Detection of Repackaged Android Malware with Code-Heterogeneity Features

Ke Tian, Danfeng (Daphne) Yao, *Member, IEEE*, Barbara G. Ryder, Gang Tan and Guojun Peng

Abstract—During repackaging, malware writers statically inject malcode and modify the control flow to ensure its execution. Repackaged malware is difficult to detect by existing classification techniques, partly because of their behavioral similarities to benign apps. By exploring the app's internal different behaviors, we propose a new Android repackaged malware detection technique based on *code heterogeneity analysis*. Our solution strategically partitions the code structure of an app into multiple dependence-based regions (subsets of the code). Each region is independently classified on its behavioral features. We point out the security challenges and design choices for partitioning code structures at the class and method level graphs, and present a solution based on multiple dependence relations. We have performed experimental evaluation with over 7,542 Android apps. For repackaged malware, our partition-based detection reduces false negatives (i.e., missed detection) by 30-fold, when compared to the non-partition-based approach. Overall, our approach achieves a false negative rate of 0.35% and a false positive rate of 2.97%.

Index Terms—Android Security, Malware Detection, Repackaged Malware.

1 Introduction

The ease of repackaging Android apps makes the apps vulnerable to software piracy in the open mobile market. Developers can insert or modify parts of the original app and release it to a third party market as new. The modification may be malicious. Researchers found 80.6% of malware are repackaged malware, which demonstrates the popularity and severity of repackaged malware [2].

There are two categories of techniques for detecting repackaged malware, *i)* similarity-based detection specific to repackaged malware and *ii)* general purpose detection. Specific solutions for repackaged Android apps aim at finding highly similar apps according to various similarity measures. For example, in ViewDroid [3], the similarity comparison is related to how apps encode the user's navigation behaviors. DNADroid [4] compares the program dependence graphs of apps to examine the code reuse. MassVet [5] utilizes UI structures to compare the similarity among apps. Juxtapp [6] and DroidMOSS [7] examine code similarity through features of opcode sequences.

Although intuitive, similarity-based detection for repackaged malware may have several technical limitations. The detection typically relies on the availability of original apps for comparison, thus is infeasible without them. The pairwise based similarity computation is of quadratic complexity $O(N^2)$ in the number N of apps analyzed. Thus, the analysis is extremely time-consuming for large-scale screening.

General-purpose Android malware detection techniques (e.g., permission analysis [8], dependence analysis [9], API mining [10]) have a limited capability in detecting repackaged malware. The reason is that these analyses are performed on the *entire app*, including both the injected malicious code and the benign code inherited from the original app. The presence of benign code in repackaged malware substantially dilutes malicious features. It skews the classification results, resulting in false negatives (i.e., missed detections). In a recent study [11], researchers found that most missed detection cases are caused by repackaged malware. Thus, precisely recognizing malicious and benign portions of code in one app is important in improving detection accuracy.

We aim to significantly improve repackaged malware detection through designing and evaluating a new partition-based classification technique, which explores code heterogeneity in an app. Repackaged malware is usually generated by injecting malicious components into an original benign app, while introducing no control or data dependence between the malicious component and the original app.

We examine Android programs for code regions that seem unrelated in terms of data/control dependences. Regions are formed through data/control dependence analysis and their behavior is examined with respect to security properties (e.g., calling sensitive APIs). We refer to code in different regions as *heterogeneous code* if regions of the program exhibit distinguishable security behaviors.

Recognizing code heterogeneity in programs has security applications, specifically in malware detection. Repackaged Android malware is an example of heterogeneous code, where the original app and injected component of code have quite different characteristics (e.g., the frequency

- *K. Tian, D. Yao and B. Ryder are with the Department of Computer Science, Virginia Tech, Blacksburg, VA, 24060.*
E-mail: {ketian, danfeng, ryder}@cs.vt.edu
- *T. Gan is with the Department of Computer Science and Engineering, Penn State University, University Park, PA 16802.*
E-mail: gtan@cse.psu.edu
- *G. Peng is with the School of Computer, Wuhan University, Wuhan, China 430072.*
E-mail: guojpeng@whu.edu.cn

A preliminary version of the work appeared in the Proceedings of the IEEE Mobile Security Technologies (MoST) workshop, in conjunction with the IEEE Symposium on Security and Privacy [1].

of invoking critical library functions for accessing system resources). We are able to locate malicious code by distinguishing different behaviors of the malicious component and the original app.

Our main technical challenge is how to identify integrated coherent code segments in an app and extract informative behavioral features. We design a partition-based detection to discover regions in an app, and a machine-learning-based classification to recognize different internal behaviors in regions. Our detection leverages security heterogeneity in the code segments of repackaged malware. Our algorithm aims to capture the semantic and logical dependence in portions of a program. Specifically, we refer to a *DRegion* (Dependence Region) as a partition of code that has disjoint control/data flows. DRegion is formally defined in Def. 3. Our goal is to identify DRegions inside an app and then classify these regions independently. Malware that is semantically connected with benign and malicious behaviors is out of scope of our model and we explain how it impacts the detection.

While the approach of classifying partitioned code for malware detection appears intuitive, surprisingly there has not been systematic investigation in the literature. The work on detecting app plagiarism [12] may appear similar to ours. It decomposes apps into parts and performs similarity comparisons between parts across different apps. However, their partition method is based on rules extracted from empirical results, and cannot be generalized to solve our problem. A more rigorous solution is needed to precisely reflect the interactions and semantic relations of various code regions.

