# The Exception Handling Riddle: An Empirical Study on the Android API

Maria Kechagia[a,b,1,*], Marios Fragkoulis[a], Panos Louridas[a], Diomidis Spinellis[a]

[a] *Athens University of Economics and Business, 76 Patission str., 104 34 Athens, Greece*
[b] *Delft University of Technology, van Mourik Broekmanweg 6, 2628 XE, Delft, Netherlands*

**Abstract**

We examine the use of the Java exception types in the Android platform's Application Programming Interface (API) reference documentation and their impact on the stability of Android applications. We develop a method that automatically assesses an API's quality regarding the exceptions listed in the API's documentation. We statically analyze ten versions of the Android platform's API (14–23) and 3,539 Android applications to determine inconsistencies between exceptions that analysis can find in the source code and exceptions that are documented. We cross-check the analysis of the Android platform's API and applications with crash data from 901,274 application execution failures (crashes). We discover that almost 10% of the undocumented exceptions that static analysis can find in the Android platform's API source code manifest themselves in crashes. Additionally, we observe that 38% of the undocumented exceptions that developers use in their client applications to handle API methods also manifest themselves in crashes. These findings argue for documenting known might-thrown exceptions that lead to execution failures. However, a randomized controlled trial we run shows that relevant documentation improvements are ineffective and that making such exceptions checked is a more effective way for improving applications' stability.

*Keywords:* exceptions, application programming interfaces, documentation

*Corresponding author
*Email address:* mkechagia@aueb.gr; m.kechagia@tudelft.nl (Maria Kechagia)

[1] Part of this work was conducted in the Delft University of Technology.

## 1. Introduction

Exceptions aim to assist developers to program robust software with limited execution failures (**crashes**) [1, 2, 3]. Exception types vary based on programming languages' design and syntax. Two opposing historical cases are C and Ada. C has no exception handling mechanism but error codes that manifest at runtime when an application programming interface (API) fails. On the other hand, Ada supports a mechanism that checks for exceptions at compile time. Modern programming languages, such as C++, C#, Objective–C, and scripting languages (Python, Ruby, PHP) offer exception handling mechanisms but without enforcing developers to handle exceptions. However, Java, which we take into account in this paper, borrows parts from both C and Ada's design concept supporting unchecked, compile time checked exceptions, and errors.

**Checked exceptions** refer to "exceptional conditions that a well-written application should anticipate and recover from."[2] A method must mention the checked exceptions that throws, in its signature. When compiling a program, the compiler ensures that the callers of this method either catch the exceptions or they explicitly mention, using the `throws` keyword, the exceptions in their own method signatures (i.e. developers are always forced to handle checked exceptions in their programs)[4].

On the other hand, **unchecked exceptions** are exceptional conditions that can be either external (`Error`) or internal (`RuntimeException`) from

*Journal of Systems and Software*, 142:248–270, August 2018. doi:10.1016/j.jss.2018.04.034



[2]https://docs.oracle.com/javase/tutorial/essential/exceptions/catchOrDeclare.html. All links were accessed on the 2nd of August, 2017.

which applications usually cannot recover, resulting to crashes. In practice, API designers should explicitly throw unchecked exceptions, listing their types in the API reference documentation, when the former believe that client applications' developers can overcome from particular execution failures [5, 6]. The usefulness of well-designed and documented unchecked exceptions is twofold. Either developers can catch unchecked exceptions to prevent their applications from recoverable crashes or they can consider documented unchecked exceptions in post-mortem analysis [7].

Theoretically, exception handling can improve applications robustness. However, there are many arguments that confirm the challenges that are associated with these mechanisms, especially concerning Java APIs. Briefly, we refer some of them.

- It is impossible for API designers and clients to document and handle, respectively, the right exceptions in order to prevent all possible application crashes [8, 9].
- It is difficult for developers to handle crash causes mainly associated with generic exceptions (e.g. `Exception`, `RuntimeException`) [10, 11, 12].
- Exception handling code is buggy and of poor quality [13, 14, 15].
- Many caught exceptions can increase a program's complexity [16, 17, 18].

In this context, we investigate: 1) the exception handling mechanisms of a large Java API, the Android platforms' API, to understand when and how API designers use exceptions and 2) how API methods are used with exceptions in Android applications' source code. Our goal is to provide suggestions about exceptions that should be listed in an API documentation reference. We hope that our approach can simplify current exception handling practices and improve client applications' stability (i.e. executions with fewer failures).

To meet our objectives, we conducted an empirical study on the Android ecosystem. Specifically, we processed: 1) ten versions of the Android platform's API (levels 14–23) to identify might-thrown exceptions, 2) more than three thousand Android applications to find what exceptions client applications' developers use to handle called API methods, and 3) a dataset of almost one million stack traces from Android application crashes to cross-check APIs associated with execution failures. We applied static analysis

on the Android API and on client applications, as well as post-mortem root cause analysis on the stack traces. Then, we pinpointed API methods that can cause application crashes and we identified undocumented exceptions that both API designers and clients can use to prevent client applications from further execution failures related to those API methods. Finally, to evaluate our approach and suggestions, we run a randomized controlled trial on Android and Java programmers.

Overall, we discovered that almost 10% undocumented exceptions that static analysis can find in the Android platform's API source code manifest themselves in crashes. Similarly, we observed that 38% undocumented exceptions that developers use in their client applications to handle API methods manifest themselves in crashes. Even though these findings argue for documenting known might-thrown exceptions that lead to execution failures, our trial revealed that documentation (i.e. if an unchecked exception is documented or not) does not affect developers on how they handle possible crash causes. Taking into account related work [1, 19, 20, 21, 3] we also argue that some exceptions (e.g. those related to invalid user inputs and erroneous resource identifiers) should be converted to checked. Alternatively, the type system [22] should be used to improve the robustness of modern applications and the productivity of client applications' developers.

Briefly, for the identification of uncaught exceptions in ML programs, Kwangkeun and Sukyoung implemented static analysis [19] and Leroy and Pessaux introduced type-based analysis [1]. Additionally, Endrikat et al. [21] and Mayer et al. [20] conducted empirical studies on developers to evaluate static systems. They found that these systems can act as implicit documentation benefiting developers' productivity based on the context. This is in line with our observations from the controlled trial we conducted. Finally, Zhang et al. [3] introduced the concept that an exception should be created as checked based on the context and not on its type. Our findings also confirm this argument.

Our research contributions are the following:

- an approach that suggests undocumented exceptions that *risky* API methods might throw at runtime, and these exceptions should be listed (i.e. be documented) in the API reference documentation,

- an analysis of the evolution of the Android platform's API regarding its exception handling mechanisms,

- an empirical study of the exception types that client applications' developers catch when calling methods from the Android API,
- a randomized controlled trial for the evaluation of the effectiveness of different flavors of exceptions and API reference documents that may help developers in writing better code and less crash-prone applications.

Our recommendations can advance the reliability of APIs and the productivity of client applications' developers. API **designers** can consider listing of the suggested exceptions in the Android platform's API. Client applications' **developers** can handle API methods, such as those presented in Table 13, using the exceptions we recommend to increase the robustness of their applications.

The structure of the remainder of the paper is organized as follows. Section 2 presents related work and Section 3 demonstrates some motivating examples of our study. Section 4 describes our experimental setup (data, methods, and metrics), Section 5 presents our findings, Section 6 refers to the method we used to evaluate our approach and findings, Section 7 lists the threats to the validity of our research. Finally, Section 8 outlines our conclusions and plans for future work.

## 2. Related work

Our study is related to previous work in the following areas: 1) exception-flow analysis, 2) exception handling evaluation, 3) mining of crash data, 4) API evaluation, and 5) software applications' robustness.

### *2.1. Exception-flow Analysis*

Many techniques and tools can be used for the identification of possible exceptions that a method can throw [23]. Robillard and Murphy implemented a static analysis tool, called Jex, that provides information about the exception types that might arise from Java systems at runtime [8]. Vallée-Rai et al. developed the Soot Java byte code optimization framework that can find might-thrown exceptions for API methods, using intra-procedural analysis [24]. Fu and Ryder also presented an exception-flow static analysis technique, based on Soot, that computes chains of semantically-related exception-flow links for the understanding of the exception handling architecture of a system [25]. Apart from static analysis, researchers have constructed

tools for finding might-thrown exceptions, based on dynamic analysis (Phosphor [26]) and model checking (Java PathFinder [27]). Here, we choose to use the Soot framework [24] on a large scale analysis of Android applications in order to identify might-thrown exceptions from the Android platform's API source code. We also implement an inter-procedural analysis approach that Soot does not currently support (see Section 4.2.1).

### *2.2. Exception Handling Evaluation*

Several works have surveyed developers regarding the understanding and usefulness of exception handling mechanisms. Most of the studies show that programmers resist to use exception handling because they found it difficult [10, 11, 12]. Also, research findings reveal that exception handling code is complex [16, 17, 18] and fault prone [13, 14]. Taking into account previous works, we make recommendations on specific documented exceptions to help developers to improve their understanding of exception handling so that they can write less crash-prone applications. To the best of our knowledge, we are the first that conduct a randomized controlled trial to validate our suggestions regarding effective exception handling.

### *2.3. Stack Trace Mining*

For stack trace mining, researchers have used heuristic methods, natural language processing, as well as machine learning. In particular, Dang et al. presented a technique based on call stack matching [28]. This technique measures the similarities of call stacks and assigns the reports to appropriate classes or "buckets". Also, Kim et al. proposed an approach based on crash graphs to provide an aggregated view of the crashes [29] and Liblit and Aiken studied the reconstruction of execution paths based on partial execution information like backtracks to find the root crash causes [30]. Recently, Coelho et al. used heuristic rules to mine exceptions from issues on Android projects that were hosted on GitHub and presented descriptive statistics based on different exception types [31]. In our study, we use both "bucketing" and heuristic rules to find method calls in the stack traces that lead applications to crashes. Our techniques are based on a previous work where we mined stack traces to classify the main reasons behind the execution failures of one million Android application crashes [32].

### *2.4. API Evaluation*

A large body of research is dedicated to the development of usable APIs for increasing developers' productivity and client applications' robustness. Robillard et al. have published a survey on tools and mining techniques regarding APIs' analysis [33]. In the following paragraphs, we list works regarding the usability of modern APIs and their documentation, discussing the contributions of this paper.

#### *2.4.1. API Usability*

The assessment of APIs' usability is a popular topic within the software engineering community. Caracteristically, Clarke developed a framework of cognitive dimensions that can be used for the assessment of the usability of large APIs [34]. Recently, Scheller and Kühn designed a framework, called API Concepts Framework, for the automated measurement of APIs' usability [35].

In this context, several works present techniques for assisting developers to use APIs. Zhong et al. implemented a tool called MAPO to mine and recommend API usage patterns [36]. Also, Buse and Weimer developed an algorithm that synthesizes human-readable API usage examples [37]. In a recent work, Santos and Myers presented a tool called Dacite that helps developers to discover API elements and relationships among them [38]. Using this tool API designers complement APIs with design annotations, from which code completion proposals regarding object creation and manipulation are provided to the clients.

Additionally, many empirical studies have been conducted for the evaluation of APIs' usability. Qiu et al. analyzed five thousand open source Java projects to understand the use of Java and third-party APIs [39]. For their analysis they used abstract syntax trees and measured the usage of metrics, such as frequency, popularity, and coverage. Furthermore, Endrikat et al. run a controlled experiment and an exploratory study to compare the impact of the usage of documentation and static or dynamic type systems at software production level [21].

This paper is closely related to the concept of the work of Endrikat et al. [21], because we also run here a randomized controlled trial and an empirical study on a large API to improve the latter's usability. However, we evaluate the use of checked and documented unchecked exceptions.

*2.4.2. Documentation Quality*

Documentation quality is associated with developers' productivity and effective software maintainability. Robillard and DeLine surveyed developers and found that an API's reference documentation with code examples, use scenarios, good formatting and presentation, can increase programmers' productivity and help them to overcome APIs' documentation learning barriers [40]. Similarly, Maalej and Robillard pinpointed knowledge patterns in API reference documentation and grouped them into a taxonomy that can be used for APIs' evaluation and content organization [41]. Also, Zhong and Su developed an approach called DocRef that identifies documentation errors (e.g. broken code names and obsolete code samples) to prevent developers from reading misleading API documents [42].

Additionally, recent studies indicate that there is a gap between APIs' documentation and source code, confusing developers of client applications. In particular, McDonnell et al. conducted a study on the co-evolution behavior of the Android API and dependent applications [43]. They found that client applications slowly adapt with the pace of API's evolution and that the API usage adaptation code can lead to more defects. Taking into account inconsistencies between APIs' source code and documentation, Dagenais and Robillard proposed a technique for automatic discovery of documentation patterns [44]. Their tool tracks source code changes and determines which changes are significant enough to be documented.

Several tools have been also constructed for the identification of missing or misleading information from APIs' documentation regarding exceptional conditions. Buse and Weimer developed an automatic approach based on symbolic execution and inter-procedural data flow analysis that identifies might-thrown exceptions and their causes of manifestation [45]. The causes can be used to generate more informative documentation about possible runtime exceptions. Also, Saied et al. [46] examined the existence of variables' constraints in the reference documentation of a set of Java APIs and made related design suggestions.

We add to the previously discussed works by statically identifying insufficient documentation about exceptions that are associated with recoverable software crashes and we make relevant recommendations for re-documentation. We also analyze several versions of the Android API as well as applications to study how developers use APIs, such as the Android one. However, we focus on the examination of the design and use of exception handling mechanisms.

### *2.5. Android Applications' Robustness*

Software responsiveness and robustness are crucial features for modern mobile applications' user experience and success.

To improve applications' stability, researchers have invented testing methods to predict possible execution failures. In particular, Machiry et al. developed a system, called Dynodroid, that generates inputs for event-driven applications to find bugs [47]. Similarly, Moran et al. implemented an automated approach, called CrashScope, for testing Android applications using generated user inputs to trigger, identify, and reproduce crashes [48]. Likewise, Yan et al. proposed a novel approach for systematic testing for resource leaks in Android software [49] and Anand et al. presented a technique based on concolic testing that generates sequences of events to show the effectiveness of mobile applications [50].

