Are Multi-language Design Smells Fault-prone? An Empirical Study

MOUNA ABIDI, DGIGL, Polytechnique Montreal, Canada
MD SAIDUR RAHMAN, DGIGL, Polytechnique Montreal, Canada
MOSES OPENJA, DGIGL, Polytechnique Montreal, Canada
FOUTSE KHOMH, DGIGL, Polytechnique Montreal, Canada

Nowadays, modern applications are developed using components written in different programming languages and technologies. The cost benefits of reuse and the advantages of each programming language are two main incentives behind the proliferation of such systems. However, as the number of languages increases, so does the challenges related to the development and maintenance of these systems. In such situations, developers may introduce design smells (i.e., anti-patterns and code smells) which are symptoms of poor design and implementation choices. Design smells are defined as poor design and coding choices that can negatively impact the quality of a software program despite satisfying functional requirements. Studies on mono-language systems suggest that the presence of design smells may indicate a higher risk of future bugs and affects code comprehension, thus making systems harder to maintain. However, the impact of multi-language design smells on software quality such as fault-proneness is yet to be investigated.

In this paper, we present an approach to detect multi-language design smells in the context of JNI systems. We then investigate the prevalence of those design smells and their impacts on fault-proneness. Specifically, we detect 15 design smells in 98 releases of nine open source JNI projects. Our results show that the design smells are prevalent in the selected projects and persist throughout the releases of the systems. We observe that in the analyzed systems, 33.95% of the files involving communications between Java and C/C++ contain occurrences of multi-language design smells. Some kinds of smells are more prevalent than others, e.g., Unused Parameters, Too Much Scattering, Unused Method Declaration. Our results suggest that files with multi-language design smells can often be more associated with bugs than files without these smells, and that specific smells are more correlated to fault-proneness than others. From analyzing fault-inducing commit messages, we also extracted activities that are more likely to introduce bugs in smelly files. We believe that our findings are important for practitioners as it can help them prioritize design smells during the maintenance of multi-language systems.


ACM Reference Format:

Authors’ addresses: Mouna Abidi, DGIGL, Polytechnique Montreal, Montreal, QC, Canada, mouna.abidi@polymtl.ca; Md Saidur Rahman, DGIGL, Polytechnique Montreal, Montreal, QC, Canada, saidur.rahman@polymtl.ca; Moses Openja, DGIGL, Polytechnique Montreal, Montreal, QC, Canada, moses.openja@polymtl.ca; Foutse Khomh, DGIGL, Polytechnique Montreal, Montreal, QC, Canada, foutse.khomh@polymtl.ca.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Association for Computing Machinery.
XXXX-XXXX/2020/10-ART $15.00
https://doi.org/10.1145/nmnnnn.nnmmnnn
1 INTRODUCTION

Modern applications are moving from the use of a single programming language to build a single application towards the use of more than one programming language [1–3]. Capers Jones reported in his book published in 1998, that at least one third of the software application at that time were written using two programming languages. He estimated that 10% of the applications were written with three or more programming languages [4]. Kontogiannis argued that these percentages are becoming higher with the technological advances [2]. Developers often leverage the strengths and take benefits of several programming languages to cope with the pressure of the market.

A common approach to develop multi-language system is to write the source code in multiple languages to capture additional functionality and efficiency not available in a single language. For example, a mobile development team might combine Java, C/C++, JavaScript, SQL, and HTML5 to develop a fully-functional application. The core logic of the application might be written in Java, with some routines written in C/C++, and using some scripting languages or other domain specific languages to develop the user interface [5]. The cost benefits of reuse and the advantages of each programming language are increasingly powerful reasons behind the proliferation of multi-language systems.

However, despite the numerous advantages of multi-language systems, they are not without some challenges. During 2013, famous web sites, e.g., Business Insider, Huffington Post, and Salon were inaccessible, redirecting visitors to a Facebook error page. This was due to a bug related to the integration of components written in different programming languages. The bug was in JavaScript widgets embedded in Facebook and their interactions with Facebook’s servers.1 Another example related to multi-language design smells is a bug reported early in 2018, which was due to the misuse of the guideline specification when using the Java Native Interface (JNI), to combine Java with C/C++ in libguests.2 There were no checks for Java exceptions after all JNI calls that might throw them. In JRuby, several problems were also reported mainly related to incompatibilities between languages and missing checks of return values and crashes related to the C language.3

Software quality has been widely studied in the literature and was often associated with the presence of design patterns, anti-patterns and code smells in the context of mono-language systems. Several studies in the literature have investigated the popularity and challenges of multi-language systems [3, 6–9], but very few of them studied multi-language patterns and practices [7–9]. Kochhar et al. [3] claims that the use of several programming languages significantly increases bug proneness. They assert that design patterns and design smells are present in multi-language systems and suggest that researchers study them thoroughly. Mono-language design smells are conjectured in the literature to hinder software reliability. While a design smell may not definitively identify an error, its presence suggests a potential trouble spot, that is, a place where there is an increased risk of bugs or potential failure in the future.

However, despite the importance and increasing popularity of multi-language systems, to the best of our knowledge, no approach has been proposed to detect multi-language smells. Also, there is no existing study that empirically evaluates the impacts of multi-language smells on the software fault-proneness. Through this paper, we aim to fill this gap in the literature. We present an approach to detect multi-language design smells. Based on our approach, we detect occurrences of 15 multi-language design smells in 98 releases of nine open source multi-language projects (i.e., VLC-android, Conscrypt, Rocksdb, Realm, Java-smt, Pljava, Javaaccp, Zstd-jni, and Jpyte). We focus on the analysis of JNI systems because they are commonly used by developers and also

1https://www.wired.com/2013/02/facebook-widget-snafu/
2https://bugzilla.redhat.com/show_bug.cgi?id=1536762
3https://www.jruby.org/2012/05/21/jruby-1-7-0-preview1.html
introduce several challenges [6, 10, 11]. Our analysis is based on a previously published catalog comprising of anti-patterns and code smells related to multi-language systems [12, 13]. In this paper, we aim to investigate the evolution of multi-language design smells and the relations between these smells and software fault-proneness. More specifically, we investigate the prevalence of 15 multi-language design smells in the context of JNI open source projects and evaluate their impact on fault-proneness.

Our four key contributions are: (1) an approach to automatically detect multi-language design smells in the context of JNI systems, (2) evaluation of the prevalence of those design smells in the selected projects, (3) empirical evaluation of the impacts of multi-language design smells on software fault-proneness, and (4) text-based analysis to identify activities that are more likely to introduce bugs once performed in files with design smells.

Our results show that in the analyzed systems, 33.95% of the files involving communication between Java and C/C++ contain occurrences of the studied design smells. Some types of smells are more prevalent than others, e.g., Unused Parameters, Too Much Scattering, Unused Method Declaration. We bring evidence to researchers that (1) the studied design smells are prevalent in the selected projects and persist within the releases, (2) some types of design smells are more prevalent than others, (3) files with the studied multi-language design smells are more likely to be the subject of bugs than files without these smells, (4) some specific smells are more correlated to fault-proneness than others i.e., Unused Parameters, Too Much Scattering, Too Much Clustering, Hard Coding Libraries, and Memory Management Mismatch, and (5) data conversion, memory management, restructuring the code, API usage and exception management activities could increase the risk of inducing bugs once performed in smelly files. We believe that our results could help not only researchers but also practitioners involved in the development of multi-language software systems. We also provide evidence to developers and quality assurance teams of the importance and usefulness of avoiding multi-language design smells.

The remainder of this paper is organized as follows. Section 2 discusses the background of multi-language systems and the design smells studied in this paper. Section 3 describes our methodology. Section 4 reports our results, while Section 5 discusses these results for better insights and implications. Section 6 summarises the threats to the validity of our methodology and results. Section 7 presents related work. Section 8 concludes the paper and discusses future works. Appendix A describes the detection rules of the proposed approach. Appendix B presents an overview of the approach validation.

2 BACKGROUND

To study the impact of multi-language design smells on fault-proneness, we first introduce a brief background on multi-language (JNI) systems. We then discuss different types of multi-language design smells and illustrate them with examples.

2.1 Multi-language Systems

Nowadays, multi-language application development is gaining popularity over mono-language programming, because of their different inherent benefits. Developers often leverage the strengths of several languages to cope with the challenges of building complex systems. By using languages that complement one another, the performance, productivity, and agility may be improved [14–16].

Java Native Interface (JNI) is a foreign function interface programming framework for multi-language systems. JNI enables developers to invoke native functions from Java code and also Java methods from native functions. JNI presents a simple method to combine Java applications with either native libraries and/or applications [17, 18]. It allows Java developers to take advantage of specific features and functionalities provided by native code. We present in Fig. 1 an example
of a JNI code extracted from [17]. Fig. 1(a) presents a Java class that contains a native method declaration `Print()` and loads the corresponding native library while Fig. 1(b) presents the C file that contains the implementation of the native function `Print()`.

(a) JNI method declaration

```java
class HelloWorld {
    static {
        System.loadLibrary("HelloWorld");
    }
    private native void print();
    public static void main(String[] args) {
        new HelloWorld().print();
    }
}
```

(b) JNI implementation function

```c
#include <jni.h>
#include <stdio.h>
#include "HelloWorld.h"

JNIEXPORT void JNICALL Java_HelloWorld_print(JNIEnv *env, jobject obj) {
    printf("Hello World!
");
    return;
}
```

Fig. 1. JNI HelloWorld Example

2.2 Anti-patterns and Code Smells

Patterns were introduced for the first time by Alexander in the domain of architecture [19]. From architecture, design patterns were then introduced in software engineering by Gamma et al. [20]. They defined design patterns as common guidelines and “good” solutions based on the developers’ experiences to solve recurrent problems. Design smells (i.e., anti-patterns and code smells), on the other hand, are symptoms of poor design and implementation choices. They represent violations of best practices that often indicate the presence of bigger problems [21, 22]. There exist several definitions in the literature about code smells, anti-patterns, and their distinction [23, 24]. However, in this paper we consider design smells, in general, to refer to both code smells and anti-patterns. Several studies in the literature studied the impacts of design smells for mono-language systems and reported that classes containing design smells are significantly more fault-prone and change-prone compared to classes without smells [25–28].

2.3 Multi-language Design Smells

Design patterns, anti-patterns, and code smells studied in the literature are mainly presented in the context of mono-language programming. While they were defined in the context of object oriented programming and mainly Java programming language, most of them could be applied to other programming languages. However, those variants consider mono-language programming and do not consider the interaction between programming languages. In a multi-language context, design smells are defined as poor design and coding decisions when bridging between different programming languages. They may slow down the development process of multi-language systems or increase the risk of bugs or potential failures in the future [12, 13].

Our study is based on the recently published catalog of multi-language design smells [12, 13]. The catalog was derived from an empirical study that mined the literature, developers’ documentation, and bug reports. This catalog was validated by the pattern community and also by surveying professional developers [11–13]. Some of those design smells could also apply to the context of mono-language systems, however, in this study we focus only on the analysis of JNI systems.

, Vol. 1, No. 1, Article . Publication date: October 2020.
this paper, since we are not analyzing anti-patterns and code smells separately but as the same entity, we will use the term design smells for both anti-patterns and code smells. In the following paragraphs, we elaborate on each of the design smells; providing an illustrative example. More details about these smells are available in the reference catalog [12, 13].

(1) **Not Handling Exceptions**: The exception handling flow may differ from one programming language to the other. In case of JNI applications, developers should explicitly implement the exception handling flow after an exception has occurred [10, 29, 30]. Since JNI exception does not disrupt the control flow until the native method returns, mishandling JNI exceptions may lead to vulnerabilities and leave security breaches open to malicious code [10, 29, 30]. Listing 1 presents an example of occurrences of this smell extracted from IBM site4. In this example, developers are using predefined JNI methods to extract a class field that was passed as a parameter from Java to C code. However, they are returning the result without any exception management. If the class or the field C is not existing, this could lead to errors. A possible solution would be to use the function `Throw()` or `ThrowNew()` to handle JNI exception, and also to add a return statement right after one of these functions to exit the native method at a point of error.

Listing 1. Design Smell - Not Handling Exceptions Across Languages

```c++
/* C++ */
(JNIEnv *)objectClass;
(JNIEnv *)fieldID;
(JNIEnv *)result = 0;
(JNIEnv *)objectClass= (*env)->GetObjectClass(env, obj);
(JNIEnv *)fieldID= (*env)->GetFieldID(env, objectClass, "charField", "C");
(JNIEnv *)result= (*env)->GetCharField(env, obj, fieldID);
```

(2) **Assuming Safe Return Value**: Similar to the previous design smell, in the context of JNI systems, not checking return values may lead to errors and security issues [13, 29]. The return values from JNI methods indicates whether the call succeeded or not. It is the developers’ responsibility to always perform a check before returning a variable from the native code to the host code to know whether the method ran correctly or not. Listing 2 presents an example of occurrences of this smell. If the class `NIOAccess` or one of its methods is not found, the native code will cause a crash as the return value is not checked properly. A possible solution would be to implement checks that handle situations in which problems may occur with the return values.

(3) **Not Securing Libraries**: A common way to load the native library in JNI is the use of the method `loadLibrary` without the use of a secure block. In such situation, the code loads a foreign library without any security check or restriction. However, after loading the library, malicious code can call native methods from the library, this may impact the security and reliability of the system [13, 31]. Listing 3, presents an example of a possible solution by loading the native library within a secure block to avoid malicious attacks.

(4) **Hard Coding Libraries**: Let us consider a situation in which we have the same code to run on different platforms. We need to customize the loading according to the operating system. However, when those libraries are not loaded considering operating system specific conditions

---

Listing 2. Design Smell - Assuming Safe Multi-language Return Values

```c++
/* C++ */
static void nativeClassInitBuffer(JNIEnv *_env){
    jclass nioAccessClassLocal= _env->FindClass("java/nio/NIOAccess");
    nioAccessClass=(jclass) _env->NewGlobalRef(nioAccessClassLocal);
    bufferClass=(jclass) _env->NewGlobalRef(bufferClassLocal);
    positionID= _env->GetFieldID(bufferClass, "position", "I");
}
```

Listing 3. Securing Library Loading

```java
/* Java */
static { AccessController.doPrivileged(
    new PrivilegedAction<Void>() {
        public Void run() {
            System.loadLibrary("osxsecurity");
            return null; }
    });
```

and requirements, but for instance with hard coded names and a try-catch mechanism, it is hard to know which library has really been loaded which could bring confusion especially during the maintenance tasks. Listing 4 provides an example of native libraries loaded without any information about how to distinguish between the usage of those libraries.

Listing 4. Design Smells - Hard Coding Libraries

```java
/* Java */
public static synchronized Z3SolverContext create(
    try { System.loadLibrary("z3"); System.loadLibrary("z3java");
    } catch (UnsatisfiedLinkError e1) {
    try { System.loadLibrary("libz3");
        System.loadLibrary("libz3java");
    } catch (UnsatisfiedLinkError e2) {...}
```

(5) **Not Using Relative Path**: This smell occurs when the library is loaded by using an absolute path to the library instead of the corresponding relative path. Using a relative path, the native library can be loaded and installed everywhere. However, the use of an absolute library path can introduce future bugs in case the library is no longer used. This may also impact the reusability of the code and its maintenance because the library can become inaccessible due to incorrect path. `System.loadLibrary("osxsecurity")` is an example of this design smell.

(6) **Too Much Clustering**: Too many native methods declared in a single class would decrease readability and maintainability of the code. This will increase the lines of code within that class and thus make the code review process harder. Many studies discussed good practices about the number of methods to have within the same class, some examples are the rule of 30
introduced by Martin Lippert [32], or the 7 plus/minus 2 rule stating that a human mind can hold and comprehend from five to 9 objects. Most of the relevant measures are the coupling, cohesion, the single principle responsibility, and the separation of concerns. In this context, a bad practice would be to concentrate multi-language code in few classes, regardless of their role and responsibilities. This may result in a blob multi-language class with many methods and low cohesion. We present in Fig. 2 an example that we extracted from ZMQJNI. In this example, native methods related to cryptographic operations are mixed in the same class as the methods used for network communication. This merging of concerns resulted in a blob multi-language class that contains 29 native declaration methods and 78 attributes. In the current study we are considering the case of having an excessive number of calls to native methods within the same class.

(7) Too Much Scattering: Similar to too much clustering, when using multi-language code, developers and managers often have to decide on a trade-off between isolating or splitting the native code. Accessing this trade-off is estimated to improve the readability and maintainability of the systems [13]. This design smell occurs when classes are scarcely used in multi-language

communication without satisfying both the coupling and the cohesion. In Figure 3 extracted from a previous work [12], we have three classes with only two native methods declaration with duplicated methods. A possible good solution would be to reduce the number of native method declaration by removing the duplicated ones possibly by regrouping the common ones in the same class. This will also reduce the scattering of multi-language participants and concerns by keeping the multi-language code concentrated only in specific classes.

(8) **Excessive Inter-language Communication**: A wrong partitioning in components written in different programming languages leads to many calls in one way or the other. This may add complexity, increase the execution time, and may indicate a bad separation of concerns. Occurrences of this design smell could be observed in systems involving different layers or components. For example, the same object could be used and-or modified by multiple components. An excessive call of native code within the same class, could be illustrated whether by having too many native method calls in the same class or having the native method call within a large range loop. In Godot, the function process() is called at each time delta. The time delta is a small period of time that the game does not process anything i.e., the engine does other things than game logic out of this time range. The foreign function process() is called multiple times per second, in this case once per frame.\(^6\)

(9) **Local References Abuse**: For any object returned by a JNI function, a local reference is created. JNI specification allows a maximum of 16 local references for each method. Developers should pay attention on the number of references created and always deleted the local references once not needed using JNIDeleteLocalRef(). Listing 5 illustrates an example of this design smell in which local references are created without deleting them.

(10) **Memory Management Mismatch**: Data types differ between Java and C/C++. When using JNI, a mapping is performed between Java data types and data types used in the native code.\(^7\) JNI handles Java objects, classes, and strings as reference types. JVM offers a set of predefined methods that could be used to access fields, methods, and convert types from Java to the native code. Those methods return pointers that will be used by the native code to perform the calculation. The same goes for reference types, the predefined methods used

---

\(^6\)https://github.com/godotengine/godot-demo-projects/blob/master/2d/pong/paddle.gd

\(^7\)https://www.developer.com/java/data/jni-data-type-mapping-to-cc.html
Listing 5. Design Smell - Local References Abuse

```c++
/* C++ */
for (i=0; i < count; i++) {
    jobject element = (*env)->GetObjectArrayElement(env, array, i);
    if((*env)->ExceptionOccurred(env)) { break;}
```

allow to either return a pointer to the actual elements at runtime or to allocate some memory and make a copy of that element. Thus, due to the differences of types between Java and C/C++, the memory will be allocated to perform respective type mapping between those programming languages. Memory leaks will occur if the developer forgets to take care of releasing such reference types. Listing 6 presents an example in which the memory was not released using ReleaseString or ReleaseStringUTF.

Listing 6. Refactoring - Memory Management Mismatch

```c++
/* C++ */
str = env->GetStringUTFChars(javaString, &isCopy);
```

(11) Not Caching Objects: To access Java objects’ fields from native code through JNI and invoke their methods, the native code must perform calls to predefined functions i.e., FindClass(), GetFieldID(), GetMethodID(), and GetStaticMethodID(). For a given class, IDs returned by GetFieldID(), GetMethodID(), and GetStaticMethodID() remain the same during the lifetime of the JVM process. The call of these methods is quite expensive as it can require significant work in the JVM. In such situation, it is recommended for a given class to look up the IDs once and then reuse them. In the same context, looking up class objects can be expensive, a good practice is to globally cache commonly used classes, field IDs, and method IDs. Listing 7 provides an example of occurrences of this design smell that does not use cached field IDs.