Our contributions can be summarized as follows:

- We provide a new code-heterogeneity-analysis framework to classify Android repackaged malware with machine learning approaches. Our prototype **DR-Droid**, realizes static-analysis-based program partitioning and region classification. It automatically labels the benign and malicious components for a repackaged malware.
- We utilize two stages of graphs to represent an app: a coarse-grained *class-level dependence graph* (CDG) and a fine-grained *method-level call graph* (MCG). The reason for these two stages of abstraction is to satisfy different granularity requirements in our analysis. Specifically, CDG is for partitioning an app into high-level DRegions; MCG is for extracting detailed call-related behavioral features. CDG provides the complete coverage for dependence relations among classes. In comparison, MCG provides a rich context to extract features for subsequent classification.
- Our feature extraction from individual DRegions (as opposed to the entire app) is more effective under existing repackaging practices. Our features cover a wide range of static app behaviors, including user-interaction related benign properties.
- Our experimental results show a 30-fold improvement in repackaged malware classification. The average false negative rate for our partition- and machine-learning-based approach is 30 times lower than the conventional machine-learning-based approach (non-partitioned equivalent). Overall, we achieve a low false negative rate of 0.35% when evaluating malicious apps, and a false positive rate of 2.96% when evaluating benign apps.

The significance of our framework is the new capability to provide in-depth and fine-grained behavioral analysis and classification on programs.

2 Overview and Definitions

In this section, we present our attack model, technical challenges associated with partitioning, and the definitions needed to understand our algorithms.

Repackaged malware seriously threatens both data privacy and system integrity in Android. There are at least two types of malware abuse through repackaged malware, data leak and system abuse. The danger of repackaged malware is that the malicious code is deeply disguised and is difficult to detect. Repackaged malware appears benign and provides useful functionality; however, they may conduct stealthy malicious activities such as botnet command-and-control, data exfiltration, or DDoS attacks. Our work described in this paper can be used to screen Android apps to ensure the trustworthiness of apps installed on mission-critical mobile devices, and to discover new malware before they appear on app markets.

Assumption. Our security goal is to detect repackaged malware that is generated by trojanizing legitimate apps with a malicious payload, where the malicious payload is logically and semantically independent of the original benign portion. This assumption is reasonable because all the repackaged malware in the existing dataset contains disjoint code.

How to analyze the more challenging case of connected graphs in repackaged malware is out of the scope of our detection. Mitigations are discussed in Section 6. Our approach is focused on automatically identifying independent partitions (DRegions) of an app, namely partitions that have disjoint control/data flows. We perform binary classification on each element of the DRegion.

2.1 Challenges and Requirements

We analyze dependence-based connectivity as the specific heterogeneous property in code. Heterogeneous code can then be approximated by finding disjoint code structures in Android event relation/dependence graphs. We aim to detect repackaged malware by identifying different behaviors in its heterogeneous code. Therefore, how to achieve an efficient partition and to acquire representative behaviors of each partition are key research questions.

Partition Challenges: One may analyze dependence relations for the purpose of code partition. A straightforward approach is to partition an app into clusters of methods based on function call relations [13]. However, this straightforward approach cannot solve the following challenges:

- *Inaccurate representation of events*. Method-level representation is less informative than class-level representation for profiling relations of *events*. An Android app is composed of different types of events (e.g., activities, services and broadcasts). An Android event

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2745575, IEEE Transactions on Dependable and Secure Computing

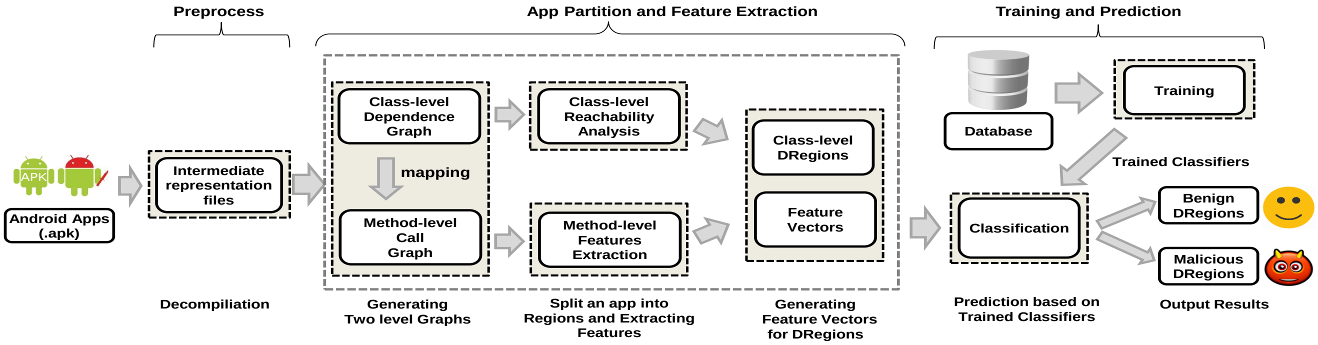

Preprocess
App Partition and Feature Extraction
Training and Prediction
APK
Android Apps (.apk)
Intermediate representation files
Decompiliation
Class-level Dependence Graph
mapping
Method-level Call Graph
Generating Two level Graphs
Class-level Reachability Analysis
Method-level Features Extraction
Split an app into Regions and Extracting Features
Class-level DRegions
Feature Vectors
Generating Feature Vectors for DRegions
Database
Training
Trained Classifiers
Classification
Prediction based on Trained Classifiers
Benign DRegions
Malicious DRegions
Output Results

Fig. 1: Workflow of our partition-based Android malware detection.

is implemented by extending an Java class. Class information for events is scattered or lost in conventional method-level graphs. Furthermore, method-level call analysis cannot resolve the implicit calls within a life-cycle of event methods (e.g., OnCreate, OnStart, and onPause). There are no direct invoking relations among event methods. (These methods are managed by an activity stack by the Android system.) Thus, method-level call partition would generate an excessive number of subcomponents, which unnecessarily complicates the subsequent classification.