Several empirical studies refer to the assessment of the reliability of the Android API itself, as well as its client applications' robustness. For instance, Felt et al. [51] applied static analysis on Android's source code to identify API permission leaks. In addition, Gorla et al. [52] used similar analysis' techniques and found Android applications that perform different actions from their descriptions (e.g. a weather application that sends messages).

Studies have been also conducted on the evolution of the Android API. Linares-Vásquez et al. [53] examined the implementation of different Android API versions and argued that bug fixes and changes in the API can have negative impact on the ratings of Android third-party applications. Li et al. analyzed several versions of the Android platform's API and Android applications to study accessible and inaccessible (hidden) APIs [54]. The authors searched how third-party applications adopt inaccessible APIs and they found that APIs are differently used among malicious and benign applications. Recently, Oliveira et al. [55] investigated the evolution of used exception handling mechanisms in Java and Android applications. They examined the impact of these mechanisms' changes on the robustness of the analyzed applications. The core finding of that study is in line with the observations we present here, suggesting that current exception handling mechanisms should be improved to support developers' attempts on maintaining exception handing code and client applications' robustness.

In this paper, we present an empirical study on the evolution of the Android ecosystem, making suggestions on exception handling mechanisms for the correct use of API methods. Contrary to the related work, we examine different sources of data (APIs, applications, and stack traces) to investigate

**Listing 1: Source code snippet from the `insert` method**

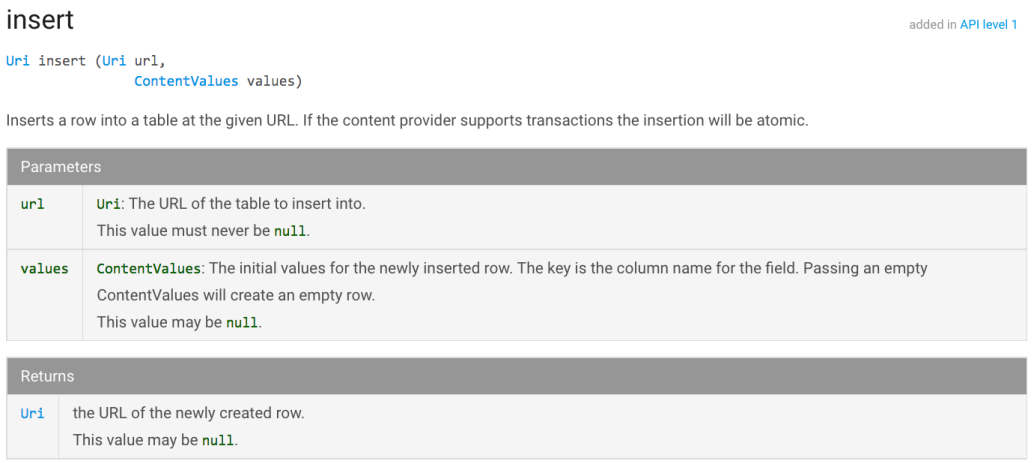
```
/**
  * Inserts a row into a table
  * at the given URL.
  *
  * If the content provider supports
  * transactions
  * the insertion will be atomic.
  *
  * @param url The URL of the table
  * @param values The initial values
  * for the newly inserted row.
  * ...
  * @return the URL of the newly created row.
  */
  public final Uri insert(Uri url, ContentValues values) {
    IContentProvider provider = acquireProvider(url);
    if (provider == null) {
        throw new IllegalArgumentException("Unknown URL" + url);
    }
    try {
        ...
        return createdRow;
    } catch (RemoteException e) {
        // ...
        return null;
    } finally {
        releaseProvider(provider);
    }
   }
```

how the use of exception handling mechanisms impact client applications' robustness. We hope that our suggestions can help in the improvement of existing exception handling mechanisms, which are crucial for the stability and responsiveness of client applications.

## 3. Motivating Examples

In this section, we illustrate a representative example of an unchecked exception that is not listed in the documentation of an API method—even though the exception is declared as might-thrown in the source code of this method. If a developer is not careful enough to catch this exception, the application will crash in case of an **invalid** method argument.

Listing 1 shows a source code snippet from the method `insert` of the Android API (level 15). When passing an unknown URL to the method, an `IllegalArgumentException` will be thrown. However, this information is



insert

added in API level 1

```
Uri insert (Uri url,
            ContentValues values)
```

Inserts a row into a table at the given URL. If the content provider supports transactions the insertion will be atomic.

| Parameters | |
|---|---|
| url | Uri: The URL of the table to insert into.<br>This value must never be null. |
| values | ContentValues: The initial values for the newly inserted row. The key is the column name for the field. Passing an empty ContentValues will create an empty row.<br>This value may be null. |

| Returns | |
|---|---|
| Uri | the URL of the newly created row.<br>This value may be null. |

**Figure 1: Android API reference documentation for the `insert` method**

not documented in the *javadoc* comments of the `insert` method. Therefore, as Figure 1 illustrates, in the online reference of the Android API the `IllegalArgumentException` is not included (**undocumented** exception). This means that the developer of a client application that uses the `insert` method is not explicitly informed about a possible crash related to an illegal URL. Novice or careless developers could be totally unaware about this and they will not guard their applications from such crashes—or they would apply the bad practice of catching generic fault-prone exceptions like `Exception` and `Throwable` [56].

Listing 2 shows the result of an application that has been crashed (the stack trace is sanitized according to the methods presented in [32]) because of a wrong URL (see lines 23, 27, 28). Looking at the stack trace (Listing 2) and going back to the API reference (Figure 1), it is not straightforward for one to understand the cause of the crash (the cause is not explained in the documentation.) Thus, we argue, here, that exceptions that can be used for recovering from an application crash should be explicitly declared in the API reference; otherwise, the API reference documentation makes debugging difficult and leaves developers unaware of catching exceptions associated with (recoverable) crashes.

In addition, we *hypothesize* that if an unchecked exception is worth documenting sometime, then, all known might-thrown unchecked exceptions

**Listing 2: Stack trace caused by the `insert` method**

```
1  dalvik.system.NativeStart.main
2  com.android.internal.os.ZygoteInit.main
3  com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run
4  java.lang.reflect.Method.invoke
5  java.lang.reflect.Method.invokeNative
6  android.app.ActivityThread.main
7  android.os.Looper.loop
8  android.os.Handler.dispatchMessage
9  android.view.ViewRootImpl.handleMessage
10 android.view.ViewRootImpl.processInputEvents
11 android.view.ViewRootImpl.handleMessage
12 android.view.ViewRootImpl.deliverPointerEvent
13 android.view.View.dispatchPointerEvent
14 com.android.internal.policy.impl.PhoneWindow$DecorView.dispatchTouchEvent
15 android.inputmethodservice.SoftInputWindow.dispatchTouchEvent
16 android.app.Dialog.dispatchTouchEvent
17 com.android.internal.policy.impl.PhoneWindow.superDispatchTouchEvent
18 com.android.internal.policy.impl.PhoneWindow$DecorView.
       superDispatchTouchEvent
19 android.view.ViewGroup.dispatchTouchEvent
20 android.view.ViewGroup.dispatchTransformedTouchEvent
21 android.view.ViewGroup.dispatchTouchEvent
22 android.view.ViewGroup.dispatchTransformedTouchEvent
23 android.view.View.dispatchTouchEvent
24 com.example.onTouchEvent
25 com.example.methodA
26 com.example.methodB
27 com.example.methodC
28 android.content.ContentResolver.insert
29 !java.lang.IllegalArgumentException
```

## createView

added in API level 1

```
View createView (String name,
                 String prefix,
                 AttributeSet attrs)
```

Low-level function for instantiating a view by name. This attempts to instantiate a view class of the given ***name*** found in this LayoutInflater's ClassLoader.

There are two things that can happen in an error case: either the exception describing the error will be thrown, or a null will be returned. You must deal with both possibilities -- the former will happen the first time createView() is called for a class of a particular name, the latter every time there-after for that class name.

| Parameters | |
|---|---|
| name | String: The full name of the class to be instantiated. |
| prefix | String |
| attrs | AttributeSet: The XML attributes supplied for this instance. |

| Returns | |
|---|---|
| View | View The newly instantiated view, or null. |

| Throws | |
|---|---|
| ClassNotFoundException | |
| InflateException | |

**Figure 2: Android API reference documentation for the `createView` method**

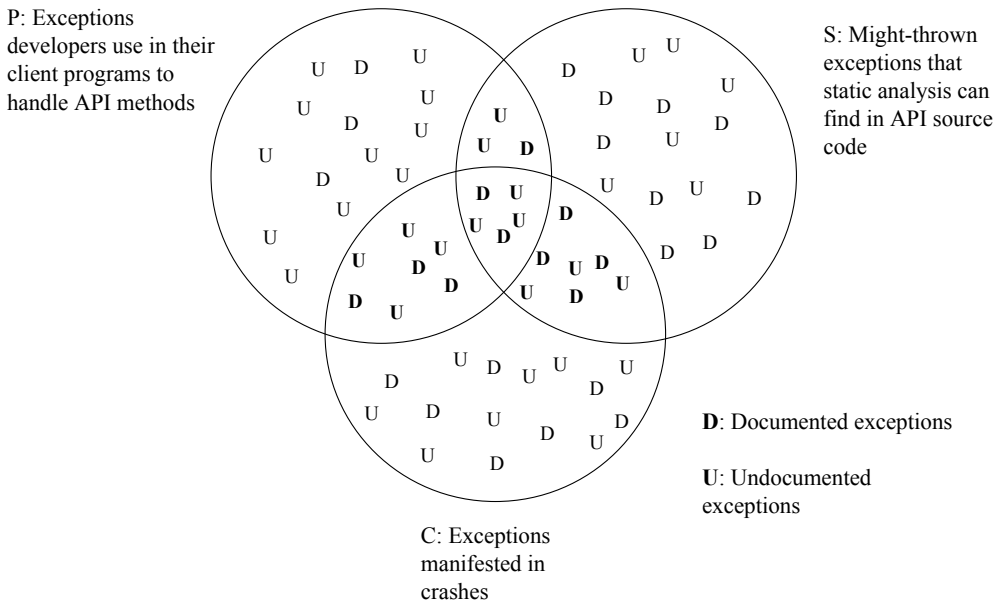



**Figure 3: Examined sets**

should be worth documenting. For instance, for the `createView` method, in Figure 2, the unchecked exception `InflateException` is correctly listed in the API reference (**documented** exception). On the contrary, for the method `insert`, in Figure 1, the unchecked exception `IllegalArgument-Exception` is missing from the API reference; however, it is mentioned in a `throw new` statement (see Listing 1).

## 4. Experimental Setup

In this section, we present the data, the methods, and the metrics we used in our empirical study.

### *4.1. Data*

For our study, we used three different data sources: 1) the Android platform, 2) Android applications, 3) crash data from Android applications. Fig-

**Table 1: Exceptions for Analyzed API Methods**

| Set | Mined Exceptions |
|---|---|
| Documented (D) Exceptions | documented exceptions |
| Undocumented (U) Exceptions | undocumented exceptions |
| App Programs (P) | exceptions (in `try-catch` constructs) used for handling called API methods in client programs (applications) |
| API Source (S) | exceptions (in `throw new` statements) that *intra-procedural* static analysis can find in API source code and propagated might-thrown exceptions that *inter-procedural* static analysis can find in API source code |
| Crashes (C) | root exceptions in crash data and API exceptions in crash data |

ure 3 illustrates our data sets (U and D are elements of the sets of Documented (D) and Undocumented (U) exceptions) and Table 1 explains the elements (**exceptions**) of each set.

Regarding the Android platform, we used ten versions of the Android API, from level 14 to level 23. We chose these API levels for three reasons: 1) these versions were available through the Android SDK manager, 2) from version 14 the Android API differs a lot from previous versions, and 3) the majority of our applications and crash reports were associated with the Android API level 14 or later. We selected the Android platform as the subject of our research, because it is available as open source code and millions of developers around the world use it to develop applications.

In total, from all the examined API versions, we analyzed 309,394 method signatures from `public` classes (with embedded classes) that belong to the `android` package of the Android source code. Table 2 lists the number of analyzed methods per Android API level. In particular, we examined API methods that have `public` and `protected` modifiers, as well as `abstract` methods. However, we excluded native methods because we had limited information about them. As far as exceptions are concerned, we took into

**Table 2: Examined Android Versions**

| API level | API Methods (#) | Abstract Methods (#) | Documented Exceptions (%) | Apps (#) |
|---|---|---|---|---|
| android-14 | 21,392 | 2,206 | 12.4 | 208 |
| android-15 | 21,789 | 2,226 | 12.5 | 399 |
| android-16 | 24,789 | 2,626 | 11.9 | 314 |
| android-17 | 26,085 | 2,677 | 12.2 | 858 |
| android-18 | 28,532 | 3,055 | 11.6 | 381 |
| android-19 | 29,932 | 3,306 | 11.5 | 1,131 |
| android-20 | 31,801 | 3,544 | 11.0 | 175 |
| android-21 | 38,757 | 4,220 | 10.5 | 69 |
| android-22 | 40,056 | 4,562 | 10.5 | 3 |
| android-23 | 46,261 | 5,202 | 10.6 | 1 |

account all the types of the found exceptions (checked and unchecked).

Additionally, we used 3,539 Android applications that use Android API versions from 14 to 23. We downloaded the applications (playdrone-apk-e8) from Archive.org[3] which hosts Android applications from Columbia University's PlayDrone project [57]. We chose to analyze Android applications, in order to find possible crucial undocumented exceptions, because: 1) API designers have to know how developers program client applications and 2) it is easier for API designers to get access to client applications' byte code rather than to runtime crash data. Specifically, from the applications we isolated application and third-party method calls to Android API methods.

