Mind the Gap: Finding what Updates have (really) changed in Android Applications

Johannes Feichtner[1,2], Lukas Neugebauer[1], Dominik Ziegler[3]
[1]*Institute of Applied Information Processing and Communications (IAIK), Graz University of Technology, Austria*
[2]*Secure Information Technology Center – Austria (A-SIT), Austria*
[3]*Know-Center GmbH, Austria*
{*firstname.lastname*}*@iaik.tugraz.at*

Keywords: Android, Code Comparison, Application Security, Static Analysis, Obfuscation, Smali

Abstract: Android apps often receive updates that introduce new functionality or tackle problems, ranging from critical security issues to usability-related bugs. Although developers tend to briefly denote changes when releasing new versions, it remains unclear what has actually been modified in the program code. Verifying even subtle changes between two Android apps is challenging due to the widespread use of code transformations and obfuscation techniques. In this paper, we present a new framework to precisely pinpoint differences between Android apps. By pursuing a multi-level comparison strategy that targets resources and obfuscation-invariant code elements, we succeed in highlighting similarities and changes among apps. In case studies, we demonstrate the need and practical benefits of our solution and show how well it is suited to verify changelogs.

1 INTRODUCTION

Developers of Android applications regularly distribute and update their apps via Google Play or third-party distribution channels. Featured by descriptions, screenshots, and further promotional information, users can pick from a large pool of often similar applications. Many vendors reuse their own code and offer multiple revisions of the same app for different devices or with adaptations, e.g., for learning various languages, local weather, and city travel guides. While these apps are usually clearly distinguishable by their visual appearance, the opposite is the case when third-party developers distribute repackaged versions of existing apps with barely noticeable modifications. Often, these changes introduce malware or code to hijack revenues for advertisements.

Upon the release of new app versions, developers can provide a changelog that typically includes a list of modifications, fixed issues, and new functionality. In practice, however, release notes are not necessarily accurate. For instance, if an author states that a known security-critical problem has been fixed, users have to put their trust into that statement. Besides not being able to check whether indicated changes have indeed and thoroughly been implemented, additional code modifications that are not exposed in the changelog are impossible to reproduce.

Whenever updates are published for Android apps, or applications appear to be repackaged versions or clones, it is crucial to see what has actually changed in the program code. From a research perspective, there is a strong need for a solution that can reliably highlight differences among various app versions while being able to distinguish between functional changes by developers and structural adaptations by compilers. Especially, regarding the rising complexity and size of today's apps, a comprehensive comparison could help to lower the costs and efforts needed for app analysis and would allow for easier verification of bugfixes.

Manually finding similarities and differences between two implementations is challenging due to the widespread use of code transformation techniques that are applied not only to optimize code but also to harden against reverse engineering and tampering.

Automated solutions, on the other side, are usually targeted to return a verdict (yes/no) on whether an app appears to be a cloned or repackaged version of another one (Chen et al., 2015; Wang et al., 2015). In a trivial case, this decision can be made by checking if the majority of files of one app sample are also contained in the comparison object. However, if code transformation techniques are used, a content or hash-based comparison would lead to spurious results. Likewise, the verdict will be tainted if obfuscation and related techniques are not taken into account.

Problem. When comparing Android apps with each other, we aim to disclose changes in the implemented program behavior, rather than finding stylistic or other non-functional changes without influence on execution or visual appearance. However, precisely uncovering only real differences between apps is challenging due to code transformations at compile time and a variety of ways, how developers can express semantically similar program statements. Also, if obfuscation techniques are used, the identifiers of classes, methods, and variables are replaced by meaningless placeholders.

Existing state-of-the-art efforts focus on detecting repackaged or cloned apps based on heuristics but are unable to highlight individual code parts that are similar or different between two Android apps. Mostly operating on the Dalvik bytecode of apps, these work derive a verdict based on a custom similarity metric that yields accurate but often irreproducible results (Zhan et al., 2019; Guan et al., 2016). Other approaches for similarity analysis, as implemented by the tools *Androguard*[1] and *FSquaDRA* (Zhauniarovich et al., 2014), deliver a deterministic score but give no explanation on how and where code relates to each other.

