize related work in chapter 7. We conclude this paper in chapter 8.

2 Background

This section provides basic knowledge required for the subsequent chapters. In particular, we explain important features of the Android Studio IDE and its code abstraction API for traversing the *Abstract Syntax Tree (AST)*.

2.1 Features

Google advertises Android Studio as the official IDE for Android app development and Kotlin as the programming language of choice. Android Studio is based on the JetBrains IntelliJ IDE for Java, which is bundled with a custom Android plug-in from Google.[1] Consequently, Android Studio is compatible with most IntelliJ extensions and supports the same APIs for code manipulation. Due to these similarities, for the remainder of this work we only refer to the JetBrains IntelliJ IDE.

2.1.1 Inspection

A basic feature of IntelliJ is that of inspections. Inspections can highlight source code in the IDE based on rules, *i.e.*, custom code that matches a specific pattern. Whenever the user hovers with the mouse cursor over code with an active inspection, it displays a tool tip text with relevant information. The code is highlighted differently depending on

[1] `https://developer.android.com/studio/intro`

the severity level of the inspection.[2] Using the default settings, for example, code that raises the severity level *error* is underlined in red, whereas it has a yellow highlighting without any underlines when it raises a *warning*. These levels have distinct meanings, *i.e.*, *error* is used for syntax errors that prevent the successful execution of an app, code that executes but might break during run time, *e.g.*, unchecked type conversions, is labelled as a *warning*, and *weak warning* is used to indicate potential optimizations, such as possibly redundant code. Ultimately, their visuals can be customized and new severity levels can be introduced by the user.

In IntelliJ a background task iterates after each change over all registered inspections for each project file and determines what part of the source code requires visual indicators. Plug-ins can implement and register such inspections; the severity level, the displayed information, and the algorithm that determines which code to highlight are parameterized.

2.1.2 Quick fix

Another important feature of the IntelliJ platform is that of quick fixes. Quick fixes contain recipes, *i.e.*, actions to fix specific problems in source code. On any code, the user can use a keyboard shortcut to display available actions, including those offered by a quick fix implementation.[3] The availability of actions depends on the context and specifically on the position of the cursor. For instance, an action to annotate a method with `@Override` is only presented when the cursor is placed on a method declaration which indeed overrides a method of its parent class. It is common to have a quick fix for problematic code that triggers inspections of the severity level *error*, *e.g.*, when there is a mismatch between a class name and its constructor, a quick fix offers a refactoring to rename either the class or the constructor to resolve the issue. As for inspections, the quick fix feature is part of IntelliJ's core engine and can be used in plug-ins. Moreover, a quick fix can be attached to a specific inspection so that it is automatically presented at all code instances that trigger that particular inspection.

2.2 Code Abstraction

In order to implement inspections or quick fixes, a developer has to use the JetBrains IntelliJ code representation API for accessing the source code displayed in the IDE view. Three different APIs are supported each of which provides a different level of abstraction: AST (*i.e.*, *Abstract Syntax Tree*, a tree representation of the source code), PSI (*i.e.*, *Program Structure Interface*, an AST with attached contextual information,

[2] `https://www.jetbrains.com/help/idea/configuring-inspection-severities.html`
[3] `https://www.jetbrains.com/help/idea/resolving-problems.html`

e.g., it supports the resolving of variables), and UAST (*i.e.*, *Unified AST*, a language-agnostic PSI that is primarily used to enable Kotlin language support).

2.2.1 AST

The basic API for internal code representations is called AST. Most code manipulation and analysis method calls within the IDE, including those from inspections and quick fixes, eventually reach this fundamental API. Even more, external changes to a file on disk initiate a recreation of the corresponding AST to maintain consistency between the IDE view and the file system. However, this API offers only limited functionality, *e.g.*, delete an element or find all references to a variable and primarily provides basic features for the upper layers, *i.e.*, PSI and UAST.

The interaction with AST objects is exemplified in Listing 1. Each element in the AST tree is represented by an `ASTNode` object, and the method `getPsi()` returns the corresponding `PsiElement`. The AST layer is rarely used in plug-ins as there is usually no need to access these low-level APIs. Instead, plug-in developers prefer to utilize the PSI and UAST layers.

```
1  ASTNode node;
2
3  // returns parent ASTNode
4  node.getTreeParent();
5
6  // adds child node at the end of another child node
7  node.addChild(newChild, anchor);
8
9  node.removeChild(child);
10
11 // finds first child of the specified type
12 node.findChildByType(iElementType);
13
14 // returns corresponding PsiElement
15 node.getPsi();
```

Listing 1: Code example using the AST API

2.2.2 PSI

The PSI API carries on with the tree-like structure of AST entities, but it provides a better abstraction and additional helper methods to ease its use. In fact, many features such as inspections and quick fixes leverage the PSI API,[4] and it also provides the fundamental feature set for the more abstract UAST API.[5] To maintain interoperability,

[4] https://plugins.jetbrains.com/docs/intellij/psi.html
[5] https://plugins.jetbrains.com/docs/intellij/uast.html

every PsiElement can be converted into an ASTNode from the AST API by using the method getNode().

Important methods are shown in Listing 2. The base element of the PSI API is the PsiElement. Every such entity can return the assigned project, the parent and children nodes, its textual representation in combination with the offsets, and it provides tree manipulation methods such as add() and delete(). It is important to know that the textual representation of a PSIElement always considers all subnodes. For example, a node that correlates with a println method call could return the string value println("Hello World!").

```
1  PsiElement element;
2
3  // get corresponding Project
4  element.getProject();
5
6  // parent PsiElement
7  element.getParent();
8
9  // all direct children as array of PsiElement
10 element.getChildren();
11
12 // returns String representation of the element
13 // (as displayed in the editor view of the IDE)
14 element.getText();
15
16 // offset range used in the editor view
17 element.getTextRange();
18
19 // adds a child at the end of this PSIElement,
20 // but before the specified PSIElement anchor
21 element.addBefore(PsiElement element, PsiElement anchor);
22
23 // removes the element from the PsiTree
24 element.delete();
```

Listing 2: Code example using the PSI API

Listing 3 presents elements that are typically used when working with variables in the PSI API. PsiVariable represents a variable in the code and provides useful methods such as the computation of its scope or the collection of references. There exist multiple ways to retrieve variable references from existing code as illustrated in lines five and seven. We prefer the latter approach because it leads to much cleaner code although it introduces a caveat: it does not always consider all references added by plug-ins support[6], however, this is irrelevant for us as the references we need are within the Java / Kotlin code and implemented in the base version of the IDE.

[6]https://github.com/JetBrains/intellij-community/blob/master/platform/core-api/src/com/intellij/psi/PsiElement.java

```
PsiVariable variable;

SearchScope varScope = variable.getUseScope();

Collection<PsiReference> refs =
        ReferencesSearch.search(variable, varScope, true).findAll();

PsiReference[] references = variable.getReferences();
```

Listing 3: Helper methods for variables in the PSI API

2.2.3 UAST

The UAST API extends the PSI API to facilitate uniform Java and Kotlin code manipulations.[7] Without UAST, a developer would have to implement an inspection for both languages separately, although Kotlin is very similar to Java in many aspects and it is transparently compiled to Java byte code during the build process of an app. Therefore, UAST enables more efficient development of functionality that targets both Java and Kotlin code.

```
UElement element;

// Parent element
element.getParent();

// Underlying, physical PSI element
element.getSourcePsi();
```

Listing 4: Example UAST code

Particularly important are the methods of the common UAST base class `UElement` two of which are shown in Listing 4. That is, `element.getParent()` returns the parent UAST node and `element.getSourcePsi()` returns, if available, the corresponding `PsiElement`. This conversion is necessary for code manipulation, because the UAST API is still under active development, and features that alter the tree structure are still experimental. Therefore, the manipulation of code should be implemented in the AST or preferably in the PSI layer. Since there exists no single method in `PsiElement` to access the corresponding UAST element, that conversion is more complicated, *e.g.*, a `PsiElement` can represent multiple `UElements`. Consequently, such conversions require the use of helper classes like `UastFacade` or `JavaUastLanguagePlugin`.[8] In addition, some of their methods require the specification of the `UElement` type the `PsiElement` should be converted to and

[7] https://plugins.jetbrains.com/docs/intellij/uast.html
[8] https://plugins.jetbrains.com/docs/intellij/uast.html

may return unexpected results depending on the provided type. When the conversion fails due to a combination of unsupported types the framework will instead perform a generic conversion to the `UElement` type.

3 Implementation

Our security code smell quick fix tool is a plug-in for IntelliJ, respectively Android Studio, that leverages the inspection and quick fix features for six different security code smells.