Finally, to cross-check our analysis from the Android API and the client applications, we used a set of 901,274 stack traces from Android applications that have been crashed—the crash data set has been also used in our previous work [32].

### *4.2. Methods*

We extracted the sets shown in Figure 3, by: 1) statically analyzing the Android platform's API source code, 2) statically analyzing 3,539 Android applications, 3) parsing Android *javadoc* comments (for Android API levels

[3]https://archive.org/details/android_apps

14–23), and 4) applying heuristic rules to 901,274 stack traces from application crashes. Finally, we applied set operations on the mined API methods and exceptions.

The extraction and the analysis of the documented and undocumented exceptions from the whole dataset of the `.apk` files took 11 hours and 36 minutes. The study run on an eight core (two four-core Intel Xeon E5-1410 processors operating at 2.80GHz machine with 12GB of RAM, and five 1.8TB disks attached through a 6Gb/s PER H 710 PCI Express RAID controller, running the Debian GNU/Linux 8.6 (jessie) distribution. In the following paragraphs, we present our methods in detail (see also Figure 4). We note that we provide as open source the software we wrote for our automated approach (named as eRec).[4] Also, we have made the used datasets publicly available (eRec–data).[5]

To evaluate the effect of undocumented exceptions on applications' stability, we need to know the exceptions that each API method can throw at runtime. This can be generally achieved through testing, model checking, and other static analysis methods, as well as dynamic analysis. In our case, we consider only exceptions that can be found from static analysis, since exceptions that dynamic analysis can give (e.g. `OutOfMemoryError` and `IllegalStateException`) are dependent on the applications' context and such exceptions are not typically listed in the API reference—because most of the times the crashes that these exceptions manifest themselves in are unrecoverable.

*4.2.1. Android API*

For the static analysis of the source code of the Android platform's API, we wrote a script and added appropriate options to Soot in order to: 1) state the analysis of the Android APIs, 2) force the extraction of might-thrown exceptions from each API method, and 3) determine the representation of each examined API class. Then, Soot gave us `jimple` files including the results of the intra-procedural analysis (i.e. exceptions that each method can throw) for each API method. Given that Soot can apply only intra-procedural analysis, we wrote a program in order to get exceptions from inter-procedural analysis (i.e. propagated exceptions that can be thrown among API method

[4]https://github.com/mkechagia/eRec.git
[5]https://github.com/mkechagia/eRec-data.git

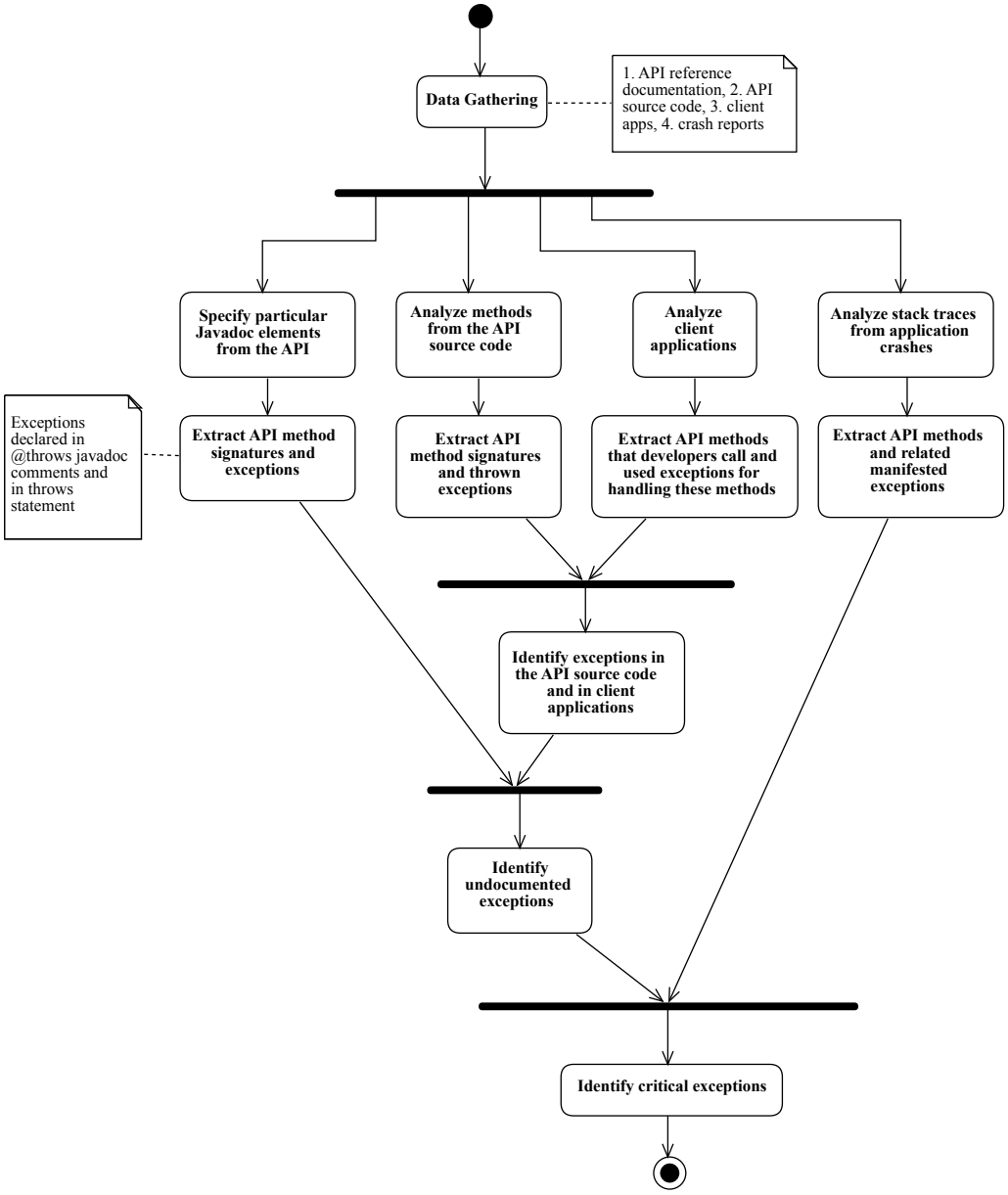



**Figure 4: Overview of the analysis**

**Algorithm 1** Extract methods and exceptions from API source code

```
1: Input: Soot files (.jimple) for API
2: Output: pairs of API methods and exceptions
3:
4: procedure ANALYZE_CODE(API)
5:     # graph for methods and exceptions
6:     G = nx.Graph()
7:     # directed graph for the edges
8:     DG = nx.DiGraph(G)
9:     for each line in file do
10:        # in case of a new method signature
11:        if (is_new_method) then
12:            G.add_node(method)
13:        end if
14:        # in case of throw new in method body
15:        if (is_throw_new) then
16:            G.node[method] ← append(exception)
17:        end if
18:        # in case of a called API method
19:        if (is_called_method) then
20:            if called_method not in G.nodes() then
21:                G.add_node(called_method)
22:            end if
23:            DG.add_edge(method, called_method)
24:        end if
25:    end for
26: end procedure
```

calls). Algorithm 1 presents the pseudocode of our technique.

In addition, to mine the Android platform's API reference documentation we wrote a program using the Java doclet API[6] and we specified particular *javadoc* elements for each API method [58].

In Table 1, S represents the exceptions we extracted for each API method, from the `.jimple` files, using intra and inter-procedural analysis. The D sign refers to documented exceptions listed in the Android platform's API reference documentation and the U sign refers to undocumented exceptions.

*4.2.2. Android applications*

For the static analysis of the Android applications we used the Soot framework [24] and its submodule, Dexpler [59]. Then, to extract API methods and exceptions from client applications, we wrote a script in Python. Algorithm 2 presents the pseudocode of our algorithm. In Table 1, P refers to Android API methods and exceptions that we extracted from the `.jimple` files (that

[6]http://docs.oracle.com/javase/7/docs/jdk/api/javadoc/doclet/

**Algorithm 2** Extract API methods and exceptions from apps' byte code

```
1: Input: Soot files (.jimple) for each app
2: Output: pairs of API methods and exceptions
3:
4: procedure ANALYZE_CODE(app)
5:     # keys: label, values: API methods and exceptions
6:     m_dict
7:     # initialize current label
8:     label = 0
9:     for each line in file do
10:        # in case of a new method signature
11:        if (is_new_method) then
12:            label = 0
13:        end if
14:        # in case of a new label in method
15:        if (is_new_label) then
16:            m_dict ← m_dict[label]
17:        end if
18:        # in case of a called API method
19:        if (is_api_method) then
20:            m_dict[label] ← append(api_method)
21:        end if
22:        # in case of exception in the current label
23:        if (is_exception) then
24:            m_dict[label] ← append(exceptions)
25:        end if
26:        # in case of catch clause
27:        if (is_catch) then
28:            # update exceptions
29:            locate_labels_in_catch()
30:            m_dict[label] ← append(exceptions)
31:        end if
32:    end for
33: end procedure
```

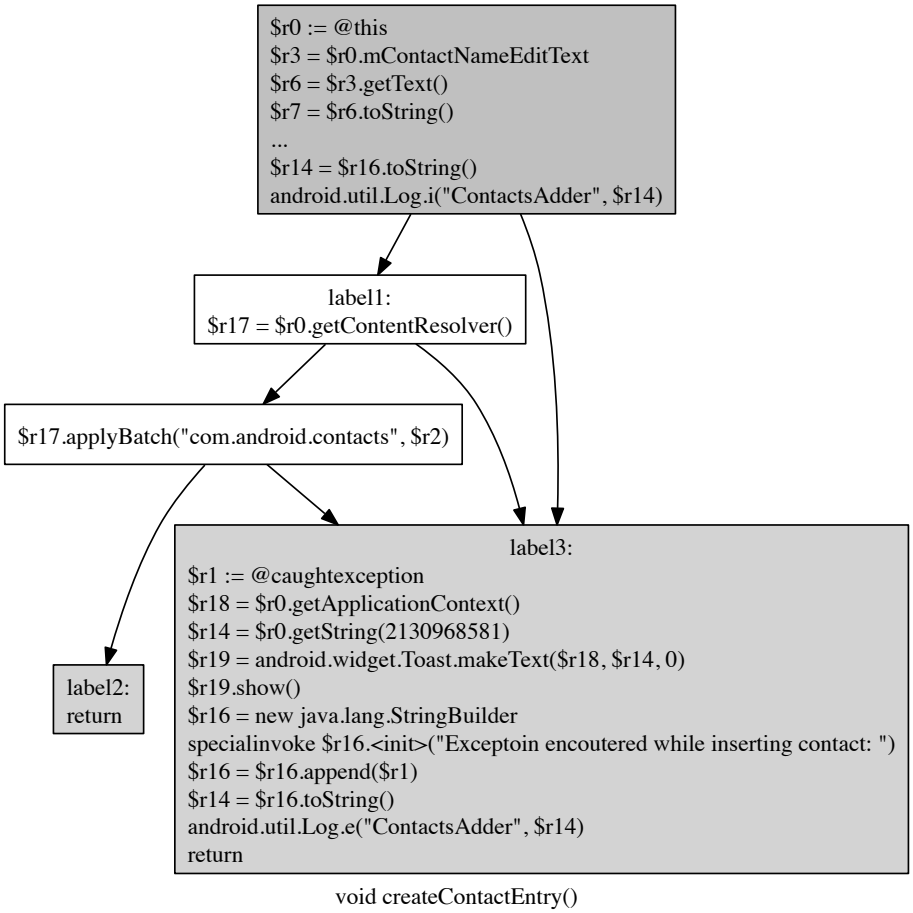



**Figure 5: Control flow graph for an app method**

Soot produces). Figure 5 graphically represents the structure of a method in a `.jimple` file. The graph shows that, the API method (`applyBatch`) is called in the body of an application method, `createContactEntry`, at `label1`. Then, the flow goes to `label3` where there is a caught exception. This means that the `applyBatch` method is handled in a `try-catch` construct.

*4.2.3. Stack Traces*

We received the stack traces from a private company, called BugSense (now it is part of Splunk),[7] which gathered crash reports through a centralized management system. Having a set of almost 1 million crash reports from Android applications presented in [32] (different from the 3,539 Android applications we analyze in Section 4.2.2, we associated risky API methods, which had caused the crashes, with the manifested root exceptions of the stack traces. In Table 1, C represents the associated exceptions with the found API methods in the stack traces. We used these pairs of API methods and exceptions to cross-check the found results from the Android platform's API and the client applications. Additionally, by analyzing the source code and the documentation of the Android platform's API, it was easy for us to find, for each pair of API method and exception retrieved from the stack traces, which of the manifested exceptions were *not* listed in the API reference documentation of the failed APIs.

To filter the stack traces, we applied the heuristic rules we present in [32]. Briefly, the stack traces we used consist of method call frames from the Android framework that lead to an exception, through an application and an API call. For the identification of *risky* API calls that lead to application crashes, one needs to locate the last instance of a call from an application to an API. By pinpointing such risky API calls we can find API methods that are possibly responsible for application crashes. For instance, in Listing 3, the interesting API call is that to the `setContentView` method, rather than the one to the `loop` method. Then, taking also into account the root exception and the exceptions from the remaining frames (for chained stack traces), we can get the concise information regarding a crash cause.

[7] https://www.splunk.com/

**Listing 3: Method calls sequence [58]**

```
com.example.Serialize$Looper.run
android.os.Looper.loop
android.os.Handler.dispatchMessage
com.example.SerializeHandler.onMessage
com.example.app.Activity$1.work
android.app.Activity.setContentView
```

### *4.3. Metrics*

We have applied set operations on the sets of exceptions for the examined API methods (see Figure 3) to pinpoint exceptions that should be documented. For instance, PCSU (i.e. PCSU = P ∩ C ∩ S ∩ U) refers that the same pairs of API methods and unchecked exceptions that appear in our three sets: applications (P), crashes (C), and API source code (S), are **not** in the API reference documentation (U).