Listing 7. Design Smell - Not Caching Objects’ Elements

```c++
/* C++ */
int sumVal (JNIEnv* env,jobject obj,jobject allVal){
    jclass cls=(*env)->GetObjectClass(env,allVal);
    jfieldID a=(*env)->GetFieldID(env,cls,"a","I");
    jfieldID b=(*env)->GetFieldID(env,cls,"b","I");
    jfieldID c=(*env)->GetFieldID(env,cls,"c","I");
    jint aval=(*env)->GetIntField(env,allVal,a);
    jint bval=(*env)->GetIntField(env,allVal,b);
    jint cval=(*env)->GetIntField(env,allVal,c);
    return aval + bval + cval;}
```

(12) Excessive Objects: Accessing field’s elements by passing the whole object is a common practice in object oriented programming. However, in the context of JNI, since the Object type does
not exist in C programs, passing excessive objects could lead to extra overhead to properly perform the type conversion. Indeed, this design smells occurs when developers pass a whole object as an argument, although only some of its fields were needed, and it would have been better for the system performance to pass only those fields except the purpose to pass the object to the native side was to set its elements by the native code using SetField methods, with x the type of the field. Indeed, in the context of object-oriented programming, a good solution would be to pass the object offering a better encapsulation, however, in the context of JNI, the native code must reach back into the JVM through many calls to get the value of each field adding extra overhead. This also increases the lines of code which may impact the readability of the code [13]. Listing 8 presents an example smell of passing excessive objects. The refactored solution of this smell would be to pass the class’ fields as a method parameters as described in our published catalog [13].

Listing 8. Design Smell - Passing Excessive Objects

```c++
/* C++ */
int sumValues(JNIEnv* env, jobject obj, jobject allVal)
{
jint avalue= (*env)->GetIntField(env,allVal,a);
jint bvalue= (*env)->GetIntField(env,allVal,b);
jint cvalue= (*env)->GetIntField(env,allVal,c);
return avalue + bvalue + cvalue;
}
```

(13) **Unused Method Implementation**: This appears when a method is declared in the host language (Java in our case) and implemented in the foreign language (C or C++). However, this method is never called from the host language. This could be a consequence of migration or refactoring in which developers opted for keeping those methods to not break any related features.

(14) **Unused Method Declaration**: Similar to **Unused Method Implementation**, this design smell occurs when a method is declared in the host language but is never implemented in the native code. This smell and the previous one are quite similar. However, they differ in the implementation part, while for the smell **Unused Method Implementation**, the method is implemented but never called, in case of the smell **Unused Method Declaration**, the unused method is not implemented and never called in the foreign language. Such methods could remain in the system for a long period of time without being removed because having them will not introduce any bug when executing the program but they may negatively impact the maintenance activities and effort needed when maintaining those classes.

(15) **Unused Parameters**: Long list of parameters make methods hard to understand [33]. It could also be a sign that the method is doing too much or that some of the parameters are no longer used. In the context of multi-language programming, some parameters may be present in the method signature however they are no longer used in the other programming language. Since multi-language systems usually involve developers from different teams, those developers often prefer not to remove such parameters because they may not be sure if the parameters are used by other components. Listing 9 presents an illustration of this design smell where the parameter acceleration is used in the native method signature. However, it is not used in the implemented function.
Listing 9. Design Smell - Unnecessary Parameters

/* C++ */
JNIEXPORT jfloat JNICALL Java_jni_distance
    (JNIEnv *env, jobject thisObject,
     jfloat time, jfloat speed,
     jfloat acceleration)
{
    return time * speed;
}

3 STUDY DESIGN

In this section, we present the methodology we followed to conduct this study. Figure 4 provides an overview of our methodology.

Fig. 4. Schematic Diagram of the Study

3.1 Setting Objectives of the Study

We started by setting the objective of our study. Our objective is to investigate the prevalence of multi-language design smells in the context of JNI systems and the relation between those smells with software fault-proneness. We also aim to investigate what kind of activities once performed in smelly files are more likely to introduce bugs. The quality focus in this study is the occurrence of bugs due to the presence of design smells in JNI systems. The perspective is that of researchers, interested in the quality of JNI systems, and who want to get evidence on the impact of design smells on the software fault-proneness. Also, these results can be of interest to professional developers performing maintenance and evolution activities on JNI projects and who need to take into account and forecast their effort, since like for mono-language projects, the presence of fault-prone files is likely to increase the maintenance effort and cost. These results are also of interest to testers since they need to know which files are more important to test. Finally, they can be of interest to quality assurance teams or managers who could use design smells detection techniques to assess the fault-proneness of in-house or to-be-acquired source code, to better quantify the cost-of-ownership of JNI systems.

We defined our research questions as follows:

RQ1: Do Multi-language design smells occur frequently in open source projects?

Several articles in the literature discussed the prevalence, detection, and evolution of design smells in the context of mono-language systems [34, 35]. Occurrences of design smells may hinder the evolution of a system by making it hard for developers to maintain the system. The detection of smells can substantially reduce the cost of maintenance and development
activities. However, most of those research are focusing on mono-language systems. Thus, we decided to fill this gap in the literature and investigate the frequency of design smells in the context of multi-language systems. This research question is preliminary to the remaining questions. It aims to examine the frequency and distribution of multi-language design smells in the selected projects and their evolution over the releases of the project. We defined the following null hypothesis: $H_1$: there are no occurrences of the multi-language design smells studied in the literature in the selected projects.

RQ2: Are some specific Multi-language design smells more frequent than others in open source projects?
Given that multi-language design smells are prevalent in the studied systems, it is important to know the distribution and evolution of the different types of smells for a better understanding of the implication of their presence for maintenance activities. Developers are likely to benefit from knowing the dominating smells to treat them in priority and avoid introducing such occurrences. Consequently, in this research question, we aim to study whether some specific types of design smells are more prevalent than others. We are also interested in the evolution of each type of smells over the releases of the project. We aim to test the following null hypothesis: $H_2$: The proportion of files containing a specific type of design smell does not significantly differ from the proportion of files containing other kinds of design smells.

RQ3: Are files with Multi-language design smells more fault-prone than files without?
Prior works reported that classes containing design smells in mono-language systems are more prone to faults than other classes [25, 36]. Due to components written in different languages, multi-language systems may have more complexities in architecture and inter-component interactions. Given the known impacts of design smells on mono-language systems, it is thus important to investigate the impacts of multi-language design smells on the corresponding software systems. To examine this, we aim to investigate whether source files containing multi-language design smells are more likely to experience faults than files without smells. We investigate whether files with multi-language design smells are more fault-prone than others by testing the null hypothesis: $H_3$: The proportion of files experiencing at least one bug does not significantly differ between files with design smells and files without.

RQ4: Are some specific Multi-language design smells more fault-prone than others?
During maintenance and quality assurance activities, developers are interested in identifying parts of the code that should be tested and/or refactored in priority. Hence, we are interested in identifying design smells that are more fault-prone than others. Thus, we defined the null hypothesis. $H_4$: There is no significant difference between the impacts of different kinds of multi-language design smells on the fault-proneness of files containing those smells.

RQ5: What are the activities that are more likely to introduce bugs in smelly files?
During the maintenance of a project, having knowledge of possible risky activities could help developers and managers to reduce the risk of bugs. They could benefit from that knowledge to capture activities that should be performed with caution in smelly files. Hence, we are interested in identifying what kinds of activities once performed in smelly files are likely to introduce bugs. Capturing such information could provide insights about what kind of activities could increase the risk of bugs in smelly files.

3.2 Data Collection
In order to address our research questions, we selected nine open source projects hosted on GitHub. We decided to analyze those nine systems because they are well maintained, and highly active. Another criteria for the selection was that those systems have different size and belong to different
Are Multi-language Design Smells Fault-prone? An Empirical Study

Table 1. Research Objectives and Research Questions

<table>
<thead>
<tr>
<th>Research Objectives</th>
<th>Methodology</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Objective 1:</strong> Detect multi-language design smells</td>
<td>Detection approach (case study of JNI systems)</td>
</tr>
<tr>
<td><strong>Objective 2:</strong> Investigate the prevalence of multi-language design smells</td>
<td>RQ1 and RQ2</td>
</tr>
<tr>
<td><strong>Objective 3:</strong> Study the relationship between multi-language design smells and fault-proneness</td>
<td>RQ3 and RQ4</td>
</tr>
<tr>
<td><strong>Objective 4:</strong> Identifying fault-inducing activities</td>
<td>RQ5</td>
</tr>
</tbody>
</table>

Table 2. Overview of the Studied Systems

<table>
<thead>
<tr>
<th>Systems</th>
<th>Domain</th>
<th>#Releases</th>
<th>#Commits</th>
<th>#Issues</th>
<th>LOC</th>
<th>Java</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rocksdb</td>
<td>Facebook Database</td>
<td>189</td>
<td>8375</td>
<td>1748</td>
<td>487853</td>
<td>11%</td>
<td>83.1%</td>
</tr>
<tr>
<td>VLC-android</td>
<td>Media Player and Database</td>
<td>176</td>
<td>12697</td>
<td>1091</td>
<td>125037</td>
<td>10.1%</td>
<td>6.7%</td>
</tr>
<tr>
<td>Realm</td>
<td>Mobile Database</td>
<td>169</td>
<td>8244</td>
<td>3886</td>
<td>171705</td>
<td>82%</td>
<td>8.1%</td>
</tr>
<tr>
<td>Conscrypt</td>
<td>Cryptography (Google)</td>
<td>32</td>
<td>3874</td>
<td>186</td>
<td>91765</td>
<td>85.3%</td>
<td>14%</td>
</tr>
<tr>
<td>Pljava</td>
<td>Database</td>
<td>27</td>
<td>1236</td>
<td>123</td>
<td>71910</td>
<td>67%</td>
<td>29.7%</td>
</tr>
<tr>
<td>Javacpp</td>
<td>Compiler</td>
<td>34</td>
<td>658</td>
<td>269</td>
<td>28713</td>
<td>98%</td>
<td>0.6%</td>
</tr>
<tr>
<td>Zstd-jni</td>
<td>Data Compression (Facebook)</td>
<td>36</td>
<td>423</td>
<td>78</td>
<td>72824</td>
<td>4.3%</td>
<td>92.1%</td>
</tr>
<tr>
<td>Jpype</td>
<td>Cross Language Bridge</td>
<td>14</td>
<td>895</td>
<td>305</td>
<td>53826</td>
<td>7.8%</td>
<td>58%</td>
</tr>
<tr>
<td>Java-smt</td>
<td>Computation</td>
<td>22</td>
<td>1822</td>
<td>146</td>
<td>42049</td>
<td>88%</td>
<td>4.6%</td>
</tr>
</tbody>
</table>

Among the nine selected systems, VLC-android is a highly portable multimedia player for various audio and video formats. Rocksdb is developed and maintained by Facebook, it presents a persistent key-value store for fast storage. It can also be the foundation for a client-server database. Realm is a mobile database that runs directly inside phones and tablets. Conscrypt is developed and maintained by Google, it is a Java Security Provider (JSP) that implements parts of the Java Cryptography Extension (JCE) and Java Secure Socket Extension (JSSE). Java-smt is a common API layer for accessing various Satisfiability Modulo Theories (SMT) solvers. Pljava is a free module that brings Java Stored Procedures, Triggers, and Functions to the PostgreSQL backend via the standard JDBC interface. Javacpp provides efficient access to native C++ inside Java, not unlike the way some C/C++ compilers interact with assembly language. Zstd-jni present a binding for Zstd native library developed and maintained by Facebook that provides fast and high compression lossless algorithms for Android, Java, and all JVM languages. Jpype is a Python module to provide full access to Java from within Python.
Table 3. Analyzed Releases in Each Project

<table>
<thead>
<tr>
<th>Systems</th>
<th>#Releases Analyzed</th>
<th>Releases</th>
<th>Analysis Periods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rocksdb</td>
<td>10</td>
<td>5.0.2 - latest release</td>
<td>2017-18-01 - 2019-14-08</td>
</tr>
<tr>
<td>VLC-android</td>
<td>10</td>
<td>3.0.0 - latest release</td>
<td>2018-08-02 - 2019-13-09</td>
</tr>
<tr>
<td>Realm</td>
<td>10</td>
<td>0.90.0 - 5.15.0</td>
<td>2016-03-05 - 2019-04-09</td>
</tr>
<tr>
<td>Conscrypt</td>
<td>11</td>
<td>1.0.0.RC11 - 2.3.0</td>
<td>2017-25-09 - 2019-25-09</td>
</tr>
<tr>
<td>Javacpp</td>
<td>13</td>
<td>0.5 - 1.5.1-1</td>
<td>2013-07-04 - 2019-05-09</td>
</tr>
<tr>
<td>Zstd-jni</td>
<td>11</td>
<td>0.4.4 - latest release</td>
<td>2015-17-12 - 2019-19-08</td>
</tr>
<tr>
<td>Jpype</td>
<td>11</td>
<td>0.5.4.5 - latest release</td>
<td>2013-25-08 - 2019-13-09</td>
</tr>
<tr>
<td>Java-smt</td>
<td>10</td>
<td>0.1 - 3.0.0</td>
<td>2015-27-11 - 2019-30-08</td>
</tr>
</tbody>
</table>

3.3 Data Extraction

To answer our research questions, we first have to mine the repositories of the nine selected systems to extract information about the occurrences of smells existing in each file and also the bugs reported for those systems.

3.3.1 Detection of Design Smells.

Detection Approach: Because no tools are available to detect design smells in multi-language systems, we extended the Ptidej Tool suite\(^\text{17}\) by building a new detection approach for JNI smells. Specifically, we extended DECOR [35] to integrate new rules related to the combination of Java and C/C++. We used srcML\(^\text{18}\), a parsing tool that converts source code into srcML, which is an XML format representation. The srcML representation of source code adds syntactic information as XML elements into the source code text. Listing 11 presents the srcML representation of the code snippet presented in 10. The main advantage of srcML, is that it supports different programming languages, and generates a single XML file for the supported programming languages. For now, our approach includes only Java, C, and C++, however, it could be extended to include other programming languages in the future. SrcML provides a wide variety of predefined functions that could be easily used through the XPath to implement specific tasks. XPath is frequently used to navigate through XML nodes, elements, and attributes. In our case, it is used to navigate through srcML elements generated as an AST of a given project. The ability to address source code using XPath has been applied to several applications [37].

Our detection approach reports smell detection results for a selected system in a CSV file. The report provides detailed information for each smells detected such as smell type, file location, class name, method name, parameters (if applicable). We then used a python script to post-process the results to create a summary file. The summary results contain the total number of occurrences of each type of smell in a specific file or class in a specific release of the selected system. Two members of our research team manually validated the results of smell detection for five systems.

Detection Rules: The detection approach is based on a set of rules defined from the documentation of the design smells. Those rules were validated by the pattern community during the Writers’ workshop to document and validate the smells. For example, for the design smell Local Reference Abuse, we considered cases where more than 16 references are created but not deleted with the DeleteLocalRef function. The threshold 16 was extracted from developers blogs discussing best

\(^\text{17}\)http://www.ptidej.net/tools/
\(^\text{18}\)https://www.srcml.org/
practices and the Java Native Interface specification [17]. We present in the following two examples of rules as well as the thresholds used to define them, and their detection process. All the other rules are available in Appendix A. We also provide in the replication folder all the detection rules for the design smells studied in this paper and the detection results.

*Listing 10. Example of Java Code*

```java
public class HelloWorld {
    public static void main(String[] args) {
        // Prints "Hello World!" to stdout
        System.out.println("Hello World!");
    }
}
```