- *Incompleteness of dependence relations*. Call relations alone cannot accurately represent all possible dependence relations. Dependences may occur through data transformation. Android also has asynchronous callbacks, where the call relations are implicit. Thus, focusing on call dependence relations alone is insufficient.

Our approach for partitioning an app is by generating the class-level dependence graph (CDG) through exploring different categories of dependence relations. To partition an app into semantic-independent regions, a class-level representation is more suitable to measure the app semantic dependence relations.

Classification Challenges: Extracting meaningful features to profile each region is important for classification. In our partitioned setting, obstacles during feature extraction may include the following:

- *Inaccurately profiling behaviors*. Class-level dependences are coarse-grained. They may not provide sufficient details about region behaviors needed for feature extraction and classification. For example, the interactions among components within the Android framework may not be included.
- *Insufficient representative features*. Features in most of the existing learning based solutions are aimed at characterizing malicious behaviors, e.g., overuse of sensitive APIs. This approach fails to learn benign properties in apps. This bias in recognition may result in missed detection and evasion.

Our approach for achieving highly accurate classification is by extracting semantic features from method-level call graph (MCG). With the help of the MCG, we extract features (e.g., sensitive APIs and permission usage in existing approaches [10], [14]) to monitor malicious behaviors. Furthermore, we discover new user interaction features with the combination of graph properties to screen benign behaviors.

2.2 Definitions

We describe major types of class-level dependence relations later in Def. 1. These class-level dependence relations emphasize on the interactions between classes.

Definition 1. *We define three types of class-level dependence relations in an Android app.*

- ***Class-level call dependence.*** *If method m' in class C' is called by method m in class C, then there exists a class-level call dependence relation between C and C', which is denoted by $C \to C'$.*
- ***Class-level data dependence****. If variable v' defined in class C' is used by another class C, then there exists a data dependence relation between C and C', which is denoted by $C \to C'$.*
- ***Class-level ICC dependence****. If class C' is invoked by class C through explicit-intent-based inter-component communication (ICC), then there exists an ICC dependence relation between C and C', which is denoted by $C \to C'$.*

The ICC dependence is specific to Android programs, where the communication channel is constructed by using *intents* [15]. For the ICC dependence definition, our current prototype does not include implicit intent, which is not common for intra-app component communication. The dependence relations via implicit-intent based ICCs cannot be determined precisely enough in static program analysis.

Definition 2. ***Class-level dependence graph*** *(CDG) of an app $G = \{V, E\}$ is a directed graph, where V is the vertex set and E is the edge set. Each vertex $n \in V$ represents a class. The edge $e = (n_1, n_2) \in E$, which is directed from n_1 to n_2, i.e., $n_1 \to n_2$. Edge e represents one or more dependence relations between n_1 and n_2 as defined in Definition 1.*

The purpose of having our customized class-level dependence graphs is to achieve complete dependence coverage and event-based partition. The graph needs to capture interactions among classes. We define method-level call dependence and how to build the method-level call

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2745575, IEEE Transactions on Dependable and Secure Computing

graph (MCG) based on this definition. We formally define DRegions through class-level dependence connectivity.

Definition 3. *Given class-level dependence graph $G(V, E)$ of an Android application, DRegions of the application are disjoint subsets of classes as a result of a partition that satisfies following two properties.*

1) *Dependence relations among the classes within the same DRegion form a directed connected graph. Formally, given a DRegions R, for any two classes $(C_i, C_j) \in R$, $\exists$ a path $\vec{p} = (C_i = C_0, C_1, \ldots, C_k = C_j)$ that connects C_i and C_j.*
2) *There exist no dependence relations between classes appearing in two different DRegions. Formally, given two DRegions R_i and R_j, for any class $C_i \in R_i$ and any class $C_j \in R_j$, $\nexists$ a path $\vec{p} = (C_i = C_0, C_1, \ldots, C_k = C_j)$ that connects C_i and C_j.*

Definition 4. ***Method-level call dependence.*** *If method m calls method m', then there exists a method-level call dependence relation between m and m', which is denoted by $m \rightarrow m'$. Method m and m' may belong to the same or different classes and one of them may be an Android or Java API.*

The purpose of constructing method-level call graphs is to extract detailed behavioral features for classifying each DRegion. The method-level call graph contains the app's internal call relations, and the interactions with the Android framework and users.

2.3 Workflow

Figure 1 shows the framework of our approach. Our approach can be divided into the following major steps:

1) **IR Generation**. Given an app, we decompile it into the intermediate representations (IR), which may be Java bytecode, Smali[1] code, or customized representation. The IR in our prototype is Smali code.
2) **CDG and MCG generation**. Given the IR, we generate both class-level and method-level dependence relations through the analysis on the Smali opcodes of instructions. We use the obtained dependence relations to construct the class-level dependence graphs (CDG) and method-level call graphs (MCG).
3) **App partition and mapping**. Based on the CDG, we perform reachability analysis to partition an app into disjoint DRegions. We map each method in MCG to its corresponding class in CDG by maintaining a dictionary data structure.
4) **Generating feature vectors**. We extract three categories of features from each DRegion in MCG. We construct a feature vector of each DRegion to describe its behaviors.
5) **Training and classification**. We train classifiers on the labeled data to learn both benign and malicious behaviors of DRegions. We apply classifiers to screen new app instances by individually classifying their DRegions and integrating the results.

In order to determine the original app, from which a flagged malware is repacked, similarity comparisons need to be performed. Our comparison complexity $O(mN)$ would be much lower than the complexity (N^2) of a straightforward approach, where m is our number of flagged malware and N is the number of total apps analyzed. $m \ll N$, as the number of malware is far less than the total number of apps on markets.