When an exception for a specific API method belongs to set PCSU the argument that the exception should be listed in the documentation of the API method is strong, whereas, if an exception belongs only to one set (e.g. PU = P ∩ U, CU = C ∩ U) the evidence that the exception should be documented is less stronger. However, we assume that if an exception is declared as might-thrown (in a `throw new` statement) in the API's source code (i.e. S), then, it is meaningful for the exception to be also documented. Finally, it is more crucial for exceptions that belong to PSU or CSU to be documented than for the exceptions that belong to SU alone—the exceptions in PSU are also caught by developers and in CSU exceptions also manifest themselves in crashes.

### *4.4. Validation*

To show that our automated techniques, illustrated in Figure 4, can help in the assessment of APIs regarding exceptions, we validated our approach by extending our experiments to cover other sets of Java APIs. Table 3 lists the publicly available software and data we used to demonstrate the generality of the proposed technique. We selected popular Java libraries that stem from: 1) the Java API itself, 2) the Apache ecosystem (Commons, HttpComponents, Groovy, and Felix), as well as from 3) Google (Guava and Guice).

**Table 3: Java Libraries**

| API | Methods (#) | Documented Exceptions (%) |
|---|---|---|
| java–7 | 12,757 | 37.8 |
| java–8 | 15,485 | 37.9 |
| commons–io–2.5 | 849 | 55.6 |
| commons–lang3–3.6 | 2,343 | 25.1 |
| commons–collections–3.2.1 | 2,375 | 24.9 |
| httpcomponents–client–4.5.3 | 1,146 | 23.3 |
| groovy–2.4.5 | 1,502 | 6.5 |
| felix–2.0.2 | 496 | 4.2 |
| guava–19.0 | 3,050 | 25.7 |
| guice–4.0 | 724 | 4.0 |

Table 3 shows the percentages of API methods with documented exceptions listed in the documentation reference of the examined APIs. We found the reported percentages by using the doclet presented in Figure 4 and a Python script to extract from the APIs public API methods and documented exceptions. In contrast to Android (see Table 2), we identified that more than 35% methods in the Java API have documented exceptions, whereas for popular third-party libraries, including Apache Commons and Google collections (i.e. Guava), the number of the documented exceptions drops to 25%. Interestingly, the commons–io library has the highest number of documented exceptions (more than 50%). This possibly occurs because exceptions, such as the `IOException` are checked and they are always in the documentation. The large variance in the results may indicate that the number of documented exceptions depends on the application domain and architectural decisions, such as the type of API methods.

Furthermore, we extended our experiments by applying our metrics (discussed in Section 4.3) on 182 Java projects to evaluate how developers handle methods from the Java API. We examined 133 Java projects from the Maven Repository[8] and 49 projects from the DaCapo benchmark suite.[9] Given that we did not have stack traces from these projects, we used stack traces from Android application crashes caused due to deficiencies associated with meth-

[8]https://mvnrepository.com/
[9]http://www.dacapobench.org/

**Table 4: Undocumented Exceptions (of the commons–io) Used in Java Projects (PU = P ∩ U)**

| API Method | Exception |
|---|---|
| output.ByteArrayOutputStream.toByteArray | IOException |
| output.DeferredFileOutputStream.getFile | IOException |
| output.ByteArrayOutputStream.toByteArray | IllegalArgumentException |
| EndianUtils.swapInteger | NoClassDefFoundError |
| EndianUtils.swapInteger | IOException |

**Table 5: Undocumented Exceptions in Java Projects (P), Java API Source Code (S) and Crashes (C), (PCSU = P ∩ C ∩ S ∩ U)**

| API Method | Exception | Crashes (#) |
|---|---|---|
| lang.String.substring | StringIndexOutOfBoundsException | 1,168 |
| lang.StringBuilder.append | OutOfMemoryError | 1,102 |
| lang.StringBuffer.append | OutOfMemoryError | 112 |
| lang.String.getBytes | UnsupportedEncodingException | 42 |
| lang.String.charAt | StringIndexOutOfBoundsException | 23 |

ods from the Java API. To validate our metrics, we concentrated on the Java API for several reasons. First, because the Java API is comparable to the Android one, in terms of magnitude, complexity, and popularity. Second, because the Java API is the most widely used among the publicly available Java libraries. Third, because we had sufficient available data (Java projects and stack traces) to use as inputs in our method.

However, in order to show that our approach also works for third-party libraries, we analyzed four Java projects from our data set that use commons–io, the most popular Apache Commons library. Through this exercise we found undocumented exceptions that developers handle in their projects (see Table 4). We are, also, planning to conduct a large-scale study on the dependencies (Java and third-party libraries) of Java projects and related crashes (e.g. extracted from Apache Jira).[10]

Table 6, summarizes the percentages of the undocumented exceptions we found after applying the metrics presented in Section 4.3 on sets (see Figure 3) from the Android and Java ecosystems (for all the API versions

[10]https://issues.apache.org/jira/secure/Dashboard.jspa

**Table 6: Results from Android and Java APIs**

| | Methods with Undocumented Exceptions | |
|---|---|---|
| Set | Android API (%) | Java API (%) |
| SU | 20.8 | 28.3 |
| PU | 81.2 | 91.6 |
| PSU | 6.2 | 6.1 |
| SCU | 49.7 | 26.3 |
| PCU | 38.4 | 21.8 |
| PCSU | 15.5 | 4.8 |

examined here). In particular, we consider the undocumented exceptions that belong in the PCSU set as documentation bugs. See Table 5 for Java and Table 13 for Android APIs.

## 5. Empirical Results

In the following paragraphs, we discuss the results we got by applying the metrics described in Section 4.3 on the sets of Figure 3—S (API source code), P (application programs), and C (crash data)—in order to reveal: 1) what exceptions API designers list in the Android platform's API reference documentation, 2) what exceptions developers of Android client applications catch to handle API methods, and 3) what exceptions actually manifest in application crashes.

### *5.1. What Exceptions Do API Designers Document?*

To find what exceptions API designers decide to include in an API reference documentation, we analyzed the Android platform's API reference documentation and source code. Regarding the API documentation, we searched for documented exceptions in `@throws` and `throws` statements in method signatures. We extracted 84,421 **distinct** methods from the `android` package of levels 14–23 of the Android API. From these methods, we used 52,465 valid distinct non-private API methods from the `android` package of all the levels 14–23 of the Android API.

Table 2 lists the number of pairs of API methods and documented exceptions for each Android API level. We found that on average around 12% API methods have documented exceptions among all Android API versions. As API designers enhance the Android platform's source code with new methods,

the number of the documented exceptions remains almost steady. In fact, the percentage of the methods with documented exceptions drops 2%. This possibly occurs for three reasons. First, most of the times, API designers do not have stack traces from client applications, at their disposal, to list the manifested exceptions into APIs' reference documentation. Second, API designers are aware about the fact that developers of client applications use development tools that make suggestions (based on the declared exceptions in `throws` clauses) about the exceptions that called API methods might throw and they possibly leave several exceptions undocumented. Third, this could be related to the maturity of the APIs (i.e. how often these APIs have been used and if there are reported issues associated with them.)

Figures 6 and 7 show the evolution of the exception types among the examined versions of the Android platform's API. We examine the API methods that have declared exceptions in `@throws` *javadoc* comments and in their signature (next to the `throws` keyword). We note, here, that from the exception types we have considered in this analysis, we have excluded the generic `Exception`, which we observed that it is mostly thrown (81.6% on average) by test methods (5.7% on average) listed in the API reference documentation. Apart from the test methods, it seems that for the remaining methods of the API reference documentation, API designers avoid to use the generic `Exception`, which reveals a good design practice [12].

Figure 6 shows the evolution of the top ten exception types among the versions 14–23 of the Android platform's API reference documentation. As we expected, there are checked exceptions among the top documented exceptions (`FileNotFoundException`, `IOException`, `NameNotFoundException`, `RemoteException`, and `XmlPullParserException`), since checked exceptions should be always listed in method signatures. However, API designers also list specific types of unchecked exceptions (`IllegalArgumentException`, `IllegalStateException`, `NotFoundException`, `SecurityException`, `UnsupportedOperationException`) in the API reference, as the API evolves.

Figure 7 presents the evolution of the listed checked and unchecked exceptions in the Android platform's API. We can see that the most of the listed exceptions are checked for all the API versions. However, the amount of the documented unchecked exceptions is significant too. From these results, we understand that API designers are interested in identifying and reporting the exceptions API methods might throw at runtime. Thus, it seems that there is a need for tools that can assist API designers in the *systematic* detection of these exceptions.

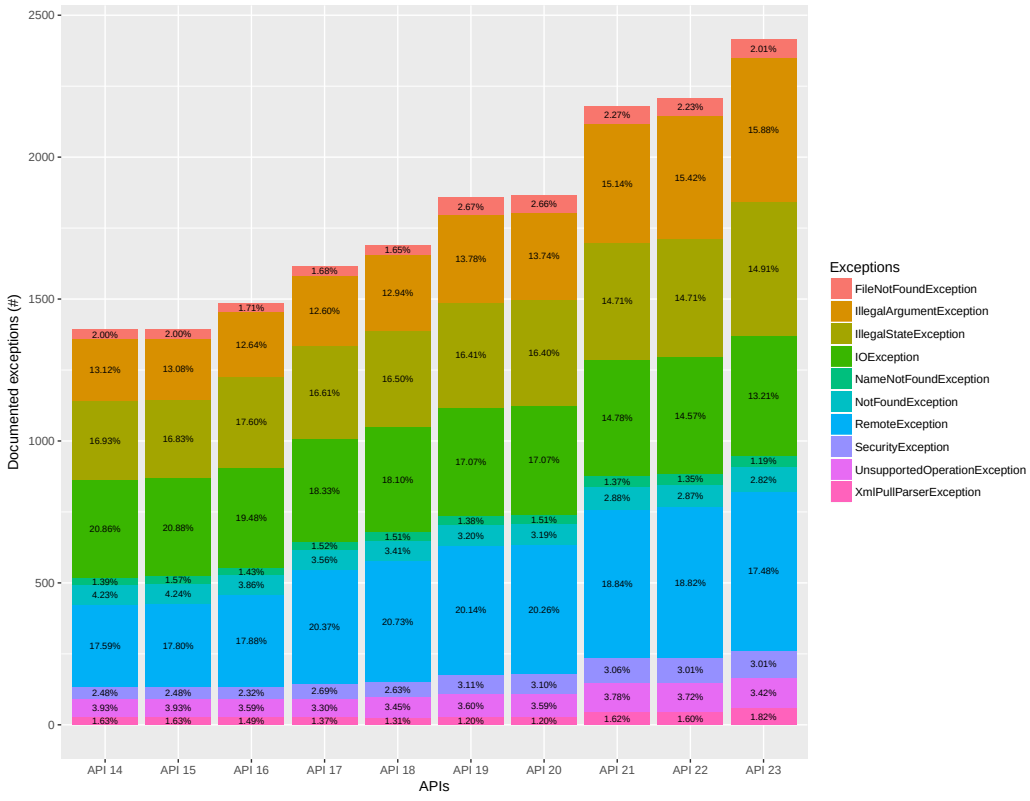



**Figure 6: Evolution of top exception types in the Android API**

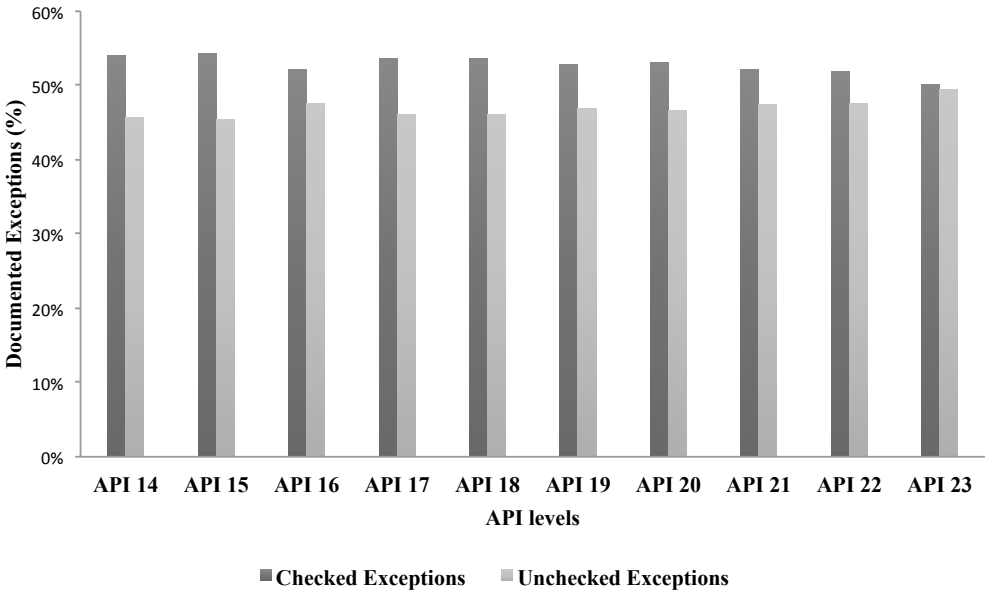



**Figure 7: Evolution of checked and unchecked exceptions throughout the versions of the Android platform's API**

We also examine changes in the exception interfaces of our sample of API methods. Information about APIs that change often can reveal *risky* APIs that may break client applications [53]. To conduct this analysis, we tracked changes (added and removed exceptions) in the exception interfaces (declared exceptions after the `throws` keyword in method signatures [60, 61]) of the API methods found in the Android platform's API source code. Table 7 lists API methods, from the Android platform's API, that their exception interfaces have been updated, in a subsequent API version of that of their first appearance in the Android platform's API. For instance, `media.MediaCodec.createByCodecName(String)` was added to the API level 16, but the `IOException` was declared, in its exception interface, in the API level 21. Here, we note that we wish to conduct, in the future, a further analysis of the changes in the exception interfaces, and see how these alterations may affect the stability of client applications. However, our current results show that the designers of the Android platform's API do not change the exception interfaces quite often. We could say that this is an indicator of the high quality of the Android platform. It reflects designers'