The ability to highlight concrete similarities and differences among apps is essential since it is crucial to understand what has really been changed by updates and in repackaged versions of apps. In addition, a reliable comparison can provide valuable insight for analysts and curious users into how developers have addressed critical bugs by updates.

Approach. To detect and visualize similarities and differences between two Android applications, we propose a multi-level comparison strategy:

1. We model the hierarchy of packages with code and resources in Merkle trees and prune parts that are equal between the two versions. For resources, we end after this step and can already tell what files are equal, deleted, or have changed among apps.
2. From the remaining code, we extract and process all classes, methods, and basic blocks. While preserving the original code semantics and data flow, we perform multiple rounds of comparison that accurately identify newly added, changed, moved, and deleted code elements.
3. We tackle common code transformation techniques by comparing only features that are invariant to obfuscation and arbitrary compiler modifications.
4. The comparison results can be inspected in a web-based view that visualizes differences similar to tools found in source code version-control systems, such as Git and Subversion.

[1]https://github.com/androguard/androguard

Results. We implemented an analysis framework that can be used for various purposes involving the direct comparison of two Android applications. In a case study, we show the practical benefits of our solution by verifying the developer-provided changelog of real-world applications. Moreover, we demonstrate that our tool cannot only help to identify repackaged apps but also to precisely isolate code that might have been added or modified by malware authors.

Contribution. Our contributions are as follows:

1. We design a framework that enables a pairwise comparison of the code and resources contained in two given Android apps. We implement similarity checks at file, class, method, and basic block level and present code differences in a format that is well-known from source code versioning systems.
2. We propose a multi-round comparison approach for code that takes compiler peculiarities and code transformation techniques, such as identifier obfuscation into account. We modify the *baksmali*[2] reverse-engineering tool to extract different representations of basic blocks from Dalvik bytecode. To counter code obfuscation, we derive segments where registers, labels, and re-arrangements cannot influence similarity matching.
3. Finally, we perform case studies on real-world applications to demonstrate how our framework can be used to explain differences and similarities among apps. We exemplify the verification of changelogs presented by popular apps and show how to identify functional changes in code.

Outline. In Section 2, we discuss related work. In Section 3, we define the requirements for our framework and summarizes the comparison workflow for code and resources. Section 4 elaborates our approach for code matching at class, method, and basic block level in more detail. Applied on real-world Android apps, in Section 5, we demonstrate the usage scenario of changelog verification by identifing similarities and differences in app code. Section 6 concludes our work.

2 RELATED WORK

Comparing the code of Android apps typically involves finding a metric that assigns scores to implementation differences. Using a static or dynamic analysis approach, most existing work extract features from meta-data (e.g., app permissions), code (e.g., call graphs or instruction subsets), or obtain them during runtime (e.g., execution traces).

[2]https://github.com/JesusFreke/smali

The reverse-engineering framework *AndroGuard* first included an algorithm for pairwise comparison using *Normalized Compression Distance* (Desnos, 2012). By comparing hashing-based fingerprints of methods, a value between 0 and 1 was derived to indicate the similarity between them. Variants of this approach focused on dependency graphs (Crussell et al., 2012), layout information (Sun et al., 2015), or combined different features (Shao et al., 2014).

To improve the scalability of pairwise comparison, other solutions summarize extracted features in vector representations. Instead of operating on code, PiggyApp (Zhou et al., 2013) fills vectors with semantic information and uses them to compute the distance between apps. For faster comparisons, other approaches abstract code parts into graph-like representations (Chen et al., 2014; Deshotels et al., 2014).

Newer approaches are directed towards measuring similarity using machine learning-based techniques. Supervised (Tian et al., 2016) and unsupervised (Shao et al., 2014) learning approaches have been proposed to tackle the recognition of malware in repackaged Android applications.