*Listing 11. Example of Java Code Converted to SrcML*

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<unit xmlns="http://www.srcML.org/srcML/src" revision="0.9.5" language="Java"
    filename="HelloWorld.java"><class><specifier>public</specifier> class<name>HelloWorld</name> <block>{
    <function><specifier>public</specifier> <specifier>static</specifier> <type><name>void</name></type><name>main</name><parameter_list>(<parameter><decl><type><name><name>String</name><index>[]</index></name></type><name>args</name></decl></parameter>)</parameter_list> <block>{
        <comment type="line">// Prints "Hello World!" to stdout</comment>
        <expr_stmt><expr><call><name><name>System</name><operator>.</operator><name>out</name><operator>.</operator><name>println</name></name><argument_list>(<argument><expr><literal type="string">"Hello World!"</literal></expr>)</argument_list></call></expr>;
    }
}</block></class></unit>
```

(1) **Rule for the smell Not Handling Exceptions**

\[
(f(y) | f \in \{\text{GetObjectClass, FindClass, GetFieldID, GetStaticFieldID, GetMethodID, GetStaticMethodID}\}) \\
\text{AND} (\text{isErrrorChecked}(f(y)) = \text{False OR ExceptionBlock}(f(y)) = \text{False})
\]

19https://www.cnblogs.com/cbcsan/articles/4733508.html
20https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#global_local
21https://github.com/ResearchML/ImapctFault

, Vol. 1, No. 1, Article . Publication date: October 2020.
Our detection rule for the smell Not Handling Exceptions is based on the existence of call to specific JNI methods requiring explicit management of the exception flow. The JNI methods (e.g., FindClass) listed in the rule should have a control flow verification. The parameter $y$ represents the Java object/class that is passed through a native call for a purpose of usage by the C/C++ side. Here, isExceptionChecked allows to verify that there is an error condition verification for those specific JNI methods, while ExceptionBlock checks if there is an exception block implemented. This could be implemented using Throw() or ThrowNew() or a return statement that exists in the method in case of errors.

If we recheck Listing 1 in Section 2, the code illustrated in this example satisfies the rule of using predefined methods to access classes and field Ids. Another condition is that those methods are not followed by an explicit exception block. Thus, this example will be captured by our approach as an occurrence of the design smell Not Handling Exceptions.

(2) Rule for the smell Local References Abuse

\[
(N_{\text{Local Reference}}(f_1(y)) > \text{MaxLocalReferenceThreshold}) \text{ AND } \\
(f_1(y) | f_1 \in \{\text{GetObjectArrayElement}, \text{GetObjectArrayElement}, \text{NewLocalRef}, \text{AllocObject}, \text{NewObject}, \text{NewObjectA}, \text{NewObjectV}, \text{NewDirectByteBuff er}, \text{ToRe flectedMethod}, \text{ToRe flectedField})\} \text{ AND } \\
(\exists f_2(y) | f_2 \in \{\text{DeleteLocalRef}, \text{EnsureLocalCapacity}\})
\]

The smell Local References Abuse is introduced when the total number of local references created inside a called method exceeds the defined threshold and without any call to method DeleteLocalRef to free the local references or a call to method EnsureLocalCapacity to inform the JVM that a larger number of local references is needed.

In the same vein, if we recall the example provided in Listing 5, in which a local reference is created to retrieve an array element. This is implemented inside a loop (for). Thus, if the total number for the count is more than 16, this indicates that we are exceeding the authorized number of local references. In this situation, our approach will capture the method exceeding the authorized number of local references and will then check for any possible usage of functions to release the memory. Since this example does not provide any functions to release the memory, this will be detected by our approach as an occurrence of the design smell Local References Abuse.

Validation Approach:

To assess the recall and precision of our detection approach, we evaluated the results of our detection approach at the first level by creating dedicated unit tests for the detector of each type of smell to confirm that the approach is detecting the smells introduced in our pilot project. We relied on six open source projects used in previous works [12, 13] on multi-language design smells. For each of the systems, we manually identified occurrences of the studied design smells. Two of the authors independently identified occurrences of the design smells in JNI open source projects, and resolved disagreements through discussions with the whole research team. Using the ground truth based on the definition of the smell and the detection results, we computed precision and recall as presented in Table 4 to evaluate our smell detection approach. Precision computes the number of true smells contained in the results of the detection tool, while recall computes the fraction of true smells that are successfully retrieved by the tool. From the six selected systems, we obtained a precision between 88% and 99%, and a recall between 74% and 90%. We calculate precision and recall based on the following equations (1) and (2) respectively:
3.3.2 Detection of Fault-inducing Commits. Our studied systems use Github as the issue tracker. We used Github APIs and PyDriller to mine the software repositories and get the list of all the commit logs and resolved issues for the systems [38]. PyDriller provides a set of APIs to extract information from Git repositories. These include important historical information regarding commits, developers, and modifications. PyDriller is very convenient for mining software repositories to analyze changes or bugs. It rely on the SZZ algorithm [39] to detect changes that introduce faults. We used PyDriller because this approach was not only evaluated regarding existing tools but also with experiments involving developers [38]. We started by retrieving all the information related to the projects. We analyzed all commit messages to identify the fault-fixing commits. We used a set of error related keywords to identify commits related to fault-fixing using a heuristic similar to that presented in the study by Mockus and Votta [40]. Our list of keywords include “fix”, “crash”, “resolves”, “regression”, “fall back”, “assertion”, “covery”, “reproducible”, “stack-wanted”, “steps-wanted”, “testcase”, “fail”, “npe”, “except”, “broken”, “bug”, “differential testing”, “error”, “addressesanitizer”, “hang”, “permaorange”, “random orange”, “intermittent”, “steps to reproduce”, “assertion”, “leak”, “stack trace”, “heap overflow”, “freez”, “str”, “problem “, “overflow”, “avoid”, “issue”, “workaround”, “break”, and “stop”.

To retrieve fault-inducing commits, given a commit, PyDriller returns the set of commits that previously modified the lines from the files included in the given commit. It applies the SZZ algorithm to find the commit when the bug was initially introduced as used in some earlier studies [41–43]. To locate the fault-inducing commits, PyDriller algorithm works as follows: for every file in the commit, it obtains the diff between the files, then obtains the list of all deleted lines. It then blames the file to obtain the commits where the deleted lines were changed. We tagged fault-inducing commits as buggy. We used this tag later to distinguish between files containing bugs and files without.

Since Pydriller’s SZZ implementation was not previously evaluated, we manually examined the bug inducing commits retrieved by Pydriller from two of our studied projects, Pljava and Zstd-jni. We performed this manual analysis in two steps. First, we executed an existing implementation of the SZZ algorithm available on GitHub on Pljava and Zstd-jni. We compared its reported results with the results obtained from Pydriller. For each bug fixing commit, we manually verified if the related bug inducing commit reported by Pydriller matches with the one reported by SZZ. For that, we used two labels (True or False) to distinguish between the bug inducing commits that match with those retrieved by SZZ and those that do not match. Next, one of the authors manually verified if the changes in the bug inducing commits reported by Pydriller were indeed related to the changes performed in the corresponding bug fixing commits. We also analyzed the commit messages. We labeled each of the bug inducing commits with three tags (True, False, and Unclear). We used the tag True in situations in which we were convinced that the change performed in the bug fixing was indeed related to the changes applied in the bug inducing. We assigned False in situations in which it was evident that the changes are not related, and Unclear, in situations in which it was not completely evident to assign a True or False tag. We analyzed for Pljava and Zstd-jni respectively a total of 113 and 96 bug-fixing commits. We performed a cleaning process
Table 4. Validation of the Smell Detection Approach

<table>
<thead>
<tr>
<th>Systems</th>
<th>True Positive</th>
<th>False Positive</th>
<th>False Negative</th>
<th>Recall</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>openj9</td>
<td>3293</td>
<td>137</td>
<td>250</td>
<td>93%</td>
<td>96%</td>
</tr>
<tr>
<td>rocksdb</td>
<td>922</td>
<td>50</td>
<td>136</td>
<td>87%</td>
<td>95%</td>
</tr>
<tr>
<td>conscrypt</td>
<td>556</td>
<td>29</td>
<td>133</td>
<td>80%</td>
<td>95%</td>
</tr>
<tr>
<td>pilot project</td>
<td>32</td>
<td>0</td>
<td>0</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>pljava</td>
<td>511</td>
<td>5</td>
<td>53</td>
<td>90%</td>
<td>99%</td>
</tr>
<tr>
<td>jna</td>
<td>375</td>
<td>50</td>
<td>127</td>
<td>74%</td>
<td>88%</td>
</tr>
<tr>
<td>jmonkey</td>
<td>2210</td>
<td>142</td>
<td>185</td>
<td>92%</td>
<td>94%</td>
</tr>
</tbody>
</table>

on those commits and removed the commits related to typos fixing and merge commits. We kept in our validation bug-fixing commits with their corresponding bug-inducing commits. Our final dataset results on 61 bug-fixing commits for Pljava and 66 bug-fixing commits for Zstd-jni. From our manual validation of fault-inducing commits reported by Pydriller for Pljava and Zstd-jni, we found respectively precision values of 78.94% and 70.83%. Those values are computed considering only the True and False tags for Java and C/C++ files resulting from our manual validation. From the comparison between Pydriller’s szz and the recent implementation of szz, we found for Pljava and Zstd-jni, respectively precision values of 85% and 80%. We did not include in our validation the recall because in our study we are considering only JNI code. However, szz is considering the whole project in general without considering multi-language interactions. So, those results may not generalize to the whole system. However, the results of our manual validation do not directly contribute to any of our empirical findings, and we did this validation as a complementary step to reduce the threats to validity of our study. We also analyzed changes related to multi-language programming. Indeed, in many situations, the Java and C/C++ code are changed within the same commit. This was helpful to validate the bug inducing commits involving Java and C/C++ code.

3.4 Analysis Method

We present in the following the analysis performed to answer our research questions.

3.4.1 Analyzing the prevalence of design smells. We investigate the presence of 15 different kinds of design smells. Each variable $s_{i,j,r}$ reflects the number of times a file $i$ has a smell $j$ in a specific release $r_k$.

For RQ1, since we are interested to investigate the prevalence of multi-language design smells, we aggregate these variables into a Boolean variable $s_{i,k}$ to indicate whether a file $i$ has at least any kind of smells in release $r_k$. We calculate the percentage of files affected by at least one of the studied design smells, $s_j$. We use our detection approach to detect occurrences of multi-language design smells following the methodology described earlier. For each file, we compute the value of a variable $Smelly_{i,r}$ which reflects if the file $i$ has a least one type of smell in a specific release $r$. This variable takes 1 if the file contains at least one design smell in a specific release $r$, and 0 otherwise. Similarly, we also compute the value of variable $Native_{i,r}$, which takes 1 if the file $i$ of a specific release $r$ is native and 0 if not. Since our tool is focusing on the combination of Java and C/C++, we compute for each release the percentage of files participating in at least one JNI smells out of the total number of JNI files (files involved in Java and C/C++).

For RQ2, we investigate whether a specific type of design smells is more prevalent in the studied systems than other types of design smells. For that, we calculate for each system the percentage of files affected by each type of the studied smells $j$. For each file $i$ and for each release $r$, we defined a flag $Smelly_{i,j,r}$ which takes the value 1 if the release $r$ of the file $i$ contains the design smell type
and 0 if it does not contain that specific smell. Based on this flag, we compute for each release the number of files participating in that specific smell. We also calculate the percentage of smelly files containing each type of smell. Note that the same file may contain more than one smell. We investigate the presence of 15 different kinds of smells. We also compute the metric $s_{i,j,k}$ which reflects the number of occurrences of smells of type $j$ in a file $i$ in a specific release $r_k$.

### 3.4.2 Analyzing the impacts of smells on bugs

For RQ3, we focus on each of the smells to study whether the proportion of files containing at least one bug, significantly differs between files containing smells and files without smells. We consider the number of bugs $c_{i,k}$ a file $i$ encountered between releases $r_k$ and $r_{k+1}$, and convert $c_{i,k}$ into a Boolean variable $f_{i,k}$ (true if the file underwent at least one bug, false otherwise). We rely on Fisher’s exact test [44] to check whether the proportion of buggy files varies between two samples (files with and without smells). This test is useful for categorical data that result from the classification of objects. It is used to examine the significance of the association between the two kinds of classification. We also calculate the odds ratio (OR) indicating the likelihood for an event (bug in our case) to occur. The odds ratio is calculated (as in Equation (3)) as the ratio of the odds $p$ of an event occurring in a sample, i.e., the odds that files with some specific smells contain a bug (defined as experimental group), to the odds $q$ of the same event occurring in another sample, i.e., the odds that files with no smells contain a bug (defined as control group):

$$OR = \frac{p/(1-p)}{q/(1-q)}$$

(3)

An OR equal to 1 indicates that the event of interest is equally likely in both samples. While an OR greater than 1 stipulates that the event is more likely to occur in the first sample (files participating in some design smells), having an OR less than 1 indicates that it is more likely to occur in the second sample (control group of files not participating in any design smell). We use the `fisher_exact` function of the `stats` module from `scipy` Python package to compute the odds ratio and the $p$-value for statistical significance of the test. By processing the commits and bug information we set different flags for each of the source files. As mentioned earlier, the `smelly` flag takes the value 1 if the associated source file contains at least one design smell of any type, and 0 otherwise. The flag `buggy`, takes the value 1 if the associated source file was identified by SZZ algorithm as related to a fault-inducing commit, and 0 otherwise. Now, for a given release of a system, we consider all JNI source files for analysis. We count the number of buggy and non-buggy files with design smells. Similarly, we also count the number of buggy and non-buggy files without design smells. With these four values, we form the 2x2 contingency table for Fisher’s exact test.

For RQ4, we investigate the relationship between different types of design smells with fault-proneness. Unlike using logistic regression for prediction purposes ([34, 45]), we use it to examine whether some types of design smells are more related to fault-proneness. Our analysis approach is similar to the one presented by Khomh et al. [34] where they investigate the impacts of different types of anti-patterns on change- and fault-proneness using logistic regression model. The multivariate logistic regression is based on the following Equation (4).

$$\pi(X_1, X_2, ..., X_n) = \frac{e^{\beta_0+\beta_1.X_1+...+\beta_n.X_n}}{1+e^{\beta_0+\beta_1.X_1+...+\beta_n.X_n}}$$

(4)

Here,
• $X_i$ are the independent variables for the logistic regression model. In our case, $X_i$ represents the number of smells of type $S_i$ in a given source file and $S = \{S_1, S_2, ..., S_n\}$ is the set of the types of smells investigated.
• $\beta_i$ are the model coefficients, and
• $0 \leq \pi \leq 1$ is the value on the logistic regression curve representing the probability of bugs for a file with smells.

In our regression model, independent variables are the number of occurrences of each type of design smells. The dependent variable is the flag (buggy) representing the presence or absence of bugs. Thus, the dependent variable is dichotomous and assumes values either 0 (non-buggy) or 1 (buggy). For each system, we build a regression model and analyze the model coefficients and p-values for individual types of smells. Each row in our data set contains the values of the metrics (number of occurrences) for different smells, file size (LOC), number of previous bug-fix, code churn, and the bug status (1 or 0).

Each logistic regression model gives the log odds (regression coefficient estimate) of individual independent variables and their corresponding p-values for a particular system. The log odds represent the factors by which the odds of the dependent variable will change for a unit change in values of corresponding independent variables. When the logistic regression coefficient is positive ($\beta_i > 0$), unit increase of the value of the corresponding independent variable will increase the log odds of the dependant variable by $\beta_i$ assuming that other independent variables are either 0 or remain unchanged. For a negative regression coefficient ($\beta_i < 0$), on the other hand, the value of the log odds of the dependant variable will decrease by $\beta_i$ for unit increase in the value of the associated independent variable. Thus, the higher the positive log odds of an independent variable, the higher is the impact of that independent variable on bug-proneness. We rank the smells based on the model coefficients and the corresponding p-values. We select files that contain at least one smell of any type. For a given type of smell, if the model coefficients show higher log odds (LO) of bugs in the majority (in percentage) of the systems, we consider the smell to be related to fault-proneness. It is important to mention that we analyzed the data for correlation among smells and dropped one independent variable from each pair of highly correlated variables. This ensures a non-redundant set of variables for the logistic regression models. From a highly correlated pair, we keep the variable representing smell type with a comparatively higher overall prevalence in the studied systems. Because the following metrics are known to be related to fault-proneness [36, 46, 47], we add the file size, code churn, and the number of the previous occurrence of faults to our model, to control their effect. Here, (i) LOC: number of lines of code in the file at that specific release; (ii) Code Churn: the sum of lines added and removed in the file before that specific release; (iii) No. of Previous-Bugs: the number of faults fixing related to that file before the particular release $r$.

### 3.4.3 Topic Modeling to Identify Fault-inducing Activities

For RQ5, we are interested to investigate what kind of activities once performed in smelly files, are more likely to introduce bugs than other activities. We decided to analyze the commit messages that developers described when they performed a change that was captured by the SZZ algorithm as a fault-inducing commit. Having knowledge about those activities, developers could pay more attention to avoid introducing additional bugs. We collect all the fault-inducing commits messages related to smelly files as described earlier. We then classify those commit messages into different topics of activities based on the keywords mentioned by developers using a mix of automated and manual techniques. We decided to apply both topic modeling strategies and manual text analysis. Similar to previous work [48, 49], we used Latent Dirichlet Allocation (LDA) [50], a well-known topic modeling algorithm to analyze the text and extract a set of frequently co-occurring words (i.e., topics). We treat the
commit messages as a corpus of textual documents, that is used as a basis for topic modeling. Given a corpus of \( n \) documents \( f_1, ..., f_n \), topic modeling techniques automatically discover a set \( Z \) of topics, \( Z = z_1, ..., z_k \). The variable \( k \) presents the number of topics. It is an input that controls the granularity of the topics.