3 Graph Generation and Partition

In this section, we provide details of our customized class-level dependence analysis and our partition algorithm.

3.1 Class-level Dependence Analysis

Our class-level dependence analysis is focused on Android event relations. It obtains class-level dependence relations based on fine-grained method- or variable-level flows. We highlight the operations for achieving this transformation.

Data dependence. In the variable-level flow F, we trace the usage of a variable v' which is defined in class C'. In case v' is used by another class C, e.g., reading the value from v' and writing it into a variable v defined in class C, we add a direct data dependence edge from C to C' in CDG.

ICC dependence. ICC dependence is the Android specific data dependence, where data is transformed by intents through ICC. An ICC channel occurs when class C initializes an explicit intent. Method m (generally OnCreate function) in class C' is invoked from class C. By finding an ICC channel between class C and C' through pattern recognition, we add a direct ICC dependence edge from C to C' in CDG.

Call dependence. We briefly describe the operations for obtaining class-level call dependence when given the method-level call graph. We first remove the callee functions that belong to Android framework libraries. For the edge $e = \{m, m'\}$ that indicates method m' is called by method m, in case m belongs to a class C and m' belongs to class C', we add a direct call dependence edge from C to C' in CDG.

Pseudocode for generating the Class-level Dependence Graph is shown in Algorithm. 1. Function FindDependentClass (m_k^i) is used to find the class set $S(m_k^i)$ that any $C_j \in S(m_k^i)$ contains dependence relations with a class C_i $(m_k^i \in C_i)$ through control-/data-flow in method m_k^i. Functions IsDataDependent(C_i,C_j), IsICCDependent(C_i,C_j), and IsCallDependent(C_i,C_j) are used to detect our defined dependence relations between classes.

We give our implementation details to statically infer these relations in Section 5.1. All four dependence relations can be identified by analyzing instructions in IR. The complexity of connecting the class-level call graph is $O(\mathcal{N})$, where $\mathcal{N}$ is the total number of the instructions in the IR decompiled from an app. We do not distinguish the direction of the edges when partitioning the CDG.

3.2 App Partition and Mapping Operations

The goal of app partition operation is to identify logically disconnected components. The operation is based on the class-level dependence graph (CDG). We use reachability analysis to find connected DRegions. Two nodes are regarded as neighbors if there is an edge from one node to the

[1] https://ibotpeaches.github.io/Apktool/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2745575, IEEE Transactions on Dependable and Secure Computing

Algorithm 1 Class-level Dependence Graph Generation

```
Require: the class-set $S_C = \{C_i, C_2, \dots, C_n\}$, each class $C_i$ represents a list of methods. the Method-set of $C_i$: $S_m^i = \{m_1^i, \dots, m_k^i\}$, where $m_j^i \in C_i$ is a list of instructions in IR.
Ensure: class-level dependence graph $CDG = \{V, E\}$.
$V = \emptyset$, $E = \emptyset$
function GEN_CDG($CDG, S_C$)
    for each $C_i \in S_C$ do
        for each $m_k^i \in S_m^i$ do
            $S(m_k^i)$ = FINDDEPENDENTCLASS($m_k^i$)
            for each $C_j \in S(m_k^i)$ do
                if ISDATADEPENDENT($C_i$,$C_j$) then
                    UPDATECDG($C_i$,$C_j$,$CDG$)
                end if
                if ISICCDEPENDENT($C_i$,$C_j$) then
                    UPDATECDG($C_i$,$C_j$,$CDG$)
                end if
                if ISCALLDEPENDENT($C_i$,$C_j$) then
                    UPDATECDG($C_i$,$C_j$,$CDG$)
                end if
            end for
        end for
    end for
    return $CDG$
end function
function UPDATECDG($C_i$,$C_j$,$CDG$)
    if $C_i \notin V$ then
        $V \leftarrow V \cup C_i$
    end if
    if $C_j \notin V$ then
        $V \leftarrow V \cup C_j$
    end if
    Edge $e = \{C_i, C_j\}$
    if $e \notin E'$ then
        $E \leftarrow E \cup e$
    end if
end function
```

other. Our algorithm starts from any arbitrary node in the CDG, and performs breadth first search to add the neighbors into a collection. Our algorithm stops when every node has been grouped into a particular collection of nodes. Each (isolated) collection is a DRegion of an app. In other words, classes with any dependence relations are partitioned into the same DRegion. Classes without dependence relations are in different DRegions.

Our mapping operation projects a method m in method-level call graph (MCG) to its corresponding class C in CDG. Mapping is uniquely designed for our feature extraction. Specifically, its purpose is to map extracted features to the corresponding DRegion. The mapping operation is denoted by $F_{mapping} : S_c \rightarrow P^m_{S_c} = \{G'_{c1}, G'_{c2}, ...\}$, where input S_c is a DRegion in CDG, and output $P^m_{S_c}$ is a set of call graphs in MCG. The mapping algorithm projects a method in MCG to a DRegion in CDG by using a lookup table. We refer to $P^m_{S_c}$ as the *projection* of S_c. Features extracted from $P^m_{S_c}$ belong to the DRegion S_c. Suppose that a method m^i exclusively belongs to a class C_i, and a class C_i exclusively belongs to a DRegion S_C, thus we have the mapping as $m^i \in C_i \in S_C \rightarrow P^m_{S_c}$. Figure 2 illustrates an example of the mapping function. Property 1 demonstrates the non-overlapping property of the mapping function.