All related work presented implement a pairwise comparison and differ mostly in the algorithm used as a distance metric, as well as in the features that are used to derive fingerprints from. Although the problem domain is strongly related to our research, related work does not yet *(1)* tackle the problem of uncovering individual code differences instead of deriving a summarizing similarity score and *(2)* cannot distinguish between functional changes by developers and structural adaptations by compilers.

3 System Design

We design a framework to list differences and similarities between two given Android application archives. The primary workflow can be split into two phases:

1. In the **resource comparison** phase, our solution extracts and compares static content and all non-code files that are referenced in code, such as bitmaps, layout definitions, and user interface strings. We then organize these files in Merkle trees and leverage the resulting set of hashes to efficiently determine newly added, changed, and deleted resource files or folders. As resources are never affected by obfuscation or related techniques, the comparison already ends after a single step. Two applications with identical resources can indicate that applications are repackaged or cloned versions (Sun et al., 2015; Shao et al., 2014).
2. In the subsequent **code comparison** phase, we first convert the Dalvik bytecode of both comparison objects into Smali code and, similar to resources, organize the resulting hierarchy of files and directories as Merkle trees. Considering the large amount of classes in today's applications, we leverage this data structure to quickly filter classes and packages that are equal among both apps. For the remaining code, we continue with a more fine-grained comparison at class, method, and basic block level. In multiple rounds, we rewrite Dalvik bytecode and derive different code representations where possible compiler modifications or effects of transformation techniques, such as code obfuscation, are mitigated. Due to the wide range of possibilities how code can be transformed, we elaborate multiple semantically equivalent code representations tackling different obfuscation aspects and apply them for comparison. Overall, this approach ensures that only code parts are matched that are indeed related with each other.

After analyzing two apps, our tool collects all found differences and similarities in separate files for later interpretation in a web-based comparison view. Based on HTML and JavaScript, differences between apps are visualized in a representation that is well-known from established management systems for source code, such as Git and Subversion. Deleted basic blocks are marked in red color, whereas newly added or changed code fragments are highlighted in green. Existing or unchanged parts are not underlined.

We replicate the visual representation of code changes, as displayed by our framework, in a case study of real-world applications (see Section 5).

4 Code Similarity in Android Apps

To answer the question of how much code is shared between two applications, we apply several generic options for both comparison on class level as well as comparison on method level. Our concept is based on comparing *.smali* code to detect code similarities and differences. The idea is to iteratively split parts that do not match between apps into smaller chunks. In each step, we gradually reduce the number of identifiers and substitute them with placeholders. However, this approach alone does not account for modifications or peculiarities of used compilers that would influence the hash value of a file or block. In the following we explain the operations we apply to the original Dalvik bytecode before comparing applications.

- **Removing debug information:** Debug information typically has no impact on code execution.

Still, it might differ for two exact versions of a file, depending on the used compiler. In a first step, we, thus, strip debug information, such as line numbers and parameter names from the *.smali* code generated by our modified version of *baksmali*.

- **Implicit method and field references:** We rely on implicit method and field references for elements of a current class, when generating *.smali* code. Notably, we do not use the current class name as a prefix for method and field names. For instance, `La/b;->a:Ljava/lang/Object;` will be rewritten as `a:Ljava/lang/Object;`. By relying on this reduced variant, we can identify identical code snippets across different classes.
- **Sequential numbering of labels:** Instead of using the address information contained in the original bytecode, we apply a sequential numbering scheme for labels (for instance targets of jump instructions). This enables a semantic comparison of classes and basic blocks despite obfuscation.
- **Sorting fields and methods:** Another important task when performing hash-based comparison of classes is sorting field and methods consistently. We achieve this by linking name invariant values with each field and method in a class. Therefore, we derive a unique deterministic value, depending on the access flag, number of parameters and return type for methods, and access flag and type of field for fields. To determine the order of elements, we also leverage the method and field index stored in Dalvik bytecode. The resulting name-invariant value is finally used to sort methods and fields.