**Table 7: Changes in Exception Interfaces for Android levels 14–23**

| API | Method | Changed Exception |
|---|---|---|
| 15 | net.NetworkStats.subtract(NetworkStats) | + NonMonotonicException |
| 16 | net.NetworkStats.subtract(NetworkStats) | − NonMonotonicException |
| 17 | security.Credentials.convertFromPem(byte[]) | + CertificateException |
| 18 | drm.DrmManagerClient.finalize() | + Throwable |
| | media.MediaExtractor.setDataSource(DataSource) | + IOException |
| | media.MediaExtractor.setDataSource(FileDescriptor, long, long) | + IOException |
| | media.MediaExtractor.setDataSource(String, Map) | + IOException |
| 21 | content.pm.PackageParser.collectManifestDigest(Package) | + PackageParserException |
| | media.MediaCodec.createByCodecName(String) | + IOException |
| | content.pm.PackageParser.collectCertificates(Package, int) | + PackageParserException |
| | media.MediaDrm.openSession() | + ResourceBusyException |
| 22 | graphics.Bitmap_Delegate.createBitmap(BufferedImage, Set, Density) | − IOException |
| | system.Os.fcntlFlock(FileDescriptor, int, StructFlock) | + InterruptedIOException |
| | graphics.Bitmap_Delegate.createBitmap(BufferedImage, boolean, Density) | − IOException |

**Table 8: Examples of Undocumented Exceptions in the Android Platform's API Source Code (SU = S ∩ U)**

| API Method | Exception |
|---|---|
| app.Fragment.startActivity | IllegalStateException |
| graphics.Canvas.drawBitmap | ArrayIndexOutOfBoundsException |
| graphics.Paint.getTextWidths | IndexOutOfBoundsException |
| widget.ImageView.setImageResource | NotFoundException |
| view.LayoutInflater.onCreateView | InflateException |
| view.View.awakenScrollBars | AndroidRuntimeException |
| location.Criteria.setPowerRequirement | IllegalArgumentException |
| text.Layout.draw | NullPointerException |
| content.ContentResolver.requestSync | IllegalArgumentException |
| net.SSLCertificateSocketFactory.createSocket | SSLPeerUnverifiedException |

care of testing their APIs to find corner cases and declare, in the interfaces, related exceptions, from the first time of the publication of the APIs. Finally, we note that we have manually validated that the changes listed in Table 7 have indeed occurred in the Android platform's API.

Analyzing the Android platform's API source code (levels 14–23), we pinpointed 3,347 undocumented exceptions for 2,442 non-private methods (i.e. set SU = S∩U in Figure 3). From the undocumented exceptions, we revealed 6% checked and 94% unchecked exceptions. One would expect that since the checked exceptions are always in the API reference documentation (`throws` in method signatures) all the found undocumented exceptions would be unchecked. However, our approach can indeed find undocumented unchecked

exceptions that are more specific than the already documented ones (as we explain in the following) because our method applies inter-procedural analysis (see Section 4.2.1). Table 8 shows ten representative pairs of API methods and might-thrown exceptions that are undocumented. We also observed that for some methods with checked exceptions, such as `createSocket`, the API uses generic exceptions, such as the `IOException` for instance, whereas from the inter-procedural analysis we can find more specific ones, such as the `SSLPeerUnverifiedException` (child of `IOException`). Similarly to Kery et al., we believe that when a specific type of might-thrown exception is known, the use of specific exceptions is better for applications' debugging [12].

Finally, we saw that API designers declare might-thrown unchecked exceptions in the API source code, but they do not always include them in an API's reference documentation. We also found that API designers rarely document propagated exceptions, as Robillard and Murphy argue in another work [8].

### *5.2. What Exceptions Do Developers Use?*

To find what exceptions developers use to handle API methods, we statically analyzed 3,539 Android applications that use Android API levels 14–23. From these applications, we identified, in 3,348 valid applications, pairs of API methods (from the `android` package of the Android API) and exceptions. In the following paragraphs, we present our results regarding sets $PU = P \cap U$ and $PSU = P \cap S \cap U$ of Figure 3.

In our sample, developers handle 2,584 distinct API methods (with caught documented and undocumented exceptions). From these methods, however, only 12% (i.e. 322 API methods) have listed exceptions in their documentation (for all the Android API levels 14–23). Figure 8 graphically shows that developers mainly catch exceptions for API methods that belong to the `content`, `util`, `os`, `net`, `graphics`, `database`, and `media` Android packages. The percentages refer to the total number of API calls with caught exceptions per package. We found that 118,027 API calls have undocumented caught exceptions and 8,456 API calls that have documented caught exceptions.

Table 9 lists the top ten types of undocumented exceptions that developers catch. The percentages in the table refer to caught exceptions of 2,536 unique calls to API methods. The unique calls' exceptions come to 10,118. For the top used API methods (more than 20%) with undocumented exceptions, client applications' developers catch generic exceptions such as the `Exception`.

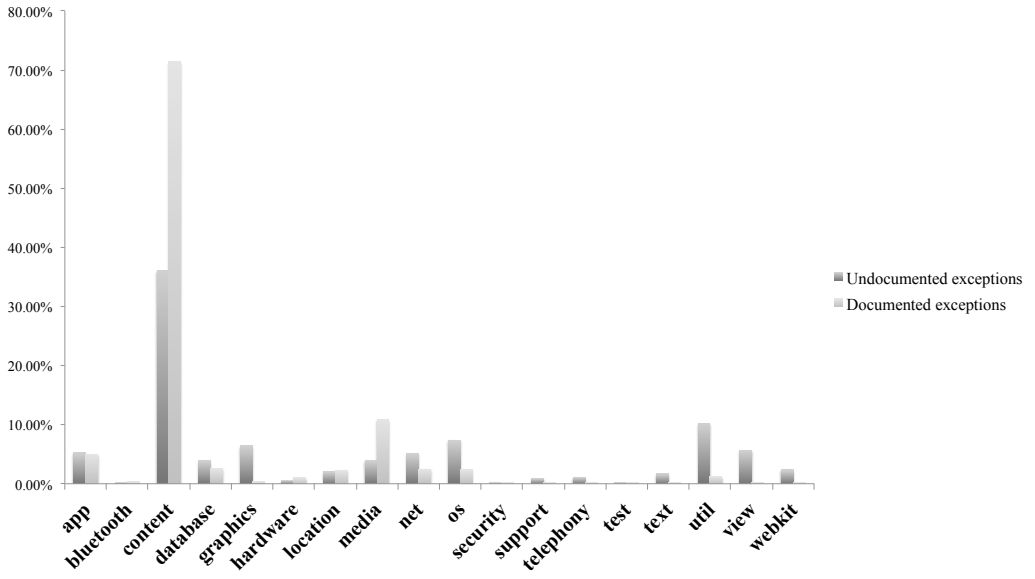



**Figure 8: APIs for which developers mostly use exceptions**

**Table 9: Top Ten Undocumented Exceptions Used in Applications ($\mathbf{PU} = \mathbf{P} \cap \mathbf{U}$)**

| Exception | Frequency (%) |
|---|---|
| Exception | 21.2 |
| IOException | 6.7 |
| JSONException | 5.5 |
| NullPointerException | 4.7 |
| RuntimeException | 3.9 |
| IllegalArgumentException | 3.8 |
| IllegalStateException | 3.1 |
| FileNotFoundException | 2.7 |
| OutOfMemoryError | 2.4 |
| NameNotFoundException | 2.2 |
| Total | 56.2 |

**Table 10: Top Ten Undocumented Exceptions Found in Applications and API Source Code (PSU = P ∩ S ∩ U)**

| API Method | Exception | Frequency in apps (%) |
|---|---|---|
| content.res.AssetManager.open | FileNotFoundException | 10.1 |
| content.res.Resources.openRawResource | FileNotFoundException | 5.5 |
| content.res.AssetManager.openFd | FileNotFoundException | 4.1 |
| net.Uri.parse | RuntimeException | 2.7 |
| database.sqlite.SQLiteDatabase.delete | IllegalStateException | 2.1 |
| widget.TextView.setText | RuntimeException | 2.0 |
| content.ContentResolver.query | RuntimeException | 2.0 |
| content.res.Resources.getConfiguration() | RuntimeException | 1.8 |
| view.View.findViewById | RuntimeException | 1.8 |
| net.ConnectivityManager.getActiveNetworkInfo() | RuntimeException | 1.3 |
| Total | | 33.4 |

Our result that `Exception` is the most significant exception for Android developers agrees with Kery's et al. findings when they investigated the use of exception types in Java programs [12]. The broad use of the `Exception` class reveals that programers need more guidance to prevent their applications from crashes. We argue that automatic assistance can derive from: 1) specific exceptions that are documented in the API reference documentation, 2) static checking of exceptions, 3) programming language's type system, as well as 4) testing and debugging tools. In this paper, we mainly make re-documentation suggestions for the Android platform's API reference and conduct a qualitative study on Android developers, in Section 6, to evaluate documented exceptions' effectiveness.

We also investigated if there are common pairs of API methods and undocumented exceptions in the source code of the Android platform's API and in client applications (PSU). We found 129 unique pairs of API methods and exceptions. Table 10 shows the top ten pairs.

### *5.3. What Exceptions Do Actually Manifest at Runtime?*

To find what undocumented exceptions, associated with API methods, are actually manifested in application crashes, we used a data set of 901,274 crash data [32] and we searched for common pairs of API methods and exceptions in the stack traces (C), source code (S), and applications (P).

From the analysis of the Android API's source code application execution failures, we found 201 unique pairs of API methods and exceptions in the stack traces and in the Android platform's API source code (levels 14–23). However,

**Table 11: Top Ten Undocumented Exceptions Found in API Source Code and Crashes (SCU = S ∩ C ∩ U)**

| API Method | Exception | Crashes (#) |
|---|---|---|
| app.Dialog.show | RuntimeException | 7,890 |
| content.res.Resources.getDrawable | FileNotFoundException | 3,843 |
| widget.Toast.makeText | InflateException | 902 |
| database.AbstractWindowedCursor.getString | CursorIndexOutOfBoundsException | 639 |
| graphics.Canvas.drawBitmap | ArrayIndexOutOfBoundsException | 628 |
| text.Html.fromHtml | RuntimeException | 565 |
| content.ContentResolver.insert | IllegalArgumentException | 552 |
| support.v4.app.Fragment.getString | IllegalStateException | 450 |
| support.v4.app.ListFragment.getListView | IllegalStateException | 346 |
| database.sqlite.SQLiteDatabase.endTransaction | IllegalStateException | 268 |

**Table 12: Top Ten Undocumented Exceptions Found in Applications and Crashes (PCU = P ∩ C ∩ U)**

| API Method | Exception | Crashes (#) |
|---|---|---|
| app.Dialog.dismiss | IllegalArgumentException | 12,637 |
| graphics.BitmapFactory.decodeResource | OutOfMemoryError | 8,766 |
| graphics.Bitmap.createBitmap | OutOfMemoryError | 8,268 |
| app.Dialog.show | BadTokenException | 7,890 |
| graphics.BitmapFactory.decodeStream | OutOfMemoryError | 6,010 |
| app.ProgressDialog.show | BadTokenException | 4,600 |
| widget.ImageView.setImageResource | OutOfMemoryError | 4,506 |
| hardware.Camera.open | RuntimeException | 3,862 |
| content.res.Resources.getDrawable | OutOfMemoryError | 3,843 |
| content.ContentResolver.query | NullPointerException | 1,836 |

these exceptions are not listed in the Android's API reference documentation and we consider them as documentation bugs. Table 11 refers to the set SCU = S ∩ C ∩ U and lists top ten representative pairs of API methods and exceptions that static analysis (intra and inter-procedurally) can locate in the source code of the Android platform's API.

We also compared pairs of API methods and exceptions that can be found in applications and in the stack traces (PCU = P ∩ C ∩ U), but again these are undocumented in all the examined Android API levels (14–23). We observed that developers tend to use exceptions they find in stack traces, since they can check what exceptions are manifested when an application crashes; whereas, API designers have not always that information to add related exceptions in the API. Specifically, in our sample, we found 610

**Table 13: Top Ten Undocumented Exceptions in Applications (P), API Source Code (S) and Crashes (C), (PCSU = P ∩ C ∩ S ∩ U)**

| API Method | Exception | Crashes (#) |
|---|---|---|
| app.Dialog.show | RuntimeException | 727 |
| text.Html.fromHtml | RuntimeException | 565 |
| content.ContentResolver.insert | IllegalArgumentException | 552 |
| database.sqlite.SQLiteDatabase.endTransaction | IllegalStateException | 251 |
| app.Activity.onBackPressed | IllegalStateException | 198 |
| app.NotificationManager.notify | NullPointerException | 148 |
| content.res.Resources.openRawResource | FileNotFoundException | 117 |
| content.ContentResolver.delete | IllegalArgumentException | 117 |
| media.MediaScannerConnection.scanFile | IllegalStateException | 105 |
| database.sqlite.SQLiteDatabase.beginTransaction | IllegalStateException | 103 |

distinct API methods that belong in client programs (P) and in crashes (C) sets, and 234 of these methods that have undocumented exceptions (PCU). Table 12 lists ten examples.

Finally, we searched for common pairs of API methods and exceptions in the Android platform's API source code (S), our Android client applications (P), and the stack traces (C). We found 283 common API methods from which we pinpointed 44 pairs of common API methods and (undocumented) exceptions in the three sets. Table 13 shows the top ten undocumented exceptions of the PCSU = P ∩ C ∩ S ∩ U set.

We note that the numbers of crashes mentioned in Tables 11, 12, and 13 refer to the sample of crashes presented in [32].

### *5.4. Discussion*

Overall, our findings show that there is significant space for improvement regarding how API designers build and maintain large APIs (such as Android). Here, we summarize the implications of the findings presented in Section 5 and we give guidelines regarding the right use of different types of exceptions.