Property 1. *If DRegion S_α and DRegion S_β are disjoint in the class-level dependence graph (CDG), then their corresponding projection $P^m_{S_\alpha}$ and projection $P^m_{S_\beta}$ are disjoint in the method-level call graph (MCG).*

Proof. We prove Property 1 by contradiction. Suppose that two methods m in class C and m' in class C' in two DRegions, there exists a path $\widehat{p} = (m, n_1, n_2, ..., m')$. For any two neighbor nodes n_i, n_{i+1} on V, n_i and n_{i+1} must be dependent through data-/control- relations: *1)* if n_i, n_{i+1} are in the same class C_1, C_1 belongs to one DRegion. *2)* if n_i, n_{i+1} are in different class C_1 and C_2, then C_1 and C_2 are connected in CDG (through a dependence edge). C_1 and C_2 are categorized to one DRegion after the partition. By induction, C and C' must belong to the same DRegion, which contradicts to our assumption. □

Our partition algorithm guarantees the non-overlapping property during the mapping operation. Features extracted from each MCG belong to exact one DRegion after partition. By the app partition and mapping, we explicitly identify each DRegion and its associated MCGs. We discuss more details on how to extract features from MCG in the following Section 4.

4 Feature Vector Generation

We analyze APIs and user interaction functions to approximate their behaviors. Our features differ from most existing features by considering DRegion behavior properties. We describe three types of features in this section.

Feature Engineering. Based on previous solutions [8] [10] as well as our own observations, we developed three feature types. Though our approach differs from whole-program strategies in that we focus only on DRegions, we too classify permissions and sensitive APIs as representative features. We also introduce new user interaction features, which until now have been ignored by previous machine learning-based solutions. Though recent rule-based approaches [11] monitor these user interaction functions for triggers to sensitive APIs, we instead extract and encode their frequencies into feature vectors. Lastly, we introduce statistic features, or coverage rate (CR), which we created after observing malware invoking numerous sensitive APIs without any user involvement. We elaborate these three types of features in Section 4.1.

4.1 Feature Extraction

Traditionally permission features analyze the registered permissions in Androidmanifest.xml as a complete unit [8]. Because our approach is focused on DRegions and different DRegions may use different permissions for various functionalities, we calculate the permission usage in each DRegion.

Type I: User Interaction Features. Malicious apps may invoke critical APIs without many user interactions [16]. User interaction features represent the interaction frequency between the app and users. Android provides UI components and each of them has its corresponding function for triggering. For example, a Button object is concatenated with onClick function, and a Menu object can be concatenated with onMenu-ItemClick function. We record the frequencies of 35 distinct user interaction functions and additional

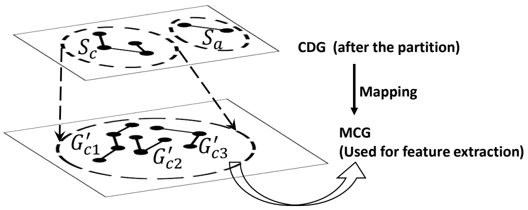


Fig. 2: An illustration of mapping operation that projects a DRegion in CDG to a set of graphs in MCG. S_c and S_a are two class-level DRegions in CDG. The *projection* of S_c consists of three method-level call graphs $\{G'_{c1}, G'_{c2}, G'_{c3}\}$.

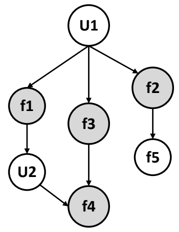


Fig. 3: An example illustrating the computation of coverage rate, U_1 and U_2 are two user interaction functions, f_1 to f_5 are five method invocations. f_1, f_2, f_3 are successors of U_1 and f_4 is the successor of U_2. The coverage rate for this DRegion is $\frac{4}{7} = 57\%$.

	True Malicious	True Benign
Detected as Malicious	TP	FP
Detected as Benign	TN	FN

TABLE 1: Semantics of true and false positive and true and false negative in our model.

2 features summarizing statistics of these functions. The statistics features represent the total number of user interaction functions and the number of different types of user interaction functions in a DRegion, respectively. We define a feature called coverage rate (CR), which is the percentage of methods directly depending on user-interactions functions. We compute the coverage rate (CR) for a *projection* $P^m_{S_c}$ of a DRegion S_c as:

$$CR(P^m_{S_c}) = \frac{\bigcup_{U \in V_i} U.successors()}{|V(P^m_{S_c})|} \tag{1}$$

The CR rate statically approximates how closely the user interacts with functions in a DRegion. In Equation (1), $P^m_{S_c}$ is the projection for a DRegion S_c in CDG. V_i is the set of user interaction methods in $P^m_{S_c}$, where $V_i \subseteq P^m_{S_c}$. $U.successors()$ is the successors vertices of method U in MCG. Any method in $U.successors()$ is *directly* invoked by U. $|V(P^m_{S_c})|$ is the total number of methods in $P^m_{S_c}$. Figure 3 shows an example to calculate the coverage rate.

Type II: Sensitive API Features. We divide sensitive APIs into two groups: Android-specific APIs and Java-specific APIs. The APIs are selected based on their sensitive operations [11]. For Android-specific APIs, we focus on APIs that access user's privacy information, e.g., reading geographic location getCellLocation, getting phone information getDeviceId. For Java-specific APIs, we focus on file and network I/Os, e.g., writing into files Write.write(), and sending network data sendUrgentData(). We extract 57 most critical APIs and 2 features on their statistic information (e.g., total count and occurrence of APIs) as features.

Type III: Permission Request Features. We analyze whether a DRegion uses a certain permission by scanning its corresponding systems calls or permission-related strings (e.g., Intent related permissions) [17]. We specify a total of 137 distinguished permissions and 2 features on permission statistics (e.g., total count and occurrence of permissions). The Android framework summarizes all the permissions into 4 groups: normal, dangerous, signature and signatureOrSystem. We record the permission usage in each group and the statistics about these groups.