Figure 3.1 presents the IntelliJ environment. As we can see, multiple JetBrains IntelliJ IDEs for various programming languages exist that share the same core engine, *e.g.*, Python (PyCharm) or C/C++ (CLion). In fact, Android Studio extends *IntelliJ IDEA*, which specialises in Java and Kotlin programming.[1] The core engine includes extension points, which allow the engine to be extended by plug-ins for specific purposes. However, the presented extension points in Figure 3.1 are simplified: In reality, there exist not a single but multiple extension points to modify the code completion behavior of the IDE, and IntelliJ features hundreds of extension points that plug-ins can optionally use to extend its functionality.[2] Therefore, IntelliJ's plug-in system is very flexible and supports a wide range of applications: From simple utilities for string manipulation[3] over style and theme customizations[4] to entirely new support for programming languages such as R.[5]

Our plug-in attaches to various extension points of the expandable core function-

[1] https://developer.android.com/studio/intro
[2] https://plugins.jetbrains.com/docs/intellij/extension-point-list.html
[3] https://plugins.jetbrains.com/plugin/2162-string-manipulation
[4] https://plugins.jetbrains.com/plugin/8006-material-theme-ui
[5] https://plugins.jetbrains.com/plugin/6632-r-language-for-intellij

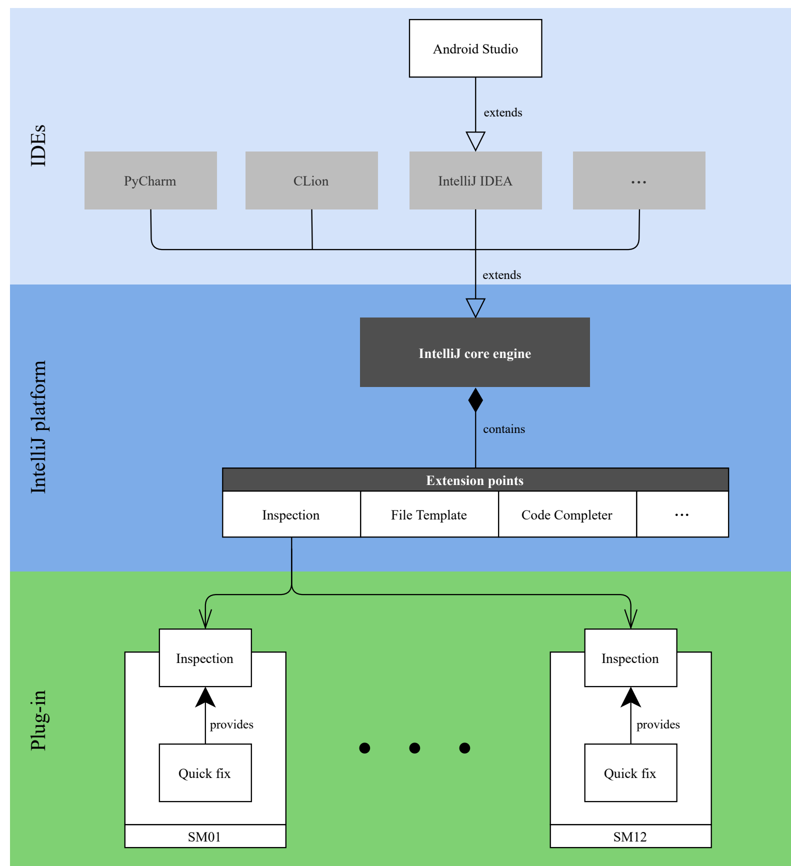

Figure 3.1: The IntelliJ environment

ality, *e.g.*, file templates, inspections. That is, every supported security smell requires two items: an inspection that highlights the code relevant for the smell, and a quick fix that is attached to it to let the developer fix affected code with ease.

In the remainder of this section, we discuss the high-level implementation of each security code smell quick fix based on the findings from Gadient *et al.* [8]. The implementation is discussed in more detail in Appendix A.

3.1 Persisted Dynamic Permission

An app may grant another app access to a protected resource during run time. Such dynamic permissions must be revoked during run time as well. If a dynamic permission is not revoked, it may persist longer than intended, which results in another app having continuous access to a potentially sensitive resource. Therefore, an unintentional perpetual permission may cause a severe threat for data leaks. Such permanent permissions should always be granted using the permission system available in the manifest file.

Problem in Code: An app grants access to a protected resource using `Context.grantUriPermission(toPackage, uri, modeFlags)` but does not revoke it with the corresponding `Context.revokeUriPermission(toPackage, uri, modeflags)` method call.

Implementation: Our tool heuristically checks whether there are more `grantUriPermission` than `revokeUriPermission` method calls for each found URI. If there is an URI with more grants than revokes, a quick fix is displayed which proposes a separate method that revokes the URI permission in the corresponding class. The developer then needs to call the inserted method himself from the desired place as the quick fix cannot predict when the permission should be revoked. More specifically, the process is presented in Listing 5.

```
1  // initial computations, only performed once per inspection session
2
3  // will hold occurrences of grantUriPermission(...);
4  pGrants := new Map(String, List);
5  // will hold occurrences of revokeUriPermission(...);
6  pRevokes := new Map(String, List);
7
8  for all permission calls in a file:
9      arg := URI argument of a permission call;
10
11     switch (arg):
12
13         case 1: arg is method call URI.create("..."):
```

```
14            id := "...";
15            break;
16
17        case 2: arg is any other method call:
18            id := method name and parameters;
19            break;
20
21        // qualified expression => arg is of the form selector.receiver
22        case 3: arg is a qualified expression:
23            recursively call this code block with arg := arg.selector;
24            break;
25
26        case 4: arg is a variable:
27            initializer := variable initializer of arg;
28            if initializer is a method:
29                recursively call this code block with arg := initializer;
30            else
31                id := initializer;
32            break;
33
34        default:
35            id := current line number of inspection;
36
37    if element is a granting permission call:
38        pGrants(id).add(element);
39    else
40        pRevokes(id).add(element);
41 store pGrants, pRevokes in session;
42
43
44 // inspection of specific element
45 if element is permission call:
46    arg := URI argument of permission call;
47    id := id from arg according to method above;
48
49    grants = session.pGrants(id).length;
50    revokes = session.pRevokes(id).length;
51
52    if grants > revokes:
53        trigger relevant inspection in the IDE;
```

Listing 5: Pseudo-code for the shortlisting of permissions

Limitations: First, the quick fix only checks for granted and revoked permissions within the current file since IntelliJ's just-in-time inspection framework does not favor look-ups to other files for performance reasons. Second, finding the relevant URI is not trivial. For example, the tool has to resolve the URI string variable if the grant call is not created with an inlined URI parameter in the method `URI.create(...)`.

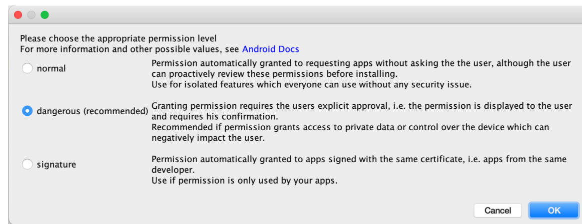

Figure 3.2: The quick fix view for the smell *Missing Protection Level*

However, if the variable is a parameter of the current method, we may need to resolve the variable beyond class boundaries, which is unsupported. Similar complications arise if the URI parameter is the result of an inline method call. Likewise, variable assignments from other classes are not considered by the tool. For instance, consider a URI variable that has been created with the value "A" and a URI grant that uses this variable as a parameter. Later, if the value "B" is dynamically assigned to the URI variable before it is used in a revoke call, the grant and the revoke call target different URIs. Our implementation, however, does not notice any difference regarding such URIs, *i.e.*, it treats them as if one dynamic URI permission was granted and then later revoked since both use the same variable.

3.2 Missing Protection Level

An app may introduce custom permissions to control the access to features and resources it provides. Custom permissions with no specified protection level may grant unwanted apps access to a protected feature, therefore facilitating data leaks. Custom permissions must use an appropriate protection level.

Problem in Code: A custom permission without an `android:protectionLevel` attribute is registered in the manifest file.

Implementation: The quick fix appears in the IDE if a custom permission without a protection level attribute is found in the manifest file. Before the quick fix is applied, it opens the view (shown in Figure 3.2) in which the issue is explained to the developer providing additional information on the different protection levels including examples on when to use them. The developer can then choose the desired protection level and the quick fix will automatically perform all the necessary changes in the code.

Limitations: While detecting a missing protection level attribute is rather trivial, accurately selecting the correct protection level for a custom permission is not feasible in the context of a quick fix, because the selection varies with the future intention of the developer. We abandoned trivial solutions such as the guessing of a protection level or

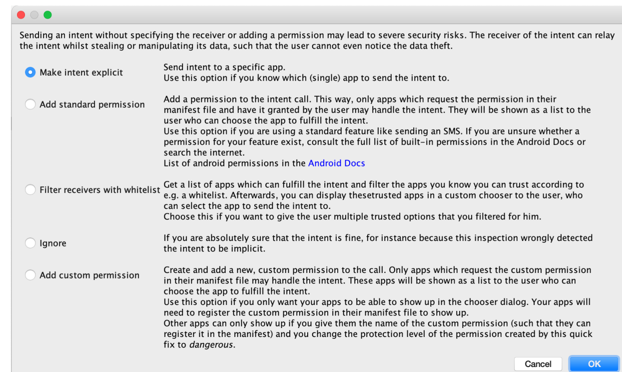

Figure 3.3: The quick fix view for the smell *Unprotected Implicit Intent*

always selecting the most stringent protection level as these would incur a risk of data leaks or denial.

3.3 Unprotected Implicit Intent

Any app can send unprotected implicit intents or register itself to receive them. Since any app can receive such an intent, an app could gain access to possibly sensitive data that is attached to the intent. It may even forward a potentially manipulated intent to a valid receiver to make data leak or to interfere with the further processing of it. Moreover, an app can send an unprotected implicit intent to target apps that hold more permissions to bypass access restrictions, *e.g.*, for the photo library. There exist five solutions for this problem: i) using only explicit intents, ii) presenting a secured app chooser, iii) requesting an existing permission, iv) requesting a new custom permission, or v) ignoring the problem (provider or client side). However, some of these solutions are only available for certain kinds of implicit intents. For instance, an implicit intent can be sent as a broadcast so that the receiver must obtain a permission before it can access the attached information, whereas an activity started with `startActivity` does not support such an authorization scheme.

Problem in Code: An app sends an unprotected implicit intent that has been constructed without a protection argument or component class, and one of the following method calls is involved: `startActivity`, `startActivityForResult`, `startService`, `sendBroadcast`, `sendOrderedBroadcast`, `sendBroad-castAsUser`, or `sendOrderedBroadcastAsUser`.

Implementation: When the tool encounters an unprotected implicit intent that is sent it presents the available solutions in the quick fix view (shown in Figure 3.3) to the developer on how to resolve the issue. For every solution the view briefly explains

what the consequences are and when it should be used. Options which are not feasible for the current situation are not displayed, *e.g.*, the option for a secured app chooser will not appear for implicit intents sent through broadcasts.

