Automatic Components Separation of Obfuscated Android Applications: An Empirical Study of Design Based Features

Amit Kumar Mondal Chanchal Roy Banani Roy Kevin A. Schneider
University of Saskatchewan, Canada
{amit.mondal, chanchal.roy, banani.roy, kevin.schneider}@usask.ca

Abstract—**In modern days, mobile applications (apps) have become omnipresent. Components of mobile apps (such as 3rd party libraries) require to be separated and analyzed differently for security issue detection, repackaged app detection, tumor code purification and so on. Various techniques are available to automatically analyze mobile apps. However, analysis of the app's executable binary remains challenging due to required curated database, large codebases and obfuscation. Considering these, we focus on exploring a versatile technique to separate different components with designed based features independent of code obfuscation. Particularly, we conducted an empirical study using design patterns and fuzzy signatures to separate programming design components such as 3rd party libraries. In doing so, we built a system for automatically extracting design patterns from both the executable package (APK) and Jar of an Android application. The experimental outcome with various standard datasets containing 3rd party libraries, obfuscated apps and malwares reveals that design features like these are present significantly within them (within 60% APKs including malware). Moreover, these features remain unaltered even after app obfuscation. Finally, as a case study, we found that the design patterns alone can detect 3rd party libraries within the obfuscated apps considerably (F1 score is 32%). Overall, our empirical study reveals that design features might play a versatile role in separating various Android components for various purposes.**

Index Terms—**Obfuscation; Design patterns; libraries; components separation;**

I. Introduction

Billions of Android applications (app) are being activated in every year. Various security flaws in those apps are also being increased [28]. A new update of the mobile platform may introduce new flaws and cause app crashes [12]. Moreover, apps are repackaged [14] by inserting junk code to make it appear as different one by stealing the original app or malicious behaviours are inserted with the original app. *Ad* libraries or analytic plugins [15, 34] are integrated for earning revenue and an app may be exposed to potential security threats for that. To deal with these concerns, the research community has been exploring various dimensions [24] of Android app development. Despite much efforts in the community, the state of the art tools are still challenged in the automatic analysis due to the requirement of curated database [34], large codebase [32] and obfuscation [19]. Available studies reveal that obfuscation makes the Android malware detection worse [19] as it hides informative data in the software. Various libraries, one of the most essential components of an app, are treated in different ways [14] during app analysis. With the latest techniques [25, 33], 40% of these libraries remain undetectable in the obfuscated apps. However, many other distinguishing components are present within an app that are related to the discussed concerns.

Typically, from the programming component perspective, an application consists of built-in code from Android native framework [9], code written by the app developers, code inserted by the obfuscators [19], code integrated through the usage of functions of 3rd party libraries [31] or ad-wares [15], and tumor payload [34]. Handwritten code components further can be treated into two categories concerning different contexts: (i) code tightly interweaving with the Android components such as activity callbacks, and (ii) code written in the external context but used within the first context such as game playing logic [15]. Many components [34] (such as 3rd party libraries) introduce noise or affect mobile app analysis [31]. For example, in app clone (or repackaged) detection process [14], 3rd party libraries must be removed at the first step. In the instrumented packaged app, code components of the Android framework and written code within its close context remain mostly unchanged and detection or separation of them is straightforward. However, that is not the case for the other code components mentioned above. Consequently, different components need to be detected or removed first or analyzed separately. Several approaches are available for detecting 3rd party libraries and ad-wares within the instrumented apps which require a previously-collected database, and the search process introduces both time and memory overhead [32] along with the challenges of altered and hidden code. Therefore, more fruitful, versatile and complementary techniques are required to be explored to advance the available efforts of mobile application analysis.

Considering the above mentioned challenges, in this paper, we conducted an empirical study on object-oriented design features [32] that are present within the Android applications having distinguishable properties to partition, filter, and detection of various components. Android application development framework is mostly based on object-oriented programming language Java (Kotlin and C++ can also be used). Therefore, we assume that the components of an application can be defined by certain design features to a considerable extent. For example, most of the widely used

3rd party libraries [31] within Android apps are developed collaboratively and remotely. Particular design choices are likely to be adopted by the developers of those libraries due to collaboration, and continuous extension. Besides, during development, design features are inherently generated by many model-driven engineering [26] tools. These design features can be object models, binary class relationship [16], micro-architecture with defined motifs [17], design patterns [13], anti-patterns [20], design principals, fuzzy signatures [33] and so on. Overall, design features [17] express traits, acts, tendencies, recurring solutions, structural consumption and other common observable characteristics within the source/executable code.