Although the applied operations already ascertain consistent comparison results, additional measures are necessary to tackle dynamic decisions during compilation. The following four operations are designed to address these differences among apps:

- **Deterministic register labels:** Our customized version of *baksmali* enforces the assignment of registers in a deterministic way instead of using variable, compiler-chosen registers. For this purpose, we sequentially assign registers depending on their first occurrence. This operation enables us to detect two semantically identical blocks where only the register names were assigned differently during compilation. However, without additional checks, a sequential labeling could lead to false results. For instance, if a simple instruction is inserted in between two app versions, all registers might be shifted and consequently, all remaining basic blocks cannot be identified as matching.
- **Static register labels:** Instead of replacing register names with deterministic assignments, our solution also supports static register placeholders. As a result, hashes of blocks are independent of used registers. However, this measure can influence accuracy. For instance, if a method changes the return value by referencing another register, this change would be unnoticed by this approach. We, thus, apply this step in conjunction with others to prevent possible mismatches.
- **Resolve resource identifiers:** The third important measure to facilitate the analysis due to compiler decisions is replacing resolved resource identifiers with the original type and content defined in `/res/default.xml`. For example, the value `const v0, 0x123cafe0` could be replaced by `const v0, **APKCOMPARE.<some_value>**`. Due to arbitrary assignments of resource identifiers during compilation, these replacements are inevitable for a distinctive comparison of code with references to application resources.
- **Excluding:** Our solution supports the exclusion of classes in the `android` package from analysis. As these classes are typically unmodified included from the Android runtime, they do not contribute to showing functional differences between apps.

4.1 Matching Classes

Matching classes is based on the idea of gradually trimming classes. For each app under comparison, we build a Merkle trees that holds all packages, classes, and methods. As depicted in Figure 1, the analysis process consists of six rounds. In each iteration step already matched classes are excluded from the next:

1. **Unmodified files:** In the first step, we compare the hash value of unmodified *.smali* files. Unmodified in this step means that no additional measures, besides those necessary for coherent generation of classes, have been applied. Subsequently, we compare the generated Merkle trees. All identical classes found in both apps are determined and removed from the tree before the next step.
2. **Files with replaced super class names:** In the second step, we aim at detecting classes which differ only in the names of the used superclasses. Hence, we generate *.smali* files, where occurrences of superclasses have been substituted with placeholders. Again, we store all matching classes and remove them from our analysis set at this point.
3. **Files with replaced class names:** After comparing all classes with changed super class names, we now substitute only the class name with a placeholder. This approach gives us the ability to detect renamed classes. After this step, our analysis set

only contains classes which either implement a different interface or differ in their class signature.

4. **Files with replaced interfaces names:** In the fourth step, we try to detect classes with changed interface names. Hence, we introduce a placeholder for all used interfaces in our class files. Like in the previous steps, we reduce our analysis set by all classes matching in both apps.
5. **Files with replaced class signatures:** For all classes for which no corresponding match was found, we repeat the search with a combination of all previous variants, in this step. In detail, we introduce placeholders for class and super class names as well as implemented interfaces. We refer to this step as *replacement of class signatures*.

As can be seen in Figure 1, steps 1-5 can be repeated multiple times. Between each iteration, we rewrite the Dalvik bytecode of both apps and apply additional operations, e.g., replacing register names with placeholders, to increase the detection rate. These steps can be repeated as long as new matches are found.

However, classes where fields or methods have been renamed, are not covered by the first five steps. Additionally, code obfuscation tools like ProGuard have been found to be resistant against these first five steps. Consequently, we introduce one final step, where we replace all identifiers with placeholders:

6. **Files with all identifiers replaced:** In the last step, we remove all identifiers and names and substitute them with placeholders. An example of this approach is depicted in Figure 2. In this step, we replace method, field, class and type names with placeholders. However, we leave method names like `<init>`, certain package names like `android/` or basic data types unchanged, as they cannot be renamed by code obfuscation tools.

4.2 Matching Methods and Basic Blocks

The pairwise comparison of classes does not lead to positive matches if methods were added, removed, or changed. For all code classes that did not match in the previous step, we propose a fine-grained inspection that focuses on individual methods and basic blocks.