Using only explicit intents: The quick fix transforms the implicit intent into an explicit one with descriptive placeholder arguments that must be replaced by the programmer. As a result, only the explicitly referenced component can receive the intent. This solution should be used if the developer requires access to the feature of a particular component, *e.g.*, a custom messaging application.

Presenting a secured app chooser: This solution adds custom code that compiles a list of app components that support the desired implicit intent type of which the developer can shortlist the preferred receivers. These preferred receivers are then presented to the user who can securely choose one of them. In more technical terms, the quick fix adds code for the custom app chooser along with the appropriate parameters and a comment that explains the entire process. This is the preferred solution when users should be able to use their own installed apps.

Requesting an existing permission: The quick fix injects an existing Android permission with a placeholder argument that must be replaced by the developer. In addition, a to-do comment is placed above the added permission which contains a web link to the relevant Android SDK documentation that presents all existing permissions. This solution protects from unauthorized requests, but it is only applicable if an existing permission matches the desired use case.

Requesting a new custom permission: This solution injects a custom permission into the manifest file, and adds a custom permission argument in corresponding method calls. When the quick fix is applied, the code view immediately switches to the manifest file view so that the developer can seamlessly continue with renaming the added permission or adding other relevant attributes, *e.g.*, `android:description` to provide a meaningful description of the provided service and the use to its users. For this solution, it is important to note that apps must be aware of the custom permission before they can request it. This solution protects from unauthorized requests and it can always be applied.

Ignoring the problem: A developer may want to ignore the warning as it could be incorrect or not a security risk after all, *e.g.*, if no sensitive data is involved. The implications of this decision are explained to the developer before that particular inspection is disabled. In other words, the tool remembers the execution of the quick fix so that the affected code will not be highlighted again by that inspection.

Limitations: The decision whether the transformation of an implicit to an explicit intent or the injection of a custom permission is a suitable solution requires contextual information. This information is only partially available in the source code, because the recommendations adjust to the use of the features, which is usually not entirely

known at compile time. Hence, a fully automated solution is not feasible and manual intervention is still required for some key aspects.

3.4 Sticky Broadcast

A broadcast is a message sent by a component to any other component that registers to receive it. Whereas an ordinary broadcast is received by the registered receivers before it terminates, a sticky broadcast remains active after it has been received by recipients. This enables the propagation of information over a longer period of time. The problem with sticky broadcasts is that they can be modified by any receiver. As a result, subsequent receivers could receive altered content from a sticky broadcast without notice. Therefore, sticky broadcasts are a security risk and, consequently, have been deprecated in the Android API level 21.[6] Developers should use a non-sticky broadcast, or if "stickiness" is required, they should migrate to a different communication scheme, *e.g.*, one that is based on regular intents.

Problem in Code: A method call related to sticky broadcasts exists in the code, *i.e.*, `sendStickyBroadcast`, `sendStickyBroadcastAsUser`, `sendStickyOrderedBroadcast`, or `sendStickyOrderedBroadcastAsUser` is called, and the permission `android.permission.BROADCAST_STICKY` is requested in the manifest file.

Implementation: The quick fix transforms the sticky broadcast into a non-sticky one. Moreover, a comment is added that sketches the procedure to follow for the implementation of a messaging scheme based on explicit intents which ensures the accessibility of "sticky" data.

Limitations: We cannot accurately detect whether the "stickiness" property of a broadcast is important for a given implementation. Therefore, transforming a sticky broadcast into a non-sticky one will mitigate the associated security risks, however it might corrupt the notification mechanism depending on the use case.

3.5 Implicit Pending Intent

A pending intent is an intent designed to be executed in the future on behalf of the app that originally placed the intent. Unfortunately, any receiving app that supports the intent protocol can obtain the permissions of the initiator through the received intent.

[6] `https://developer.android.com/reference/android/content/Context.html#sendStickyBroadcast(android.content.Intent)`

Moreover, the intent can be executed in a delayed manner, *i.e.*, even if the initiating app is currently not running. Therefore, an app can misuse this feature to acquire permissions of other apps, posing an obvious security threat. As a result, pending intents should always be avoided.

Problem in Code: An implicit pending intent is set up in the code using one of the methods `PendingIntent.getActivity()`, `PendingIntent.getBroadcast()`, or `PendingIntent.getService()` with an implicit intent argument.

Implementation: The quick fix transforms an implicit pending intent into an explicit intent with empty arguments whenever a method related to the construction of an implicit pending intent is encountered. The empty arguments must be set accordingly by the programmer, because those are use case dependent. In addition, the tool remembers the execution of the quick fix so that after an undo operation the affected code will not be highlighted again by that inspection until it changes.

Limitations: The detection of implicit intents is not fully accurate, *i.e.*, the algorithm has a recursion limit for performance reasons and assumes the intent to be implicit if that limit is reached.

3.6 Common Task Affinity

A task is a group of activities that relate to the same job from a user's perspective, *e.g.*, all activities from an email app can be grouped in one task. Consequently, the task affinity attribute defines the affiliation with a task for an activity.[7] Unfortunately, activities that share an identical task affinity value can overlay each other even if they are not from the same app. For example, an app using an existing task affinity id could display an invisible voice record button on top of another app. If no task affinity value is specified, the default value will be applied which does not protect against this security risk. The solution is to use an empty string for the task affinity attribute that will instruct the Android operating system to use a secure random identifier.

Problem in Code: The `android:taskAffinity` in the manifest file is missing or set to a non-empty value. Both the `<application>` and `<activity>` tags may have the attribute set. The value in the `<application>` task affinity will get used if a corresponding `<activity>` has no task affinity set.

Implementation: The code is highlighted and a quick fix offered in two cases:

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

- If `<application>` level task affinity is missing or non-empty and there are one or multiple `<activity>` tags in the `<application>` tag, the `<application>` tag is marked
- If an `<activity>` level task affinity is non-empty, the `task affinity` is highlighted

The quick fix acts differently depending on the case:

- For the `<application>` level, the task affinity is set to an empty string.
- On the `<activity>` level, the task affinity is removed entirely if the `<application>` level task affinity is set to an empty string, as this value is inherited. Else, it is set to an empty string.

A comment explaining the necessity of the change is always added above the changed tag. With all quick fixes executed, the manifest will contain an empty task affinity in the `<application>` tag with no or empty task affinities in the `<activity>` tags, depending on the execution order. It is of course possible to make all the necessary changes in one quick fix, however, we believe that missing feedback would be a problem. A manifest can be hundreds of lines long; if it all was fixed with one quick fix, the user might not realize what was changed and a task affinity might be deleted without his knowledge and approval. Affected code is highlighted using the security level *error* since the resulting problem is severe and the smell is very common.

Limitations: We cannot detect whether an application uses this feature intentionally, *e.g.*, for displaying additional calling information while being on a phone call. For such apps, the proposed changes would remove support for any overlaid activities which would break the implementation. However, for the majority of the apps this quick fix provides additional security.

4

Framework Limitations

Static code analysis and in particular IntelliJ tool support are subject to certain limitations that affected the quick fix implementation for some of the security code smells.

4.1 IntelliJ Framework

The IntelliJ framework makes only a subset of its features accessible to inspections and the corresponding quick fixes. Moreover, it does not acquire any high-level contextual knowledge a developer has and some features are rather difficult to use due to frequent code changes, ambiguities in the use of APIs, and the incomplete official documentation.

4.1.1 Lack of Context

Quick fixes require substantially more contextual information than inspections, because they must propose adequate solutions that fix a problem without altering the app logic, and at the same time remain compliant to the existing programming environment. Even more, for a specific problem there exist usually numerous solutions, each with different advantages and limitations. But unfortunately, holistic contextual information extracted from code is out of reach for static code analysis tools, because some information remains hidden in developers' minds and thus inaccessible. For example,

the proposal of an appropriate protection level for a permission requires a comprehensive understanding of the responsibilities of each class in an app and the intentions behind them. To mitigate this lack of context the required context must be provided by the developer, *e.g.*, by displaying a dialog to obtain feedback from the developer. Using such feedback the tool can then accurately perform the requested change, or hide the inspection if desired.

4.1.2 Lack of Continuity

The UAST API was newly introduced when this work started. In consequence, the API underwent massive changes that broke parts of the implementation with every new release. For example, in the early days of UAST many helper methods had to be used from the PSI and AST APIs. However, over time these helper methods have been migrated to the UAST implementation and the original methods became deprecated or were removed entirely. We strived to use newly introduced methods where possible and continuously patched our code in order to make it work with recent releases.

4.1.3 Lack of Documentation

The IntelliJ documentation for AST APIs and GUI classes was at the time of writing largely non-existent, and it is still not complete. For instance, most of the AST manipulation methods were undocumented and UI-related classes that have been adapted by JetBrains lacked explanations. Therefore, we had to follow a trial and error strategy.

4.1.4 Lack of Performance

IntelliJ performs an extraordinary amount of background operations that ensure a smooth user experience, *e.g.*, by identifying potential issues or performing background indexing and compilation of resources. Such background operations consume many system resources and therefore limit the resources available for a plug-in. In fact, there exist strict guidelines on what operations should be used.[1] If such guidelines are not followed, the IDE might become unresponsive. This is more relevant for inspections than for quick fixes, because quick fixes must be executed manually by the developer and do not run repeatedly after each code change. In order to avoid slow-downs of the IDE, if possible, we closely followed these guidelines and only perform light-weight computations in the background. In particular for computationally expensive look-ups, we limit the maximal execution time using loop counters which can terminate an inspection after a certain number of iterations. For instance, when recursively resolving an intent to decide whether it is implicit or not we break the recursive look-up cycle after ten iterations and assume that the intent is implicit.