Among many features, design patterns, recurring solutions to common design problems in the organization, are independent [18] of both the context and the programming language. Particularly, in this study, we focused on design patterns and fuzzy signatures to separate programming design components such as 3rd party libraries. In doing so, we have built a system for automatically extracting design patterns from both the Android executable package called APK and Jar of an Android application; we are unaware of any such tool that can extract design patterns from an APK. With the help of this tool, we have experimented the distribution of design patterns in 3rd party libraries, usual apps and Android malware. Within standard datasets, we found that around 50 to 98% obfuscated apps containing libraries have design patterns which are changed neither in the original code nor in the 3rd party libraries. Furthermore, among studied collection, 61% malwares contain design patterns. Finally, as a case study, we have shown that the design patterns alone can detect 3rd party libraries within obfuscated apps considerably (F1 score is 32%). Overall, our empirical study reveals that design features might play a versatile role in separating various Android components for various purposes. The contribution of this paper are:

- Extracted design features of various Android apps focusing five research questions that direct the researchers of exploring a versatile approach for separating components.
- Presented the distribution of design patterns in the 3rd party libraries, obfuscated apps and malware.
- We have shown that design patterns can be treated as versatile features that can predict 3rd party library components in the obfuscated apps significantly without the database of libraries.
- Conducted a case study on the impact of the design features for detecting 3rd party libraries within the obfuscated apps.

The paper continues as follows. In Section II we designed our empirical study. Section III describes our methodologies and tools. Section IV describes the experimental dataset. Section V report our experimental outcome and Section VII concludes our paper.

II. Study Design

In our empirical study, we consider various design patterns [13] as key design features - how an application is intentionally or unintentionally developed on underlying object-oriented principals and behaviors. More specifically, we focus on the following research questions:

RQ1: *How design features are contained in 3rd party libraries?* If design features can define a 3rd party library then it is likely to be defined with those features within an app which integrates it. Researchers will benefit from RQ1 for developing a versatile technique to detect 3rd party libraries within an app.

RQ2: *How significantly design patterns are present in Android apps?* It is important to know the presence of certain design features before any efforts done to develop a design-based methodology.

RQ3: *Can obfuscator alter the design patterns?* If obfuscator can alter design patterns then it poses a challenge to develop independent technique. Therefore, we also investigate whether obfuscation modifies the design features or not. Practitioners will benefit from RQ3 to adopt a design-based methodology to separate components of an obfuscated APK.

RQ4: *Does Android malware contain design patterns?* For designed based security and privacy leaks analysis, it is essential to figure out the design choices used in malware.

RQ5: *Can design features detect 3rd party libraries in obfuscated Android apps?* Finally, the implication of the defined features needs to be tested for the practical cases of Android app analysis. In order to do that we conducted a case study for 3rd party library detection utilizing fuzzy signatures and fuzzy hash along with the design patterns. This research question will allow us to understand the effectiveness of design-based features for separating 3rd party libraries.

For our empirical study, we develop a tool for extracting design patterns from Android APK. In the subsequent sections, we discuss the methodology, tool, fuzzy signatures and fuzzy hash deployed for answering the aforementioned research questions.

III. Methodology and Tool for Our Study

From the object-oriented design perspective, the design of an application can be described in many ways. As an Android app is built with various components discussed in Introduction, most of them are not developed by the development-owner of the app, we assume that there might be design characteristics that are omnipresent among the components of an application. Primarily, in this study, we focus on object-oriented design features defined by the researchers: design patterns [13] and fuzzy signatures [33].