In a first step, we leverage our customized version of *baksmali* to extract all methods from remaining classes that could not be matched in the previous step. For each method, we retrieve two representations: first the original code with all identifiers and in addition, to overcome code transformations, we rewrite the Dalvik bytecode to generate a semantically equivalent version where identifiers are replaced with placeholders.

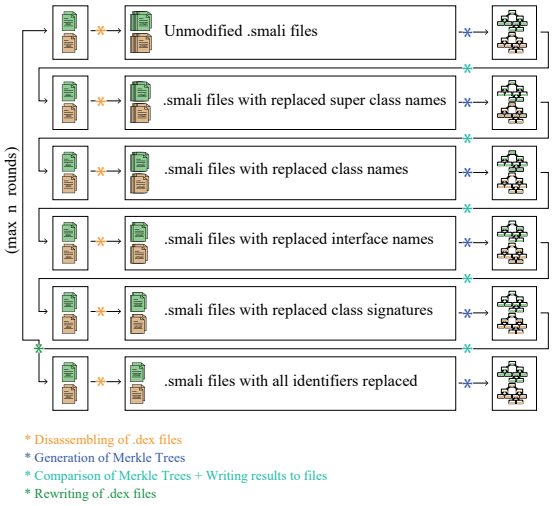

Figure 1: Multi-round code comparison at class level.

For all methods that exceed the lower bound of code lines needed for matching, we derive a hash value of the method signature and the body with all basic blocks. This implies that the order to basic blocks is also considered and re-arranged, moved, added, or deleted basic blocks will thwart a successful match. As a remedy, in cases where methods cannot be matched, we propose to extend the matching process to the level of basic blocks for a more in-depth comparison.

Overall, the analysis procedure of methods and basic blocks can be split into four steps:

1. **Methods with identifiers:** After extracting all methods of unmatched classes from Dalvik bytecode in their original format (with potentially obfuscated identifiers), their hash representation is stored in a sorted list. Methods that fall below the configured minimum threshold for needed code lines are winnowed and not considered. Then, for all generated hash values of one Android app a match is looked up in the set of methods in the second app provided for comparison.

   In this step, we determine all methods that have not been changed but were moved to another class.
2. **Methods without identifiers:** For all methods for which no corresponding match was found in the first step, the search is repeated with the semantically equivalent but obfuscation-invariant representation of the method body.

   This step finds all methods that exhibit the same control and data flow with differently named identifiers. In practice, this is the case, if one version of an Android app is obfuscated and the comparison object is not. Likewise, this comparison step matches method bodies that were applied differ-

```
.method public final run()V
.registers 3

iget-object v0, p0, b:Lb/f;
iget-object v0, v0, Lb/f;->a:Lb/e;
iget-object v1, p0, a:L
      renamed/by/apkcompare/number124;

invoke-static {v0, v1}, Lb/e;->a
    (Lb/e;Lrenamed/by/apkcompare/number124;)V
return-void

.end method
```

```
.method public final _METHOD_NAME_()V
.registers 3

iget-object v0, p0, _FIELD_NAME_:_TYPE_;
iget-object v0, v0, _CLASS_->_FIELD_NAME_:_TYPE_;
iget-object v1, p0, _FIELD_NAME_:
      Lrenamed/by/apkcompare/number124;

invoke-static {v0, v1}, _CLASS_->_METHOD_NAME_
    (_TYPE_;Lrenamed/by/apkcompare/number124;)V
return-void

.end method
```

Figure 2: Comparison of Smali code at class level without (left) and with (right) replaced identifier names.

ent code transformation techniques and were e.g., compiled using different obfuscation settings.

3. **Basic blocks with identifiers:** If the method body has changed, a more fine-grained matching using the contained basic blocks is inevitable. Similar to the first step, we derive the hash value of all basic blocks in so far unmatched methods, collect them in a sorted list, and compare them with the hash values of basic blocks of a comparison object. If basic blocks are found in multiple methods, the resulting candidates are sorted by the overall amount of matching basic blocks in the same method. This decision logic resembles the matching behavior of the Git source code versioning system.