[1] `https://plugins.jetbrains.com/docs/intellij/performance.html`

4.1.5 Lack of User Characteristics

We have no information about the expertise of developers on the subject of a specific security issue nor their Android app programming experience. Therefore, we were forced to find solutions that are acceptable for every developer. For example, well-versed app developers might prefer a simplistic tool that only reports a problem without any functionality to resolve it, because large refactorings applied to complex projects often require some manual effort to ensure a successful transition. On the other hand, inexperienced developers who create their first app in their spare time might require more background information, *e.g.*, a web link to the relevant documentation and a list of all options with an explanation for each of them whether it is suitable for a given scenario. To mitigate this lack of user characteristics we present the information in a way that an experienced developer can quickly skip every step, but at the same time a rather inexperienced developer can find all the essential information. A well-thought GUI certainly supports this task, *e.g.*, by following basic GUI design principles like hiding unrelated content to keep users focused. Advancing the previous example, a developer who knows exactly the required protection level can immediately choose the correct protection level from a list, whereas novices read all the text content to make their informed decision.

4.1.6 Lack of User Feedback

Collecting user feedback is valuable for the further development of the tool. However, IntelliJ does not provide any facilities to directly communicate with users of their software. In general, feedback can be collected in various ways, *e.g.*, by email or web APIs that are accessed through a GUI. Whereas email communication requires a manual effort, web APIs can provide a seamless experience. However, they should not be used in a disruptive way, *e.g.*, asking developers for feedback after they just have executed a quick fix, or spawning pop-up windows from inspection tasks that are run in the background. Such behavior would confuse developers and eventually drop their interest. Since IntelliJ already lists on their website the email address of every plug-in author it is natural to use that feature. Nevertheless, it does not provide immediate communication and requires additional effort on the developer's side.

4.1.7 Lack of UI Integration

The GUI of IntelliJ-based IDEs are built on *Swing* which is a widget toolkit for Java. Unfortunately, custom views do not automatically respond to changes in the main window. For instance, a developer can always change the default font size for code in the IDE, but this will not affect the font size used in custom views. To reduce the impact of these issues, we relied primarily on basic Swing features for the GUI implementation.

4.2 Unsupported Security Code Smells

Some of the reported smells in [8] are very complex and their accurate resolution is barely possible with the current analysis framework. In the remainder of this subsection, we discuss the reason for exclusion of every reported smell that suffered from such framework limitations.

4.2.1 Custom Scheme Channel

An Android app can register a custom scheme, *e.g.*, `someapp://` to receive all requests for such hyperlinks. However, an app cannot prevent others from registering the same custom URI scheme. Thus any app can potentially receive such URIs that might contain sensitive data like access tokens, user identifiers, or passwords. If the receiver forwards the data accordingly it may be even leaked without any notice of the user. Besides that sensitive data should never occur in custom scheme URIs, a possible fix for this smell is the use of the system reserved scheme `intent://`, which allows an intent to be sent to a specific app based on its package name instead of an arbitrary app that was successful in registering the scheme. In more recent operating system releases developers can use *Deep Links* or *App Links* that provide better security.[2] In order to fix this smell much additional contextual information is required, because there exists no universal solution to determine how the request should be handled in an app. In addition, changes on the web server are required for more recent measures. Asking the developers whether their code is secure is not an option.

4.2.2 Slack WebViewClient

The `WebViewClient` component is used for web browsing within Android apps. `WebViewClient` has no default mechanism to prevent users from accessing malicious websites. As a result, apps that use this component potentially expose users to threats such as phishing or denial of service attacks. Although the loading of websites can be prevented by overriding the method call `shouldOverrideUrlLoading()`, it is extremely difficult to determine whether the provided implementation is sane, *i.e.*, the validation of custom code is correct. Furthermore, we have no accurate measures to judge white or black-listed URLs beyond simple safety checks provided by online services such as *SafetyNet*.

4.2.3 Broken Service Permission

A service is an application component performing long-running operations in the background.[3] An arbitrary app accessing the service may possess different permissions than

[2] `https://developer.android.com/training/app-links/`

[3] `https://developer.android.com/guide/components/services`

the service provider and thus the provider should verify the caller's permissions before performing any privileged operation on its behalf. However, certain permission checks such as `checkCallingOrSelfPermission()` check the service provider's permissions instead of the caller's permission. Fixing this smell is highly context sensitive: The automated validation of a custom permission verification algorithm is not feasible due to a lack of context. Similarly, a white-listing of client apps is not reasonable, because such lists are not comprehensive and hard to maintain.

4.2.4 Insecure Path Permission

The `UriMatcher` of the Android framework does not distinguish between path permissions using single and double slashes. However, the path permission element in the manifest file uses this distinction. This can lead to unauthorized apps being granted access to sensitive resources when the standard `UriMatcher` is used. In order to fix this smell, a custom `UriMatcher` must be implemented. However, we cannot verify such an implementation for correctness, because we lack context regarding the intended use of a path.

4.2.5 Broken Path Permission Precedence

The path-permission element which is more fine grained should always take precedence in a content provider. However, due to a bug in certain versions of the Android framework, this may not be the case and as a consequence, access may be granted to apps by mistake. Consequently, a distinct content provider should be used for each path, however this might require major code changes that require detailed user feedback. In addition, we do not know whether the nested permissions have been assigned properly.

4.2.6 Unprotected Broadcast Receiver

A broadcast receiver component receives broadcasted messages from arbitrary apps and processes them. It is unprotected if it does not validate the received content and thus can be exploited with spoofed broadcast messages. In other words, an app registered to receive a certain broadcast is vulnerable to any other app, as any app can initiate said broadcast with a spoofed intent. Answering a spoofed intent may lead to a data leak or unintended behavior. Thus, in order to check the validity of the received broadcast the permissions of the broadcast sender must be verified. The check for missing permissions is feasible, but it is not possible to accurately judge whether a used permission is appropriate due to a lack of context. Furthermore, in some cases a spoofed broadcast represents no security risk at all, *e.g.*, if the broadcast is truly intended to be public and always returns the same static response.

4.3 Discussion

In general, the lack of information introduces an inherent difficulty in satisfying the different requirements of certain user groups and tasks within a single user interface. For instance, on the one hand a quick fix that is understandable for beginners must be self-contained, *e.g.*, comprehensive information must be provided such as the benefits, the drawbacks, or potential side-effects. On the other hand, for experienced developers it should be non-intrusive and effective to use by displaying only essential information, *e.g.*, the available options to choose from. Another such conflict is performance and accuracy: the decision to support one of them might depend on the personal preference of a developer, or is defined by the project. Where contradictory design considerations require trade-offs we optimized our tool towards novice users.

5 Evaluation

In this section, we present the evaluation of our plug-in. We explain our methodology before we present the results.

5.1 Methodology

We validated the tool's output against the results from prior work [8], and we manually validated a subset of the output by inspecting the corresponding decompiled app code. For the initial data set we used 25 random *F-Droid* projects that have been downloaded around 11-NOV-2018. In a first step, we evaluated whether each detected security code smell represented a real issue (true positive), or it was a false alarm (false positive). For each reported smell we then evaluated whether the quick fix was successful in resolving the issue. We did not investigate false negatives nor true negatives, because we assume that our detection routines are rather opportunistic with an increased likelihood of false positives. We did not evaluate the user interaction and information displayed during the quick fix, because the evaluation would require additional user experiments including numerous developers with different levels of experience. We did not find for every security smell an affected app and thus we performed for such smells an additional evaluation, *i.e.*, for each smell detector that did not report at least three occurrences for the 25 random apps, we used additional random apps that are known to be affected from prior work to evaluate the smell detector and its quick fix.

5.2 Results

In the source code of 25 random apps our plug-in could identify 77 smell occurrences in 24 apps, *i.e.*, only one app was not affected by a security smell. In other words, 96% of the evaluated apps are affected by at least one security smell. The two security smells *Common Task Affinity* and *Unprotected Implicit Intent* account for 75 smell occurrences, *i.e.*, 97% of all smell occurrences. The remaining two smell occurrences were categorized as *Missing Protection Level* and *Persisted Dynamic Permission*. Two security smells, *i.e.*, *Sticky Broadcast* and *Implicit Pending Intent* did not occur in the data set and required the targeted analysis of additional apps. In the remainder of this section, we discuss the detailed results for each reported security smell.

5.2.1 Persisted Dynamic Permission

Prevalence: Since only one occurrence has been reported in the 25 apps, we evaluated the two additional apps that were known to be affected from the previous study. In these two apps, our tool could identify two additional occurrences, which added to a total of three occurrences. *Detection:* Our manual evaluation revealed that all four reported occurrences were indeed instances of this security smell. *Quick fix:* However, in every case the quick fix was not entirely resolving the issue, because it only added a ready to use method to revoke URI permissions with a comment that instructs developers to call the inserted method from the "correct" place as we do not know when the permission should be revoked. Moreover, in one case the grant was inside a static utility method and would have required a static method that we did not provide, *i.e.*, we only provided the non-static method. *Success rate:* After all, our quick fix was useful for two security smell instances (67%).

5.2.2 Missing Protection Level

Prevalence: Since this smell has been reported only once in the 25 apps, we evaluated the three additional apps that were known to be affected from the previous study. The plug-in reported seven additional occurrences in these three apps. *Detection:* The reported occurrence in our primary data set was correct and referred to a permission with the word "example" in its path. It is not used anywhere and should have been deleted. The seven remaining occurrences were also valid and caused by custom permissions that lacked a protection level. *Quick fix:* The quick fix for the example permission was inappropriate, because the permission should have been deleted and thus did not require a protection level argument. A similar problem revealed the quick fix for the permission `android.permission.MEDIA_CONTENT_CONTROL`, which is "not for use by third-party applications due to privacy of media consumption" according to

the official Android documentation.[1] For the remaining six occurrences, the quick fix was useful and successfully inserted a protection level. *Success rate:* After all, our quick fix was useful for six security smell instances (75%).