**Summary and implications**. Concerning API *designers*, we discovered that they do not document all the thrown exceptions declared in `throw new` statements in the APIs' source code. We also confirmed related work's insights that API designers rarely document propagated exceptions [8]. The referred findings maybe result from the fact that: 1) the majority of API designers lack stack traces from client applications that have crashed—which the former can consider in order to improve the reliability of their APIs, and 2) there is limited use of tools that can predict possible application crashes.

This study can help API designers to make the right design and implementation choices concerning risky API methods (see Table 13). Even though this paper refers to the Android platform's API, we present automated techniques (see the algorithms in Section 4.2 and Figure 4) that can help in the assessment (regarding exceptions) of other APIs.

As far as client applications' *developers* are concerned, we found that they mostly use generic exceptions ($> 20\%$). Also, only 12% of the exceptions they use are documented. It seems that they mostly handle exceptions based on the stack traces they get from their applications' crashes.

Providing developers with more informative documentation and suggestions for might-thrown exceptions related to risky API methods, programming can become more productive and flawless. Our goal is to inform client applications' developers—at production level—about risky API methods that can cause application crashes. We envisage that our approach can be incorporated in integrated development environments (IDEs), such as Eclipse, so that developers can receive on the fly recommendations about possible runtime exceptions that risky API methods can generate.

**Guidelines**. Table 18 lists might-thrown exceptions that eRec finds that should be caught by developers for implementing robust applications. Taking into account the types of the reported APIs and the exceptions, one can consider to protect similar APIs by handling relevant exceptions. In the following, we summarize our guidelines.

- API methods that may throw an exception when they run out of resources should use checked exceptions, in resource-constrained systems (i.e. mobile devices). This applies for instance in programs that deal with bitmap images.

- An API method should *document* any exception (including propagated exceptions [61, 8]) which it might throw.

- API methods that can potentially receive external inputs should be declared with checked exceptions. For instance, consider when creating a `new URL` and the input comes from the user. The value given by the user could be malformed causing a runtime error. Here, the use of a checked exception can prevent this error and recover the application.

- API methods that receive statically checked inputs (e.g. constant literals) should throw unchecked exceptions. The validity of such APIs can

**Table 14: Study Design**

| Yate's Notation for treatment | Factor A Exception Type | Factor B Documentation |
|---|---|---|
| (1) | − (unchecked) | − (undocumented) |
| $a$ | − (unchecked) | + (documented) |
| $\{b\}$ | $\{+\}$ | $\{-\}$ |
| $ab$ | + (checked) | + (documented) |

be verified at compile-time. For instance, consider when compiling a pattern to a regular expression and the pattern is given by the developer. The developer can correct any invalid pattern before releasing the software. Thus, there is no need for checked exceptions, which can decrease code readability [13, 14, 15].

- Indexing-related APIs can throw unchecked exceptions, which are typically resulted by programming errors. In such cases, Java conventions call for throwing unchecked exceptions. However, these API methods should be verified through unit tests, too.

- When APIs interact with the environment (i.e. they are used to open files, connect to databases, etc.) and synchronization issues can arise, relevant API calls should be used with checked exceptions. We propose this in cases where: these API methods receive external inputs, which can be possibly malformed, and possible related input errors are recoverable.

## 6. Validation

In the previous section, we argued that the API methods mainly listed in Table 13 should have documented exceptions in their API reference. Here, we present a qualitative study we conducted to evaluate our arguments.

### *6.1. Experimental Design*

We conducted a randomized controlled trial (RCT) with a 2x2 (two-level) factorial design. RCT has been also used in related work for the testing of coding practices [21, 62]. We examine the effect of two independent variables: **exception type** (having as treatments unchecked and checked exceptions)

**Table 15: Descriptive Statistics of the Participants**

| Education Level | | Programming Experience | | Android Expertise | |
|---|---|---|---|---|---|
| Degree | N | Years | N | Level | N |
| BSc | 15 | <3 | 5 | None | 5 |
| MSc | 7 | 3 − 6 | 9 | Medium | 8 |
| PhD | 3 | 6 − 9 | 8 | High | 7 |
| Other | 0 | >9 | 3 | Professional | 5 |

and API **reference documentation** (having as treatments documented and undocumented exceptions) on the dependent variables of **task completion** (successful built) and **program robustness** (in terms of handled exceptions). Table 14 graphically shows our design, using Yate's notation [63].

Regarding the treatments in Table 14, combination (1) refers that all factors are at their lowest values. Given that our values are not numerical, (1) is our control with lowest values denoted with − (i.e. refers to undocumented and unchecked exceptions). Each of the (1), $a$, $b$, and $ab$ represents a combination treatment of the two independent variables.

Even though in Table 14 we have four treatments, we used the incomplete version of the 2x2 factorial design, because the $b$ treatment (checked and undocumented exceptions) is not applicable. Incomplete factorial design is a design proposed in cases where it is not reasonable, interesting, and feasible for researchers to study all treatment combinations required in a complete or fractional factorial design [64]. Thus, our experimental design consists of three groups. All subjects were randomly assigned to one of these three groups and, finally, each group had approximately the same number of participants.

The participants of our study were 16 professionals and 9 students that have experience in the development of Android and Java applications. Four students (one undergraduate, two master, and one PhD student) took part in a pilot experiment that we conducted to test our subjects' tasks. Table 15 presents the demographics of the participants of our study. In particular, with the term "Programming Experience", we mean experience in writing at least Java programs. Also, in the "Android Expertise" column, we define as *medium* expertise, participants' experience in writing small Android applications (e.g. in a programming course), whereas we state as *high* expertise, participants' experience that have developed and launched at least one An-

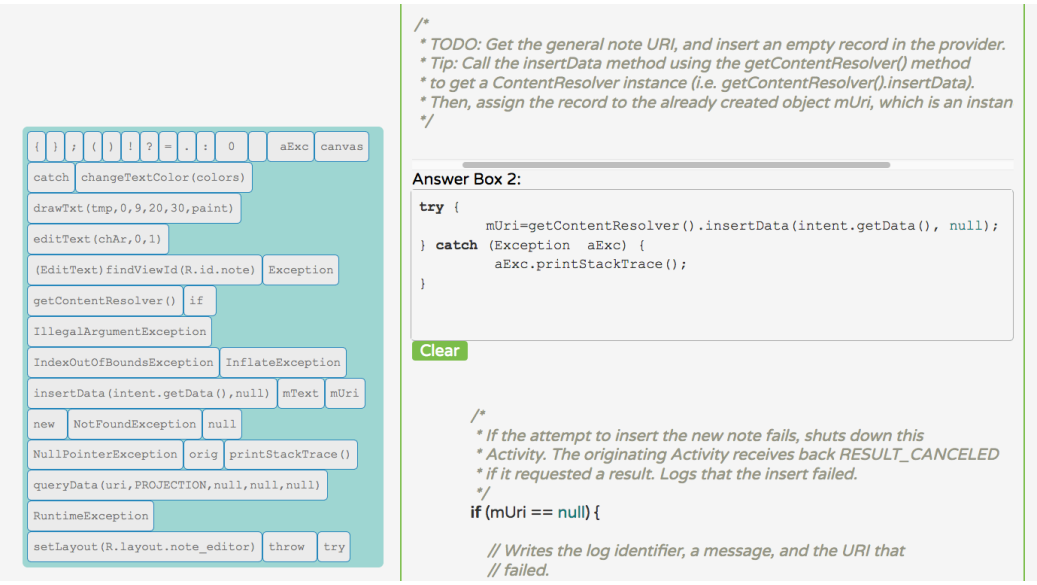



**Figure 9: Android survey**

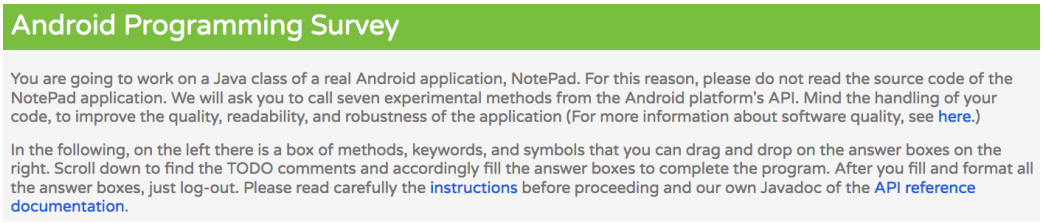



**Figure 10: Initial instructions of the survey**

droid application. Finally, with the term *professional* expertise, we refer to participants that work as Android software engineers in the industry. Our trial took place from the 30th of September until the 15th of October, 2016.

### *6.2. Survey Tool*

To conduct our trial, we developed an open source software web survey tool[11] where participants first signup to fill some demographic details and then access the survey [65]. We implemented three versions of the same survey, related to the treatments of Table 14. When one signs up, the sys-

[11]https://github.com/mkechagia/android-survey

**Table 16: Method Mapping**

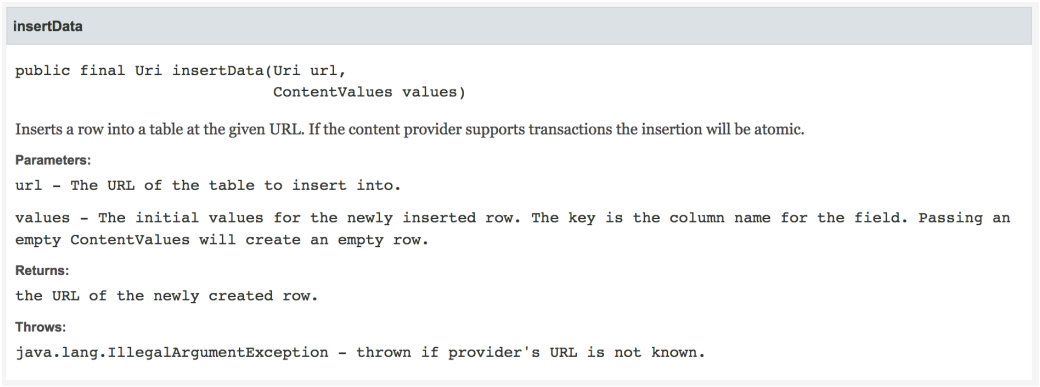
| Real method | Experimental method | Experimental exceptions in (1), *a* | Experimental exceptions in *ab* |
|---|---|---|---|
| drawText | drawTxt | IndexOutOfBoundsException | InvalidArrayIndexException |
| insert | insertData | IllegalArgumentException | InvalidUriException |
| setContentView | setLayout | InflateException | InsufficientMemoryForResourceException |
| findViewById | findViewId | NotFoundException | MissingResourceException |
| setTextColor | changeTextColor | NullPointerException | NullResourceException |
| query | queryData | RuntimeException | UnauthorizedAccessException |
| setText | editText | IndexOutOfBounds | InvalidArrayIndexException |

tem randomly assigns them to one of the three survey types. All subjects worked on a Java class (NoteEditor.java) from a simple Android application, NotePad, where they were asked to handle seven **experimental** methods through `TODO` comments (see Figure 9). We instructed the participants to handle the experimental API methods in a way that will ensure the robustness of the given application (see Figure 10). The whole task for each participant took almost 30 minutes.

The differentiate factors of the survey for the three groups was the given API reference documentation and the types of the exceptions that the subjects could use to complete the Java code. To ensure that participants will only use the given API reference, we invented seven experimental API methods that resemble Android API's methods and we added the methods to the Android platform's API source code. Then, we generated three different *javadoc* references.

The first group received the current version of the Android platform's API reference documentation, with unchecked and undocumented exceptions, the second group received an API reference documentation with documented (listed) unchecked exceptions, and the last group received an API reference documentation with checked exceptions. Table 16 presents the mapping between the real Android methods and the experimental methods and exceptions we gave to the participants. The selected methods and exceptions come from the empirical results of Section 5 and mainly from the PSCU set.

To gather the metrics of our survey (presented in Table 17) we used several techniques. First, we used timestamps to track when the subjects switch the documentation pages (DS), such as listing in Figure 11. Also, we gave to the participants the opportunity to build the program and resubmit their answers until they got the successful build message (see Figure 12). Thus, we also counted the number of successful builds per participant (SB). Finally, we used heuristic rules to automatically parse the submitted answers



**insertData**

```
public final Uri insertData(Uri url,
                            ContentValues values)
```

Inserts a row into a table at the given URL. If the content provider supports transactions the insertion will be atomic.

**Parameters:**

`url - The URL of the table to insert into.`

`values - The initial values for the newly inserted row. The key is the column name for the field. Passing an empty ContentValues will create an empty row.`

**Returns:**

`the URL of the newly created row.`

**Throws:**

`java.lang.IllegalArgumentException - thrown if provider's URL is not known.`

**Figure 11: Generated documentation**

## Compiler output

```
[subant] No sub-builds to iterate on
[javac] warning: [options] source value 1.5 is obsolete and will be removed in a future release
[javac] warning: [options] target value 1.5 is obsolete and will be removed in a future release
[javac] warning: [options] To suppress warnings about obsolete options, use -Xlint:-options.
[javac] Note: /home/maria/android-survey/NotePad/src/com/example/android/notepad
/NoteEditor.java uses or overrides a deprecated API.
[javac] Note: Recompile with -Xlint:deprecation for details.
[javac] 3 warnings

BUILD SUCCESSFUL
Total time: 10 seconds
```

View source code

Press refresh on your browser to see the last source code version on a new tab.

Close the popup.