To generate the topic of activities introducing bugs, we combine both manual and automated approaches to build a categorization of risky activities. Based on developers’ commit messages, similar to previous work [51], we used MALLET\(^{23} \), a specific type of LDA implementation to generate a set of topics based on frequently co-occurring words. We removed stop words using MALLET stop words list (e.g., a, the, is, this, punctuation marks, numbers, and non-alphabetical characters). We also used Porter stemmer to reduce words to their root words (e.g., programmer became program) [52]. Since our objective is to study the activities that could introduce bugs once performed in smelly files, we limited our study to the smelly files in which a bug was introduced (Flag = 1). Thus, our data-set resulted in 2707 commit messages. We manually inspected the commit messages to estimate the number of possible topics for each system and also to assign a meaningful name to each topic. Once the number of possible topics was fixed, we used a python script that takes as input the list of all the commits messages in a CSV file and returns the list of commit messages with common keywords that could be used to build the topic. Two of the authors went through all the topics extracted for all the systems, and manually assigned meaningful names to each topic. The name of the topic was decided based on manual inspections of the commit messages and the keywords used to build that topic. We relied on the keywords generated by MALLET but also on frequent keywords captured during the manual analysis. We manually analyzed a total of 500 commits. To resolve the disagreements, two of the authors went through those commit messages and discussed the main topics of activities performed on those commits. Through those analysis, we aim to capture the possible types of activities that were described in the commit messages of the bug-inducing commits.

4 STUDY RESULTS

In this section, we report on the results of our study by addressing the five research questions defined in Section 3. We focus on the three key research objectives of our study. First, research questions RQ1 and RQ2 investigate the prevalence of the multi-language design smells in software systems. Then, research questions RQ3 and RQ4 evaluate the impacts of the design smells on the fault-proneness of JNI systems. Finally, RQ5 investigates fault-inducing activities. We present additional insights into the findings from the research questions later in Section 5.

4.1 RQ1: Do Multi-language design smells occur frequently in open source projects?

We use our detection approach to detect occurrences of multi-language design smells following the methodology discussed in Section 3. For each file, we compute the value of a variable \( Smelly_{i,r} \) that takes 1 if the file \( i \) contains at least one design smell in a specific release \( r \), and 0 otherwise. We also compute \( Native_{i,r} \) which takes 1 if the file \( i \) in a specific release \( r \) is native and 0 if not, following the rules discussed in Section 3.4.1. Since our tool is focusing on the combination of Java and C/C++, we compute for each release the percentage of files participating in at least one JNI smell out of the total of JNI files (files involved in Java and C/C++).

Table 5 summarises our results on the percentages of files with JNI smells in each of the studied systems. We report in this table the average number of JNI files participating in, at least one of the studied design smells for each system. The percentage for each release is available in our replication folder.\(^{21} \) Our results show that indeed, the JNI smells discussed in the literature are prevalent in the

\(^{21}\text{http://mallet.cs.umass.edu/} \)
nine studied open source projects with average occurrences from 10.18% in Jpype system to 61.36% in Zstd-jni. The percentage of files with smells differ from one project to another. We compute the average of the percentage of smells in all the systems. We find that on average, one-third (33.95%) of the JNI files in the studied systems contain multi-language design smells.

Besides analyzing in each system the percentage of files affected by each of the studied JNI smells, we also investigate their evolution over the releases. Figure 5 presents an overview of the evolution of the percentage of files participating in multi-language design smells in the releases of each system. All the details and data are available in the replication folder. The X-axis in Fig. 5 represents the releases analyzed. The Y-axis represents the percentage of files affected by at least one of the studied design smells, while the lines are related to each system. Results show that these percentages vary across releases in the nine systems with peaks as high as 69.04%. Some of these systems i.e., Realm and Jpype contain respectively 4.61% and 6.41% in the first releases, but the occurrences of smells increased over time to reach respectively 15.66% and 32.94%. Overall, the number of occurrences of smells are increasing over the releases. Although, in some cases such as in Rocksdb, the number of occurrences seems to decrease from one release to the next one, (from 43.78% to 31.76%). The fact that developers might not be aware of occurrences of such smells and the lack of tools for their detection might explain the observed prevalence. The observed decrease in the number of occurrences observed in certain cases could be the result of fault-fixing activities, features updates, or any other refactoring activities. In general, as one can see in Fig. 5, these decreases are temporary; the number of occurrences often increase again in the next releases. Overall, the proportions of files with smells are considerably high and the smells persist, thus allowing to reject $H_1$.

**Summary of findings (RQ1):** JNI smells discussed in the literature are prevalent and persistent in open source projects. The number of their occurrences even increases over the releases.

### 4.2 RQ2: Are some specific Multi-language design smells more frequent than others in open source projects?

Similar to RQ1, we use our approach from Section 3 to detect the occurrence of the 15 design smells in the nine subject systems. For each file and for each release, we defined a metric $\text{Smelly}_{i,r}$ which takes the value 1 if the release $r$ of the file contains the design smell type $i$ and 0 if it does not.
Are Multi-language Design Smells Fault-prone? An Empirical Study

Fig. 5. Evolution of Design Smells in the Releases of the 9 Systems

not contain that specific smell. We compute for each release the number of files participating in
that specific smell. Note that the same file may contain more than one smell.

Table 6 shows the distribution of the studied smells in the analyzed open source systems. We
calculate the percentage of files containing these smells and compute the average. Since our goal is
to investigate if some specific smells are more prevalent than others, we compute the percentage
of files containing that specific smell out of all the files containing smells. Our results show that
some smells are more prevalent than others, i.e., Unused parameter, Too much scattering, Too much
clustering, Unused Method Declaration, Not securing libraries, Excessive Inter-language communication.
In studied releases from Jpype, on average, 89.24% of the smelly files contain the smell Unused
parameter. In Java-smt, on average, 94.06% of the smelly files contain the smell Unused Parameters.
Our results also show that some smells discussed in the literature and developers’ blogs have a low
diffusion in the studied systems, i.e., Excessive objects, Not caching objects, Local reference abuse,
while the other smells are quite diffused in the analyzed systems. Conscrypt presents 79.60% occurrences
of the design smell Unused Parameters. As described in the commit messages in Conscrypt, this could
be explained by the usage of BoringSSL which has many unused parameters. Results presented in
Table 6 report a range of occurrences from 0% to 94.06%. Some specific types of smells seem to be
more frequent than others. On average Unused Parameters represents 57.36% of the existing smells,
followed by the smell Too Much Clustering with 20.91%. We also report in Table 7, the distribution
of smells normalized by the number of KLOC.

For each system, in addition to analyzing the percentage of files affected by each type of smell,
we also investigate the evolution of the smell over the releases. Figures 6, 7, 8, 9, 10, and 11
provide an overview of the evolution of smells respectively in Rocksdb, Java-cpp, Pljava, Realm,
Jpype, and Java-smt releases. The X-axis in these figures represents the releases analyzed. The
Y-axis represents the number of files in that specific system affected by that kind of design smells,
We studied the source code files containing some occurrences of the design smell while the lines are related to the different types of smells we studied. Depending on the system, some smells seem more prevalent than the others. In `Java-smt`, `Too Much Scattering`, and `Excessive Inter-language Communication` seem to be the predominant ones, while `Unused Parameters` is less frequent in this system. However, in general, for other systems including `Rocksdb` and `Realm`, `Unused Parameters` seems to be dominating. Results show that most of the smells generally persist within the project. The smells tend to persist in general or even increase from one release to another. Although, in some specific cases, for example, the design smell `Unused Parameters in Rocksdb`, presented a peak of 82 and decreased to 28 in the next release. However, the number of files containing this smell increased in the next releases and reached to 34 in the last release analyzed. We found that some method parameters were unused on `Rocksdb` between releases (5.11.2 and 5.14.3) of `Rocksdb` containing this smell increased in the next releases and reached to 34 in the last release analyzed. However, in general, for other systems including `Rocksdb` and `Realm`, `Unused Parameters` seems to be dominating. Results show that most of the smells generally persist within the project. The smells tend to persist in general or even increase from one release to another. Although, in some specific cases, for example, the design smell `Unused Parameters in Rocksdb`, presented a peak of 82 and decreased to 28 in the next release. However, the number of files containing this smell increased in the next releases and reached to 34 in the last release analyzed. We studied the source code files containing some occurrences of the design smell `unused parameters` between releases (5.11.2 and 5.14.3) of `Rocksdb` to understand the reasons behind the peak and the decrease. We found that some method parameters were unused on `Rocksdb` (5.11.2) and have been refactored during the next releases by removing occurrences of this smell and also due to project migration features. Another example of refactoring of the code smell `Unused Parameters` from one release to another was observed in `Conscript`, where they refactored `Unused Parameters` occurrences due to errors generated by those occurrences in the release 1.0.0.RC14 (“commit message: Our Android build rules generate errors for unused parameters. We cant enable the warnings in the external build rules because BoringSSL has many unused parameters”). From our results, we can clearly observe

Table 6. Percentage of JNI Files Participating in Design Smells in the Releases of the Studied Systems

<table>
<thead>
<tr>
<th>System</th>
<th>Smells</th>
<th>UP</th>
<th>UM</th>
<th>TMS</th>
<th>TMC</th>
<th>UM</th>
<th>ASR</th>
<th>EO</th>
<th>EILC</th>
<th>NHE</th>
<th>NCO</th>
<th>NSL</th>
<th>HCD</th>
<th>NURP</th>
<th>MMM</th>
<th>LRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conscript</td>
<td>Unused Parameter</td>
<td>79.6%</td>
<td>4.4%</td>
<td>0%</td>
<td>1.9%</td>
<td>0%</td>
<td>3.9%</td>
<td>0%</td>
<td>1.9%</td>
<td>5.9%</td>
<td>5.7%</td>
<td>3.8%</td>
<td>3.7%</td>
<td>1.7%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Realm</td>
<td>Unused Method Declaration</td>
<td>67.6%</td>
<td>3.06%</td>
<td>9.5%</td>
<td>14.8%</td>
<td>2.3%</td>
<td>4.3%</td>
<td>0%</td>
<td>12.5%</td>
<td>5.1%</td>
<td>2.1%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>2.9%</td>
<td>0%</td>
</tr>
<tr>
<td>Java-smt</td>
<td>Hard Coding Libraries</td>
<td>94.06%</td>
<td>2.9%</td>
<td>0%</td>
<td>2.9%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>2.9%</td>
<td>2.9%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Zstd-jni</td>
<td>Unused Method Implementation</td>
<td>10.46%</td>
<td>0.95%</td>
<td>13.98%</td>
<td>12.36%</td>
<td>3.47%</td>
<td>17.98%</td>
<td>0%</td>
<td>23.55%</td>
<td>21.45%</td>
<td>5.74%</td>
<td>3.47%</td>
<td>0%</td>
<td>2.25%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>Rocksdb</td>
<td>Assuming Safe Return Value</td>
<td>44.55%</td>
<td>5.48%</td>
<td>34.48%</td>
<td>23.47%</td>
<td>0%</td>
<td>0.67%</td>
<td>0%</td>
<td>14.35%</td>
<td>0.67%</td>
<td>0%</td>
<td>2.85%</td>
<td>0.95%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Java-cpp</td>
<td>Not Using Relative Path</td>
<td>2.53%</td>
<td>31.70%</td>
<td>74.19%</td>
<td>19.49%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>69.14%</td>
<td>0%</td>
<td>6.48%</td>
<td>2.51%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>Jytype</td>
<td>Unused Method Implementation</td>
<td>89.24%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>1.78%</td>
<td>0%</td>
<td>0.55%</td>
<td>1.78%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>8.25%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Pljava</td>
<td>Excessive Inter-lang Communication</td>
<td>64.45%</td>
<td>35.62%</td>
<td>31.02%</td>
<td>8.42%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>4.36%</td>
<td>2.04%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>2.04%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>VLC-android</td>
<td>Unused Parameters</td>
<td>63.67%</td>
<td>23.71%</td>
<td>24.74%</td>
<td>17.10%</td>
<td>7.34%</td>
<td>3.67%</td>
<td>0%</td>
<td>13.29%</td>
<td>3.67%</td>
<td>0%</td>
<td>3.92%</td>
<td>0%</td>
<td>0%</td>
<td>6.01%</td>
<td>0%</td>
</tr>
<tr>
<td>Median</td>
<td>Unused Parameter</td>
<td>64.43</td>
<td>4.4</td>
<td>13.98</td>
<td>12.36</td>
<td>0</td>
<td>1.78</td>
<td>0</td>
<td>12.58</td>
<td>2.04</td>
<td>0</td>
<td>2.96</td>
<td>0</td>
<td>0.79</td>
<td>0</td>
<td>0.1</td>
</tr>
<tr>
<td>Average</td>
<td>Unused Method Implementation</td>
<td>57.36</td>
<td>12.21</td>
<td>20.91</td>
<td>11.17</td>
<td>1.69</td>
<td>3.60</td>
<td>0.09</td>
<td>15.50</td>
<td>4.31</td>
<td>0.10</td>
<td>3.31</td>
<td>1.75</td>
<td>1.9</td>
<td>1.05</td>
<td></td>
</tr>
</tbody>
</table>

Acronyms: **Up:** Unused Parameter, **UM:** Unused Method Declaration, **TMS:** Too Much Scattering, **TMC:** Too Much Clustering, **UMI:** Unused Method Implementation, **ASR:** Assuming Safe Return Value, **EO:** Excessive Objects, **EILC:** Excessive Inter-lang Communication, **NHE:** Not Handling Exceptions, **NCO:** Not Caching Objects, **NSL:** Not Securing Libraries, **HCD:** Hard Coding Libraries, **NURP:** Not Using Relative Path, **MMM:** Memory Management Mismatch, **LRA:** Local References Abuse

Table 7. Number of Design Smells per KLOC in the Releases of the Studied Systems

<table>
<thead>
<tr>
<th>System</th>
<th>Smells</th>
<th>UP</th>
<th>UM</th>
<th>TMS</th>
<th>TMC</th>
<th>UM</th>
<th>ASR</th>
<th>EO</th>
<th>EILC</th>
<th>NHE</th>
<th>NCO</th>
<th>NSL</th>
<th>HCD</th>
<th>NURP</th>
<th>MMM</th>
<th>LRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conscript</td>
<td>Unused Parameter</td>
<td>6.091</td>
<td>7.089</td>
<td>0.0</td>
<td>0.022</td>
<td>0.0</td>
<td>0.07</td>
<td>0.0</td>
<td>0.07</td>
<td>0.16</td>
<td>0.0</td>
<td>0.15</td>
<td>0.0</td>
<td>0.25</td>
<td>0.02</td>
<td>0.13</td>
</tr>
<tr>
<td>Realm</td>
<td>Unused Method Declaration</td>
<td>1.81</td>
<td>0.086</td>
<td>0.075</td>
<td>0.097</td>
<td>0.024</td>
<td>0.04</td>
<td>0.0</td>
<td>2.36</td>
<td>0.12</td>
<td>0.0</td>
<td>0.01</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.00</td>
</tr>
<tr>
<td>Java-smt</td>
<td>Hard Coding Libraries</td>
<td>8.34</td>
<td>16.56</td>
<td>0.0</td>
<td>0.05</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.2</td>
<td>0.15</td>
<td>0.78</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Zstd-jni</td>
<td>Unused Method Implementation</td>
<td>2.0</td>
<td>0.22</td>
<td>0.11</td>
<td>0.23</td>
<td>1.09</td>
<td>1.08</td>
<td>0.0</td>
<td>1.60</td>
<td>1.15</td>
<td>0.0</td>
<td>0.078</td>
<td>0.0</td>
<td>0.50</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Rocksdb</td>
<td>Assuming Safe Return Value</td>
<td>1.32</td>
<td>0.18</td>
<td>0.34</td>
<td>0.23</td>
<td>0.0</td>
<td>0.02</td>
<td>0.0</td>
<td>5.72</td>
<td>0.03</td>
<td>0.01</td>
<td>0.081</td>
<td>0.019</td>
<td>0.02</td>
<td>0.02</td>
<td>0.0</td>
</tr>
<tr>
<td>Java-cpp</td>
<td>Not Using Relative Path</td>
<td>0.05</td>
<td>7.06</td>
<td>1.93</td>
<td>0.5</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>8.06</td>
<td>0.0</td>
<td>0.0</td>
<td>0.20</td>
<td>0.04</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Jytype</td>
<td>Unused Method Implementation</td>
<td>3.18</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.13</td>
<td>0.0</td>
<td>0.007</td>
<td>1.32</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>1.5</td>
<td>0.08</td>
<td></td>
</tr>
<tr>
<td>Pljava</td>
<td>Excessive Inter-lang Communication</td>
<td>5.10</td>
<td>1.7</td>
<td>0.41</td>
<td>0.06</td>
<td>0.02</td>
<td>0.0</td>
<td>0.0</td>
<td>0.04</td>
<td>0.11</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.14</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>VLC-android</td>
<td>Unused Parameters</td>
<td>4.18</td>
<td>4.75</td>
<td>0.46</td>
<td>0.4</td>
<td>0.55</td>
<td>0.1</td>
<td>0.010</td>
<td>5.47</td>
<td>0.1</td>
<td>0.07</td>
<td>0.37</td>
<td>0</td>
<td>1.25</td>
<td>0.0</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Acronyms: **Up:** Unused Parameter, **UM:** Unused Method Declaration, **TMS:** Too Much Scattering, **TMC:** Too Much Clustering, **UMI:** Unused Method Implementation, **ASR:** Assuming Safe Return Value, **EO:** Excessive Objects, **EILC:** Excessive Inter-lang Communication, **NHE:** Not Handling Exceptions, **NCO:** Not Caching Objects, **NSL:** Not Securing Libraries, **HCD:** Hard Coding Libraries, **NURP:** Not Using Relative Path, **MMM:** Memory Management Mismatch, **LRA:** Local References Abuse
that occurrences of JNI smells are not equally distributed. We conclude that the proportions of files with specific smells vary significantly between the different kinds of smells. We, therefore, reject hypothesis $H_2$.

![Fig. 6. Evolution of the Different Kinds of Smells in Rocksdb Releases](image)

**Summary of findings (RQ2):** Some JNI smells are more prevalent than others, e.g., *Unused Parameters*, *Too Much Scattering*, *Unused Method Declaration* while others are less prevalent, e.g., *Excessive Objects* and *Not Caching Objects*. Most of the smells persist with an increasing trend from one release to another in most of the systems.

### 4.3 RQ3: Are files with Multi-language design smells more fault-prone than files without?

Prior works show that design smells increase the fault-proneness of Java applications[25, 36]. Since JNI systems introduce other kinds of design smells and those smells are prevalent as observed in research questions RQ1 and RQ2, we are interested in studying the impacts of those design smells on the fault-proneness of JNI systems. For that, we applied Fisher’s exact test [44] to check whether...
the proportion of bugs varies between two samples (files with and without smells) as discussed in Section 3.4.2. The columns Smelly-buggy (SB), Buggy-NotSmelly (BNS), Smelly-NotBuggy (SNB), NotBuggy-NotSmelly (NBNS) in Tables 8 and 9 contain the values of the contingency tables for the Fisher’s exact test; each row corresponding to a single release. The numbers reported in the cells of these columns are the total number of JNI source code files (for the specific release) with or without smells and with or without bugs, depending on the column. More specifically, these columns present respectively: the total number of source code files with smells and are buggy (SB), the total number of source code files containing bugs without occurrences of smells (BNS), smelly source code files that are not buggy (SNB), and source code files that do not present any occurrences of smells or bugs (NBNS). The value of the odds ratio (OR) greater than 1 from Fisher’s exact test indicates that files with design smells have higher odds of being buggy compared to files without design smells. The values $OR < 1$ indicate that files with design smells have lower odds of having faults, while $OR = 1$ refers to no impact of design smells on fault-proneness of the source files. The $p$-value shows the probability of observing the odds ratio by chance, and thus lower values ($< 0.05$) of $p$-value confirm the significance of the impacts of design smells on fault-proneness. In addition to significant $p$-values, we examine the confidence intervals of the odds ratios. A confidence interval specifies the range where the true odds ratio lies in. A significant $p$-value value ($< 0.05$) of an odds ratio ($> 1.0$) with confidence interval not containing 1 confirms a true relationship between design smells and fault-proneness. We marked the $p$-values of such cases with (*) in Table 8 and Table 9.