A. Design Patterns

Design patterns are one of the most used features for developing complex and large systems based on object-oriented platforms. A design pattern [17, 30] is *"a standard solution to a common programming problem which is a design or implementation structure that achieves a particular purpose"*. Although not compulsory, as design patterns guide the implementation structure in the code-base, they are being widely used [11] in

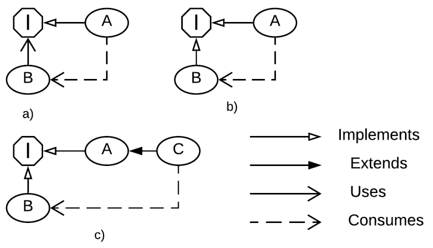

Fig. 1. Basic class relation in design patterns: (a) Observer, (b) Adapter, (c) Composite.

developing applications. Purposes of various design patterns are shown in Table I. Android framework itself and the app developers are likely to adopt design patterns. Design patterns in Android app codebase might introduce different relationships and interactions among classes, objects, and actions. Therefore, we assume that design patterns in the codebase can be a representational and complementary characteristic for Android app analysis in various dimensions. For example, adapter pattern [13] orchestrates incompatible classes to work together by converting the interface of one class to an interface expected by the clients. A basic representation of a few of the design patterns is shown in Figure 1. The relationship presented in Figure 1(a) is from the example observer pattern in Listing 1. In the sample adapter pattern in Figure 1(b), an interface *I* is implemented by both classes *A* and *B*; class *A* consumes the object of class *B*. In an abstract view, a design pattern forms a complex relationship [30] among classes concerning interface implementation, inheritance, consuming objects, and calling methods. Adapter pattern can hide a sensitive method call through updating the method signature. In practice, a design pattern can be employed in various ways. In this study, we are restricted into 11 design patterns with two variations of each. A total list of design patterns is found in $www.oodesign.com/$, and $sourcemaking.com/design\_patterns$.

Listing 1. Observer pattern. Subject broadcasts events to registered Observer

```
abstract class Observer {
   protected Subject subject;
   public abstract void update();
}
class Subject {
  public void add(Observer o) {
      observers.add(o);
   }
   public int getState() {
      return state;
   }
   public void setState(int value) {
      this.state = value; execute(); }
   private void execute() {
      ..  }
}
class BinObserver extends Observer {
   public BinObserver(Subject subject) {
      ......... }
   public void update() {
   ..(subject.getState()); }
}
public class ObserverDemo {
   public static void demo( ) {
      Subject sub = new Subject();
      new BinObserver(sub);
      ...  }       }
```

B. Fuzzy Signature:

Fuzzy signature [35] of a method as a simplified method signature is generated by (1) removing the method name with access modifier and (2) replacing all built-in classes defined within the app with a single placeholder name, and (3) removing all variables. For example, in Listing 1, the fuzzy signature of the *add* method is *void(X)*, and fuzzy signature of the method *setState* is *void(int)*. Fuzzy signature [35] can be used for similarity matching of a component which is obfuscated within the APK. In our study, we utilize the fuzzy signature along with a fuzzy hash to match two design patterns from two apps/libraries.

C. Design Pattern Extraction Tool from APK

A methodology [6] is available for detecting UI (User Interface) design patterns (layout) of the Android app. Some of the tools [17, 30] are also available for design pattern detection from source code of a typical project. But, no tool is available to detect object-oriented design patterns from Android APK or dex file. Therefore, to conduct our study, at first, we developed a tool named DPAK for detecting design patterns from both APK and Jar file format using Soot [22] tools. Design pattern is extracted based on the standard assumptions found in the literature [17, 30]: (i) most design patterns involve class hierarchies since they usually include at least one abstract class/interface in one of their roles except single-tone pattern, and (ii) each pattern role is associated with one class although there are exceptions for few of the cases. We primarily used the abstract class relationship to detect each of the patterns as shown in Figure 1, then method call is used to detect more subtle patterns. Our tool can detect design patterns from obfuscated apps as well. We developed a demo library which contains those design patterns and a demo APK integrating that library for testing the design patterns detection tool. We also manually verified some random design patterns in the APKs used for our experiments. Each extracted design pattern is presented as a design digest as follows: $< P\ n > b = F_b; ob = F_{ob}; ca = F_{ca}; cb = F_{cb}; sub = F_{sub}; sup = F_{sup}....$

Here, P is the pattern name, n total number of candidates, b is the base class or interface which is the major entity over which a pattern ecosystem is evolved. For the example pattern in Listing 1, we consider $Observer$ abstract class as the base (in other patterns we treat it as the super class (sup) since it is extended by others). $BinObserver$ class in the Listing is the client class a (ca) which we treat as sub class (sub) for other pattern. Where $Subject$ class is the other client class (cb). Here, F_b is the computed fuzzy signature of the base class (b) and others are the fuzzy signatures of the corresponding classes. We integrated ORLIS [3] in our tool for computing fuzzy signature from a pattern digest, and integrated SSDeep source code [5] for computing fuzzy hash which is used for comparing two patterns based on the fuzzy signature.