   This step discloses basic blocks that were moved to other methods without changing any identifiers.

4. **Basic blocks without identifiers:** Finally, the hash values of obfuscation-invariant basic blocks are compared with hash values of basic block representations where identifiers have been replaced by placeholders.

   In this step, we find deleted and newly added basic blocks. Due to the lookup without identifiers, we also discover basic blocks that have been moved to other methods but where identifiers have been renamed, e.g., due to different obfuscation settings or membership in a different code package. This step also covers typical obfuscation settings, such as *code merging* and *inlining*. Conscious about preserving control flow integrity, code transformations are usually applied on entire basic blocks, rather than single code lines. By comparing representations without identifiers, our approach enables to effectively keep track of affected basic blocks.

The described comparison strategy evolves from entire methods to a fine-grained matching on basic block level. Considering that today's Android apps often include tens of thousands of methods, our approach reasonably reduces the set of objects to compare in each step. Evidently, newly added or deleted basic blocks always come at full analysis cost as they are compared with the hash values of all other blocks, until we can conclude that they exist in only one of the two given Android applications.

5 CASE STUDY

We evaluate our framework by comparing subsequent versions of Android apps and validate whether provided release notes are accurate. For demonstration purposes, we select two popular apps with a large amount of code: the messenger app *Skype*[3] and the password manager *1Password*[4].

For both apps, the source code is non-public, which means that only changelog information and the results of our tool can be used for analysis. We could find security-relevant release notes for both apps and will use our tool to verify if the changes that have been made correspond to the statements in the changelogs.

5.1 1Password

1Password is a password manager app for Android with more than 1.000.000 installations, according to Google Play. Besides the brief changelog denoted in the official distribution platform, full release notes are presented at the developer website[5]. In version 6.4.1, the authors addressed several security-relevant vulnerabilities that were present in previous versions. In the following, we apply our tool to compare the versions 6.4 (build 58) and 6.4.1 (build 59) of *1Password* and validate how the security issues were handled in code. According to the developer-provided changelog, the app update involved functional code changes:

- **Improved domain matching**

  In versions 6.4 and older, the *1Password* app did not consider subdomains when parsing URLs due

[3] https://play.google.com/store/apps/details?id=com.skype.raider

[4] https://play.google.com/store/apps/details?id=com.agilebits.onepassword

[5] https://app-updates.agilebits.com/product_history/OPA4

Listing 1: *1Password*: Added method `getLoginsForUrl`.

```
@@ diff: com/agilebits/onepassword/support/Utils.java <->
@@       com/agilebits/onepassword/support/Utils.java
   ...
+  public static List<GenericItemBase> getLoginsForUrl(
+    List<GenericItemBase> paramList, String paramString)
+  {
+    paramString = PublicSuffix.registrableDomainForUrl(paramString);
+    ArrayList localArrayList = new ArrayList();
+    if ((paramList != null) && (!TextUtils.isEmpty(paramString)))
+    {
+      paramList = paramList.iterator();
+      while (paramList.hasNext())
+      {
+        GenericItemBase localGenericItemBase = (GenericItemBase)paramList.next();
+        if ((!TextUtils.isEmpty(mLocation)) &&
+          (paramString.equals(PublicSuffix.registrableDomainForUrl(mLocation)))) {
+          localArrayList.add(localGenericItemBase);
+        }
+      }
+    }
+    return localArrayList;
+  }
   ...
+  public static URI parseURIFromUrl(String paramString)
+  {
+    ...
+    return createURIFromUrlStr("https://" + paramString);
+  }
   ...
```

Listing 2: *1Password*: Changed URL input check.

```
@@ diff: com/agilebits/onepassword/activity/AutologinActivity.java <->
@@       com/agilebits/onepassword/activity/AutologinActivity.java
   ...
   public void loadUrl(String paramString)
   {
-    paramString = Utils.uriFromUrl(paramString);
+    paramString = Utils.parseURIFromUrl(paramString);
     if (paramString != null) {
-      mWebView.loadUrl(paramString.toString());
+      mWebView.loadUrl(paramString.toASCIIString());
+      return;
     }
+    ActivityHelper.getAlertDialog(this, 2131231548, 2131231547).show();
   }
   ...
```

to a mismatching regular expression. The update to version 6.4.1 replaced the check with a call to the newly added method `getLoginsForUrl`, which our tool successfully located in the class `Utils` (Listing 1). An inspection of the new code behavior reveals that the changes indeed fixed the domain matching problem.