Tables 8 and 9 report the results of applying Fisher’s exact test and present the values of odds ratios for the studied systems. Each row of those tables shows, for each system and each release, the odds for a file containing at least one type of design smells to be involved in a bug inducing change. In most of the analyzed releases, Fisher’s exact test indicates a significant difference of proportions between fault-prone JNI files with and without design smells. In some systems (e.g., Rocksdb, Javacpp, and Java-smt), odds ratios for specific releases are less than one, or the p-value is not statistically significant. However, in general, the values of odds ratios are high (in general greater than 2) in most cases. For Zstd-jni, we found odds ratios always higher than 13. Having this high odds could be explained by the large number of smells contained in that system, as described in Table 5, but also by the nature of smells existing in this system. The higher values of statistically significant odds ratios in most cases and the confidence intervals of those significant odds ratios being above the value 1 in some cases show that multi-language design smells are related to fault-proneness. However, this relationship varies with systems and further investigation is necessary to generalize.

From analyzing fault-fixing commit messages, we identified some commits reporting a refactoring for specific smells e.g., “removing unused parameter”, “implementing the handling of exception”. This could explain cases where Smelly-Buggy values decrease from one release to the other, while the overall number of occurrences of smells are in general increasing from one release to the other, as shown in Fig. 5. For example, in Rocksdb, Smelly-Buggy values are decreasing from one release to the other, while Smelly-NonBuggy is increasing. This could be explained by the nature of the smells and the refactoring applied. Since one file can contain more than one type of smell, the refactoring of some specific types of smells could decrease the risk of bugs while leaving the file still smelly. This suggests that some specific smells could be more correlated with bugs than others. These hypotheses motivate us to further investigate the relationship between specific types of smells and
fault-proneness (RQ4) and also the activities that once performed in smelly files could lead to bugs (RQ5).

We, therefore, conclude that, in most cases, there is a relation between multi-language design smells and fault-proneness in the context of JNI systems: a greater proportion of JNI files participating in design smells experienced bugs compared to other classes. We therefore reject $H_3$. The rejection of $H_3$ and the statistically significant odds ratios provide a posteriori concrete evidence of the impact of multi-language design smells on fault-proneness in the context of JNI files.

Fig. 9. Evolution of the Different Kinds of Smells in Realm Releases

Fig. 10. Evolution of the Different Kinds of Smells in Jpype Releases
Summary of findings (RQ3): Our results suggest that files with occurrences of the studied smells are more likely to be associated with faults than files without these smells and this relationship is statistically significant in most cases.
<table>
<thead>
<tr>
<th>System</th>
<th>Releases</th>
<th>SB</th>
<th>BNS</th>
<th>SNB</th>
<th>NBNS</th>
<th>Odds Ratios</th>
<th>p-values</th>
<th>Confidence Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rocksdb</td>
<td>rocksdb-5.0.2</td>
<td>82</td>
<td>85</td>
<td>17</td>
<td>108</td>
<td>6.1287</td>
<td>&lt; 0.01*</td>
<td>(1.2184, 2.4076)</td>
</tr>
<tr>
<td></td>
<td>rocksdb-5.4.6</td>
<td>89</td>
<td>90</td>
<td>23</td>
<td>98</td>
<td>4.2135</td>
<td>&lt; 0.01</td>
<td>(0.8979, 1.9787)</td>
</tr>
<tr>
<td></td>
<td>rocksdb-5.6.2</td>
<td>90</td>
<td>80</td>
<td>24</td>
<td>107</td>
<td>5.0156</td>
<td>&lt; 0.01*</td>
<td>(1.0771, 2.1480)</td>
</tr>
<tr>
<td></td>
<td>rocksdb-5.9.2</td>
<td>97</td>
<td>84</td>
<td>30</td>
<td>101</td>
<td>3.8876</td>
<td>&lt; 0.01</td>
<td>(0.8563, 1.8592)</td>
</tr>
<tr>
<td></td>
<td>rocksdb-5.11.2</td>
<td>99</td>
<td>86</td>
<td>42</td>
<td>95</td>
<td>5.0100</td>
<td>&lt; 0.01*</td>
<td>(1.0765, 2.1482)</td>
</tr>
<tr>
<td></td>
<td>rocksdb-5.14.3</td>
<td>97</td>
<td>84</td>
<td>30</td>
<td>101</td>
<td>3.8872</td>
<td>&lt; 0.01</td>
<td>(0.8563, 1.8592)</td>
</tr>
<tr>
<td></td>
<td>rocksdb-5.17.2</td>
<td>97</td>
<td>84</td>
<td>30</td>
<td>101</td>
<td>3.8872</td>
<td>&lt; 0.01</td>
<td>(0.8563, 1.8592)</td>
</tr>
<tr>
<td></td>
<td>rocksdb-5.18.3</td>
<td>97</td>
<td>84</td>
<td>30</td>
<td>101</td>
<td>3.8872</td>
<td>&lt; 0.01</td>
<td>(0.8563, 1.8592)</td>
</tr>
<tr>
<td></td>
<td>rocksdb-6.1.1</td>
<td>97</td>
<td>84</td>
<td>30</td>
<td>101</td>
<td>3.8872</td>
<td>&lt; 0.01</td>
<td>(0.8563, 1.8592)</td>
</tr>
<tr>
<td></td>
<td>rocksdb-latest release</td>
<td>88</td>
<td>88</td>
<td>40</td>
<td>100</td>
<td>3.8872</td>
<td>&lt; 0.01</td>
<td>(0.8563, 1.8592)</td>
</tr>
<tr>
<td>Pljava</td>
<td>pljava-1_4_3</td>
<td>0</td>
<td>36</td>
<td>0</td>
<td>100</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>pljava-rel1_5_stable</td>
<td>33</td>
<td>38</td>
<td>13</td>
<td>78</td>
<td>5.2105</td>
<td>&lt; 0.01</td>
<td>(0.9008, 2.4005)</td>
</tr>
<tr>
<td></td>
<td>pljava-1_5_0</td>
<td>33</td>
<td>37</td>
<td>13</td>
<td>79</td>
<td>5.4199</td>
<td>&lt; 0.01*</td>
<td>(0.9388, 2.4413)</td>
</tr>
<tr>
<td></td>
<td>pljava-1_5_1b1</td>
<td>32</td>
<td>38</td>
<td>14</td>
<td>78</td>
<td>5.7489</td>
<td>&lt; 0.01*</td>
<td>(0.8077, 2.2839)</td>
</tr>
<tr>
<td></td>
<td>pljava-1_5_1b2</td>
<td>39</td>
<td>36</td>
<td>14</td>
<td>76</td>
<td>5.8809</td>
<td>&lt; 0.01*</td>
<td>(1.0436, 2.4998)</td>
</tr>
<tr>
<td></td>
<td>pljava-1_5_2</td>
<td>38</td>
<td>34</td>
<td>15</td>
<td>78</td>
<td>5.8117</td>
<td>&lt; 0.01*</td>
<td>(1.0392, 2.4806)</td>
</tr>
<tr>
<td></td>
<td>pljava-latest release</td>
<td>39</td>
<td>35</td>
<td>16</td>
<td>76</td>
<td>5.2928</td>
<td>&lt; 0.01</td>
<td>(0.9600, 2.3727)</td>
</tr>
<tr>
<td>Realm</td>
<td>realm-java-0.90.0</td>
<td>21</td>
<td>89</td>
<td>2</td>
<td>365</td>
<td>43.0617</td>
<td>&lt; 0.01*</td>
<td>(2.2938, 5.2315)</td>
</tr>
<tr>
<td></td>
<td>realm-java-1.2.0</td>
<td>20</td>
<td>169</td>
<td>2</td>
<td>285</td>
<td>16.8672</td>
<td>&lt; 0.01*</td>
<td>(1.3592, 4.2912)</td>
</tr>
<tr>
<td></td>
<td>realm-java-2.3.2</td>
<td>33</td>
<td>177</td>
<td>3</td>
<td>269</td>
<td>16.7317</td>
<td>&lt; 0.01*</td>
<td>(1.6194, 4.0135)</td>
</tr>
<tr>
<td></td>
<td>realm-java-3.7.2</td>
<td>43</td>
<td>165</td>
<td>8</td>
<td>271</td>
<td>8.8280</td>
<td>&lt; 0.01*</td>
<td>(1.3988, 2.9570)</td>
</tr>
<tr>
<td></td>
<td>realm-java-4.4.0</td>
<td>48</td>
<td>166</td>
<td>18</td>
<td>262</td>
<td>4.2088</td>
<td>&lt; 0.01</td>
<td>(0.8616, 2.0127)</td>
</tr>
<tr>
<td></td>
<td>realm-java-5.4.0</td>
<td>50</td>
<td>165</td>
<td>21</td>
<td>261</td>
<td>3.7662</td>
<td>&lt; 0.01</td>
<td>(0.7804, 1.8718)</td>
</tr>
<tr>
<td></td>
<td>realm-java-5.7.1</td>
<td>52</td>
<td>164</td>
<td>22</td>
<td>260</td>
<td>3.7472</td>
<td>&lt; 0.01</td>
<td>(0.7856, 1.8565)</td>
</tr>
<tr>
<td></td>
<td>realm-java-5.9.0</td>
<td>54</td>
<td>161</td>
<td>23</td>
<td>260</td>
<td>3.7915</td>
<td>&lt; 0.01</td>
<td>(0.8066, 1.8589)</td>
</tr>
<tr>
<td></td>
<td>realm-java-5.11.0</td>
<td>54</td>
<td>161</td>
<td>23</td>
<td>260</td>
<td>3.7915</td>
<td>&lt; 0.01</td>
<td>(0.7668, 1.8059)</td>
</tr>
<tr>
<td></td>
<td>realm-java-5.15.0</td>
<td>54</td>
<td>162</td>
<td>24</td>
<td>259</td>
<td>3.7915</td>
<td>&lt; 0.01</td>
<td>(0.7668, 1.8059)</td>
</tr>
<tr>
<td>VLC-android</td>
<td>vlc-android-3.0.92</td>
<td>19</td>
<td>23</td>
<td>8</td>
<td>40</td>
<td>4.1304</td>
<td>&lt; 0.01</td>
<td>(0.4460, 2.3907)</td>
</tr>
<tr>
<td></td>
<td>vlc-android-3.1.6</td>
<td>22</td>
<td>22</td>
<td>6</td>
<td>40</td>
<td>6.6666</td>
<td>&lt; 0.01</td>
<td>(0.8552, 2.3930)</td>
</tr>
<tr>
<td></td>
<td>vlc-android-3.1.0</td>
<td>22</td>
<td>22</td>
<td>6</td>
<td>40</td>
<td>6.6666</td>
<td>&lt; 0.01</td>
<td>(0.8552, 2.3930)</td>
</tr>
<tr>
<td></td>
<td>vlc-android-3.0.13</td>
<td>18</td>
<td>24</td>
<td>7</td>
<td>41</td>
<td>4.3928</td>
<td>&lt; 0.01</td>
<td>(0.4720, 2.4879)</td>
</tr>
<tr>
<td></td>
<td>vlc-android-latest release</td>
<td>21</td>
<td>23</td>
<td>13</td>
<td>33</td>
<td>2.3177</td>
<td>0.081</td>
<td>(-0.0322, 1.7134)</td>
</tr>
<tr>
<td></td>
<td>vlc-android-3.0.11</td>
<td>19</td>
<td>24</td>
<td>6</td>
<td>41</td>
<td>5.4097</td>
<td>&lt; 0.01</td>
<td>(0.6411, 2.7352)</td>
</tr>
<tr>
<td></td>
<td>vlc-android-3.0.0</td>
<td>19</td>
<td>22</td>
<td>6</td>
<td>43</td>
<td>6.1893</td>
<td>&lt; 0.01</td>
<td>(0.7709, 2.8747)</td>
</tr>
<tr>
<td></td>
<td>vlc-android-3.0.96</td>
<td>19</td>
<td>23</td>
<td>8</td>
<td>40</td>
<td>4.1304</td>
<td>&lt; 0.01</td>
<td>(0.4460, 2.3907)</td>
</tr>
<tr>
<td></td>
<td>vlc-android-3.1.2</td>
<td>22</td>
<td>22</td>
<td>6</td>
<td>40</td>
<td>6.6666</td>
<td>&lt; 0.01</td>
<td>(0.8552, 2.3930)</td>
</tr>
</tbody>
</table>

For Pljava, Realm, VLC-android, Jpype:

* = significant p-values for odd ratios with confidence intervals not containing 1
Table 9. Fisher’s Exact Test Results for the Fault-proneness of Files with and without Design Smells (2)

<table>
<thead>
<tr>
<th>System</th>
<th>Releases</th>
<th>SB</th>
<th>BNS</th>
<th>SNB</th>
<th>NBNS</th>
<th>Odds Ratios</th>
<th>p-values</th>
<th>Confidence Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Javacpp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>javacpp-0.5</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>5</td>
<td>-</td>
<td>1.0</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>javacpp-0.9</td>
<td>10</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2.5</td>
<td>0.612</td>
<td>(-1.0599, 2.8926)</td>
<td></td>
</tr>
<tr>
<td>javacpp-1.1</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>4</td>
<td>-</td>
<td>1.0</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>javacpp-1.2</td>
<td>9</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>0.72</td>
<td>1.0</td>
<td>(-2.2994, 1.6424)</td>
<td></td>
</tr>
<tr>
<td>javacpp-1.2.1</td>
<td>12</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>2.4</td>
<td>0.574</td>
<td>(-1.3449, 3.0958)</td>
<td></td>
</tr>
<tr>
<td>javacpp-1.2.7</td>
<td>7</td>
<td>4</td>
<td>7</td>
<td>3</td>
<td>0.75</td>
<td>1.0</td>
<td>(-2.1148, 1.5395)</td>
<td></td>
</tr>
<tr>
<td>javacpp-1.3</td>
<td>10</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>15.0</td>
<td>&lt; 0.05</td>
<td>(0.2942, 5.1218)</td>
<td></td>
</tr>
<tr>
<td>javacpp-1.3.2</td>
<td>11</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>0.6111</td>
<td>1.0</td>
<td>(-2.9646, 1.9797)</td>
<td></td>
</tr>
<tr>
<td>javacpp-1.4</td>
<td>12</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>1.2</td>
<td>1.0</td>
<td>(-1.8101, 1.7477)</td>
<td></td>
</tr>
<tr>
<td>javacpp-1.4.2</td>
<td>14</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3.5</td>
<td>0.306</td>
<td>(-0.6953, 3.2011)</td>
<td></td>
</tr>
<tr>
<td>javacpp-1.4.4</td>
<td>11</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>2.75</td>
<td>0.378</td>
<td>(-0.9147, 2.9397)</td>
<td></td>
</tr>
<tr>
<td>javacpp-1.5</td>
<td>14</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>0.56</td>
<td>1.0</td>
<td>(-2.9573, 1.7977)</td>
<td></td>
</tr>
<tr>
<td>javacpp-1.5.1-1</td>
<td>10</td>
<td>4</td>
<td>9</td>
<td>2</td>
<td>0.5555</td>
<td>0.660</td>
<td>(-2.5093, 1.3337)</td>
<td></td>
</tr>
<tr>
<td>Zstd-jni</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>zstd-jni-0.4.4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>21</td>
<td>-</td>
<td>&lt; 0.01</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>zstd-jni-1.3.0-1</td>
<td>13</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>-</td>
<td>&lt; 0.01</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>zstd-jni-1.3.2-2</td>
<td>15</td>
<td>2</td>
<td>7</td>
<td>13</td>
<td>13.9285</td>
<td>&lt; 0.01</td>
<td>(0.8958, 4.3721)</td>
<td></td>
</tr>
<tr>
<td>zstd-jni-1.3.3-1</td>
<td>16</td>
<td>2</td>
<td>7</td>
<td>13</td>
<td>14.8571</td>
<td>&lt; 0.01</td>
<td>(0.9649, 4.4320)</td>
<td></td>
</tr>
<tr>
<td>zstd-jni-1.3.4-1</td>
<td>20</td>
<td>1</td>
<td>8</td>
<td>12</td>
<td>30.0</td>
<td>&lt; 0.01*</td>
<td>(1.2025, 5.5998)</td>
<td></td>
</tr>
<tr>
<td>zstd-jni-1.3.4-4</td>
<td>20</td>
<td>1</td>
<td>8</td>
<td>12</td>
<td>30.0</td>
<td>&lt; 0.01*</td>
<td>(1.2025, 5.5998)</td>
<td></td>
</tr>
<tr>
<td>zstd-jni-1.3.5-3</td>
<td>20</td>
<td>1</td>
<td>8</td>
<td>12</td>
<td>30.0</td>
<td>&lt; 0.01*</td>
<td>(1.2026, 5.5998)</td>
<td></td>
</tr>
<tr>
<td>zstd-jni-1.3.7</td>
<td>20</td>
<td>1</td>
<td>8</td>
<td>12</td>
<td>30.0</td>
<td>&lt; 0.01*</td>
<td>(1.2026, 5.5998)</td>
<td></td>
</tr>
<tr>
<td>zstd-jni-1.3.8-1</td>
<td>20</td>
<td>1</td>
<td>8</td>
<td>12</td>
<td>30.0</td>
<td>&lt; 0.01*</td>
<td>(1.2026, 5.5998)</td>
<td></td>
</tr>
<tr>
<td>zstd-jni-1.4.0-1</td>
<td>22</td>
<td>1</td>
<td>7</td>
<td>12</td>
<td>37.7142</td>
<td>&lt; 0.01*</td>
<td>(1.4198, 5.8403)</td>
<td></td>
</tr>
<tr>
<td>zstd-jni-latest release</td>
<td>22</td>
<td>1</td>
<td>7</td>
<td>12</td>
<td>37.7142</td>
<td>&lt; 0.01*</td>
<td>(1.4198, 5.8403)</td>
<td></td>
</tr>
<tr>
<td>Conscrypt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>conscrypt-1.1.1</td>
<td>42</td>
<td>52</td>
<td>12</td>
<td>46</td>
<td>3.0961</td>
<td>&lt; 0.01</td>
<td>(0.3758, 1.8844)</td>
<td></td>
</tr>
<tr>
<td>conscrypt-1.0.0.RC14</td>
<td>4</td>
<td>64</td>
<td>0</td>
<td>54</td>
<td>-</td>
<td>0.128</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>conscrypt-1.0.1</td>
<td>38</td>
<td>55</td>
<td>11</td>
<td>43</td>
<td>2.7008</td>
<td>&lt; 0.02</td>
<td>(0.2123, 1.7743)</td>
<td></td>
</tr>
<tr>
<td>conscrypt-2.1.0</td>
<td>47</td>
<td>53</td>
<td>17</td>
<td>42</td>
<td>2.1908</td>
<td>&lt; 0.05</td>
<td>(0.0975, 1.4711)</td>
<td></td>
</tr>
<tr>
<td>conscrypt-1.0.2</td>
<td>38</td>
<td>55</td>
<td>11</td>
<td>43</td>
<td>2.7008</td>
<td>&lt; 0.02</td>
<td>(0.2123, 1.7742)</td>
<td></td>
</tr>
<tr>
<td>conscrypt-1.4.2</td>
<td>6</td>
<td>0</td>
<td>55</td>
<td>97</td>
<td>-</td>
<td>&lt; 0.01</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>conscrypt-1.2.0</td>
<td>45</td>
<td>52</td>
<td>15</td>
<td>46</td>
<td>2.6538</td>
<td>&lt; 0.01</td>
<td>(0.2697, 1.6823)</td>
<td></td>
</tr>
<tr>
<td>conscrypt-1.0.0.RC11</td>
<td>37</td>
<td>55</td>
<td>11</td>
<td>43</td>
<td>2.6297</td>
<td>&lt; 0.02</td>
<td>(0.1844, 1.7493)</td>
<td></td>
</tr>
<tr>
<td>conscrypt-1.0.0.RC2</td>
<td>23</td>
<td>20</td>
<td>6</td>
<td>90</td>
<td>17.25</td>
<td>&lt; 0.01*</td>
<td>(1.8270, 3.8686)</td>
<td></td>
</tr>
<tr>
<td>conscrypt-1.0.0.RC8</td>
<td>26</td>
<td>59</td>
<td>11</td>
<td>46</td>
<td>1.8428</td>
<td>0.172</td>
<td>(-0.1922, 1.4148)</td>
<td></td>
</tr>
<tr>
<td>Java-smt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java-smt-0.60</td>
<td>0</td>
<td>23</td>
<td>0</td>
<td>7</td>
<td>-</td>
<td>1.0</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>java-smt-1.0.1</td>
<td>21</td>
<td>20</td>
<td>5</td>
<td>9</td>
<td>1.89</td>
<td>0.3667</td>
<td>(-0.6165, 1.8896)</td>
<td></td>
</tr>
<tr>
<td>java-smt-2.0.0-alpha</td>
<td>22</td>
<td>16</td>
<td>11</td>
<td>12</td>
<td>1.5</td>
<td>0.5966</td>
<td>(-0.6357, 1.4467)</td>
<td></td>
</tr>
<tr>
<td>java-smt-2.2.0</td>
<td>30</td>
<td>19</td>
<td>9</td>
<td>10</td>
<td>1.7543</td>
<td>0.4132</td>
<td>(-0.5061, 1.6304)</td>
<td></td>
</tr>
<tr>
<td>java-smt-3.0.0</td>
<td>19</td>
<td>17</td>
<td>22</td>
<td>12</td>
<td>0.6096</td>
<td>0.3414</td>
<td>(-1.4556, 0.4658)</td>
<td></td>
</tr>
</tbody>
</table>

* = significant p-values for odd ratios with confidence intervals not containing 1
<table>
<thead>
<tr>
<th>Design Smells</th>
<th>Conscrypt</th>
<th>Java-smt</th>
<th>Java-pp</th>
<th>Jpype</th>
<th>Pjave</th>
<th>Realm</th>
<th>Rocksdb</th>
<th>VLC-android</th>
<th>Zstd-jni</th>
</tr>
</thead>
<tbody>
<tr>
<td>Excessive Inter-language Communication</td>
<td>-9.538e+15</td>
<td>NA</td>
<td>-1.765e+14</td>
<td>-5.380e+01</td>
<td>-4.649e+01</td>
<td>4.495</td>
<td>1.640e+01</td>
<td>4</td>
<td>-7.653e+13</td>
</tr>
<tr>
<td>Too Much Clustering</td>
<td>3.280e+15</td>
<td>1</td>
<td>-6.864e+03</td>
<td>3.280e+15</td>
<td>1</td>
<td>NA</td>
<td>-5.488e+02</td>
<td>-9.514e+01</td>
<td>8.593e+02</td>
</tr>
<tr>
<td>Too Much Scattering</td>
<td>2.304e+14</td>
<td>4</td>
<td>NA</td>
<td>2.304e+14</td>
<td>4</td>
<td>NA</td>
<td>1.492e+02</td>
<td>1</td>
<td>9.310e+01</td>
</tr>
<tr>
<td>Unused Method Declaration</td>
<td>-6.275e+13</td>
<td>1</td>
<td>9.799e+13</td>
<td>6</td>
<td>NA</td>
<td>4.883e+01</td>
<td>2</td>
<td>-6.092e+01</td>
<td>-6.785e+01</td>
</tr>
<tr>
<td>Unused Method Implementation</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>-3.446e+02</td>
<td>-2.288e+02</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Unused Parameter</td>
<td>8.145e+13</td>
<td>3</td>
<td>8.497</td>
<td>2</td>
<td>7.421e+14</td>
<td>2</td>
<td>-5.527e-02</td>
<td>-7.098e-02</td>
<td>4.828</td>
</tr>
<tr>
<td>Excessive Objects</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Not Handling Exceptions</td>
<td>2.089e+14</td>
<td>2</td>
<td>NA</td>
<td>NA</td>
<td>1.993e+01</td>
<td>1</td>
<td>-1.999e+02</td>
<td>-9.445e+01</td>
<td>-4.270e+01</td>
</tr>
<tr>
<td>Not Caching Objects</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Not Securing Libraries</td>
<td>-2.915e+13</td>
<td>3</td>
<td>3.308e+14</td>
<td>3</td>
<td>NA</td>
<td>NA</td>
<td>-9.201e+01</td>
<td>-2.498e+03</td>
<td>-8.185e+14</td>
</tr>
<tr>
<td>Hard Coding Libraries</td>
<td>-8.665e-01</td>
<td>1.235e+14</td>
<td>5</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>4.936e+03</td>
<td>1</td>
</tr>
<tr>
<td>Memory Management Mismatch</td>
<td>-1.180e+15</td>
<td>NA</td>
<td>NA</td>
<td>1.162e-01</td>
<td>2</td>
<td>NA</td>
<td>NA</td>
<td>-1.393e+03</td>
<td>NA</td>
</tr>
<tr>
<td>Local References Abuse</td>
<td>-1.301e+15</td>
<td>NA</td>
<td>NA</td>
<td>-2.462e+02</td>
<td>NA</td>
<td>-1.891e+02</td>
<td>-3.474e+04</td>
<td>-8.620e+15</td>
<td>NA</td>
</tr>
<tr>
<td>LOC</td>
<td>-8.459e+11</td>
<td>-4.280e-03</td>
<td>1.159e+11</td>
<td>8.468e-05</td>
<td>5.328e-04</td>
<td>2.488e-03</td>
<td>8.713e-01</td>
<td>1.091e+11</td>
<td>5.331e+10</td>
</tr>
<tr>
<td>Previous bug-fix</td>
<td>6.363e+14</td>
<td>1.864e+02</td>
<td>1.293e+15</td>
<td>5.728e+01</td>
<td>3.018e+02</td>
<td>9.480e+01</td>
<td>1.488e+03</td>
<td>7.873e+14</td>
<td>7.437e+14</td>
</tr>
<tr>
<td>Null deviance</td>
<td>2034.3</td>
<td>5.201e+02</td>
<td>349.15</td>
<td>1.0902e+03</td>
<td>2.3975e+03</td>
<td>6.6378e+03</td>
<td>4048.24</td>
<td>1245.10</td>
<td>589.62</td>
</tr>
<tr>
<td>Residual deviance</td>
<td>1513.8</td>
<td>2.725e-09</td>
<td>2595.14</td>
<td>3.562e+10</td>
<td>4.8079e-10</td>
<td>8.8774e-10</td>
<td>8.5475</td>
<td>360.44</td>
<td>2595.14</td>
</tr>
<tr>
<td>AIC</td>
<td>1535.8</td>
<td>16</td>
<td>2615.1</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>34.55</td>
<td>384.44</td>
<td>1107.3</td>
</tr>
</tbody>
</table>
4.4 RQ4: Are some specific Multi-language design smells more fault-prone than others?

Findings from RQ3 suggest that source code files with smells in JNI systems are often more prone to faults than files without smells. Although these findings give a general impression of the impacts of smells on the fault-proneness of JNI systems, it is important to know which smell(s) are more related to faults. When we are able to identify some specific smells to be more related to faults, we can prioritize those smells during the maintenance of the JNI systems.

As presented in our methodology described in Section 3.4.2, we apply multivariate logistic regression to examine whether some types of design smells are more related to fault-proneness. In our logistic regression models, independent variables are the number of occurrences of each type of design smells. The dependent variable is a dichotomous flag (buggy) that assumes values either 0 (non-buggy) or 1 (buggy). For each system, we build a logistic regression model and analyze the model coefficients and p-values for individual types of smells. To address multicollinearity among the independent variables, we drop one of the variables from each highly correlated pair of variables from the models. From our analysis, we observed two pairs of smells highly correlated—(Not Handling Exceptions, Assuming Safe Return Value) and (Not Securing Libraries, Not Using Relative Path) with correlation (Spearman’s) coefficients of 0.91 and 0.60, respectively. We keep Not Handling Exceptions and Not Securing Libraries as they are more prevalent in the systems compared to the other smell in each correlated pair. Similarly, we drop the variable code churn from the model as we found it to be highly correlated (0.99) with the file size (LOC). We chose Spearman’s rank correlation as it is non-parametric and does not require data to be normally distributed.