D. Code Obfuscation:

Code obfuscation [19, 25, 33] is widely employed for preventing reverse engineering through information hiding.

Obfuscation tools such as Allatori [1] can opaque identifier names, method names, class names, and package names. Moreover, the tools can change the package hierarchy through (1) repackaging classes from several packages into a new, different package and (2) flattening the package hierarchy. Both Allatori [1] and DashO [2] can alter the program's control flow and encrypt the constant strings. The tools also can remove unused code and add utility methods in the new code.

E. Fuzzy Hash:

Fuzzy hash is originally a Context Triggered Piecewise Hash (CTPH) [21] and used in computer forensic – *such hashes can be used to identify ordered homologous sequences between unknown inputs and known files even if the unknown file is a modified version of the known file*. For example, the usual Fuzzy signature of the $Subject$ class [*void(X), int, void(int), void*] might be changed to [*void(X), int, void, void(int), void*] due to obfuscator added an extra dummy method [19]. CTPH can return a matching score for these two almost identical data. The main difference of fuzzy hash [21] is that current piecewise hashing programs used fixed offsets to determine when to start and stop the typical hash algorithm, a CTPH algorithm uses the *rolling hash*. Consequently, if one or two bytes are changed in the input, instead of changing total hash only one of the hash values is changed. Since the major portion of the signature remains the same, contents with reformations can still be related with the CTPH signatures of known contents. This algorithm is implemented as SSDeep tool.

IV. Dataset

For our empirical study, we collected various standard datasets widely used in the published research projects. We collected 453 3rd party libraries, 266 general APKs, and three types (ProGuard [4], Dasho [2], and Allatori [1]) of obfuscated apps from ORLIS [33] project. There are 659 obfuscated apps in the ORLIS collection. We also collected 5,560 malware from the Drebin [7] project which contains both general and obfuscated APKs. For testing purpose of our DPAK tool, we develop a demo library Jar and demo APK containing the example design patterns which we make available online [1].

V. Experimental Outcome

A. Answering RQ1: Design features in 3rd Party Libraries

For answering RQ1, we experimented with the 453 collection of Java 3rd party libraries. Then we run the DPAK tool in the Jar mode with this dataset. The distribution of different patterns within the library dataset are presented in Table I. In the output of the tool, 33% libraries (149) among 453 3rd party libraries contain 11 design patterns. Given the only design feature, this is a significant presence. When these libraries are integrated with a project, these design patterns might be preserved irrespective of code obfuscation (we will investigate in the next section). The collection of libraries are open source and cover diverse areas of functionalities (not only Android applications). From the distribution, we observe that different types of object-oriented design solutions are adopted by the developers. Other categories (such as design principals) of design features might be possible to extract and track the distinguishing design behaviours which are also true for Android apps. Therefore, design features are contained in the 3rd party libraries considerably.

TABLE I
Distribution of design patterns in libraries and APKs

Patterns	#in libs	#in APKs	Properties [13]
observer	559	1222	*Defines a one to many relationship, so that when one object changes state, the others are notified and updated automatically*
factory	508	1127	*Defines an interface for creating objects, subclasses decide which classes to instantiate*
builder	270	626	*Separates the construction of a complex object, so the same construction process can create different representations*
visitor	305	796	*Represents an operation to be performed on the elements of an object structure, without changing the classes on which is operates*
abstract-factory	305	652	*Provides an interface for creating related objects without specifying concrete classes*
objectpool	168	311	*Used in situations where the cost of initializing a class instance is high*
adapter	176	454	*Allows incompatible classes to work together by converting the interface of one class to an interface expected by the clients*
composit	76	138	*Composes objects into tree structures, and lets clients treat individual objects and compositions uniformly*
chain	184	598	*Avoids coupling the sender of a request to the receiver*
decorator	40	74	*Attaches additional responsibilities to an object dynamically*
prototype	83	243	*Specifies the kind of objects to instantiate using a prototypical instance*