- **New default scheme: HTTPS instead of HTTP**

  In prior app versions, the internal browser of the app used HTTP as the default scheme, if no full URL was provided by the user. As also shown in lower part of Listing 1, code has been added to prepend URLs with the *https://* prefix.

- **Prevent access to non-web URLs**

  In older versions, users could read private data from the app folder by using URLs with the *file:///* scheme in the built-in web browser. As shown in Listing 2, the update introduced a limitation to web URLs and printed an error with all other URLs.

- **Informative dialogs in the case of TLS errors**

  According to the changelog, the update also improved messages that are shown when SSL/TLS errors occur. Inspecting the newly added class `CommonWebViewClient`, we can verify how the improvements have been realized.

Summarizing, the verification of changes between two versions of the *1Password* app shows that all indicated security-relevant modifications have indeed been carried out as described and that the developer-provided release notes covered all changes.

5.2 Skype

In 2011, the developers of the *Skype* messenger app distributed an update via Google Play that raised the version from 1.0.0.831 to 1.0.0.983. The provided changelog reported the fix of a recent security issue[6]. As also reported by related work (Desnos, 2012), in the vulnerable version of the messenger, sensitive profile data, such as the account balance, date of birth, email address, etc. were stored unencrypted on the device. In addition, wrong access permissions have been chosen for files, allowing other apps to read and write them.

By applying our tool on both versions, we find that security-relevant changes have been made in the class `com/skype/ipc/SkypeKitRunner`. The most significant modifications with relevance to fixed security issue have been compiled in Listing 3.

As shown in the upper part of the listing, two new methods have been introduced by the update. The methods `fixPermissions(File[])` and `chmod(File, String)` are used to change access permissions for files that were created with previous versions of the *Skype* app. In the second half of the listing, we see that a string value used to define the Unix access permissions for files was reset from `777` (*rwxrwxrwx*) to the value `750` (*rwxr-x—*), followed by an invocation of the newly added method `fixPermissions`.

Overall, the comparison study of *Skype* underlined that our tool can reproduce the fix of security issues in accordance with the findings reported by related work.

[6] http://www.helloandroid.com/content/skype-security-vulnerability-fixed

Listing 3: *Skype*: Update with security-related changes.

```
@@ diff: com/skype/ipc/SkypeKitRunner.smali <->
     com/skype/ipc/SkypeKitRunner.smali
  ...
  .end method

+ .method private fixPermissions([Ljava/io/File;)V
+    .registers 7
+
+    array-length v0, p1
+    ...
+
+ .end method
+
+ .method private chmod(Ljava/io/File;Ljava/lang/String
     ;)Z
+    .registers 7
     ...
     const-string v6, "csf"
-    const/4 v7, 0x3
+    const/4 v7, 0x0

     invoke-virtual {v4, v6, v7}, L
          android/content/Context;->
      openFileOutput(Ljava/lang/String;I)L
           java/io/FileOutputStream;
     ...

     invoke-direct {v2}, Ljava/lang/StringBuilder;-><
          init>()V

-    const-string v4, "chmod 777 "
+    const-string v4, "chmod 750 "
     ...

     move-result-object v1
+    move-object/from16 v3, p0
+
+    iget-object v3, v3, mContext:L
       android/content/Context;
+    move-object v2, v3
+
+    invoke-virtual {v2}, Landroid/content/Context;->
       getFilesDir()Ljava/io/File;
+    move-result-object v2
+
+    invoke-virtual {v2}, Ljava/io/File;->listFiles()[L
       java/io/File;
+    move-result-object v2
+
+    move-object/from16 v3, p0
+    move-object v18, v2
+
+    invoke-direct {v3, v18}, fixPermissions([L
       java/io/File;)V

     invoke-static {}, Ljava/lang/Runtime;->getRuntime()
          Ljava/lang/Runtime;
     ...
```

6 CONCLUSION

Android apps often receive updates that provide new functionality and bugfixes. Verifying what has really been changed in the code is challenging due to compiler peculiarities and code transformations.