We rank the independent variables based on the logistic regression model coefficients (log odds) and the corresponding p-values. Table 10 presents the model coefficients and their ranking for each system. The coefficients with significant p-values (<0.01) are presented in boldface. To evaluate the relationships of individual types of smells with bug-proneness, we summarize the data from Table 10 into Table 11 to identify the top five smell types that are more related to bugs. For each smell type, Table 11 presents the percentage of systems where the smell type has positive log odds, the number of times the smell type is in the top five in the ranking of positive log odds, and the number of systems where the log odds are statistically significant. For each smell, we consider only the system where we are able to calculate the model coefficients and thus we exclude the systems where we do not get the coefficient values due to singularities. For the smell Too Much Clustering for example, in Table 10, for eight (8) out of nine (9) systems (i.e., except Jpype) we have values for model coefficients. Out of these eight systems, for five (5) systems (Conscrypt, Javacpp, Rocksdb, VLC-android, and Zstd-jni i.e., 5/8 (62.5%) times) Too Much Clustering has positive values for log odds. All of these five times (systems) the log odds were ranked in the top five, having significant p-values in four systems (Conscrypt, Javacpp, VLC-android, and Zstd-jni). For all smell types in Table 10, we present such summary in Table 11. We then report the top five smell types (underlined in Table 11) based on the percentage of systems where the smell have positive log odds, number of times the positive log odds were in the top five ranking, and the number of systems in which the log odds have significant p-values respectively as shown in Table 11. For the control variables LOC and the number of previous bug-fix, we observed positive log odds in most of the systems. So, these coefficients for the control factors agree with the known impacts of these two variables on fault-proneness. We also observed negative log odds for the smells from the logistic regression models for the studied systems. The negative regression coefficients might be interpreted as an indication that the corresponding smells are negatively related to fault-proneness. However, this scenario varies across the studied systems.
Table 11. Fault-proneness of Different Types of Smells Based on Logistic Regression Analysis

<table>
<thead>
<tr>
<th>Smell Types</th>
<th>Number and Percentage of Systems</th>
<th>LO &gt; 0</th>
<th>LO in Top 5</th>
<th>(LO&gt;0 and p&lt;0.01)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Excessive Inter-language Communication</td>
<td>25%(2/8)</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Too Much Clustering</td>
<td>62.5%(5/8)</td>
<td>5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Too Much Scattering</td>
<td>100%(6/6)</td>
<td>6</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Unused Method Declaration</td>
<td>37.5%(3/8)</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Unused Method Implementation</td>
<td>25%(1/4)</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Unused Parameters</td>
<td>66.6%(6/9)</td>
<td>5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Not Handling Exceptions</td>
<td>42.8%(3/7)</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Not Securing Libraries</td>
<td>28.5%(2/7)</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Hard Coding Libraries</td>
<td>75%(3/4)</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Memory Management Mismatch</td>
<td>50%(2/4)</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Local References Abuse</td>
<td>0%(0/5)</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Excessive Objects</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td></td>
</tr>
<tr>
<td>Not Caching Objects</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td></td>
</tr>
</tbody>
</table>

LO = Log Odds of the corresponding smell from the logistic regression model.
NA = Corresponding Log odds are not available from the LR models due to singularities.

Table 12. Fault-proneness of Different Types of Smells Based on Logistic Regression Model for All Systems

<table>
<thead>
<tr>
<th>Smell Types</th>
<th>Log Odds</th>
<th>p-values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Excessive Inter-language Communication</td>
<td>2.985e-01</td>
<td>1.11E-07 (&lt;0.01)</td>
</tr>
<tr>
<td>Too Much Clustering</td>
<td>4.262e+00</td>
<td>0.000898 (&lt;0.01)</td>
</tr>
<tr>
<td>Too Much Scattering</td>
<td>8.359e+00</td>
<td>1.82e-15 (&lt;0.01)</td>
</tr>
<tr>
<td>Unused Method Declaration</td>
<td>9.078e-01</td>
<td>&lt; 2e-16 (&lt;0.01)</td>
</tr>
<tr>
<td>Unused Method Implementation</td>
<td>-1.255e+01</td>
<td>0.894915</td>
</tr>
<tr>
<td>Unused Parameters</td>
<td>5.695e-01</td>
<td>&lt; 2e-16 (&lt;0.01)</td>
</tr>
<tr>
<td>Not Handling Exceptions</td>
<td>3.248e+00</td>
<td>0.000217 (&lt;0.01)</td>
</tr>
<tr>
<td>Not Securing Libraries</td>
<td>-4.469e-01</td>
<td>0.813090</td>
</tr>
<tr>
<td>Hard Coding Libraries</td>
<td>1.841e+00</td>
<td>0.546194</td>
</tr>
<tr>
<td>Memory Management Mismatch</td>
<td>-9.255e+00</td>
<td>0.998258</td>
</tr>
<tr>
<td>Local References Abuse</td>
<td>-1.172e+01</td>
<td>0.999968</td>
</tr>
<tr>
<td>Excessive Objects</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Not Caching Objects</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

NA = Corresponding Log odds are not available from the LR models due to singularities.

As shown in Table 10, the log odds of the independent variables vary across the systems. In four of the systems (Conscrypt, Javacpp, VLC-android, and Zstd-jni), we observe that the log odds for the smells are statistically significant (<0.01). These four systems reject the hypothesis $H_4$, meaning that different smells have different impacts on fault-proneness. However, we cannot generalize it to other systems to have a concrete conclusion. Thus, given the varying log odds of the smells from our regression models for individual systems, we conclude that the relationships between different types of multi-language smells and fault-proneness are system dependent.
Given that we have limited evidence to draw a firm conclusion on the strength of the relationships between different types of smells and fault-proneness, we focus on identifying smells that are relatively more related to faults based on the ranking of the values of the log odds and their significance. In Table 11, the smell type *Too Much Clustering* has positive log odds in 62.5% (5/8) of the systems. Each time, log odds were among the top 5 and was statistically significant in three systems. Similarly, *Too Much Scattering, Unused Parameters, Hard Coding Libraries* and *Memory Management Mismatch* are among the top five smells with positive log odds in 100%(6/6), 66.6%(6/9), 75%(3/4), and 50%(2/4) systems, respectively. These smells are likely to have a strong relation with fault-proneness. Besides, the smells *Not Handling Exceptions, Unused Method Declaration* and *Not Securing Libraries* have significant positive log odds for 2 (Conscrypt, Zstd-jni), 1 (Javacpp), and 1 (Javacpp) system(s), respectively; indicating some degree of relation with fault-proneness. *Excessive Inter-language Communication* and *Local References Abuse* which have no significant positive log odds are less likely to be associated with faults.

We also build a single logistic regression model for all the systems combined to evaluate how the findings from individual systems generalize. We presented the regression results for the smells in Table 12. We observed that smells *Excessive Inter-language Communication, Too Much Clustering, Too Much Scattering, Unused Method Declaration, Unused Parameters,* and *Not Handling Exceptions* have positive log odds with significant p-values (<0.01). This is an indication that these smells have statistically significant relationships with fault-proneness. This finding corroborates our findings from the analysis of individual systems for most cases. However, we did not observe significant relationships between multi-language smells and fault-proneness for the remaining smell types (in the models for all systems combined). Now, if we consider positive log odds with significant p-values in the logistic regression model for all systems combined (Table 12) and the percentage of positive log odds for regression models for individual systems (Table 11), we observe that the smell types *Too Much Clustering, Too Much Scattering, Unused Parameters, Not Handling Exceptions,* and *Hard Coding Libraries* are the most related to fault-proneness. However, this relationship varies with systems. One important point to note is the fact that smells suggested by our empirical results to be more related to fault-proneness constitute roughly over 80% of the smells in the studied systems (details in Table 6). This further shows that it is important to detect and remove these smells from the systems as soon as possible.

To study the relationships between multi-language smells and fault-proneness, we also investigate the correlation (Spearman’s) between the number of smells of individual types in a file and the number of bugs associated with the corresponding file. We observed that *Not Using Relative Path* (0.48), *Not Handling Exceptions* (0.32), *Excessive Inter-language Communication* (0.30), *Local References Abuse* (0.24), *Hard Coding Libraries* (0.19), and *Too Much Clustering* (0.18) are the top smells based on the correlation with faults, although most of these and the remaining correlations are weak. Also, we mentioned that *Not Using Relative Path* was dropped from our logistic regression models because of its high correlation with *Not Securing Libraries*. So, we cannot draw a firm conclusion on the impacts of smells on fault-proneness based on these correlation results.

To have better insights into the identified relationships between the different types of JNI smells and bugs and to understand the bug-smell contexts in the studied systems, we further manually investigated a random sample of commit messages associated with bugs. From analyzing these commit messages, we found some commit messages clearly suggesting that some specific smells are often related to bugs. For example, in the release 1.0.0.RC14 of *Conscrypt*, a commit message is clearly specifying errors related to the code smell *Unused Parameters* (“Our Android build rules generate errors for unused parameters. We cant enable the warnings in the external build rules because BoringSSL has many unused parameters”). The same goes for *Memory Management Mismatch* in *Realm*, a commit message was discussing errors related to memory management ”DeleteLocalRef
when the ref is created in loop (#3366) Add wrapper class for JNI local reference to delete the local ref after using it”. Another example from Conscrypt discussing bugs related to native memory management “This fixes a memory leak in NativeCrypto_i2d_PKCS7. It never frees derBytes”. The smell Not Handling Exceptions is also discussed as related to the bug 3482 in Realm (“Add cause to RealmMigrationNeededException (#3482)”). VLC-android also presents bugs related to the smell Not Handling Exceptions “rework exceptions throwing from jni”. Similarly, other commit messages were also describing bugs related to Unused Method Declaration. “There were a bunch of exceptions that are being thrown from JNI methods that aren’t currently declared”, and “Fix latent bug in unused method” present examples extracted respectively from Conscrypt and Pljava. Thus, we see that our identified smells are often related to bugs which highlight practical contexts and usability of our findings.

We also analyzed some quality attributes of our models such as Null deviance, Residual deviance, and Akaike’s Information Criterion (AIC) as presented in Table 10. We observe that there are larger differences between null deviance and the residual deviance for the models of all the systems indicating a good fit of the regression models. We observe lower values for AIC for most of the regression models indicating the simplicity of the models with comparatively higher values for Conscrypt, Java-clip, VLC-android, and Zstd-jni.

Summary of findings (RQ4): We conclude that although not always significant, there exists a relation between types of smells and the fault-proneness. The relationship is not consistent for all types of smell and across all the systems. Smell types Too much Scattering, Too Much Clustering, Unused Parameters, Hard Coding Libraries, and Not Handling Exceptions are observed to be more related to faults compared to other smells, and thus should be prioritized during maintenance.

4.5 RQ5: What are the activities that are more likely to introduce bugs in smelly files?
Since the risk of having bugs could differ from one activity to the other, we decided to investigate what kind of activities once performed in smelly files could increase the risk of bug occurrences. Having knowledge of risky activities developers and maintainers could reduce the risk of bugs in smelly files. To study the activities that could introduce bugs in smelly files, we collected the fault-inducing commits messages and performed a topic modeling, combining a mix of manual and automatic approaches as described in Section 3.

Table 13 lists 12 activities that are more likely to introduce bugs in smelly files. For each activity the table lists the systems from which the activity was extracted. For each activity, we also present examples of keywords used to build the topic for that activity. For example, the activity memory management, was extracted using a set of keywords including: buffer, memory, leak, flush, reference, local, memtable from Rocksdb, Realm, Conscrypt, and Jpype. “Add more numbers to float-conversion test, add new unit-test for float-conversion”, is an example of a commit message describing data conversion activity when the bug was introduced, extracted from Java-smt. Another example of commit message that introduced bugs extracted from Rocksdb: “Another change is to only use class BlockContents for compressed block, and narrow the class Block to only be used for uncompressed blocks, including blocks in compressed block cache”. This commit message is related to compression activities. From Zstd-jni, the following commit messages “expose faster API to allow re-using of dictionaries” refers to the usage of APIs. Those activities were extracted from commit messages of fault-inducing commits. Developers performed those activities in files containing occurrences of multi-language design smells when the bug was introduced.
Table 13. Activities Introducing Bugs in Smelly Files

<table>
<thead>
<tr>
<th>No</th>
<th>Activities</th>
<th>Example of Keywords</th>
<th>Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Compression tasks</td>
<td>compression, decode, memory, blocks, encode, compaction, streaming, frames, block, dictionary.</td>
<td>Rocksdb, Zstd-jni</td>
</tr>
<tr>
<td>2</td>
<td>Data conversion</td>
<td>parser, type, container, basic, declaration, write, invalid, convert, string, coverage.</td>
<td>Rocksdb, Realm, Pljava, Javacpp, Conscrypt, Java-smt, Zstd-jni</td>
</tr>
<tr>
<td>3</td>
<td>Memory management</td>
<td>buffer, messagesize, memory, leak, local, reference, flush, mentable, allocation, garbage.</td>
<td>Rocksdb, Realm, Conscrypt, Pljava, Javacpp</td>
</tr>
<tr>
<td>4</td>
<td>Restructuring the code</td>
<td>add, update, remove, code, reorder, native, move, public, improve, change.</td>
<td>Rocksdb, VLC-android, Conscrypt, Jpype, Pljava</td>
</tr>
<tr>
<td>5</td>
<td>Database management</td>
<td>stored, db, database, persistence, key, data, visible, size, file, timestamp.</td>
<td>Rocksdb, Pljava</td>
</tr>
<tr>
<td>6</td>
<td>API usage</td>
<td>external, library, api, include, expose, public, integrate, allow, streaming, wrapper.</td>
<td>VLC-android, Realm, Conscrypt, Rocksdb, Pljava, Zstd-jni, Java-smt</td>
</tr>
<tr>
<td>7</td>
<td>Feature migration</td>
<td>upgrade, support, migrate, integrate, create, legacy, simplify, add, format, update.</td>
<td>Realm, Javacpp, Rocksdb, Java-smt, Zstd-jni</td>
</tr>
<tr>
<td>8</td>
<td>Network management</td>
<td>sslsocket, encrypt, socket, nativessl, token, hostname, protocol, platform, sssession, activesession.</td>
<td>Conscrypt, Rocksdb</td>
</tr>
<tr>
<td>9</td>
<td>Exception management</td>
<td>occur, handle, check, exception, throw, return, fix, pointer, illegal, runtime.</td>
<td>Conscrypt, Javacpp, Jpype, Java-smt</td>
</tr>
<tr>
<td>10</td>
<td>Threads management</td>
<td>thread, pull, execution, reflect, client, transaction, monitor, notify, mutex, log.</td>
<td>Pljava, Rocksdb, Realm</td>
</tr>
<tr>
<td>11</td>
<td>Performance management</td>
<td>time, wrap, performance, execution-time, regression, cache, shared, resources, bundle, increase.</td>
<td>Zstd-jni, Rocksdb</td>
</tr>
<tr>
<td>12</td>
<td>Compiler management</td>
<td>compiler, resolve, failure, check, warnings, support, JNI_ABORT, error, illegal, dynamic.</td>
<td>Jpype, Rocksdb</td>
</tr>
</tbody>
</table>

From analyzing the commit messages and topics of activities, we found that activities related to data conversion, memory management, API usage, code restructuring, and exception management are the most common activities that could increase the risk of bugs when performed in smelly files. Activities related to the compiler management, threads management, and compression tasks could also induce bugs in smelly files.

To understand why these activities seem to be risky, we decided to investigate further these activities at the source code level. For example, zstd.java is a native class from Zstd-jni system. This class contains 1351 lines of code with 75 native methods and exhibits the smell Too Much Clustering. This class combines methods performing distinct responsibilities, i.e., compression, decompression, computation, data access, and utility methods. As per commit messages identified as introducing bugs, the developer was reordering the statics methods, adding JNI wrappers, and performing compression tasks “Reorder the static methods, All compression first then all decompression then the rest, inputs checking + utility methods”, “Add Java wrappers and C implementations of compress/decompress using direct ByteBuffer”. This class contains two types of smells, Too Much Clustering and Excessive Inter-language Communication. The nature of those two smells is by definition adding complexity to the code by making the readability of such classes hard. Thus, restructuring the code of a large native class could have increased the risk of introducing bugs. Indeed, applying changes on multi-language code could bring some confusion if the developer is not familiar with the components involved in the multi-language interaction. Similarly, activities
related to compression are declared in Java side and mainly implemented in the C/C++ side (jni_zstd.c). Developers should have knowledge of both implementations to correctly perform a change, especially that this class contains Excessive Inter-language Communication between Java and C/C++ when performing compression activities. Another example is illustrated by Listing 12. It presents a function extracted from the C file jni_zdict.c from Zstd-jni system. A developer was “adding support for legacy dictionary trainer” in this smelly function when the bug was introduced. However, in the context of JNI, it is important to always perform checks to ensure that the native execution was performed correctly. As described in Section 2.3, when checking JNI exceptions, we should add a return statement just after throwing the excepting to interrupt the execution flow and exit the method in case of errors. The ThrowNew() functions do not interrupt the control flow of the native method. In case an error occurred when retrieving the jclass, the exception will not be thrown in the JVM until the native method returns. Developers should be aware of how to implement the exception in the context of JNI systems to avoid introducing bugs related to mishandling JNI exceptions. Activities related to the conversion of types could also introduce bugs as expressed by a commit message extracted from javacpp; i.e., “Provide ‘BytePointer’ with value getters and setters for primitive types other than ‘byte’ to facilitate unaligned memory accesses”.

Another example of bugs related to the management of the memory is extracted from Pljava, c source code file JNICalls.c, “Eliminate threadlock ops in string conversion”. Both of those files exhibit the smell Memory Management Mismatch. Activities related to data and type conversion could increase the risk of bug because when converting types from Java to C/C++, the conversion will raise two categories of types; primitive types and reference types. Primitive types are simple to convert, we usually add j in front of the type e.g., int become jint, float become jfloat, etc. However, for the reference types i.e., Class, Object, String, developers should use the predefined method to correctly perform the conversion. However, it happens that they forget to release the memory after such conversion which could introduce additional bugs including memory leaks. Listing 13 presents an example extracted from Pljava as introducing bugs. In this example, the method GetObjectArrayElement is used to capture a Java array. However, the memory is not released after usage as done in Listing 12. From the above examples, we conclude that some specific types of activities are relatively more frequently associated with bugs, especially in the context of multi-language design smells. Developers should be cautious while performing those activities.

/*
...*/
jsize num_samples = (*env)->GetArrayLength(env, sampleSizes);
jint *sample_sizes_array = (*env)->GetIntArrayElements(env, sampleSizes, 0);
sizet *samples_sizes = malloc(sizeof(sizet) * num_samples);
if (!samples_sizes) {
    jclass eClass = (*env)->FindClass(env, "Ljava/lang/OutOfMemoryError;);
    (*env)->ThrowNew(env, eClass, "native heap");
}
for (int i = 0; i < num_samples; i++) {
    samples_sizes[i] = sample_sizes_array[i];
}
(*env)->ReleaseIntArrayElements(env, sampleSizes, sample_sizes_array, 0);

Listing 12. Example of Bug in Smelly Method 1/2
Listing 13. Example of Bug in Smelly Method 2/2