B. Answering RQ2 and RQ3: Design features in Android apps

In the previous section, it is evident that the design features in 3rd party libraries play an influential role. Those design features will likely be present in an Android application as well. To figure out the importance of design features in apps (i.e., the answer of RQ2), we run our tool (excluding the libraries provided by the Android framework) with the ground truth of 266 apps in ORLIS study [33]. Among these apps, 226 contains 3rd party libraries. The outcome is promising as we found that 157 are detected (60%) as containing design patterns. All the app that contain these patterns, 95% of them contains 3rd party libraries. The distribution of individual category of design patterns are shown in Table I. Moreover, we run our tool with various obfuscated APKs which outcome is shown in Table II, where DF is design features and "%DF=Libs" is percentage of apps that contain DF also contain 3rd party libraries. In the Proguard collection, 46% apps containing design patterns also contain 3rd party libraries, whereas this is 98% for the Allatori collection. The lower outcome in Proguard is due to 203 collections contain many APKs originated from the original 266 ground truth where the 11 design patterns do not exist. Overall, we can deduct that these features can predict 85% of the existence of 3rd party libraries in obfuscated APKs without prior knowledge-database. Table II also displays unsuccessful

[1]github.com/akm523/AndroidApps

TABLE II
DESIGN FEATURES IN VARIOUS OBFUSCATED APPS.

Data	#APKs	#Contains DF	#Contains Libs	%DF=Libs	Unsuccessful
Allatori	241	212	208	98%	0
Dasho	215	149	180	83%	2
Proguard	203	78	171	46%	0

TABLE III
LIBRARY DETECTION OUTCOME OF VARIOUS OBFUSCATED APPS.

Data	APK+Lib	#R	# P_F	#Libs	#R	T	F1
Allatori	208	209	3	431	84	53	31%
Dasho	180	132	3	318	86	33	32%
Progurd	171	52	0	239	102	27	33%

R-recall, P_F-false positive, T-true positive, F1- F1 score calculated from R and T

obfuscated APKs to extract design patterns by our tool which are also contained in the 266 ground truth without obfuscation. From Table II we notice that obfuscation does not impact design pattern extraction as there are no unsuccessful APKs except two which answers our RQ3; design patterns for two apps in Dasho dataset are unsuccessful due to (i) Dasho removes some candidate components, and (ii) exception during parsing the APK by the Soot [22] tool. We can conclude that many of these design patterns are inherited due to the adoption of 3rd party libraries within an Android application, and these patterns are obfuscation resilient.

C. Answering RQ4: Design features in Android malware

In Section V-B, our experiment reveals that in general Android apps contain design patterns. However, how malwares applications are defined using design features would be an important direction of research for overcoming the challenges in detecting and vetting security and privacy leaks. Many automated malware analysis process [8, 29] can not be completed using the existing techniques for large call graph or flow graph of the code base. Therefore, components need to be segregated during analysis process to overcome such challenges. Focusing this, we investigate how design features are present in malwares. So, to answer RQ4, we run our DPAK tool with the widely used Android malware collection of drebin dataset [7]. In Drebin-5 collection (random pick) of Android malware, 61% malware contain design patterns. This finding is very much encouraging as it helps to develop techniques to accelerate malware detection with more precision rate irrespective of obfuscations for one of the two purposes: (i) partitioning and filtering components, (ii) different components are analyzed differently than the written code.

D. Answering RQ5: Library detection using Design Features

Third party libraries are present [32] within 60% APKs, and one of the most important components to be separated. Existing techniques [14, 25, 33] detect 3rd party libraries with a promising outcome. According to our analysis, library detection within the mobile apps still requires improvement in many areas such as accuracy, less prior database dependency and obfuscation resiliency. In the Section V-B, we notice that design patterns have 85% predictability of existence of library within an obfuscated APK without prior knowledge. In this section, we discuss how design patterns alone can be utilized to detect obfuscation resilient libraries in the APK. At first, a database is created with the summary of fuzzy signature of extracted design patterns for the collection of 453 3rd party libraries with our developed tool discussed in Section III-C. Then, design pattern digest and their fuzzy signatures within each APK, designated as the candidate components, are matched adopting context triggered piece-wise hashes (CTPH) [21] computed by the SSDeep library. An app might contain multiple libraries and their corresponding design patterns. Our matching technique performs similarity check on individual design pattern digest i.e., fuzzy signatures of all the classes involved within a pattern as in Section III-C. However, there are many libraries which have same types of design pattern with overlapping fuzzy components since a fuzzy signature consists of mostly the combination of *int,void,X,boolean* and so on. Fuzzy hash provides a score on block-wise matches. Consequently, a common threshold of similarity score detects many libraries falsely. Therefore, we compute similarity score cumulatively on all categories of patterns and ranked the matched libraries and only consider the top ranked libraries to reduce the false positive. Detection logic is presented in Algorithm 1. Design pattern directly detects libraries among 101 APKs (out of 226, 45%) in the ground truth collection; only false positive is 5, where 102 libraries are detected (among 465). Therefore, for non-obfuscated dataset, recall R=22%, precision P=51% and F1=31%. Moreover, the experimental outcome with the three types of obfuscated datasets is presented in Table III. F1 score for the obfuscated datasets is also around 32%. The F1 score of Orlis [33] and LibDetect [14] for the same dataset are 67% and 17% respectively. However, those techniques require to calculate fuzzy signature and match fuzzy hash for all the classes and methods within an APK. In summary, given the challenges, design features show encouraging outcome for detecting the 3rd party libraries.