5.2.3 Unprotected Implicit Intent

Prevalence: The tool reported 41 security smell occurrences for the 25 apps in our dataset. 29 of these occurrences were found in two apps, and the remaining 12 occurrences were spread across seven apps. 16 apps did not suffer from this security smell. *Detection:* Every reported occurrence was correct, *i.e.*, the reported intent was not protected. *Quick fix:* The application of this quick fix requires a decision from the developer on how to proceed with the identified issue, because we do not know the required protection of the intent. Every offered option could be applied to every reported occurrence. Except for the "ignore" option, all offered quick fix options result in improved security. The threat was always mitigated if the developer chose the "correct" fix for the reported issue. *Success rate:* Therefore, we consider these quick fixes useful for every reported security smell (100%).

5.2.4 Sticky Broadcast

Prevalence: Since this smell did not occur in the apps from our dataset, we evaluated three additional apps that were known to be affected from the previous study. The tool reported three security smell occurrences for these apps. *Detection:* Every reported occurrence was indeed a sticky broadcast and thus correct. *Quick fix:* The quick fix successfully transformed every found sticky broadcast into a non-sticky one. *Success rate:* Consequently, the quick fix was useful for every reported security smell (100%).

5.2.5 Implicit Pending Intent

Prevalence: Since this smell did not occur in the apps from our dataset, we evaluated three additional apps that were known to be affected from the previous study. The tool reported three security smell occurrences for these apps. *Detection:* Every reported occurrence was indeed an implicit pending intent and thus correct. *Quick fix:* The quick fix successfully transformed every found implicit pending intent into an explicit intent. *Success rate:* Consequently, the quick fix was useful for every reported security smell (100%).

[1] `https://developer.android.com/reference/android/Manifest.permission#MEDIA_CONTENT_CONTROL`

5.2.6 Common Task Affinity

Prevalence: The tool reported 34 security smell occurrences in the 25 evalauted apps. Most apps were not using the task affinity feature, *i.e.*, the `taskAffinity` attribute in the manifest file was missing. In a minority of the smell occurrences (21%), the task affinity was set to a non-empty value, *i.e.*, our tool could identify seven non-empty `taskAffinity` attributes in three apps. *Detection:* The use of the `taskAffinity` attribute in combination with a static value was always correctly detected. For example, a developer deliberately used a template and did not remove the provided static value. Moreover, every app which did not specify the `taskAffinity` attribute has been reported. *Quick fix:* Although we do not offer a functional replacement for the task affinity feature, the reported apps did not require it after all. Therefore, the quick fix could resolve the problem for every reported security smell occurrence by either replacing the particular value with an empty value for the `taskAffinity` attribute, or by adding a `taskAffinity` attribute with an empty value. On the contrary, if this feature would be essential for the functionality of an app, the developer could find more advice in the provided comment, which contains a URL that points to further information, because we cannot programmatically decide whether this feature is required for a specific app. *Success rate:* The quick fix was useful for every discovered security smell occurrence (100%).

5.3 Discussion

Most smells rarely occurred in the evaluated projects from our dataset. As in the prior study, the most prevalent smells were *Unprotected Implicit Intent* and *Common Task Affinity*. That is, because implicit intents are a core feature of Android, but they were usually not protected with additional code, and the task affinity feature was usually not used by developers. Interestingly, the occurrences of the *Common Task Affinity* smell were spread among numerous apps, whereas the occurrences of the *Unprotected Implicit Intent* smell were primarily found in a few apps. It appears that the task affinity feature was misused in templates that have been used as a starting point for app projects in which it can occur once for every registered activity in the manifest file. Moreover, we observed that apps either use many implicit intents or they use them not at all. We believe that this is a result of the flexibility of some apps: apps that can share content usually support many different providers, and each provider requires a different intent. Moreover, four apps contained multiple occurrences of *Common Task Affinity*, and two of these apps account for 71% of all *Unprotected Implicit Intents* instances. Generally, a high number of *Common Task Affinity* occurrences indicates a large and complex app as multiple activities are necessary to suffer from more than one *Common Task Affinity* instance.

Our quick fixes suffer from limitations. Regarding the *Persisted Dynamic Permission* security smell, for example, if the developer calls the provided method which correctly revokes the granted permission, the grant call might remain highlighted by our inspection, because we currently cannot accurately match grant and revoke calls that concern the same URI. Similarly with respect to the *Missing Protection Level* security smell, we currently cannot reason about the legitimacy of existing permissions. We realized that for many encountered issues the problem is caused by the lack of information regarding developer interactions and intentions. Therefore, we see a possibility in future work to improve the tool's accuracy by gathering such information. Similarly, the usability study remains future work.

Nevertheless, our tool identifies security smell occurrences accurately for most smells and is very useful for fixing the identified smells, *i.e.*, on average it can successfully fix the existing security smells in more than 91% of all cases.

6
Threats to Validity

We see four major threats to validity: the completeness of this study, the usability in practice for developers, the existence of software bugs, and the validity of the results.

Completeness of this study. Our evaluation was performed on a small subset of the existing Android apps. There is a risk that the results do not apply for Android apps beyond our study. We mitigate this threat with our random selection of apps. Moreover, in our tool evaluation we did not consider false negatives or true negatives of the smell detection as their manual search in random apps is a very time demanding task, because many smells only occur in a rather small subset of apps. Therefore, we only have a limited understanding of our implementation's precision and recall.

Usability for developers. We focused in this study on the feasibility of quick fixes for security smells and not on their usability. We evaluated whether the smells were accurately detected and the fixes worked as desired, but we gathered no data whether our tool could be used by developers and, consequently, would provide a benefit for them. We did not evaluate the perception of developers regarding user interface components or the information presented in them.

Existence of software bugs. During the implementation of our tool we created test files based on our own observations that allowed us to continuously validate our implementation. However, there may still exist bugs in our tool that might alter the results.

Validity of the results. All experiments were performed by the author of the tool,

which introduces a threat to validity due to the experimenter's expectancy.

7
Related Work

We present related work in three major areas, *i.e.*, the tool support, the tool usage, and the security smells.

7.1 Tool Support

Scientific work in the tool support domain primarily targets the most prominent IDEs, *i.e.*, either the Eclipse or the JetBrains IntelliJ IDE[1] and focuses on desktop applications instead of mobile apps. Both environments are feature-wise very similar, *i.e.*, there is only little functionality that remains exclusive to either one of them, *e.g.*, the aggregated project view in Eclipse. In general, the researchers provide their implementations as plug-ins that improve the functionality of the existing IDEs.

Improving the refactoring capabilities of IDEs was a popular topic among the researchers. For example, Mahmood *et al.* performed a major overhaul of the Java literal expression refactoring in IntelliJ, and proposed a more comprehensive implementation that investigates the project-wide use of variable names to avoid refactorings that could cause name conflicts. Before the refactoring is applied, the plug-in presents the user a view of the current variable uses, and allows the developer to perform an informed decision [17].

[1] https://www.baeldung.com/java-ides-2016

Another topic that received much attention in research were code clones. Jablonski *et al.*, for instance, developed an Eclipse plug-in that consistently renames variable names in copy-pasted code [15]. Without using their tool, if a variable assignment is duplicated within the same method, the renaming refactoring could become confused and change the names in the entire method instead of only in the newly inserted block.

Besides code refactoring, teaching assistants used the plug-in system for educational purposes. They implemented a plug-in that let them manage the distribution of assignments, their collection, and their grading of participating students for IntelliJ [5] and Eclipse [1].

A field related to education is code comprehension, *i.e.*, ability to learn and understand provided code. For that purpose, Guzzi *et al.* created a tool that records the user-interface actions of developers together with additional feedback to determine the important code sections of an application. Based on this feedback, they present other developers the relevant code locations to let them grasp the code more efficiently [13]. Better code comprehension can also be achieved by better code navigation. Therefore, Smith *et al.* implemented a plug-in that allows developers to easily trace values through code by making the relevant elements in the code clickable [21].

In order to collect such information, researchers worked on generic plug-ins that can closely monitor user interactions and report the aggregated data. For example, Levaja implemented a plug-in that can track the time spent on writing code, or the average number of test executions per day [16]. On the contrary, Ioannou *et al.* focused rather on high-level user actions such as saving a file, or how the search feature has been used [14]. Interestingly, in their small study with only six participants they found that saving a file is the most common operation performed by developers, before using the undo feature, or the copy-paste mechanism.

Besides plug-ins with rather constrained functionality, there exist complex plug-ins that introduce support for additional computer languages, *e.g.*, for the Java Modeling Language (JML) [18] and the Answer-Set Programming (ASP) [4]. There further exist plug-ins for test generation [2], library selection [6], and security [24]. The security plug-in highlights potential issues for web services, *e.g.*, when untrusted content from the web is used in the application without prior sanitization. In such cases it provides a quick fix for the developer that adds additional validation code.

Finally, researchers altered entire IDEs to suit their needs. For example, Van Deursen *et al.* changed an existing Eclipse instance so that it can be used over the web [22], Biegel *et al.* added touch support for mobile devices [3], and Gasparic *et al.* experimented with recommender systems that guide developers through the debugging process [10].

Currently, support for security and in particular security code smells is rather limited, *i.e.*, there exists linting support in Android Studio, but the provided suggestions provide no possibility for immediate quick fixes. In this work we advance the current

state and provide quick fixes for a subset of the reported smells.