```java
/*
... */
jsize idx;
jboolean foundNull = JNI_FALSE;
BEGIN_JAVA
idx = (*env)->GetArrayLength(env, array);
while(--idx >= 0)
{
    if((*env)->GetObjectArrayElement(env, array, idx) != 0)
        continue;
    foundNull = JNI_TRUE;
    break;
}
END_JAVA
return foundNull;
```

Summary of findings (RQ5): Activities related to data conversion, memory management, code restructuring, API usage, and exception management are the most common activities that could increase the risk of bugs once performed in smelly files, and thus should be performed carefully.

5 DISCUSSION

This section discusses the results reported in Section 4.

5.1 Multi-language Design Smells

Detection of Smells. We used srcML parser due to its ability to provide a single xml file combining source code files written in more than one programming language. Languages supported in the current version of srcML include Java, C, C++, and C#.

However, this could be extended to include other programming languages [53]. The detection approach presents some limitations. The recall and precision vary depending on the type of design smells and mainly on the naming convention used to implement the JNI projects. For the smell Unused Method Declaration, we are missing some occurrences due to the syntax used in the C implementation that is not completely following the JNI naming convention (e.g., `Pljava jobject pljava_DualState_key`). For Local References Abuse, we are not considering situations in which predefined methods could be used to limit the impact of this design smell, i.e., `PushLocalFrame` and `PopLocalFrame`. These methods were excluded because by a manual validation when defining the smells, we found that those methods do not always prevent occurrences of the design smells and inclusion of those may result in false negatives. Our detection approach also presents some limitations in the detection of Not Using Relative Path, particularly in situations where the path could be retrieved from a variable or concatenation of

---

24https://www.srcml.org/about.html
25https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#PushLocalFrame
26https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html#PopLocalFrame
strings. However, this was not captured as a common practice in the analyzed systems. We refined our detection rules to favor the recall over precision, as was done for smells detection approaches for mono-language systems [35, 54]. However, by refining some rules as explained earlier for the smell Local References Abuse, and mainly due to some situations that are not coherent with the standard implementation of JNI code, we ended up having on average a better precision. The same goes for the smell Memory Management Mismatch. Indeed, we implemented a simple detection approach that could be applied to detect the smell following the definition and rule presented in this paper. Thus, this could not be generalized to all memory allocation issues. The detection approach relies on rules specific to the JNI usage. Thus, other native methods that could be implemented without considering JNI guidelines could lead to false positives and false negatives. To reduce threats to the validity of our work, we manually verified instances of smells reported by our detection approach on six open source projects along with our pilot project and measured the recall and precision of our detection approach as described in Section 3.

**Distribution of JNI Smells.** From our results we found that most of the studied smells specific to JNI systems are prevalent in the selected projects. Results from the studied systems reflect a range from 10.18% of smelly files in Jpype system to 61.36% of smelly files in Zstd-jni. On average, 33.95% of the JNI files in the studied systems contain multi-language design smells. Multi-language systems offer numerous benefits, but they also introduce additional challenges. Thus, it is expected to have new design smells specific to such systems due to their heterogeneity. The prevalence of multi-language smells in the selected projects highlights the need for empirical evaluation targeting the analysis of multi-language smells and also the study of their impact on software maintainability and reliability. We also analyzed the persistence of these smells. Our results show that overall the number of smells usually increases from one release to the other. Such systems usually involve several developers working in the same team and who might not have a good understanding of the architecture of the whole project. Thus, the number of smells may increase if no tools are available to detect those smells and-or to propose refactored solutions.

We observed situations in which the number of smells could decrease from one release to the next one. From investigating the commit message, we observed that some smells were refactored from one release to the other. Most of them due to the side effect of other refactoring activities, but also due to specific refactoring activities, e.g., removing Unused Parameters, unused methods, implementing the handling of native exceptions, etc. This suggests that some developers might be aware of the necessity to remove those smells. However, since no tools are available to automatically detect such occurrences, it is hard for a developer to manually identify all the occurrences. However, we plan in another study to investigate the developers’ perceptions and opinions about those smells as well as their impacts on software quality.

**Distribution of specific kinds of smells.** We investigated in RQ2, if some specific smells are more prevalent than others. We found that the smells are not equally distributed within the analyzed projects. We also investigated their evolution over the studied releases. Our results show that the studied smells either persist or even mostly increase in number from one release to another. We observed some cases in which there was a decrease from one release to the other, and where smells occurrences were intentionally removed (Rocksdb, Conscrypt) by refactoring. Those systems are emerging respectively from Facebook and Google. In Realm, we also observed the awareness of developers about the bad practice of not removing local references (commit message: “DeleteLocalRef when the ref is created in loop (#3366) Add wrapper class for JNI local reference to delete the local ref after using it”). This could explain the decrease of smells occurrences in some situations. However, since no automatic tool is available, it could be really hard to identify all the occurrences, especially
since such systems usually include different teams, which could explain the increase and decrease of multi-language design smells occurrences.

Our results show that *Unused Parameters* is one of the most frequent smells in the analyzed projects. This could be explained by the nature of the smell. This smell is defined when an unnecessary variable is passed as a parameter from one language to another. Since multi-language systems are emerging from the concept of combining heterogeneous components and they generally involve different developers who might not be part of the same team, it could be a challenging task for a developer working only on a sub-part of a project to clearly determine whether that specific parameter is used by other components or not. Thus, developers will probably tend to opt for keeping such parameters for safety concerns. The same goes for *Too Much Scattering* and *Unused Method Declaration*, these smells are defined respectively by occurrences in the code of native methods declarations that are no longer used, and separate and spread multi-language participants without considering the concerns. The number of these smells seems to increase over the releases as shown in Fig. 7. Under time pressure the developers might not take the risk to remove unused code, especially since in the case of JNI systems, such code could be used in other components. Similarly, the high distribution and increase of *Too Much Scattering* could be explained in situations where several developers are involved in the same projects, bugs related to simultaneous files changes may occur. When features are mixed together, a change to the behavior of one may cause a bug in another feature. Thus, developers might try to avoid these breakages by introducing scattered participants. Similarly, the design smell *Not Securing Libraries* is prevalent in the analyzed systems. We believe that developers should pay more attention to this smell. Malicious code may easily access such libraries. Occurrences of this smell can introduce vulnerabilities into the system, especially JNI systems that have been reported by previous studies to be prone to vulnerabilities [6, 10]. Several problems may occur due to the lack of security checking. An unauthorized code may access and load the libraries without permission. This may have an adverse impact especially in industrial projects that are usually developed for sale or are available for online use, or other safety-critical systems.

5.2 Smells and Faults

*Relation Between Smells and Faults*. In RQ3, we analyzed the relation between smells and fault-proneness. We used Fisher’s exact test and the odds ratios to check whether the proportion of buggy files varies between two samples (with and without design smells). From our results, we found that in general odds ratios are higher than one. This confirms previous insights from mono-language studies in which researchers claimed that design smells could increase the risk of faults [25, 55]. We cannot claim causation as we do not know whether such faults could have been caused by other factors. Although, our results suggest that files with JNI systems are more likely to be associated with faults than files without. In Zstd-jni, we found higher ORs than those of other systems from 13.9285 to 37.7142; this could be explained by the nature of smells involved in this system as reported in Table 6. Some types of smells could be more related to bugs than other types. Out of all the 98 releases analyzed, we found eight releases with ORs less than one, however, none of them was with a significant \( p \)-value. In Java-smt and Javacpp, \( p \)-values are not statistically significant (higher than 0.05) in most releases.

From studying bug-fix commit messages, we observed that the impact is also smell-dependent. Occurrences of some types of smells seem more related to bugs than others, which motivates us to perform the RQ4. Some occurrences of smells related to bugs have been refactored from one release to the next one. In many cases, we find a description in the commit message indicating refactoring for removing specific smells that caused the bugs (commit message: e.g., “There were a bunch of exceptions that are being thrown from JNI methods that aren’t currently declared”, “cleaning
up JNI exceptions (#252), “removed a few unused JNI methods”). Mono-language smells have been widely studied in the literature and were reported to negatively impact systems by making classes more change-prone and fault-prone. Multi-language systems could introduce additional challenges compared to mono-language systems. Those challenges are mainly related to the incompatibilities of programming languages and the heterogeneity of components. Thus, the design smells occurring on those systems are expected to increase the challenges related to the maintenance of these systems. Even if some smells e.g., Unused Method Declaration, Unused Method Implementation could not be directly related to bugs, they seem to increase the maintenance efforts because some of them are intentionally removed by developers. Thus, we believe that developers should be cautious about files with JNI smells, because they are more likely to be subject to faults and thus may incur additional maintenance efforts. Developers should also pay attention to avoid introducing occurrences of such design smells when dealing with JNI systems.

Relation between Specific Smells and Faults. Results from RQ4 show that some smells seem more related to faults than others: Unused Parameters, Too Much Clustering, Too Much Scattering, Hard Coding Libraries, and Not Handling Exceptions. The smell types Memory Management Mismatch and Not Securing Libraries are also found to be related to bugs. We believe that files containing these smells should be considered in priority for testing and refactoring. The smell Not Handling Exceptions was previously reported as related to bugs [6, 56]. In fact, we discussed a bug related to this smell early in Section 1. A bug related to this smell was reported in Conscrypt, developers were not checking for Java exceptions after all JNI calls that might throw them. The management of exceptions is not automatically ensured in all the programming languages. Incompatibility between the programming languages may lead to bugs and challenges related to the conversion, management of memory, and other mismatches between programming languages. In JNI projects, developers should explicitly implement the exception handling flow. Similarly, bugs in files containing the smells Unused Parameters and Too Much Clustering could be explained as the impacts of the noises that these two smells could introduce. Indeed, unused code or huge files with JNI code could impact code maintainability and the comprehension of JNI systems, which may lead to the introduction of bugs. From our detection approach, we identified files containing more than 200 method declarations that are not necessarily related in terms of responsibilities and that do not follow the principle of separating the concerns. We believe that faults could be easily introduced in such files, especially when dealing with JNI code; a developer might not be an expert on all the languages used and the inter-language interfaces. Developers should be concerned about the types of smells that are more likely to introduce bugs. The code containing these smells should be prioritized for testing and refactoring.

5.3 Risky Activities

From RQ5, we found that activities related to data conversion, memory management, restructuring the code, API usage, and exception management are among the activities that could increase the risk of bugs when performed on smelly files. This is not surprising as several articles and developers’ blogs discussed bugs related to the management of the memory in JNI systems [6, 10]. Some of these activities are directly related to design smells discussed in this paper, e.g., Memory Management Mismatch, Local References Abuse. Developers who do not follow good practices to avoid such design smells could perform activities that could increase the risk of future bugs in those files. In the context of JNI systems, it is the developers’ responsibility to take care of the management of memory because of the incompatibility between Java and C/C++. The same goes for data conversion when using JNI. We should consider specific rules to convert and access data between Java and C/C++. Primitive types could be easy to convert from Java to C/C++. However, reference types
are more complex and require additional knowledge on what kind of methods to use to apply a proper conversion. Several studies also discussed issues related to exceptions in JNI context. Unlike Java, C/C++ does not support the automatic handling of exceptions. Developers could introduce bugs if they do not have enough knowledge about how to implement the exception handling flow in JNI context. Such incompatibility between programming languages could introduce bugs and other maintenance challenges including checking exceptions, buffer overflows, and memory leaks [30]. Following formal guidelines and being aware of the practices to follow could help to improve the quality of those systems [6, 10, 12, 13]. We also noticed that in some systems, developers started paying more attention to this smell to avoid bugs related to the management of exceptions.

Conscrypt: “This works towards issue #258. So the exception can be routed out properly, this moves the SSL_get0_peer_certificates call to after doHandshake completes in ConscryptFileDescriptorSocket”.

Another example from Realm, developers started paying more attention to the smell Local References Abuse “DeleteLocalRef when the ref is created in a loop (#3366) Add wrapper class for JNI local reference to delete the local ref after using it”. We believe that further investigations should be performed to better understand the reasons for bug introduction in the presence of this smell.

5.4 Implications of the Findings

Based on our results we formulate some recommendations and highlight the implications of our findings that could help not only researchers but also developers and anyone considering using more than one programming language in a software system:

Our main goal was to investigate the existence of multi-language design smells and their impact on software quality. We found that multi-language code smells frequently occur within the selected projects and that they may increase the risk of bugs occurrence. Our results also highlight that the frequency and impact differ from one smell to the other. We also studied the activities that could introduce bugs once performed in smelly files.

Some of the implications of this study could be derived directly from the outcome of our research questions. First, researchers could find interest in studying why and how some specific types of smells are more frequent than others and the reasons behind their increase over time. They could also investigate the reasons why some specific types of smells are more related to bugs than others. The same goes for the activities, they could investigate further reasons behind the introduction of bugs when those specific activities are performed. They could also explore the existence of other activities that could introduce bugs. Second, practitioners could also take advantage of the outcome of this paper to reduce the maintenance cost of multi-language systems. In fact, most of the smells discussed in this paper (even those that are not always related to bugs) could introduce additional challenges and increase the effort of maintenance activities. Having knowledge of their existence and the potential impact could help to improve the quality of multi-language systems, and avoid their introduction in systems during evolution activities. In fact, as reported earlier, we found multiple commit messages in which developers explicitly mentioned issues caused by the occurrence of a smell studied in this paper. Studying each type of smell separately also allowed us to capture their impact individually. The insights from this study could help developers to prioritize multi-language smells for maintenance and refactoring activities. The same goes for the activities introducing bugs. Being aware of those activities could help developers avoid issues when performing them. Finally, the catalog of design smells studied in this paper is not exhaustive and presents only a small sample of possible multi-language smells and practices. Therefore, researchers and developers could further investigate smells and practices in multi-language software development. Our focus in this paper was on the JNI systems, and the researchers could also investigate other combination of programming languages. Additionally, they can also examine the impact of design smells on other quality attributes.
We recommend that developers pay more attention to the design patterns and design smells discussed in the literature that could be applied to the context of multi-language systems. Our results highlight the need for more empirical studies on the impact of multi-language smells on maintainability and program comprehension. We recommend to developers to be cautious when editing files containing design smells *Unused Parameters, Too Much Clustering, Too much Scattering, Not Handling Exception, Hard Coding Libraries* since their occurrence seems to increase the risk of fault introduction.

6 THREATS TO VALIDITY

In this section, we shed light on some potential threats to the validity of our methodology and findings following the guidelines for empirical studies [57].

*Threats to Construct Validity.* These threats concern the relation between the theory and the observation. In this study, these threats are mainly due to measurement errors. Most of the studied projects rely on Github issues to report bugs. Therefore, we identified fault-fixing commits by mining the Github commit logs using a set of keywords extracted from the literature [40, 48, 58]. We used a set of keywords similar to those previously used in studies focusing on bug prediction. However, this technique may not capture all the commits related to fault-fixing if the commit messages were not representative enough of the developer’s intention or were not containing any of those keywords. Nevertheless, this methodology was successfully used in multiple previous empirical studies [34, 46, 47, 58]. Moreover, in [59], the authors report that this technique can achieve a precision of 87.3% and a recall of 78.2%. Another threat to construct validity is related to the accuracy of the SZZ heuristic used to identify fault-inducing commits. Although this heuristic does not achieve a 100% accuracy, it has been successfully employed and reported to achieve good results in multiple empirical studies from the literature [60–62]. We also did a manual validation of the bug-inducing commits as described in Section 3.3.2 by inspecting the changes of a small sample of bug-inducing commits. For our smell detection approach, we applied simple rules. We adapted our detection approach to ensure a balanced trade-off between the precision and the recall. For some smells, *e.g., Memory Management Mismatch*, we considered specific situations in which the smell occurs following simple rules and the definition presented earlier in Section 2.3. Thus, this is not currently covering all possible issues related to memory management. However, the approach could be extended to include other contexts and types of memory issues following other rules.

When analyzing the smelliness of files that experienced bugs, we considered the whole file as participating in the design smell. Hence, the smell present in the file could be in different code lines than the bug. There is a similar threat in our analysis of the activities introducing bugs. We rely on commit messages provided by developers to identify the activities. We are aware that in some cases, developers might not have provided all the details of the activities performed or might have used some abbreviations. However, we mitigated this threat by combining both manual and automatic approaches to capture the possible activities that were performed. We are aware that the retrieved topics may not be 100% accurate. However, we followed the coding methodology applied in previous studies [63, 64] and two of the authors manually validated a subset of the commit messages. Through the manual analysis, we found that some commit messages describe more than one activity in the same commit (*e.g.*, Commit extracted from *Zstd-jni*: “Align the JNI names with the new streaming API, Move to the new streaming API, Use the ZBUFF based streaming compression”) while they assigned by the automatic approach to a single category. Although, the category to which they are assigned is based on the frequencies of the related keywords. We mitigated this threat by performing a manual validation over 500 commit messages. We also investigated some examples of activities at the source code level in the smelly code as described in Section 4.5. The
list of the activities may not be exhaustive and do not present a 100% recall and precision. However, in this paper we are reporting our observation on the activities that once performed in smelly files could introduce bugs without any empirical comparison of the risk introduced by each activity. However, we consider this as our future work in which we plan to perform a full manual validation approach to capture individual activities and the risk of introducing bugs related to each of them.

**Threats to Internal Validity.** We do not claim causation and only relate the presence of multi-language design smells with the occurrences of faults. We report our observations based on empirical results and explain these observations with manually analyzed examples from the studied systems to better contextualize our findings. We are aware that smells can depend on each other and we select the subset of non-correlated smells while building the logistic regression models. However, the variations in the distribution of smells, and some smells being very infrequent can have negative impacts on the regression models. As our model for each system considers all releases of a particular system than individual releases separately, it helps compensate for the infrequent classes by boosting the per-class data size. Our study is an internal validation of multi-language design smells that we previously defined and cataloged. Thus, this may present a threat to validity. However, this threat was mitigated by publishing our catalog in a pattern conference. The paper went through rounds of a shepherding process. In this process, an expert on patterns provided three rounds of meaningful comments to refine and improve the patterns. The catalog then went through the writers’ workshop process, in which five researchers from the pattern community had two weeks before the writers’ session to carefully read the paper and provide detailed comments for each defined smell. The catalog was then discussed during three sessions of two hours each. During these sessions, each smell was examined in detail along with their definition and concrete examples. The conference chair also provided additional comments to validate the catalog. In addition, the results of this paper have shown that the studied smells are related to bugs. From the commit messages, we also found that some smells were explicitly discussed by developers who contributed in the smelly files. For example, one developer discussed exception handling as “There were a bunch of exceptions that are being thrown from JNI methods that aren’t currently declared”. Therefore, we believe that the studied smells should be considered with caution by developers since they may hinder the software maintenance and may lead to bugs.

**Threats to External Validity.** These threats concern the possibility to generalize our results. We studied nine JNI open source projects with different sizes and domains of application. We focused on the combination of Java and C/C++ programming languages. Nevertheless, further validation of a larger number of systems with other sets of languages would give more opportunities to generalize the results. We studied a particular yet representative subset of multi-language design smells. Future works should consider analyzing other sets of design smells.

**Threats to Conclusion Validity.** These threats are related to the relationship between the treatment and the outcome. We were careful to take into account the assumptions of each statistical test. We mainly used non-parametric tests that do not require any assumption about the data set distribution.

**Threats to Reliability Validity.** We mitigate the threats by providing all the details needed to replicate our study in section 3. We analyzed open source projects hosted in GitHub. We also provide an online access to all the data and scripts used to conduct this study.21

7 RELATED WORK

We now discuss the literature related to this work.
7.1 Multi-language Systems