**Figure 12: Successful build of an Android application**

**Table 17: Study Results for SB: Successful Builds, DS: Documentation Switches, CE: Caught Exceptions, TE: Thrown Exceptions, NC: Null Checks, and NHE: Number of Handled Exceptions**

| Groups | (1): Unchecked and Undocumented | | | | | | *a*: Unchecked and Documented | | | | | | | | | | *ab*: Checked | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Subject (#) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
| SB | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ |
| DS | 0 | 25 | 0 | 35 | 1 | 38 | 8 | 20 | 0 | 0 | 37 | 0 | 3 | 13 | 4 | 0 | 0 | 0 | 18 | 14 | 20 | 0 | 0 | 0 | 6 |
| CE | 0 | 2 | 0 | 3 | 0 | 0 | 0 | 2 | 3 | 2 | 7 | 0 | 1 | 4 | 1 | 2 | 1 | 0 | 7 | 7 | 0 | 0 | 5 | 7 | 7 |
| TE | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| NC | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| NHE | 0 | 5 | 0 | 3 | 0 | 0 | 1 | 3 | 6 | 3 | 7 | 0 | 1 | 4 | 1 | 2 | 4 | 0 | 7 | 7 | 0 | 0 | 6 | 7 | 7 |

to find where methods have been handled with exceptions.

### *6.3. Experiment Results and Observations*

In Table 17 we present the results of the randomized controlled trial. The dependent variables are: 1) SB: shows whether the built was successful (task completion), and 2) NHE: represents the number of handled exceptions (through catching and throwing other exceptions or using null checks).

Looking at the first metric, successful builds (SB), we can see that all subjects from treatment (1) (undocumented and unchecked) and treatment *a* (documented and unchecked) successfully built the application, whereas only 33% of the subjects of treatment *ab* (checked exceptions) managed to successfully built it. Even though most developers of the treatment involving checked exceptions were not able to complete the task of handling them, those who managed to successfully build the application also wrote the most robust programs. Interestingly, two novice Android developers (one student and one practitioner) who managed to build the application, said that they liked that the compiler checked their programs' exception handling so that they can avoid related crashes at runtime.

To examine whether the subjects read the documentation, we used the values of the DS (documentation switches) metric. We first applied one-sample t-test to check if the true mean is equal to 0. We got a low p value (0.0009) from the test and therefore rejected the null hypothesis. This result implies that the subjects of our sample read the given documentation. In addition, we used the two-sample Mann-Whitney-Wilcoxon test to check if some groups read the documentation more often. For all tested group combinations, the p value was greater than 0.05. Thus, we accept the null hypothesis implying that developers read the same amount of documentation in all groups.

Given that our experiment is based on a 2x2 factorial design, we could use a standard analysis technique for this design, two-factor ANOVA [63]. Our two factors are: exception type and documentation (see Table 14). To use two-way ANOVA, however, one needs to test two assumptions first: the homogeneity of variance among the treatment groups and that the residuals are normally distributed.

To validate the homogeneity of the three groups variances, we applied on the values of the number of handled exceptions (NHE) metric the Levene test [66]. The Levene test's p value was 0.44 which is greater than 0.05, and equal variances can be assumed. However, it is easily observable from the values of NHE that the residuals of our sample are not normally distributed. Thus, we decided to use the Kruskal-Wallis test for one-way analysis of variance [67]. This test does not assume that the groups should follow a normal distribution and it can detect if the means of the values of different groups are statistically similar.

The Kruskal-Wallis test gave p value 0.14 meaning that the three groups of our experiment have equal means. In other words, no one of the treatments have greater effect on exception handling (i.e. NHE) that developers conduct to prevent their programs from crashes.

One explanation for this could be that developers do not carefully read the documentation for possible exceptions associated with the API methods they call. In fact, one experienced Android developer told us that he handles possible crash causes when the IDE suggests it and he does not pay much attention to the API reference documentation.

From the values of the metrics DS (documentation switches) and NHE (number of handled exceptions) we can deduce that developers often ignore the API documentation. Taking also into account the values of the SB (successful builds) metric we observe that developers who caught all the crash-causing exceptions and successfully built the application came from the third group (those that were given checked exceptions). Accordingly, we must revise our argument regarding the undocumented exceptions that we reported in Section 5. We propose that exceptions that demonstrably lead to crashes should become checked.

Based on previous works [3, 68, 69], we also think that applications' robustness could increase even more if some exceptions that called API methods can throw become checked. Here, we have already validated the exceptions of methods in Table 16; i.e. one could convert the exceptions of column Experimental exceptions in (1), *a* to checked (such as in the column Ex-

perimental exceptions in *ab*) to increase applications' robustness. We recommend that all the exceptions in the PSCU set (the real methods listed in Table 16 also come from this set) should become checked. Alternatively, to handle these problems type systems can be extended using tools such as the Checker Framework [70].[12] This also agrees with the results of previous studies [20, 21] that consider static type systems a form of implicit documentation. In such a case, further studies would be needed to investigate how the proposed changes would affect developer productivity and code maintainability.

## 7. Threats to Validity

In this section, we discuss the internal validity of our techniques, the external validity of our findings regarding the examined data sets and our trial's subjects, as well as the construct validity of our survey's design and the reliability of our study.

### *7.1. Internal Validity*

Internal validity refers to possible issues of our techniques that can lead to false positives and imprecision. We used specific Android API levels (14–23), associated with the Android API versions that our Android applications use. This implies that we have yet to analyze the remaining methods of the Android API and relevant applications and crash data, so that we can paint the whole picture of the evolution of the Android platform.

Also, given that we had only the `.apk` files of the client applications, we analyzed only the `.jimple` files that Soot produces. This means that maybe there are missing files from Android applications that we did not examine. However, due to the volume of our applications, we think that we did not lose precision in our results.

In addition, not all the methods that static analysis finds will actually be called always. We recognize that this can lead to the over-approximation of our results. To find, however, the most risky methods that should be handled with specific exceptions, we used for validation a set of Android applications and a set of stack traces from application crashes. Furthermore, in our analysis we did not consider dynamic reflection calls that can cause execution failures, too. We aim to investigate such crashes in a future work.

[12] https://checkerframework.org/

Finally, the use of heuristic techniques to pinpoint critical exceptions implies that there could be exceptions that our pattern-matching approaches do not evaluate. To validate our results (i.e. found undocumented exceptions), we validated the Android API reference documentation and our results manually. Thus, we could not exclude the possibility of human error.

### *7.2. External Validity*

External validity refers to the extent to which the results of our study can be generalized to other APIs. As we examined the APIs of a specific platform, Android, and we used particular stack traces, from Android applications, our results are related to the Android API. However, we argue that because we conducted a large scale analysis on the Android ecosystem our guidelines (see Section 5.4), which stemmed from our results, can be generalized for other Java APIs, too. We have also conducted an empirical study in Section 4.4 to prove that our approach presented in Figure 4 can be applied on other Java APIs.

In addition, the number of subjects that participated in our trial can be considered small. Nevertheless, as we have seen in related work [20, 21, 62], such a number is common for studies where developers are asked to complete programming tasks (e.g. to write a piece of code).

### *7.3. Reliability Validity*

Reliability validity refers to the repeatability of our study. For this, we have made our source code (eRec)[13] and data (eRec-data)[14] publicly available. However, we acknowledge that in contrast to the source code of the Android API and the Android applications, we cannot provide the whole dataset of the stack traces used in our study, because these data come from an industrial partner. Our methods are generic though, and they can be applied on any data set of Java stack traces provided by the user. Finally, regarding our developers' survey, we have made the source code of our Android survey tool[15] publicly available for a possible study with more developers, in the future.

[13]https://github.com/mkechagia/eRec
[14]https://github.com/mkechagia/eRec-data
[15]https://github.com/mkechagia/android-survey-tool

### *7.4. Construct Validity*

Construct validity refers to any possible bias in our experimental design. Regarding the developers' study (see Section 6) the reader should take into account that our results are susceptible to the following two factors. First, our trial is dependent on the expertise of our trial's subjects. To limit any related bias, we tried to have in our experiment developers from different levels of expertise—from novice to experts—(see Table 15). Using our Android survey tool, we wish to run similar trials on groups with different levels of expertise, in the future, to further validate our results. Second, our trial could be susceptible to the participants' behavior (i.e. whether they all had the same and right information for conducting the survey and so on). To avoid any possible bias, we conducted an online survey and we asked from all the participants of each group (see Figure 10) to use the provided experimental API reference documentation and build a simple Android application.

## 8. Conclusions

In this paper, we investigated the exception handling of the Android platforms' API to understand when and how developers use exceptions, and make suggestions on ways that can improve exception handling practices and guarantee the robustness of client applications. We applied static analysis techniques on three sets of pairs of API methods and exceptions that came from: 1) the source code of the Android platform's API, 2) applications, and 3) crash data from execution failures. For the validation of the findings of our data-driven method we run a randomized controlled trial.

Overall, we found that almost 10% of the undocumented exceptions that static analysis can find in the Android platform's API source code can be found in actual crashes. Similarly, we discovered that 38% of the undocumented exceptions that developers use in their client applications to handle API methods also manifest themselves in crashes. However, documenting these exceptions does not seem to be a viable option, because our randomized controlled trial demonstrated that documentation does not affect the developers' handling of exceptions. Consequently, we believe that exceptions that result in crashes should be converted to checked or there will be implemented type systems (e.g. for the checking of malformed user inputs and erroneous codes of resources) in order to improve applications' stability.

Admittedly, the findings from the randomized controlled trial run contrary to our initial expectations. This means that undertaking the trial was

a truly worthwhile exercise. Based on these results, in the future we would like to investigate a) how reference documentation can better serve the developers, and b) the effects of increasing the number of checked exceptions on developer productivity and code maintainability.

## Acknowledgments

The authors would like to thank the founders of BugSense (now acquired by Splunk) Panos Papadopoulos and John Vlachogiannis for the data they provided us, as well as the participants of our trial.

## References

[1] X. Leroy, F. Pessaux, Type-based analysis of uncaught exceptions, Transactions on Programming Languages and Systems 22 (2) (2000) 340–377. doi:10.1145/349214.349230.

[2] C.-T. Chen, Y. C. Cheng, C.-Y. Hsieh, I.-L. Wu, Exception handling refactorings: Directed by goals and driven by bug fixing, Journal of Systems and Software 82 (2) (2009) 333–345. doi:10.1016/j.jss.2008.06.035.

[3] Y. Zhang, G. Salvaneschi, Q. Beightol, B. Liskov, A. C. Myers, Accepting blame for safe tunneled exceptions, in: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '16, ACM, New York, NY, USA, 2016, pp. 281–295. doi:10.1145/2908080.2908086.

[4] S. Drossopoulou, T. Valkevych, Java exceptions throw no surprises, Department of Computing, Imperial College, London, 2000.

[5] J. Bloch, How to design a good API and why it matters, in: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, OOPSLA '06, ACM, New York, NY, USA, 2006, pp. 506–507. doi:10.1145/1176617.1176622.

[6] M. Henning, API design matters, Communications of the ACM 52 (5) (2009) 46–56. doi:10.1145/1506409.1506424.

[7] A. Ganapathi, V. Ganapathi, D. A. Patterson, Windows XP kernel crash analysis., in: LISA, Vol. 6, 2006, pp. 49–159.

[8] M. P. Robillard, G. C. Murphy, Static analysis to support the evolution of exception structure in object-oriented systems, ACM Trans. Softw. Eng. Methodol. 12 (2) (2003) 191–221. doi:10.1145/941566.941569.

[9] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, P. McDaniel, FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps, in: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, ACM, New York, NY, USA, 2014, pp. 259–269. doi:10.1145/2594291.2594299.

[10] H. Shah, C. Gorg, M. Harrold, Understanding exception handling: Viewpoints of novices and experts, IEEE Transactions on Software Engineering 36 (2) (2010) 150–161. doi:10.1109/TSE.2010.7.

[11] S. Nakshatri, M. Hegde, S. Thandra, Analysis of exception handling patterns in Java projects: An empirical study, in: Proceedings of the 13th International Workshop on Mining Software Repositories, MSR '16, 2016, pp. 500–503. doi:10.1145/2901739.2903499.

[12] M. B. Kery, C. Le Goues, B. A. Myers, Examining programmer practices for locally handling exceptions, in: Proceedings of the 13th International Workshop on Mining Software Repositories, MSR '16, ACM, New York, NY, USA, 2016, pp. 484–487. doi:10.1145/2901739.2903497.

[13] W. Weimer, G. C. Necula, Exceptional situations and program reliability, ACM Transactions on Programming Language Systems 30 (2) (2008) 8:1–8:51. doi:10.1145/1330017.1330019.

[14] F. Ebert, F. Castor, A. Serebrenik, An exploratory study on exception handling bugs in Java programs, Journal of Systems and Software 106 (2015) 82–101. doi:10.1016/j.jss.2015.04.066.

[15] E. A. Barbosa, A. Garcia, Global-aware recommendations for repairing violations in exception handling, IEEE Transactions on Software Engineering PP (99) (2017) 1–1. doi:10.1109/TSE.2017.2716925.

[16] M. P. Robillard, G. C. Murphy, Designing robust Java programs with exceptions, in: Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications, SIGSOFT '00/FSE-8, ACM, New York, NY, USA, 2000, pp. 2–10. doi:10.1145/355045.355046.

[17] J. R. Kiniry, Exceptions in Java and Eiffel: Two extremes in exception design and application, in: C. Dony, J. L. Knudsen, A. Romanovsky, A. Tripathi (Eds.), Advanced Topics in Exception Handling Techniques, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 288–300. doi:10.1007/11818502_16.

[18] C. Marinescu, Should we beware the exceptions? an empirical study on the eclipse project, in: 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), IEEE, 2013, pp. 250–257.

[19] K. Yi, S. Ryu, A cost-effective estimation of uncaught exceptions in Standard ML programs, Theoretical Computer Science 277 (1) (2002) 185–217. doi:10.1016/S0304-3975(00)00317-0.

[20] C. Mayer, S. Hanenberg, R. Robbes, E. Tanter, A. Stefik, An empirical study of the influence of static type systems on the usability of undocumented software, in: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12, ACM, New York, NY, USA, 2012, pp. 683–702. doi:10.1145/2384616.2384666.

[21] S. Endrikat, S. Hanenberg, R. Robbes, A. Stefik, How do API documentation and static typing affect API usability?, in: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, ACM, New York, NY, USA, 2014, pp. 632–642. doi:10.1145/2568225.2568299.