Coverage rate (CR) is new. It is obtained by our empirical observation that malware invokes a large number of sensitive APIs without user's involvement. These complex features cover the behaviors of DRegions from various perspectives. We expect these features to be more obfuscation resilient than signature features extracted from bytecode or structures of the control-flow graph.

4.2 Feature Vector Analysis

We generate a feature vector for *each DRegion* of an app. For classification, each DRegion is independently classified into benign or malicious.

We perform the standard 10-fold cross-validation to calculate FNR (false negative rate), FPR (false positive rate), TPR (true positive rate) and ACC (accuracy rate) for each fold. These rates are defined as:

$$FNR = \frac{FN}{P}, \; FPR = \frac{FP}{N}$$
$$TPR = \frac{TP}{P}, \; ACC = \frac{TP + TN}{P + N}$$

where FN represents the number of false negative (i.e., missed detection), FP represents the number of false positive (i.e., false alerts), TP represents the number of true positive (i.e., accuracy of detection), TN represents the number of true negative (i.e., accuracy of identifying benign apps), P represents the number of malicious apps and N represents the number of benign apps.

Classification of Apps. Our classifiers can be used to classify both single-DRegion and multi-DRegion apps. For a multi-DRegion app after classification, we obtain a binary vector showing each DRegion marked as benign or malicious. We define the *malware score* r_m as follows:

$$r_m = \frac{N_{mali}}{N_{total}} \tag{2}$$

In Equation (2), N_{mali} is the number of DRegions labeled as malicious by classifiers, N_{total} is the total number of DRegions and $r_m \in [0, 1]$. If an app contains both malicious and benign DRegions, then we regard this app as a suspicious repackaged app.

5 Evaluation

The objective of our preliminary evaluation is to answer the following questions:

- **Q1)** Can our approach accurately detect non-repackaged malware that has a single DRegion (Section 5.2)?
- **Q2)** How much improvement is our approach in classifying repackaged malware that has multiple DRegions (Section 5.3)?
- **Q3)** Can our approach distinguish the benign and malicious code in repackaged malware (Section 5.3.1)?
- **Q4)** What is the false positive rate (FPR) and false negative rate (FNR) of our approach in classifying apps that have multiple DRegions (Section 5.3.2 and Section 5.4)?
- **Q5)** What is the performance of DR-Droid (Section 5.5)?
- **Q6)** Can our approach discover new malware (Section 5.6)?

We implement our prototype with Smali code analysis framework Androguard [2], graph analysis library networkX, and machine learning framework scikit-learn. Most existing machine learning based approaches (e.g, [10], [18]) are built on the intermediate representation with Smali code. Smali code analysis achieves large scale app screening with low performance overhead, because Smali code analysis is performed on the assembly code representation. Our current prototype is built on the Smali code for the scalability of large-scale app analysis. Our prototype is implemented in Python with total 4,948 lines of code [3].

We evaluated our approach on malware dataset Malware Genome [2] and VirusShare database [4]. We also screened 1,617 benign apps to compute false positive rate and 1,979 newly released apps to discover new malware.

5.1 Implementation Details

Building upon the method-level call graph construction [18] from Smali code, we construct more comprehensive analysis to approximate the class-level dependence graph and graph partitioning. We highlight how **DR-Droid** approximates various class-level dependence relations with intra-procedure analysis (i.e., discovering dependence relations) and inter-procedure analysis (i.e., connecting the edges). Our experiment results indicate that our dependence relations provide sufficient information for identifying and distinguishing different behaviors in an app.

Inferring class-level call dependence. Opcodes beginning with invoke represent a call invocation from this calling method to a targeted callee method. Call dependence can be inferred by parsing the call invocation. E.g., invoke-virtual represents invoking a virtual method with parameters. invoke-static represents invoking a static method with parameters and invoke-super means invoking the virtual method of the immediate parent class. We identify each instruction with invoke opcodes and locate the class which contains the callee method. The class-level call dependence is found, when the callee method belongs to another class inside the app. Because we focus on the interactions among classes, Android API calls are not included.

Cases	FNR(%)	FPR(%)	ACC(%)
KNN	6.43 ± 5.22	6.50 ± 2.67	93.54 ± 3.33
D.Tree	4.78 ± 2.90	3.52 ± 1.57	95.79 ± 2.14
R.Forest	3.85 ± 3.27	1.33 ± 0.78	97.30 ± 1.96
SVM	7.42 ± 4.85	1.46 ± 0.58	95.28 ± 2.58

TABLE 2: 10-fold cross-validation for evaluating the classifiers' performance in classifying single-DRegion apps.

Inferring class-level data dependence. Opcodes such as iget, sget, iput, and sput are related with data transformation. For example, the instruction "iget-object v0, v0, Lcom/ geinimi/AdActivity;-⟩d: Landroid/widget/Button;" represents reading a field instance into v0 and the instance is a Button object named d, which is defined in another class ⟨Lcom/geinimi/AdActivity;⟩.

Furthermore, there is a subset of opcodes for each major opcode, e.g., iget-boolean specifies to read a boolean instance and iget-char specifies to read a char instance. By matching these patterns, we obtain the data dependence among these classes.

Inferring class-level ICC dependence. To detect an ICC through an explicit intent, we identify a targeted class object that is initialized by using const-class, then we trace whether it is put into an intent as a parameter by calling Intent.setclass(). If an ICC is trigged to activate a service (by calling startService) or activate an activity (by calling startActivity), we obtain the ICC dependence between current class and the target class.

Method-level call graph construction. Our method-level call graph is constructed while we analyze call relations in the construction of the CDG by scanning invoke opcode, which is similar to the standard call graph construction [18]. We store more detailed information including the class name, as well as the method name for each vertex in MCG. For example, ⟨Landroid/telephony/SmsManager;⟩ is the class name for dealing with messages and sendTextMessage(...) is a system call with parameters to conduct the behavior of sending a message out. After the construction of MCG, we use a lookup table structure to store the projection for each DRegion in CDG and to maintain the mapping relation between a method and a DRegion.