In this paper, we presented a solution to accurately detect similarities and differences in the code and resources of two given Android apps. With a focus on features that are invariant to code obfuscation, we proposed a multi-round comparison approach that excels in finding matching pairs of code fragments. In a case study, we exemplified the practical use of our framework by verifying how updates have been deployed to fix security-critical issues in real-world apps.

REFERENCES

Chen, J., Alalfi, M. H., Dean, T. R., and Zou, Y. (2015). Detecting Android Malware Using Clone Detection. *J. Comput. Sci. Technol.*, 30:942–956.

Chen, K., Liu, P., and Zhang, Y. (2014). Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In *International Conference on Software Engineering – ICSE 2014*, pages 175–186. ACM.

Crussell, J., Gibler, C., and Chen, H. (2012). Attack of the Clones: Detecting Cloned Applications on Android Markets. In *European Symposium on Research in Computer Security – ESORICS 2012*, volume 7459 of *LNCS*, pages 37–54. Springer.

Deshotels, L., Notani, V., and Lakhotia, A. (2014). DroidLegacy: Automated Familial Classification of Android Malware. In *Program Protection and Reverse Engineering Workshop – PPREW*, pages 3:1–3:12. ACM.

Desnos, A. (2012). Android: Static Analysis Using Similarity Distance. In *Conference on Systems Science – HICSS 2012*, pages 5394–5403. IEEE Computer Society.

Guan, Q., Huang, H., Luo, W., and Zhu, S. (2016). Semantics-Based Repackaging Detection for Mobile Apps. In *Engineering Secure Software and Systems – ESSoS 2016*, volume 9639 of *LNCS*, pages 89–105. Springer.

Shao, Y., Luo, X., Qian, C., Zhu, P., and Zhang, L. (2014). Towards a scalable resource-driven approach for detecting repackaged Android applications. In *Annual Computer Security Applications Conference – ACSAC 2014*, pages 56–65. ACM.

Sun, M., Li, M., and Lui, J. C. S. (2015). DroidEagle: seamless detection of visually similar Android apps. In *Security and Privacy in Wireless and Mobile Networks – WISEC 2015*, pages 9:1–9:12. ACM.

Tian, K., Yao, D., Ryder, B. G., and Tan, G. (2016). Analysis of Code Heterogeneity for High-Precision Classification of Repackaged Malware. In *IEEE Security and Privacy Workshops – SPW 2016*, pages 262–271. IEEE Computer Society.

Wang, H., Guo, Y., Ma, Z., and Chen, X. (2015). WuKong: a scalable and accurate two-phase approach to Android app clone detection. In *Symposium on Software Testing and Analysis – ISSTA 2015*, pages 71–82. ACM.

Zhan, X., Zhang, T., and Tang, Y. (2019). A Comparative Study of Android Repackaged Apps Detection Techniques. In *Software Analysis, Evolution, and Reengineering – SANER 2019*, pages 321–331. IEEE.

Zhauniarovich, Y., Gadyatskaya, O., Crispo, B., Spina, F. L., and Moser, E. (2014). FSquaDRA: Fast Detection of Repackaged Applications. In *Data and Applications Security and Privacy – DBSec 2014*, volume 8566 of *LNCS*, pages 130–145. Springer.

Zhou, W., Zhou, Y., Grace, M. C., Jiang, X., and Zou, S. (2013). Fast, scalable detection of "Piggybacked" mobile applications. In *Conference on Data and Application Security and Privacy – CODASPY 2013*, pages 185–196. ACM.