Several studies in the literature discussed multi-language systems. One of the very first studies, if not the first, was by Linos et al. [65]. They presented PolyCARE, a tool that facilitates the comprehension and re-engineering of complex multi-language systems. PolyCARE seems to be the first tool with an explicit focus on multi-language systems. They reported that the combination of programming languages and paradigms increases the complexity of program comprehension. Kullbach et al. [66] also studied program comprehension for multi-language systems. They claimed that program understanding for multi-language systems presents an essential activity during software maintenance and that it provides a large potential for improving the efficiency of software development and maintenance activities. Linos et al. [1] later argued that no attention has been paid to the issue of measuring multi-language systems’ impact on program comprehension and maintenance. They proposed Multi-language Tool (MT); a tool for understanding and managing multi-language programming dependencies. Kontogiannis et al. [2] stimulated discussion around key issues related to the comprehension, reengineering, and maintenance of multi-language systems. They argued that creating dedicated multi-language systems, methods, and tools to support such systems is expected to have an impact on the software maintenance process which is not yet known. Kochhar et al. [3] investigated the impact on software quality of using several programming languages. They reported that the use of multi-programming languages significantly increases bug proneness. They claimed that design patterns and anti-patterns were present in multi-language systems and suggested that researchers study them thoroughly. Kondoh et al. [30] presented four kinds of common JNI mistakes made by developers. They proposed BEAM, a static-analysis tool, that uses a typestate analysis, to find bad coding practice pertaining to error checking, virtual machine resources, invalid local references, and JNI methods in critical code sections. Tan et al. [10] studied JNI usages in the source code of part of JDK v1.6. They examined a range of bug patterns in the native code and identified six bugs. The authors proposed static and dynamic algorithms to prevent these bugs. Li and Tan [29] highlighted the risks caused by the exception mechanisms in Java, which can lead to failures in JNI implementation functions and affect security. They defined a pattern of mishandled JNI exceptions.

7.2 Impacts of Patterns and Smells

Several studies in the literature have studied the impact of design smells on software quality but mainly for mono-language systems.

Khomh et al. [25] analyzed nine releases of Azureus and 13 releases of Eclipse to investigate if the classes with occurrences of design smells are more change-prone than classes without those occurrences. They concluded that the classes with occurrences of design smells are more likely to be the subject of changes than classes without those occurrences. Olbrich et al. [67] proposed an approach that analyses the evolution of design smells and study their impact on the frequency and size of changes. They study two design smells: God Class and Shotgun Surgery. They used an automated approach based on detection strategies to detect the occurrences of design smells. They identified different phases in the cycle of design smells evolution during the different phases of the system development. They also found that components infected by design smells exhibit different behavior. Abbes et al. [68] investigated the impact of occurrences of anti-patterns in the developers’ understandability of systems while performing comprehension and maintenance tasks. They conducted three experiments to collect data about the performance of developers and study the impact of Blob and Spaghetti Code anti-patterns and their combinations. They concluded that the occurrence of one anti-pattern does not significantly impacts comprehension while the combination of the two anti-patterns negatively impact program comprehension. This finding
was corroborated by Politowski et al. [69]. Linares et al. [70] studied the potential relationship between the occurrence of design smells and quality attributes as well as the possible relation between design smells and application domains. They analyzed 1,343 Java Mobile applications in 13 different application domains. They concluded that anti-patterns negatively impact software metrics in Java Mobile applications, in particular, fault-proneness. They observed that there is a difference in the metric values between classes containing occurrences of smells and classes without smells. They also found that some smells are more frequently present in a domain of application while other smells are more present in other domains. Soh et al. [27] performed a study with six developers, three maintenance tasks, and four equivalent functions in Java. They used the Eclipse Mimec plugin and Thinkaloud sessions to analyze the effort spent by different developers when performing different maintenance activities (editing, reading, navigating, searching, static navigation, executing, and other activities). They concluded that design smells differently impact the effort needed to perform the different activities. They also found that the effort needed for reading, navigating, and editing is affected by three smells: “Feature Envy”, “God Class”, and “ISP Violation”.

7.3 Patterns and Smells Detection Approaches

Van Emden et al. [71] proposed the JCosmo tool that supports the visualization of the code layout and design defects locations. They used primitives and rules to detect occurrences of anti-patterns and code smells while parsing the source code into an abstract model. Marinescu et al. [72] proposed an approach for design defects detection based on detection strategies. The approach captures deviations from good design principles and heuristics to help developers and maintainers in the detection of design problems. Lanza et al. [73] presented the platform iPlasma for software modeling and analysis of object oriented software systems to detect occurrences of design defects. The platform applies rules based on metrics from C++ or Java code. Moha et al. [74] introduced DECOR which detects design defects in Java programs. DECOR is based on a domain-specific language that generates the design defect detection algorithms. Khomh et al. [75] proposed a Bayesian approach to detect occurrences of design defects by converting the detection rules of DECOR into a probabilistic model. Their proposed approach has two main benefits over DECOR: (i) it can work with missing data and (ii) it can be tuned with analysts’ knowledge. Later on, they extended this Bayesian approach as BDTEx [76], a Goal Question Metric (GQM) based approach to build Bayesian Belief Networks (BBNs) from the definitions of anti-patterns. They assessed the performance of BDTex on two open-source systems and found that it generally outperforms DECOR when detecting Blob, Functional Decomposition, and Spaghetti code anti-patterns.

Kessentini et al. [77] proposed an automated approach to detect and correct design defects. The proposed approach automatically finds detection rules and proposes correction solutions in term of combinations of refactoring operations. Rasool et al. [78] proposed an approach to detect occurrences of code smells that supports multiple programming languages. They argued that most of the existing detection techniques for code smells focused only on Java language and that the detection of code smells considering other programming languages is still limited. They used SQL queries and regular expressions to detect code smells occurrences from Java and C# programming languages. In their approach, the user should have knowledge about the internal architecture of the database model to use the SQL queries and regular expressions. In addition, each language needs a specific regular expression. Fontana et al. [79] conducted a study applying machine learning techniques for smell detection. They empirically created a benchmark for 16 machine learning algorithms to detect four types of code smells. The analysis was performed on 74 projects belonging to the Qualitas Corpus dataset. They found that J48 and Random Forest classifiers attain the highest accuracy. Liu et al. [80] proposed a smell detection approach based on
Deep Learning to detect Feature Envy. The proposed approach relies on textual features and code metrics. It relies on deep neural networks to extract textual features. Barbez et al. [81] proposed a machine learning based method SMAD that combines several code smells detection approaches based on their detection rules. The core of their approach is to extract metrics based on existing approaches and use those metrics as features to train the classifier for smell detection. The proposed approach supports the detection of the smells of type God Class and Feature envy. Their approach outperforms other existing methods in terms of recall and Matthews Correlation Coefficient (MCC). Palomba et al. [82] proposed TACO, an approach that relies on textual information to detect code smells at different levels of granularity. They evaluated their approach on ten open source projects and found that the proposed approach outperforms existing approaches. While there are some studies in the literature that document the good and bad practices related to multi-language systems,[7, 8, 10, 56, 83] to the best of our knowledge, this is the first study that automatically detects occurrences of multi-language design smells in the context of JNI systems and evaluates their impact on software fault-proneness. Other studies in the literature are focusing on the detection and analysis of design smells in mono-language systems.

8 CONCLUSION

In this paper, we present an approach to detect multi-language design smells and empirically evaluate the impacts of these design smells on fault-proneness. We performed our empirical study on 98 releases of nine open source JNI systems. Those systems provide a great variety of services to numerous different types of users. They introduce several advantages, however, as the number of languages increases so does the maintenance challenges of these systems. Despite the importance and increasing popularity of multi-language systems, studying the prevalence and impact of patterns and smells within these systems is still under-investigated. In this paper, we studied the impact of multi-language design smells on the software fault-proneness. We investigated the prevalence and impact of 15 design smells on fault-proneness. We showed that the design smells are prevalent in the selected projects and persist across the releases. Some types of smells are more prevalent than others. Our results suggest that files with JNI smells are more likely to be subject of bugs than files without those smells. We also report that some specific smells, are more likely to be of a concern than others, i.e., Unused Parameters, Too Much Scattering, Too Much Clustering, Hard Coding Libraries, and Not Handling Exceptions. These smells seem more related to faults, thus we suggest that practitioners consider them in priority for testing and-or refactoring. This empirical study supports, within the limits of its threats to validity, the conjecture that multi-language design smells are prevalent in the selected projects and that similar to mono-language smells, JNI smells may have a negative impact on software reliability. From analyzing fault-inducing commits we found that data conversion, memory management, code restructuring, API usage, and exception management activities could increase the risk of bug introduction when performed on smelly files. We believe that the results of this study could help not only researchers but also practitioners involved in building software systems using more than one programming language. Our future work includes (i) replicating this study with a larger number of systems for further generalization of our results; (ii) studying the impact of design smells on change-proneness, (iii) investigating the occurrences of other patterns and defects related to multi-language systems.

REFERENCES


Are Multi-language Design Smells Fault-prone? An Empirical Study

51


., Vol. 1, No. 1, Article . Publication date: October 2020.
A APPENDIX

We present in the following the smell detection rules of the proposed approach. These rules are applied on the srcML elements generated as an XML representation of a given project as described in Section 3.3.1. Since the smells described in this paper are multi-language smells, the following rules detect the occurrences of smells by using the XPath queries in the srcML representation of the source code that contains Java and C/C++ native code.

(1) Rule 1: Not Handling Exceptions

\[(f(y) \mid f \in \{\text{GetObjectClass, FindClass, GetFieldID, GetStaticFieldID, GetMethodID, GetStaticMethodID}\}) \land (\text{isExceptionChecked}(f(y)) = \text{False} \lor \text{ExceptionBlock}(f(y)) = \text{False})\]

Our detection rule for the smell Not Handling Exceptions is based on the existence of call to specific JNI methods requiring an explicit management of the exception flow. The JNI methods (e.g., FindClass) listed in the rule should have a control flow verification. The parameter \(y\) presents the Java object/class that is passed through a native call for a purpose of usage by the C/C++ side. Here, \(\text{isExceptionChecked}\) allows to verify that there is an error condition verification for those specific JNI methods, while \(\text{ExceptionBlock}\) checks if there is an exception block implemented. This could be implemented using Throw() or ThrowNew() or a return statement that exists in the method in case of errors.

(2) Rule 2: Assuming Safe Return Value

\[x := f(y) \mid f \in \{\text{FindClass, GetFieldID, GetStaticFieldID, GetMethodID, GetStaticMethodID}\} \land \text{IsErrorChecked}(x) = \text{False} \land \text{IsReturn}(x) = \text{True}\]

This rule is quite similar to the previous rule. However, it considers the return value from the native code. Indeed, the JNI methods called in this context are used for specific calculation and the result then needs to be passed as a method return value to the Java side. Here, \(x\) presents the native variable used within the method to receive the returned value and perform computation on the Java side. \(\text{IsErrorChecked}(x)\) allows to verify if there is an error condition verification applied to the variable \(x\) that will be returned back to the Java code (\(\text{IsReturn}(x) = \text{True}\)). The use of the variable \(x\) as a return value by a native method without any check of its correctness will introduce smell of type Assuming Safe Return Value given other conditions hold.

(3) Rule 3: Not Securing Libraries

\[\text{IsNative}(\text{Lib}) = \text{True} \land \text{loadedWithinAccessBlock}(\text{Lib}) = \text{False}\]
This rule implies that in the Java code, a native library is used (IsNative(Lib) = True) and that this library is loaded outside a block AccessController.doPrivileged without a try and catch statements for safe handling of potential exceptions. This introduces smell of type Not Securing Libraries.

(4) **Rule 4: Hard Coding Libraries**

\[
\text{IsNative(Lib)} = \text{True AND AccessiblePath(Lib)} = \text{False AND OsBlock(m)} = \text{True}
\]

This rule implies that in the Java code, a native library (Lib) is used in a native method \( m \) and that the path used for accessing that library is an absolute path while the code loading the library depends on the operating systems. Here, the access to libraries is hard coded for specific operating system rather than implementing a platform independent mechanism for libraries. This limits the portability of the code and may cause issues in accessing the libraries for different operating systems.

(5) **Rule 5: Not Using Relative Path**

\[
\text{IsNative(Lib)} = \text{True AND RelativePath(Lib)} = \text{False}
\]

This rule implies that in the Java code, a native library is used. However, the native library loaded from an absolute path and not from a relative path.

(6) **Rule 6: Too Much Clustering**

\[
\text{NbNativeMethods (C)} \geq \text{MaxMethodsThreshold AND IsCalledOutside(m)} = \text{True}
\]

This rule detects cases where the total number of native methods (NbNativeMethods) within any class \( C \) is equal to or higher than a specific threshold while those methods \( m \) are used by other classes and not only the one where they are declared (IsCalledOutside(m) = True). In our case, we used the default values for the threshold eight. However, all the thresholds could be easily adjusted as discussed earlier in Section 3.3.1.

(7) **Rule 7: Too Much Scattering**

\[
\text{NBNativeClass(P)} \geq \text{MaxClassThreshold AND (NbNativeMethods(C) < MaxMethodsThreshold AND C} \in \text{P})
\]

The smell of type Too Much Scattering occurs when the total number of native classes in any package \( P \) (NBNativeClass(P)) is more than a specific threshold (MaxClassThreshold) for the number of maximum native classes. In addition, each of those native classes \( C \) contains a total number of native methods (NbNativeMethods(C)) less than a specific threshold (MaxMethodsThreshold) i.e., the class does not contain any smell of type Too Much Clustering. We used default values for the threshold three for the minimum number of classes with each a maximum of three native method each.

(8) **Rule 8: Excessive Inter-language Communication**

\[
(\text{NBNativeCalls(C,m)} > \text{MaxNbNativeCallsThreshold}) \text{ OR } (\text{NBNativeCalls(m(p))} > \text{MaxNativeCallsParametersThreshold}) \text{ OR } ((\text{NBNativeCalls (m)} > \text{MaxNbNativeCallsMethodsThreshold}) \text{ AND IsCalledInLoop(m)} = \text{True})
\]

The smell Excessive Inter-language Communication is detected based on the existence of at least one of the three possible scenarios. First, in any class \( C \) the total number of calls to a
particular native method \( m \) exceeds the specified threshold (\( NBNativeCalls(C,m) > MaxNbNativeCallsThreshold \)). Second, the total number of calls to the native methods \( m \) with the same parameter \( p \) exceeds the specific threshold (\( MaxNativeCallsParametersThreshold \)). Third, the total number of calls to a native method \( m \) within a loop is more than the defined threshold (\( (MaxNbNativeCallsMethodsThreshold) \)).

(9) **Rule 9: Local References Abuse**

\[
(NbLocalReference(f_1(y)) > MaxLocalReferenceThreshold) \text{ AND } (f_1(y) | f_1 \in \{GetObjectArrayElement, GetObjectArrayElement, NewLocalRef, AllocObject, NewObject, NewObjectA, NewObjectV, NewDirectByteBuffer, ToReflectedMethod, ToReflectedField\}) \text{ AND } (\exists f_2(y) | f_2 \in \{DeleteLocalRef, EnsureLocalCapacity\})
\]

The smell *Local References Abuse* is introduced when the total number of local references (\( NbLocalReference(f_1(y)) \)) created inside a called method exceeds the defined threshold and without any call to method \( DeleteLocalRef \) to free the local references or a call to method \( EnsureLocalCapacity \) to inform the JVM that a larger number of local references is needed.

(10) **Rule 10: Memory Management Mismatch**

\[
\]

As discussed earlier, JNI offers predefined methods to manage the access of reference types that are converted to pointers. These methods are used to create pointers and to allocate the corresponding memory. The rule described here allows to detect the native implementation in which the memory was allocated by calling one of these allocation methods, however, the memory allocated was never released. The rule detects situations in which ‘get’ methods are used to allocate memory for specific JNI elements that are not released after usage by calling the corresponding ‘release’ methods.
(11) Rule 11: Not Caching Objects


\[ ((\text{Parameter}(m, p) = \text{Object}) \ \land \ \neg \text{IsCached}(m, f_n(y)) \ \land \ \neg \text{IsLoop}(m) \ \land \ \text{NoOfIterations}(m) > \text{MaxCountThreshold}) \]

This rule allows to detect occurrences of the smell Not Caching Objects based on two situations. The first one is where the total number in which ids related to the same object are looked up for the same class through JNI allocation methods is greater than or equal to a specific threshold or the method is called within a loop. Indeed, the ids returned for a given class remain the same for the lifetime of the JVM execution. Considering that we have a native method and one of its parameter is a Java object, this type is considered in the native code as a reference type. Thus, unlike primitive types, its element could not be accessed directly by the native code but should be accessed through the usage of the methods defined in. In this first scenario, the total number of calls from the Java code to a native method is defined in a class exceeds a specific threshold (i.e., \( \text{NbCalls}(C, m) \geq \text{MaxNbCallsThreshold} \)). This last scenario includes the total number of calls to the predefined methods independent of the total number of calls to the method itself.

(12) Rule 12: Excessive Objects

\[ ((\text{Parameter}(m, p) = \text{Object}) \ \land \ \text{IsCalledInMethod}(m, f_1) = \text{True}) \ \land \ \text{NbCalls}(f_1(y)) = \text{MaxNbCallsThreshold} \]

This rule identifies situations in which a JNI object is passed as a parameter to the native code. In this context the total number of calls to allocation methods to retrieve its field id in the same method is greater than a specific threshold (i.e., \( \text{NbCalls}(f_1(y)) = \text{MaxNbCallsThreshold} \)), without a call to corresponding set functions to set the object fields by the native code. However, as described in the specification of the smell in Section 2.3, having the total number of calls to allocation methods higher than the threshold is not considered as a smell only in situations where the purpose of those calls was to set the...
object fields by the native code.

(13) **Rule 13: Unused Method Implementation**

\[
\text{IsNative}(m) = \text{True AND IsDeclared}(m) = \text{True AND IsImplemented}(m) = \text{True AND IsCalled}(m) = \text{False}
\]

This rule allows to capture the native functions \( m \) (IsNative \( m \) = True) implemented in the C/C++ (IsImplemented \( m \) = True), declared in Java with the keyword native but never used in the Java code (IsCalled \( m \) = False). It looks for the native methods that are declared using the keyword native with a header in the Java code and looks for the corresponding native implementation nomenclature.

(14) **Rule 14: Unused Method Declaration**

\[
\text{IsNative}(m) = \text{True AND IsDeclared}(m) = \text{True AND IsImplemented}(m) = \text{False}
\]

Native functions declared in Java with the keyword native (IsDeclared \( m \) = True) that are not implemented in C/C++ (IsImplemented \( m \) = False). This rule allows to retrieve the native methods that are declared with a header in the Java code using the keyword native and checks for the corresponding implementation nomenclature. However, those methods were never used or even implemented in the C/C++ code.

(15) **Rule 15: Unused Parameters**

\[
(\text{IsNative}(m(p)) = \text{True AND IsDeclared}(m(p)) = \text{True AND IsImplemented}(m(p)) = \text{True AND IsParameterUsed}(p) = \text{False})
\]

This rule reports the method parameters that are used in the Java native method declaration header using the keyword native (IsDeclared \( m(p) \) = True). However the parameter is never used in the body of the implementation of the methods, apart from the first two arguments of JNI functions in C/C++. The rule checks if the parameter \( p \) is used in the corresponding native implementation (IsParameterUsed \( p \) = False).

B APPENDIX

We present in Table 14 the results of the evaluation of the performance of our design smell detection approach. The details of the results are available in the replication folder.

The pilot project as described in Section 3 was the project we developed with the occurrences of the smells along with the clean code without any smell to test and validate our approach. This explains the 100% precision and 100% recall for all the smells. For the other projects, the precision and recall were evaluated through the investigation of the occurrence of the smell itself and the multi-language files involved on that smell. Most of the rules are trivial and the corresponding smells could easily be detected by our approach. Therefore, the reasons for false positives and false negatives are mainly related to the alternative implementation choices of the multi-language code that do not follow JNI specification guidelines and therefore are not currently covered by our approach. Indeed, our approach consider the JNI implementation with the appropriate naming convention as described in the JNI specification (e.g., using the native keyword in the Java native method declaration, using JNIENV, JNIEXPORT, JNICALL, and Java_ClassName_methodname) [84]. Thus, our detection approach could only be considered for JNI systems that follows the JNI
specification guideline. The validation on some systems were done earlier and thus on the older versions of the systems. Thus, the validation results only reports on the smell types available in the analyzed version. As described in the Section 5, our approach may present some limitations for the smell **Local References Abuse** in situations in which some specific methods are used to ensure the memory capacity. However, as per our manual analysis when defining the smells, those methods are not considered relevant to detect the smell and are not usually used. However, we are aware that in similar situations, the approach may result in false positives. For the smells **Unused Parameters** and **Unused Method Declaration**, when evaluating the recall and precision we noticed that our approach was not always able to correctly match the Java and corresponding native implementation. This was mainly due the syntax used in the C implementation that is not completely following the JNI specification for the naming convention (e.g., `Pljava jobject pljava_DualState_key`). For the smells **Assuming Safe Return Value** and **Not Handling Exceptions**, the false negatives in Conscrypt were related an intermediate step that made the detection harder. In this intermediate step, the native value was checked before returning it to the Java code. The same goes for the smell **Memory Management Mismatch**, our rules allows the detection of a specific type of memory issues and do not cover other types of issues related to the memory.
<table>
<thead>
<tr>
<th>Type of Smells</th>
<th>Design Smells</th>
<th>FP Precision</th>
<th>FN Recall</th>
<th>FP Precision</th>
<th>FN Recall</th>
<th>FP Precision</th>
<th>FN Recall</th>
<th>FP Precision</th>
<th>FN Recall</th>
<th>FP Precision</th>
<th>FN Recall</th>
<th>FP Precision</th>
<th>FN Recall</th>
<th>FP Precision</th>
<th>FN Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Excessive Inter-language</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>2 Too Much Clustering</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>3 Too Much Scattering</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>4 Unused Method Declaration</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>5 Unused Method Implementation</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>6 Unused Parameter</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>7 Assuming Safe Return Value</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>8 Excessive Objects</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>9 Not Handling Exceptions</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>10 Not Caching Objects</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>11 Not Securing Libraries</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>12 Hard Coding Libraries</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>13 Not Using Relative Path</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>14 Not Securing Libraries</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
</tr>
<tr>
<td>15 Local Reference Abuse</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
<td>100%</td>
<td>0</td>
</tr>
</tbody>
</table>