7.2 Tool Usage

Researchers were also interested in the usage of tool support. Murphy *et al.* gathered usage information from 41 Eclipse developers and found that the majority of developers use every provided view, but clearly prefer the Java and Debug perspectives [19]. The package explorer view was the most useful view for the developers, and renaming the most useful refactoring. Gómez *et al.* explored the gain that can be achieved from tool support for solo and pair programming [12]. They reported that the quality decreases when tool support is used for rather simple tasks, however it is particularly useful for more complex assignments. They assume that tool support fosters a trial and error mentality, whereas coding without tool support requires more effort on the developer's end and therefore, the developers prolonged the code compilation and execution processes to look more carefully at their code to avoid unnecessary iterations. However, they found no notable differences between solo and pair programming. Finally, researchers realized that very often developers are not aware of a particular refactoring. In order to provide assistance, they continuously observed every action of the developers and offered them tailored support based on their needs [23, 7]. Finally, Poon remarked that the automated refactoring is not always feasible, yet manual refactoring poses difficulties especially to less experienced programmers [20]. He proposed a tool for Eclipse that combines the best of both worlds, *i.e.*, the manual and the automated refactorings by guiding manual refactorings with useful information for the developers.

In our plug-in we proactively report issues in IntelliJ's problem view and offer the developers fixes that only require a few clicks.

7.3 Security Smells

The notion of security code smells that we use in this work has been defined by Ghafari *et al.* [11]. In their work they investigated the prevalence of ten security code smells in more than 46 000 Android apps and found that the majority of apps suffer from at least three different security smells, and that the identified security smells are a good indicator of security vulnerabilities. In their follow up work, they particularly investigated the existence of security smells in Android ICC [8]. Based on the analysis of more than 700 open-source apps they realized that updates rarely have any impact on ICC security, however, in case they do, they often correspond to new app features. Consequently, they 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. According to a more recent work

from Gadient *et al.* [9], the same applies to web API communication, which would also benefit from improved IDE support.

Security smells are a good indicator for vulnerabilities and should be mitigated whenever they are spotted. However, this requires comprehensive tool support that we investigate in this work. Moreover, IDE updates are essential and can further improve security. Following this principle, our tool is compatible with IDE releases of the well-known JetBrains IntelliJ.

8 Conclusion

In this work, we explore the possibility of quick fixes for security code smells, *i.e.*, we provide a plug-in that seamlessly integrates with Android Studio and supports the detection of six different security code smells. We performed a brief manual evaluation of our tool on 25 random apps from the *F-Droid* app store as well as some additional apps from the previous study and checked whether the reported security smells are correct and if they can be automatically mitigated with a quick fix. According to our manual validation, the tool identified 92 security code smells of which it was able to fix more than 91%, many of them without any additional intervention of the developer.

These results are promising and indicate that many security smells can be mitigated with quick fixes. However, additional information such as the user's programming experience or the importance of accuracy would foster tailored quick fixes that are more comprehensive and could achieve an even higher accuracy.

Bibliography

[1] A. Allowatt and S. H. Edwards. IDE support for test-driven development and automated grading in both Java and C++. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 100–104, 2005.

[2] A. Arcuri, J. Campos, and G. Fraser. Unit test generation during software development: EvoSuite plugins for Maven, IntelliJ and Jenkins. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 401–408. IEEE, 2016.

[3] B. Biegel, J. Hoffmann, A. Lipinski, and S. Diehl. U can touch this: touchifying an IDE. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 8–15, 2014.

[4] P.-A. Busoniu, J. Oetsch, J. Pührer, P. SKOČOVSKÝ, and H. Tompits. SeaLion: An Eclipse-based IDE for answer-set programming with advanced debugging support. *Theory and Practice of Logic Programming*, 13(4-5):657–673, 2013.

[5] N. Denissov et al. Creating an educational plugin to support online programming learning. Master's thesis, Aalto University, 2021.

[6] R. El-Hajj and S. Nadi. LibComp: An IntelliJ plugin for comparing Java libraries. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1591–1595, 2020.

[7] S. R. Foster, W. G. Griswold, and S. Lerner. WitchDoctor: IDE support for real-time auto-completion of refactorings. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 222–232. IEEE, 2012.

[8] P. Gadient, M. Ghafari, P. Frischknecht, and O. Nierstrasz. Security code smells in Android ICC. *Empirical Software Engineering Special Issue*, 2018.

[9] P. Gadient, M. Ghafari, M.-A. Tarnutzer, and O. Nierstrasz. Web APIs in Android through the lens of security. In *2020 IEEE 27th International Conference on*

Software Analysis, Evolution and Reengineering (SANER), pages 13–22. IEEE, 2020.

[10] M. Gasparic and F. Ricci. IDE interaction support with command recommender systems. *IEEE Access*, 8:19256–19270, 2020.

[11] 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.

[12] O. S. Gómez, A. A. Aguileta, R. A. Aguilar, J. P. Ucan, R. H. Rosero, and K. Cortes-Verdin. An empirical study on the impact of an IDE tool support in the pair and solo programming. *IEEE Access*, 5:9175–9187, 2017.

[13] A. Guzzi, M. Pinzger, and A. Van Deursen. Combining micro-blogging and IDE interactions to support developers in their quests. In *2010 IEEE International Conference on Software Maintenance*, pages 1–5. IEEE, 2010.

[14] C. Ioannou, A. Burattin, and B. Weber. Mining developers' workflows from IDE usage. In *International Conference on Advanced Information Systems Engineering*, pages 167–179. Springer, 2018.

[15] P. Jablonski and D. Hou. CReN: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 16–20, 2007.

[16] I. Levaja. WatchDog for IntelliJ: An IDE plugin to analyze software testing practices. Master's thesis, Delft University of Technology Delft, the Netherlands, 2016.

[17] J. Mahmood and Y. R. Reddy. Automated refactorings in Java using IntelliJ IDEA to extract and propogate constants. In *2014 IEEE International Advance Computing Conference (IACC)*, pages 1406–1414. IEEE, 2014.

[18] S. Monteiro, E. Sokolovas, E. Wittingen, T. v. Dijk, and M. Huisman. IntelliJML: a JML plugin for IntelliJ IDEA. In *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs*, pages 39–42, 2021.

[19] G. C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? *IEEE software*, 23(4):76–83, 2006.

[20] D. Y. Poon. Implementing refactoring guidance into Eclipse. Master's thesis, California State University, Sacramento, 2019.

[21] J. Smith, C. Brown, and E. Murphy-Hill. Flower: Navigating program flow in the IDE. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 19–23. IEEE, 2017.

[22] A. van Deursen, A. Mesbah, B. Cornelissen, A. Zaidman, M. Pinzger, and A. Guzzi. Adinda: a knowledgeable, browser-based IDE. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 203–206, 2010.

[23] P. Viriyakattiyaporn and G. C. Murphy. Challenges in the user interface design of an IDE tool recommender. In *2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, pages 104–107. IEEE, 2009.

[24] J. Xie, B. Chu, H. R. Lipford, and J. T. Melton. ASIDE: IDE support for web application security. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 267–276, 2011.

Anleitung zum wissenschaftlichen Arbeiten

The Anleitung consists of a more detailed description of how to implement, execute, and package IntelliJ inspections and quick fixes.

A.1 Implementation

In this section we discuss the default components required to implement a quick fix and the optional user-interface APIs.

A.1.1 Inspection

An inspection entity is responsible for the highlighting of relevant code fragments in the editor view. It must contain logic to decide what code must be highlighted and in which style. There exist two different types of inspections: local and global inspections. Local inspections can only process one file at a time and repeatedly run in the background as the developer edits the code.[1] In contrast, global inspections can process multiple files

[1] `https://github.com/JetBrains/intellij-community/blob/master/platform/analysis-api/src/com/intellij/codeInspection/LocalInspectionTool.java`

and only run on user demand due to the higher utilization of hardware resources.[2] An inspection is represented by a class that either extends the `LocalInspectionTool` or the `GlobalInspectionTool` class, accordingly. Moreover, they must provide a `PsiElementVisitor` object, which will traverse the entire PSI tree representation while searching for nodes that match a given criterion. We focus on local inspections for the remainder of this section since only they can provide immediate feedback to the developer and their provided features are sufficient for most code linting tasks.

In order to illustrate the implementation task, we refer in the following listings to a simplified version of our *Missing Protection Level* inspection with only basic interaction between our custom tool window and the inspections. That inspection is based on the `LocalInspectionTool` class and its stub is presented in Listing 6. The `logger` variable in line 2 contains the logging object of the IntelliJ IDE that allows us to write content into its log file. The remaining two variables in the lines four and six declare the corresponding smell id used for the log output and the available protection levels that can be set separately for each reported element.

```
1 public class UnprotectedPermissionDetection extends LocalInspectionTool {
2     private final Logger logger = Logger.getInstance(getClass());
3     // internal identifier of the smell in our tool
4     protected static final String smellId = "SM03";
5     // list of all valid protection level values
6     private static final List<String> PROTECTION_LEVEL_VALUES = Arrays.asList("normal", "dangerous", ..., "signature");
7     ...
8 }
```

Listing 6: Inspection class stub example

Listing 7 presents a typical `XmlElementVisitor` method that specializes the `PsiElementVisitor` implementation used by an inspection to determine the highlighted text areas. Such visitor implementations must be embedded in the corresponding inspection tool classes, and the visitor logic itself must be specified within the visit method, *e.g.*, `visitXmlTag(...)` (line 7) that constrains the scope to XML content, which must reside in the overridden `buildVisitor(...)` method. The provided inspection only evaluates `PsiElement` entities that are of the type `XmlTag`, *i.e.*, it ignores all non-XML content as well as XML elements that are not a tag, *e.g.*, comments. The try-catch block (lines 8 to 17) ensures that the execution of the inspection code can be interrupted at any time (*i.e.*, by receiving a `ProcessCanceledException`), *e.g.*, when a user closes an editor view in the IDE while the inspections are still running in the background or if the corresponding file content changed during the code inspection. The exception should always be rethrown as it is further processed by the IDE infrastructure.[3] If other exceptions are raised, we log these events and forward the exception through the `MessageBus` to our custom tool window to notify the user.