5.2 Non-repackaged Malware Classification

Our first evaluation is on a set of non-repackaged malicious applications and a set of benign applications. Each of them contains just a single DRegion. The DRegion is labeled as benign if the app belongs to the benign app dataset, and the DRegion is labeled as malicious if the app belongs to the malicious app dataset. There are two purposes for the first evaluation: *1)* comparing the detection accuracy of different machine learning techniques, *2)* obtaining a trained classifier for testing complicated repackaged apps. The classification result is binary (0 for benign and 1 for malicious) for single DRegion apps. We evaluated four different machine learning techniques: Support Vector Machine (SVM), K-nearest

[2] http://code.google.com/p/androguard.

[3] https://github.com/ririhedou/dr_droid

[4] http://virusshare.com/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2745575, IEEE Transactions on Dependable and Secure Computing

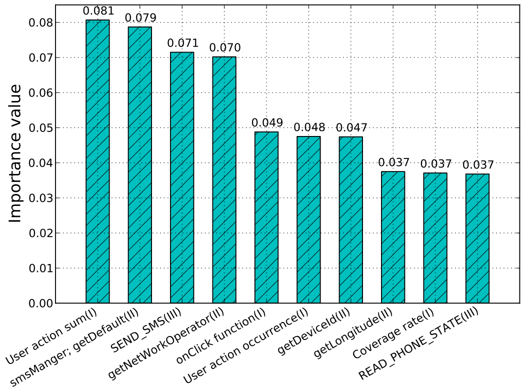


Fig. 4: Top ten important features with their ranking values, which are computed by Random Forest classifier.

neighbor (KNN), Decision Tree (D.Tree) and Random Forest (R.Forest) in non-repackaged (general) malware classification. Our training set is broadly selected from 3,325 app samples, among which 1,819 benign apps from Google Play, and 1,506 malicious apps from both Malware Genome and VirusShare.

Our feature selection step reduces the size of features from 242 to 80. We choose the radial basis function as SVM kernel, 5 as the number of neighbors in KNN, and 10 trees in the Random Forest.

We used a standard measurement 10-fold cross-validation to evaluate efficiency of classifiers. In 10-fold cross-validation, we randomly split the dataset into 10 folds. Each time, we use 9 folds of them as the training data and the 1 fold left as the testing data. We evaluate the performance of classifiers by calculating the average FPR, FNR and ACC. Our 10-fold cross-validation results are shown in Table 2, where each value is represented as the average ± the standard deviation. Figure 4 shows the top ten features with their types and ranking importance values, where coverage rate (CR) ranks the ninth. We found four of top ten important features belong to user interaction features (Type I). The user interaction features are important in our classification.

We conclude that: *1)* to answer **Q1**, DR-Droid detects non-repackaged malware with single DRegions with high accuracies. *2)* The Random Forest classifier achieves the highest AUC value 0.9930 in ROC and accurate rate (ACC) 97.3% in two different measurements. *3)* Our new user interaction features have a significant influence on the improvement of classifiers.

5.3 Repackaged Malware Classification

We tested our approach on more complicated repackaged malware which contains multiple DRegions. We calculate *malware score* r_m for each repackaged malware. Unlike binary classification in existing machine-learning-based approaches, r_m is a continuous value in $[0, 1]$ to measure DRegions with different security properties.

There are no existing solutions on the classification of multiple DRegions in an app. For comparison, we carefully implemented a control method called the *non-partition-based* classification. To have a fair and scientific comparison with the non-partition-based which does not consider code heterogeneity, DR-Droid's classification and the control method's classification use the same Random Forest classifier and the same set of features from Section 5.2. The **only difference** between our method and the control method is that the control method treats an app in its entirety. The control method represents the conventional machine-learning-based approach.

We assessed several repackaged malware families: Geinimi, Kungfu (which contains Kungfu1, Kungfu2, Kungfu3, Kungfu4) and AnserverBot multi-DRegion apps in these families. The major reason for choosing these families is that they contain enough representative repackaged malware for testing. Other malware datasets (e.g., VirusShare) do not specify the types of malicious apps. It is hard to get the ground truth of whether an app in the datasets is repackaged or not. The classification accuracy results of our partition-based approach and the non-partition-based approach are shown in Table 3.

Our partition-based approach gives the substantial improvement by achieving a lower FNR in all three families. Specifically, the non-partition-based approach misses 12 apps in Geinimi and 3 apps in AnserverBot family. In comparison, our approach accurately detects all the malicious DRegions in Geinimi and AnserverBot families. The non-partition-based approach misses 12 apps in Kungfu family. In comparison, our approach misses 4 apps in Kungfu family. The average FNR for our approach is 0.35%.

To answer **Q2**, our solution gives 30-fold improvement over the non-partition-based approach on average false negative rate in our experiment. This improvement is substantial. The control method without any code heterogeneity analysis is much less capable of detecting repackaged malware.