[22] B. C. Pierce, Types and Programming Languages, 1st Edition, The MIT Press, 2002.

[23] S. Sinha, M. Harrold, Analysis of programs with exception-handling constructs, in: Proceedings of the International Conference on Software Maintenance, 1998, pp. 348–357. doi:10.1109/ICSM.1998.738526.

[24] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, V. Sundaresan, Soot: A Java bytecode optimization framework, in: CASCON First Decade High Impact Papers, CASCON '10, IBM Corp., Riverton, NJ, USA, 2010, pp. 214–224. doi:10.1145/1925805.1925818.

[25] C. Fu, B. Ryder, Exception-chain analysis: Revealing exception handling architecture in Java server applications, in: 29th International Conference on Software Engineering, ICSE '07, 2007, pp. 230–239. doi:10.1109/ICSE.2007.35.

[26] J. Bell, G. Kaiser, Phosphor: Illuminating dynamic data flow in commodity JVMs, in: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '14, 2014, pp. 83–101. doi:10.1145/2660193.2660212.

[27] K. Havelund, T. Pressburger, Model checking Java programs using Java PathFinder, International Journal on Software Tools for Technology Transfer 2 (2000) 366–381. doi:10.1007/s100090050043.

[28] Y. Dang, R. Wu, H. Zhang, D. Zhang, P. Nobel, ReBucket: a method for clustering duplicate crash reports based on call stack similarity, in: Proceedings of the 2012 International Conference on Software Engineering, ICSE 2012, IEEE Press, Piscataway, NJ, USA, 2012, pp. 1084–1093.

[29] S. Kim, T. Zimmermann, N. Nagappan, Crash graphs: an aggregated view of multiple crashes to improve crash triage, in: Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks, DSN '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 486–493.

[30] B. Liblit, A. Aiken, Building a better backtrace: techniques for postmortem program analysis, Tech. rep., Berkeley, Berkeley, CA, USA (2002).

[31] R. Coelho, L. Almeida, G. Gousios, A. van Deursen, Unveiling exception handling bug hazards in Android based on GitHub and Google code issues, in: Proceedings of the 12th Working Conference on Mining Software Repositories, MSR '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 134–145.

[32] M. Kechagia, D. Mitropoulos, D. Spinellis, Charting the API minefield using software telemetry data, Empirical Software Engineering 20 (6) (2015) 1785–1830. doi:10.1007/s10664-014-9343-7.

[33] M. Robillard, E. Bodden, D. Kawrykow, M. Mezini, T. Ratchford, Automated API property inference techniques, Software Engineering, IEEE Transactions on 39 (5) (2013) 613–637. doi:10.1109/TSE.2012.63.

[34] S. Clarke, Measuring API usability, Dr. Dobb's Journal 29 (2004) S6–S9.
URL http://www.drdobbs.com/windows/184405654

[35] T. Scheller, E. Kühn, Automated measurement of API usability: The API concepts framework, Information and Software Technology 61 (2015) 145–162. doi:10.1016/j.infsof.2015.01.009.

[36] U. Farooq, D. Zirkler, API peer reviews: a method for evaluating usability of application programming interfaces, in: Proceedings of the 2010 ACM conference on Computer supported cooperative work, CSCW '10, ACM, New York, NY, USA, 2010, pp. 207–210. doi:10.1145/1718918.1718957.

[37] R. P. L. Buse, W. Weimer, Synthesizing API usage examples, in: Proceedings of the 2012 International Conference on Software Engineering, ICSE 2012, IEEE Press, Piscataway, NJ, USA, 2012, pp. 782–792. doi:10.1109/ICSE.2012.6227140.

[38] A. L. Santos, B. A. Myers, Design annotations to improve API discoverability, Journal of Systems and Software 126 (2017) 17–33. doi:10.1016/j.jss.2016.12.036.

[39] D. Qiu, B. Li, H. Leung, Understanding the API usage in Java, Information and Software Technology 73 (C) (2016) 81–100. doi:10.1016/j.infsof.2016.01.011.

[40] M. Robillard, R. DeLine, A field study of API learning obstacles, Empirical Software Engineering 16 (6) (2011) 703–732. doi:10.1007/s10664-010-9150-8.

[41] W. Maalej, M. P. Robillard, Patterns of knowledge in API reference documentation, IEEE Transactions on Software Engineering 99 (PrePrints) (2013) 1. doi:10.1109/TSE.2013.12.

[42] H. Zhong, Z. Su, Detecting API documentation errors, in: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '13, ACM, New York, NY, USA, 2013, pp. 803–816. doi:10.1145/2509136.2509523.

[43] T. McDonnell, B. Ray, M. Kim, An empirical study of API stability and adoption in the Android ecosystem, in: 2013 IEEE International Conference on Software Maintenance, 2013, pp. 70–79. doi:10.1109/ICSM.2013.18.

[44] B. Dagenais, M. P. Robillard, Using traceability links to recommend adaptive changes for documentation evolution, IEEE Transactions on Software Engineering 40 (11) (2014) 1126–1146. doi:10.1109/TSE.2014.2347969.

[45] R. P. Buse, W. R. Weimer, Automatic documentation inference for exceptions, in: Proceedings of the 2008 International Symposium on Software Testing and Analysis, ISSTA '08, ACM, New York, NY, USA, 2008, pp. 273–282. doi:10.1145/1390630.1390664.

[46] M. Saied, H. Sahraoui, B. Dufour, An observational study on API usage constraints and their documentation, in: Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on, 2015, pp. 33–42. doi:10.1109/SANER.2015.7081813.

[47] A. Machiry, R. Tahiliani, M. Naik, Dynodroid: An input generation system for Android apps, in: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, ACM, New York, NY, USA, 2013, pp. 224–234. doi:10.1145/2491411.2491450.

[48] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, D. Poshyvanyk, Automatically discovering, reporting and reproducing android application crashes, in: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), 2016, pp. 33–44. doi:10.1109/ICST.2016.34.

[49] D. Yan, S. Yang, A. Rountev, Systematic testing for resource leaks in android applications, in: 24th IEEE International Symposium on Software Reliability Engineering (ISSRE), ISSRE '13, 2013, pp. 411–420. doi:10.1109/ISSRE.2013.6698894.

[50] S. Anand, M. Naik, M. J. Harrold, H. Yang, Automated concolic testing of smartphone apps, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, ACM, New York, NY, USA, 2012, pp. 59:1–59:11. doi:10.1145/2393596.2393666.

[51] A. P. Felt, E. Chin, S. Hanna, D. Song, D. Wagner, Android permissions demystified, in: Proceedings of the 18th ACM conference on Computer and communications security, CCS '11, ACM, New York, NY, USA, 2011, pp. 627–638. doi:10.1145/2046707.2046779.

[52] A. Gorla, I. Tavecchia, F. Gross, A. Zeller, Checking app behavior against app descriptions, in: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, ACM, New York, NY, USA, 2014, pp. 1025–1035. doi:10.1145/2568225.2568276.

[53] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, D. Poshyvanyk, API change and fault proneness: A threat to the success of Android apps, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, ACM, New York, NY, USA, 2013, pp. 477–487. doi:10.1145/2491411.2491428.

[54] L. Li, T. F. D. A. Bissyande, Y. Le Traon, J. Klein, Accessing inaccessible Android APIs: An empirical study, in: Proceedings of the 32nd International Conference on Software Maintenance and Evolution, ICSME '16, 2016, p. 12.

[55] J. Oliveira, N. Cacho, D. Borges, T. Silva, F. Castor, An exploratory study of exception handling behavior in evolving Android and Java applications, in: Proceedings of the 30th Brazilian Symposium on Software Engineering, SBES '16, ACM, New York, NY, USA, 2016, pp. 23–32. doi:10.1145/2973839.2973843.

[56] J. Bloch, Effective Java: A Programming Language Guide, Addison-Wesley Java series, Prentice Hall, 2008, Ch. 9: Exceptions, pp. 209–240. URL http://books.google.co.uk/books?id=ka2VUBqHiWkC

[57] N. Viennot, E. Garcia, J. Nieh, A measurement study of Google Play, in: The 2014 ACM International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '14, ACM, New York, NY, USA, 2014, pp. 221–233. doi:10.1145/2591971.2592003.

[58] M. Kechagia, D. Spinellis, Undocumented and unchecked: Exceptions that spell trouble, in: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, ACM, New York, NY, USA, 2014, pp. 312–315. doi:10.1145/2597073.2597089.

[59] A. Bartel, J. Klein, M. Monperrus, Y. Le Traon, Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot, in: ACM Sigplan International Workshop on the State Of The Art in Java Program Analysis, 2012, pp. 27–38.

[60] J. Lang, D. B. Stewart, A study of the applicability of existing exception-handling techniques to component-based real-time software technology, ACM Trans. Program. Lang. Syst. 20 (2) (1998) 274–301. doi:10.1145/276393.276395.

[61] I. Garcia, N. Cacho, eFlowMining: An exception-flow analysis tool for .NET applications, in: Fifth Latin-American Symposium on Dependable Computing Workshops, LADCW '11, 2011, pp. 1–8. doi:10.1109/LADCW.2011.18.

[62] P. M. Uesbeck, A. Stefik, S. Hanenberg, J. Pedersen, P. Daleiden, An empirical study on the impact of C++ lambdas and programmer experience, in: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, ACM, New York, NY, USA, 2016, pp. 760–771. doi:10.1145/2884781.2884849.

[63] N. Salkind, Encyclopedia of Research Design, Vol. I, SAGE Publications, 2010, see p.p. 1245, 1648–1649.

[64] D. P. Byar, A. M. Herzberg, W.-Y. Tan, Incomplete factorial designs for randomized clinical trials, Statistics in Medicine 12 (17) (1993) 1629–1641. doi:10.1002/sim.4780121708.

[65] Survey: http://stereo.dmst.aueb.gr:5000/.

[66] T.-S. Lim, W.-Y. Loh, A comparison of tests of equality of variances, Computational Statistics & Data Analysis 22 (3) (1996) 287 – 301. doi:10.1016/0167-9473(95)00054-2.

[67] H. Bhattacharyya, Kruskal–Wallis Test, John Wiley & Sons, Inc., 2004. doi:10.1002/0471667196.ess1369.pub2.

[68] N. Cacho, E. A. Barbosa, J. Araujo, F. Pranto, A. Garcia, T. Cesar, E. Soares, A. Cassio, T. Filipe, I. Garcia, How does exception handling behavior evolve? an exploratory study in Java and C# applications, in: 2014 IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 31–40. doi:10.1109/ICSME.2014.25.

[69] N. Cacho, T. César, T. Filipe, E. Soares, A. Cassio, R. Souza, I. Garcia, E. A. Barbosa, A. Garcia, Trading robustness for maintainability: An empirical study of evolving C# programs, in: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, ACM, New York, NY, USA, 2014, pp. 584–595. doi:10.1145/2568225.2568308.

[70] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, M. D. Ernst, Practical pluggable types for Java, in: Proceedings of the 2008 International Symposium on Software Testing and Analysis, ISSTA '08, ACM, New York, NY, USA, 2008, pp. 201–212. doi:10.1145/1390630.1390656.

## Appendix

**Table 18: Exceptions stemming from apps (P), API source code (S) and crashes (C), (PCSU = P ∩ C ∩ S ∩ U)**

| Android API method | Recommended exception |
|---|---|
| app.Dialog.setContentView | RuntimeException |
| content.ContentResolver.update | IllegalArgumentException |
| opengl.GLUtils.texSubImage2D | IllegalArgumentException |
| content.res.Resources.openRawResource | FileNotFoundException |
| app.Activity.startActivity | RuntimeException |
| app.Activity.onBackPressed | IllegalStateException |
| view.ViewGroup.addView | RuntimeException |
| widget.TextView.setText | IndexOutOfBoundsException |
| accounts.AccountManager.getAccounts | RuntimeException |
| graphics.Canvas.setBitmap | IllegalStateException |
| text.Html.fromHtml | RuntimeException |
| os.Bundle.getInt | RuntimeException |
| view.animation.AnimationUtils.loadAnimation | RuntimeException |
| os.Handler.postDelayed | RuntimeException |
| database.sqlite.SQLiteDatabase.execSQL | IllegalStateException |
| content.ContentResolver.query | RuntimeException |
| view.View.getLocationOnScreen | IllegalArgumentException |
| database.sqlite.SQLiteOpenHelper.getWritableDatabase | IllegalStateException |
| view.LayoutInflater.inflate | InflateException, RuntimeException |
| graphics.Canvas.drawText | IndexOutOfBoundsException |
| app.PendingIntent.getActivity | RuntimeException |
| content.Intent.getStringExtra | RuntimeException |
| database.sqlite.SQLiteDatabase.endTransaction | IllegalStateException |
| widget.Toast.show | RuntimeException |
| database.sqlite.SQLiteDatabase.delete | IllegalStateException |
| database.sqlite.SQLiteDatabase.beginTransaction | IllegalStateException |
| webkit.CookieManager.setCookie | IllegalStateException |
| content.ContentResolver.bulkInsert | IllegalArgumentException |
| view.View.setLayoutParams | NullPointerException |
| content.res.Resources.getColor | FileNotFoundException |
| app.NotificationManager.notify | NullPointerException |
| support.v4.view.ViewPager.setAdapter | IllegalStateException |
| content.ContentResolver.delete | IllegalArgumentException |
| os.Bundle.getBoolean | RuntimeException |
| content.ContentResolver.insert | IllegalArgumentException |
| widget.TextView.setTextColor | NullPointerException |
| content.Intent.getParcelableExtra | RuntimeException |
| support.v4.app.Fragment.startActivityForResult | IllegalStateException |
| media.MediaScannerConnection.scanFile | IllegalStateException |
| webkit.CookieManager.getCookie | IllegalStateException |
| graphics.Color.parseColor | IllegalArgumentException |
| content.res.AssetManager.open | RuntimeException, FileNotFoundException |
| graphics.Canvas.drawBitmap | ArrayIndexOutOfBoundsException, NullPointerException, RuntimeException |