[2] `https://github.com/JetBrains/intellij-community/blob/master/platform/analysis-api/src/com/intellij/codeInspection/GlobalInspectionTool.java`

[3] `https://plugins.jetbrains.com/docs/intellij/general-threading-rules.html#background-processes-and-processcanceledexception`

```
1  @NotNull
2  @Override
3  public XmlElementVisitor buildVisitor(@NotNull final ProblemsHolder holder, boolean isOnTheFly) {
4     return new XmlElementVisitor() {
5
6        @Override
7        public void visitXmlTag(XmlTag xmlTag) {
8           try {
9              // insert inspection logic here
10          } catch (ProcessCanceledException canceled) {
11             throw canceled;
12          } catch (Exception e) {
13             logger.warn("Security Smells: Something went wrong with " + smellId + ".", e);
14             MessageBus bus = holder.getProject().getMessageBus();
15             ChangeActionNotifier publisher = bus.syncPublisher(ChangeActionNotifier.CHANGE_ACTION_TOPIC);
16             publisher.showException(e);
17          }
18       }
19    };
20 }
```

Listing 7: Example visitor required by an inspection

```
1  if (!StringUtils.isEmpty(xmlTag.getName()) && xmlTag.getParent() != null && xmlTag.getName().equalsIgnoreCase("permission")) {
2     XmlAttribute protectionLevel = xmlTag.getAttribute("android:protectionLevel");
3     if (invalidProtectionLevel(protectionLevel)) {
4        logger.info("Security smell found in project file " + xmlTag.getContainingFile().getName() + ": " + DESCRIPTION_TEMPLATE);
5        AddProtectionLevelFix quickFix = protectionLevel == null ? new AddProtectionLevelFix(xmlTag) : new AddProtectionLevelFix(protectionLevel);
6        holder.registerProblem(protectionLevel != null ? protectionLevel : xmlTag, "Unprotected Permission", ProblemHighlightType.ERROR, quickFix);
7     }
8  }
```

Listing 8: Simple inspection logic example

```
1  private boolean invalidProtectionLevel(XmlAttribute protectionLevel) {
2     if (protectionLevel == null || StringUtils.isEmpty(protectionLevel.getValue()))
3        return true;
4     else
5        return !PROTECTION_LEVEL_VALUES.contains(protectionLevel.getValue());
6  }
```

Listing 9: Example code to detect invalid protection levels

```
1  public boolean visitQualifiedReferenceExpression(@NotNull UQualifiedReferenceExpression node) {
2     UCallExpression methCall = (UCallExpression) node.getSelector();
3     UExpression receiver = node.getReceiver();
4     if (receiver == null)
5        return true;
6
7     if (!isPendingIntentObject(receiver))
8        return true;
9
10    UExpression intentArg = UastHelper.getParamOfClass(methCall, INTENT_CLASS);
11    if (intentArg == null)
12       return true;
13
14    boolean isPendingIntent = isPendingIntent(methCall);
15    if (!isPendingIntent)
16       return true;
17
18    if (qFixAlreadyPerformed(intentArg)) // check if quick fix was already applied
19       return true;
20
21    if (!isExplicitIntent(intentArg)) {
22       logger.debug("Security Smell found in: [FILE], [METHODNAME] " + methCall.getContainingFile() + ", " + methCall.getMethodName());
23       holder.registerProblem(staticMethodCall.getSourcePsi(), DESCRIPTION_TEMPLATE, new MakeIntentExplicitQuickFix(origIntentArg));
24    }
25 }
```

Listing 10: More complex inspection logic example

Listing 8 presents an example of an inspection logic. First, we verify whether an XML node is valid and whether its tag name matches the string `permission` (line 1). If a node passes this validation, we try to read the attribute name `android:protectionLevel` and its value from the found XML node (line 2). If this data is invalid or malformed (line 3, for more details see Listing 9), we log that finding to the debug console (line 4), instantiate a "quick fix instance" that characterizes the problem (line 5), and we hand over that instance to the IDE infrastructure (line 6) to let the IDE display the problem to the user, *i.e.*, the relevant text in the editor view becomes highlighted and the quick fix is offered. In other words, the provided code lets the IDE raise an error at the problematic protection level attribute and value, or at the permission tag if the protection level data is missing. Such an error will reveal the text "Unprotected Permission" on mouse hover.

The method `invalidProtectionLevel(...)` in Listing 9 that is called from the inspection logic tests whether the `android:protectionLevel` attribute data is valid. This is necessary to identify the relevant attributes for the highlighting, *i.e.*, we capture every null or empty attribute name (line 2), and we verify whether the attribute value matches one of the possible values (line 5). As shown in the listing, the text representation of an XML attribute's value can be accessed with the method `getValue()`.

Besides XML visitors, the IntelliJ framework also supports generic PSI or UAST visitors, which can visit every element in their corresponding abstract syntax tree. These visitors are more complex to implement since they visit every UAST or PSI node that corresponds to source code instead of structured XML data that follows a rather simple schema. In Listing 10, we provide an example of such a generic `PSIElementVisitor` for a qualified reference expression. As we can see, contrary to the XML API, the PSI API allows to resolve other nodes in the tree beyond parents and children, *e.g.*, with `getReceiver()` (line 3) or methods provided by our custom class `UastHelper` (line 10). However, the PSI visitor implementation is still very similar to the XML visitor: At first, several checks are performed on a particular element and if all pass, the problem is reported to the IDE. If a check fails, the method must return `true`, because returning `false` would cause the visitor to visit some elements twice, which is not desired.

A.1.2 Quick Fix

The common base interface `QuickFix` delivers a single problem descriptor for the implementation of the fix. Consequently, every quick fix can only resolve one problem found by an inspection at a time. In this subsection, we discuss the implementation of a `LocalQuickFix` that implements the common quick fix interface, because the local quick fix and the local inspection are usually used together.

Listing 11 presents a stub of a class that is used to specify a quick fix. The class has two constructors (lines 7 and 11), because the first constructor is required if the inspection passes the `permission` tag to the quick fix, whereas the second constructor is required if the `android:protectionLevel` attribute is passed to the quick fix. Moreover, the `LocalQuickFix` interface declares three important methods:

- `getName()` (line 17) returns the name of the quick fix that is displayed on mouse hover and in the problem view. The method can return different names depending on the results of the code analysis.
- `getFamilyName()` (line 25) returns the category name of the quick fix that is used for the grouping of problem categories in the problem view. This method should return a unique hard-coded string since it should not interfere with the grouping of other elements in the problem view.
- `applyFix(Project project, D descriptor)` (line 30) applies the quick fix. The `project` parameter contains a reference to the target project that requires the fix, and the `descriptor` parameter holds information regarding the problem to be fixed, *e.g.*, elements of the PSI tree that require some changes. A sample quick fix logic implementation is shown in Listing 12.

```
1  private static class AddProtectionLevelFix implements LocalQuickFix {
2     private final Logger logger = Logger.getInstance(getClass());
3
4     private XmlTag permissionTag;
5     private XmlAttribute protectionLevelAttribute;
6
7     public AddProtectionLevelFix(XmlTag permissionTag) {
8        this.permissionTag = permissionTag;
9     }
10
11    public AddProtectionLevelFix(XmlAttribute protectionLevel) {
12       this.protectionLevelAttribute = protectionLevel;
13    }
14
15    @NotNull
16    @Override
17    public String getName() {
18       if (protectionLevelAttribute == null)
19          return "Missing protection level";
20       else
21          return "Incorrect protection level";
22    }
23
24    @NotNull
25    public String getFamilyName() {
26       return "Unprotected permission";
27    }
28
29    @Override
30    public void applyFix(@NotNull Project project, @NotNull ProblemDescriptor descriptor) {
31       (...)
32    }
33 }
```

Listing 11: Quick fix class stub

```
1  try {
2     SM03Dialog dialog = new SM03Dialog(project);
3     dialog.show();
4     String choice = null;
5     if (dialog.getExitCode() == DialogWrapper.OK_EXIT_CODE)
6        choice = dialog.getChosenValue();
7
8     if (choice == null) {
9        logger.debug(”Choice dialog was skipped”);
10       return;
11    }
12
13    logger.debug(”Choice in dialog: ” + choice);
14    String finalChoice = choice;
15
16    ApplicationManager.getApplication().invokeLater(() −> WriteCommandAction.runWriteCommandAction(project, () −> {
17       if (protectionLevelAttribute != null) {
18          protectionLevelAttribute.setValue(finalChoice);
19       } else if (permissionTag != null) {
20        permissionTag.setAttribute(ATTR_PROTECTION_LEVEL, finalChoice);
21       }
22    }));
23 } catch (ProcessCanceledException canceled) {
24    throw canceled;
25 } catch (Exception ex) {
26    logger.error(”Security Smells: Something went wrong with ” + smellId + ”\nUnable to perform” +
27          ”quickfix, i.e. fix protection level attribute”, ex);
28 }
```

Listing 12: Quick fix logic

More complex quick fixes require UI dialogs, *i.e.*, views to interact with a developer. In such cases, the quick fix logic must implement UI features and code changes as shown in Listing 12. The first code block (lines 2 to 14) spawns a custom dialog, presents it to the user, and stores the user's choice for the second code block, which performs the requested changes (lines 16 to 22). In more detail, the lines 2 and 3 present the quick fix dialog, and the custom method `getChosenValue()` (line 6) returns the `android:protectionLevel` value the user selected in that dialog. The user may cancel the quick fix in the dialog by clicking on the "Cancel" button, in which case we abort the quick fix without changing any code (lines 8 to 11). In the lines 16 to 22, we adjust the value of the `protectionLevelAttribute` to the selected value, or we add the attribute with the correct value if it does not exist at all. IntelliJ demands that all code that writes data to a resource file is wrapped in a `WriteCommandAction`. The code inside the `WriteCommandAction` is then eventually executed in the single write thread of the IDE.[4] Thus any expensive calculations necessary for writing data should be performed ahead of the `WriteCommandAction` to avoid potential responsiveness issues, *e.g.*, lagging or freezing views in the UI. In the example, our write action is very cheap and can be directly applied to the `XmlTag` and `XmlAttribute` element. However, it is not always beneficial to apply such changes directly to the existing PSI elements, *e.g.*, often it is easier to replace an existing PSI element than to implement the necessary routines to transform it.