VI. RELATED WORK

A. Study of Design Features in Android Applications

A few of the tools have been proposed [6, 23] for analyzing UI design patterns of Android applications but those are not traditional object-oriented design patterns [13]. A few of other studies have explored type [10] and anti-pattern [20] (bad design choice) based characterization. However, in our empirical study, we extracted and experimented the existence of design patterns in various types of Android apps.

B. Library Detection

LibDetect [14] uses five different abstract digests of a method's bytecode to match app methods against library methods whose recall rate is only 10%. Recently, ORLis [33] is proposed for detecting 3rd party libraries within obfuscated Android APK using fuzzy signatures where fuzzy signatures from all the classes are computed and fuzzy hash is used for similarity matching. This approach has both memory and time overhead as we run the tool. In this study, we proposed a method based on design patterns which require fuzzy hash

```
Data: $F_A$ is the digest of an APK, $F_n$-is the fuzzy digest of the
      libraries, $T$-defined threshold.
Result: $M$ - Indices of matched $F_n$
begin
    A library digest $F_i - > F_{adapter}, F_{observer}....$
    A pattern digest $F_P - > F_b; F_{ob}; F_{ca}; ....$
    foreach fuzzy digest $F_i \in F_n$ do
        foreach pattern digest $F_{AP} \in F_A$ do
            totalScore = 0
            counted = 0
            foreach class digest $F_{AC} \in F_{AP} \& F_{iC} \in F_i$ do
                if $F_{AC} \neq \emptyset$ then
                    totalScore.add(SSDeepMatch($F_{AC}, F_{iC}$))
                     counted = counted+1
                end
            end
            totalScore = totalScore/counted
            if $totalScore > T$ then
                $M.apend(i)$
            end
        end
    end
end
```

Algorithm 1: Major similarity matching logic for detecting library components

but only for a handful of classes within the APKs. Libd [25] is proposed to detect 3rd party libraries using package and class relationship. Close to this technique, our empirical study also used class relationship but we adopted more complex and abstract patterns of relationship. Libradar [27] is proposed using package structures which has a significant performance gap in terms of accuracy. However, the major difference with our investigated method is that none of the above mentioned methods can predict a library component without 3rd party library database, whereas our design based features can predict 3rd party library components independently of library database and code-obfuscation.

VII. Conclusion

In this paper, we conducted an empirical study using design patterns and fuzzy signatures to separate programming design components such as 3rd party libraries. For that purpose, we have built a system for automatically extracting design patterns from both the Android executable package and Jar of an application as we are unaware of any such tool. We have experimented the distribution of design patterns in 3rd party libraries, usual apps and Android malwares. The experimental outcomes with various standard datasets are encouraging as both obfuscated and malware apps contain such features significantly (around 60%). Finally, as a case study, we have shown that the design patterns alone can detect 3rd party libraries within obfuscated apps considerably (F1 score is 32%). Overall, our empirical study confirms that design features play a versatile role in separating various Android components for various purposes. In future, we will extend our work for separating other app components with more features such as micro-architecture and design motifs.

Acknowledgment

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] Allatori: www.allatori.com.

[2] Dasho: www.preemptive.com/company.

[3] Orlis: github.com/presto-osu/orlis-orcis/tree/master/orlis.

[4] Proguard: Proguard.developer.android.com/studio/build/shrink-code.html.

[5] Ssdeep: github.com/openpreserve/bitwiser.