[4]`https://plugins.jetbrains.com/docs/intellij/general-threading-rules.html`

```
1 private PsiMethod calcNewMethod(String oldMethodText, String oldParameterList, String newParameterList, Project project) {
2   String newMethodText = oldMethodText.replace(oldParameterList, newParameterList);
3   newMethodText = addCommentLineAbove(newMethodText, newParameterList);
4   PsiFileFactory fileFactory = PsiFileFactory.getInstance(project);
5   PsiJavaFile dummyFile = (PsiJavaFile) fileFactory.createFileFromText("dummy.java", JavaFileType.INSTANCE,
        ↪ "public class dummy {\n" + newMethodText + "\n}");
6   return dummyFile.getClasses()[0].getMethods()[0];
7 }
```

Listing 13: PSI element creation

Listing 13 shows that the programmatic assembly of source code fragments required for such a quick fix becomes complex very quickly. In the listing, we first patch the parameters of an existing Java method (line 2) and add a comment above the method (line 3), before we store the method in a dummy file on the disk (lines 4 and 5). Next, we access that file with the provided methods of the `PsiJavaFile` class (line 6) that automatically build the corresponding PSI tree from which we can gather the relevant `PsiMethod`.

Listing 14 shows a typical offloading of computationally demanding work from a `WriteCommandAction` so that it does not negatively impact the performance of the IDE. In this example, we use the method presented in Listing 13 to prepare the quick fix code before we submit the `WriteCommandAction` to the IDE. As a result, the `WriteCommandAction` only has to replace the relevant nodes in the tree using the method `PsiMethod.replace(...)` in line 4, and to apply the existing formatting style guidelines defined in the IDE (line 5), which are rather simple operations that finish swiftly.

```
1 PsiMethod newMethod = calcNewMethod(oldMethodText, oldParameterList, newParameterList, project);
2
3 ApplicationManager.getApplication().invokeLater(() -> WriteCommandAction.runWriteCommandAction(project, () -> {
4   PsiMethod method = oldMethod.replace(newMethod);
5   CodeStyleManager.getInstance(project).reformat(method);
6 }));
```

Listing 14: Offloading work from IntelliJ's write thread

A.1.3 UI Controls

IntelliJ further supports custom UI dialogs that are presented to the developer, for example when executing a quick fix. Listing 15 presents the stub that is required to create such a custom dialog, which has to extend IntelliJ's `DialogWrapper` class (line 1) and override its `createCenterPanel()` method (line 16). Moreover, the method `init()` (line 11) must be called before returning the Java Swing UI `JPanel` component. The `DialogWrapper` class then automatically creates a dialog with *OK* and *Cancel* buttons at the bottom and the custom JPanel component `myComponent` centered on top. Since the entire dialog UI framework is based on Java Swing, most Swing UI classes can be used to create such a view, which is represented by a `JPanel` element.

```
public class SM03Dialog extends DialogWrapper {
    protected final JPanel myComponent = new JPanel();

    public SM03Dialog(Project project) {
        // build UI in myComponent using Java Swing
        GridBagLayout layout = new GridBagLayout();
        myComponent.setLayout(layout);

        (...)

        init();
    }

    @Nullable
    @Override
    protected JComponent createCenterPanel() {
        return myComponent;
    }
}
```

Listing 15: Class stub that provides a custom dialog for a quick fix

A.2 Execution

In this section we discuss the preliminary steps to execute an inspection and its corresponding quick fix in the IntelliJ IDE. To execute custom inspections in the IDE, the inspections must be initially registered in the file `plugin.xml` that can be found in the `META-INF` folder of an IntelliJ plug-in project.[5] Listing 16 shows such a registration of an inspection. The most important attribute is `implementationClass` (line 13), which is mandatory and must target the inspection class.[6] The other attributes (lines 7 to 12) define the behavior of the registered inspection, *e.g.*, whether it should be enabled by default, or the preferred severity level the reported issues will receive. Quick fixes do not require a registration, because they are already attached to the inspections.

```
<idea-plugin>
  <id>ch.unibe.scg.security-smell-quickfix</id>
  <name>QuickFixes for Security Smells</name>
  (...)
  <extensions defaultExtensionNs="com.intellij">
    <localInspection language="XML"
            displayName="Missing/incorrect protection level"
            groupPath="Security Smells"
            groupBundle="messages.InspectionsBundle"
            groupKey="group.names.probable.bugs"
            enabledByDefault="true"
            level="WARNING"
            implementationClass="inspections.UnprotectedPermissionDetection"/>
  (...)
  </extensions>
</idea-plugin>
```

Listing 16: Inspection registration

If the inspections and quick fixes are created using the officially recommended

[5] https://plugins.jetbrains.com/docs/intellij/plugin-extensions.html#declaring-extensions

[6] https://plugins.jetbrains.com/docs/intellij/code-inspections.html#plugin-configuration-file

Gradle project,[7] they can be run and debugged from within the IDE, *i.e.*, the project must be started using the predefined Gradle task `runIde` as shown in the run configuration of Figure A.1. After the Gradle task is started, IntelliJ opens a new IDE instance in which the inspections and quick fixes are already enabled. In this distinct IDE their code can be tested and the debug console output is forwarded to the original IDE's debug output view. In order to avoid potential timeouts during the testing of the inspections and quick fixes, the `-Didea.ProcessCanceledException=disabled` VM option must be set in the run configuration, which disables that particular problematic exception. Furthermore, an environment variable is supported that can be used to define whether the inspections and quick fixes should run in debug or production mode. Certain behavior is altered in debug mode to facilitate the testing and debugging of the implementation, *e.g.*, the execution of quick fixes is not remembered by the system such that they can be tested repeatedly without running into any caching problems.

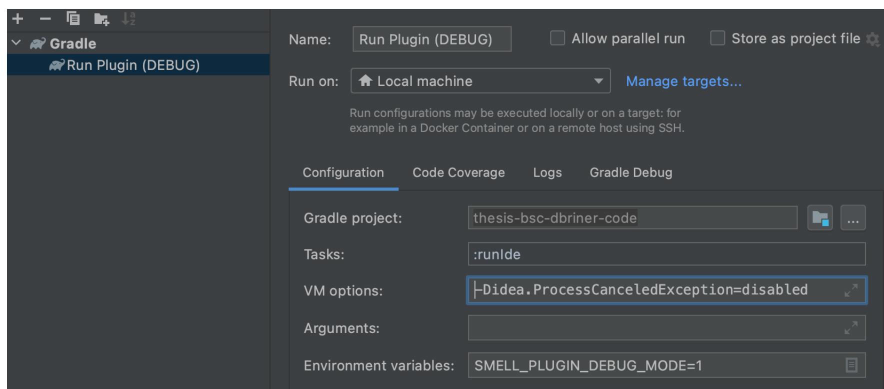

Figure A.1: Typical run configuration for inspections and quick fixes

A.3 Packaging

In this section we briefly discuss the packaging and distribution of inspections as well as quick fixes. Inspections and quick fixes are developed within a plug-in project and, consequently, can be packaged as IntelliJ plug-in. If such a plug-in file is generated, *i.e.*, a JAR archive that contains the relevant sources and metadata it can be easily integrated into existing IntelliJ installations. For that purpose, a developer has to open IntelliJ's *Settings* dialog and select the *Plugins* category. In the presented view on the right, the developer can then click on the gear icon and choose *Install Plugin from Disk...* as illustrated in Figure A.2, which will present a file chooser dialog to select

[7] `https://github.com/JetBrains/gradle-intellij-plugin`

the generated plug-in file. After these dialogs are acknowledged, the plug-in with its inspections and quick fixes will be activated immediately.

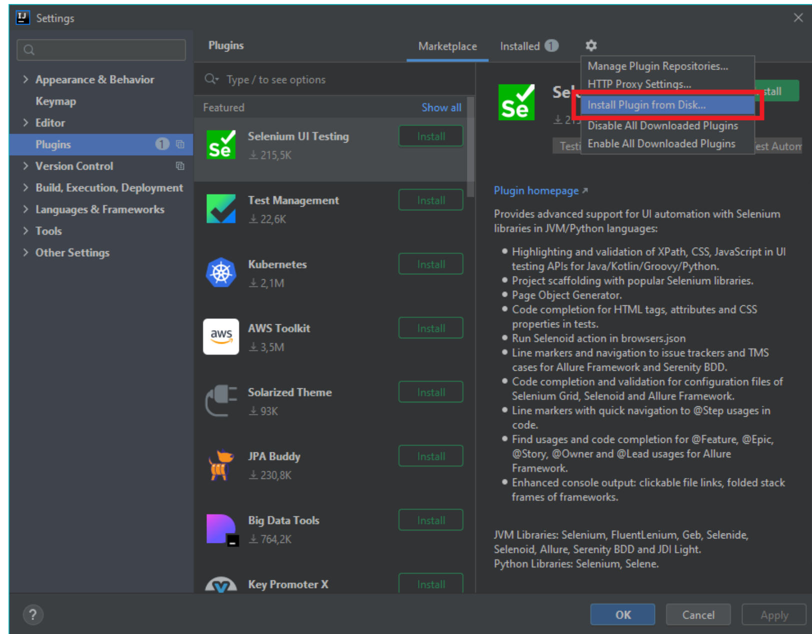

Figure A.2: IntelliJ's plug-in configuration dialog