[6] K. Alharbi and T. Yeh. Collect, decompile, extract, stats, and diff: Mining design pattern changes in android apps. In *Proc. of MobileHCI*, pages 515–524, 2015.

[7] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, pages 23–26, 2014.

[8] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Notices*, pages 259–269, 2014.

[9] M. Backes, S. Bugiel, E. Derr, S. Gerling, and C. Hammer. R-droid: Leveraging android app analysis with static slice optimization. In *Proc. of ASIACCS*, pages 129–140, 2016.

[10] K. Choi and B.-M. Chang. A type and effect system for activation flow of components in android programs. *IPL*, pages 620–627, 2014.

[11] H. Ergin. *Design Patterns for Model Transformations*. PhD thesis, The University of Alabama, 2014.

[12] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su. Large-scale analysis of framework-specific exceptions in android apps. In *Proc. of ICSE*, pages 408–419, 2018.

[13] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[14] L. Glanz, S. Amann, M. Eichberg, M. Reif, B. Hermann, J. Lerch, and M. Mezini. Codematch: obfuscation won't conceal your repackaged app. In *Proc. of FSE*, pages 638–648, 2017.

[15] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proc. of WiSec*, pages 101–112, 2012.

[16] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: Putting icing on the uml cake. In *SIGPLAN Notices*, pages 301–314, 2004.

[17] Y.-G. Guéhéneuc and G. Antoniol. Demima: A multilayered approach for design pattern identification. *TSE*, pages 667–684, 2008.

[18] Y.-G. Guéhéneuc and N. Jussien. Using explanations for design-patterns identification. In *IJCAI*, pages 57–64, 2001.

[19] M. Hammad, J. Garcia, and S. Malek. A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products. In *Proc. of ICSE*, pages 421–431, 2018.

[20] G. Hecht, R. Rouvoy, N. Moha, and L. Duchien. Detecting antipatterns in android apps. In *Proc. of MOBILESoft*, pages 148–149, 2015.

[21] J. Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital investigation*, pages 91–97, 2006.

[22] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Workshop of CETUS*, pages 15–35, 2011.

[23] J. Lehtimaki. *Smashing Android UI: Responsive User Interfaces and Design Patterns for Android Phones and Tablets*. John Wiley & Sons, 2012.

[24] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon. Static analysis of android apps: A systematic literature review. *IST*, pages 67–95, 2017.

[25] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo. Libd: scalable and precise third-party library detection in android markets. In *Proc. of ICSE*, pages 335–346, 2017.

[26] D. Lucredio, E. S. de Almeida, and R. P. Fortes. An investigation on the impact of mde on software reuse. In *Proc. of CBSOFT*, pages 101–110, 2012.

[27] Z. Ma, H. Wang, Y. Guo, and X. Chen. Libradar: fast and accurate detection of third-party libraries in android apps. In *Proc. of ICSE-C*, pages 653–656, 2016.

[28] Z. Shan, I. Neamtiu, and R. Samuel. Self-hiding behavior in android apps: detection and characterization. In *Proc. of ICSE*, pages 728–739, 2018.

[29] F. Shen, J. Del Vecchio, A. Mohaisen, S. Ko, and L. Ziarek. Android malware detection using complex-flows. *Transactions on Mobile Computing*, 2018.

[30] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design pattern detection using similarity scoring. *TSE*, pages 896–909, 2006.

[31] H. Wang and Y. Guo. Understanding third-party libraries in mobile app analysis. In *Proc. of ICSE-C*, pages 515–516, 2017.

[32] H. Wang, Y. Guo, Z. Ma, and X. Chen. Wukong: A scalable and accurate two-phase approach to android app clone detection. In *Proc. of STA*, pages 71–82, 2015.

[33] Y. Wang, H. Wu, H. Zhang, and A. Rountev. Orlis: Obfuscation-resilient library detection for android. In *Proc. of MOBILESoft*, pages 13–23, 2018.

[34] W. Yang, J. Li, Y. Zhang, Y. Li, J. Shu, and D. Gu. Apklancet: tumor payload diagnosis and purification for android applications. In *Proc. of AsiaCCS*, pages 483–494, 2014.

[35] W. Yang, M. Prasad, and T. Xie. Enmobile: Entity-based characterization and analysis of mobile malware. In *Proc. of ICSE*, pages 384–394, 2018.