5.3.1 Case Study of Heterogeneous Properties

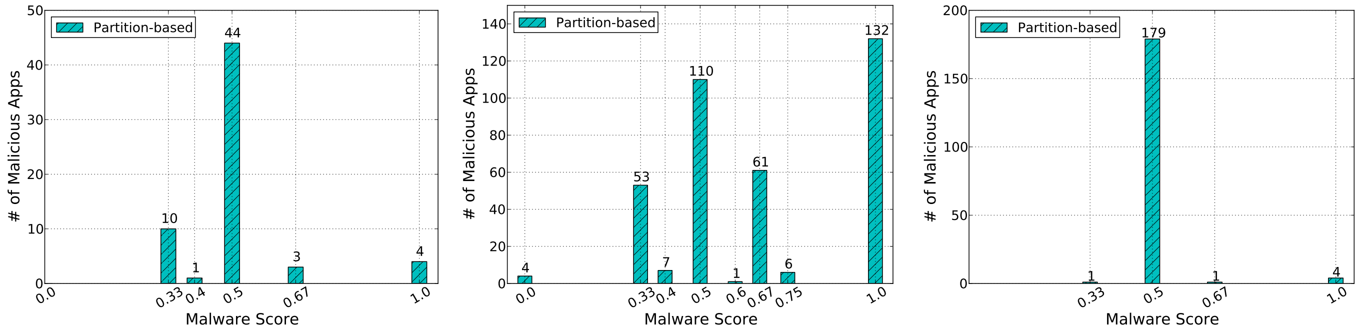
For an app (DroidKungFu1--881e*.apk) in Kungfu family, the malicious DRegion contains 13 classes whose names begin with Lcom/google/ssearch/*. The app attempts to steal the user's personal information by imitating a Google official library. The other DRegion whose name begins with Lcom/Allen/mp/* is identified as benign by our approach. There are some isolated classes such as R$attr, R$layout, which are produced by R.java with constant values. The malicious DRegion has its own life cycle which is triggered by a receiver in the class Lcom/google/ssearch/Receiver. All the processes run on the background and separately from the benign code.

Table 4 shows the distribution of a subset of representative features in two different methods. Particularly, DRegion 1 contains many user interaction functions with no sensitive APIs and permissions. However, DRegion 2 invokes a large number of sensitive APIs and requires many critical permissions. In the experiment, DRegion 1 is classified as benign and DRegion 2 is classified as malicious. The different prediction results are due to the differences in DRegion behaviors, which originally comes from their code heterogeneity. The non-partition-based approach fails to detect this instance. The experiment results validate our

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2745575, IEEE Transactions on Dependable and Secure Computing

Malware Families	Geinimi		Kungfu		AnserverBot		Average
	FN	FNR(%)	FN	FNR(%)	FN	FNR(%)	FNR(%)
Partition-based	0(62)	0	4(374)	1.07	0(185)	0	**0.35**
Non-partition-based	12(62)	19.36	12(374)	9.89	3(185)	1.62	**10.29**

TABLE 3: False negative rate for detecting three families of repackaged malware. Our partition-based approach reduces the average false negative rate by 30-fold.



(a) Geinimi family

(b) Kungfu family

(c) AnserverBot family

Fig. 5: Prediction of malware score in different Malware families.

DroidKungfu1–881e*.apk		Partition (ours)		Non-partition
Feature	Description	DRegion1	DRegion2	N/A
Type III	READ_PHONE_STATE permission	**0**	**1**	1
	READ_LOGS permission	**0**	**1**	1
Type II	getDeviceId function in Landroid/telephone/ telephoneManager	**0**	**1**	1
	read function in Ljava/io/InputStream	**0**	**3**	3
	write function in Ljava/io/FileOutput	**0**	**1**	1
Type I	onClick function occurrence	**16**	**2**	18
	# of distinct user-interaction functions	**5**	**1**	5
	onKeyDown function occurrence	**3**	**0**	3
Classification		Benign	Malicious	Benign
Correctness		✓(Yes)		✗(No)

TABLE 4: Our method shows heterogeneous properties in the repackaged app (DroidKungfu1–881e*.apk), where the no-partition based cannot.

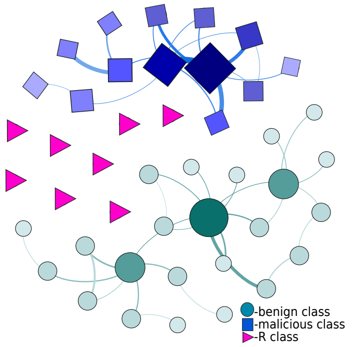


Fig. 6: A simplified class-level dependence graph for the app DroidKungFu1–881e*.apk. Each node represents a class, and the diameter of a node indicates the total degree of its corresponding class.

initial hypothesis that identifying code heterogeneity can substantially improve the detection of repackaged malware.

To answer **Q3**, our approach successfully detects different behaviors in the original and injected components, demonstrating the importance and effectiveness of code heterogeneity analysis.

5.3.2 False Negative Analysis

We discuss possible reasons that cause false negatives in our approach. *1)* Integrated benign and malicious behaviors. Well integrated benign and malicious behaviors in an app can cause false negatives in our approach. Com.egloos.dewr.ddaycfgc is identified by Virus Total as a trojan but is predicted as benign by our approach. The reason is that the malicious behavior, which communicates with a remote server, is hidden under the large amount of benign behaviors. The activities are integrated tightly and several sensitive APIs are used in the app. *2)* Low code heterogeneity in malicious components. Low code heterogeneity means that malicious code does not exhibit obvious malicious behaviors or is deeply disguised. To reduce false negatives, a more advanced partition algorithm is required to identify integrated benign and malicious behaviors. How to detect low heterogeneity malicious code is still an open question. We provide more discussion in Section 6.

5.3.3 Distribution of DRegions in Different Dataset

We evaluate the distribution of the DRegion number in three different datasets: the Genome repackaged malware dataset, the Virus-Share general malware dataset and the benign app dataset. Figure 7 shows the distribution of the number of DRegions in three datasets by randomly testing 1,000 apps. We find 66.9% of apps in Genome has multiple DRegions, in comparison, 6.5% of apps in Virus Share and 28.1% in benign app dataset have multiple DRegions. Because of repackaged malware, the distribution of Genome malware significantly differs from others in the others.

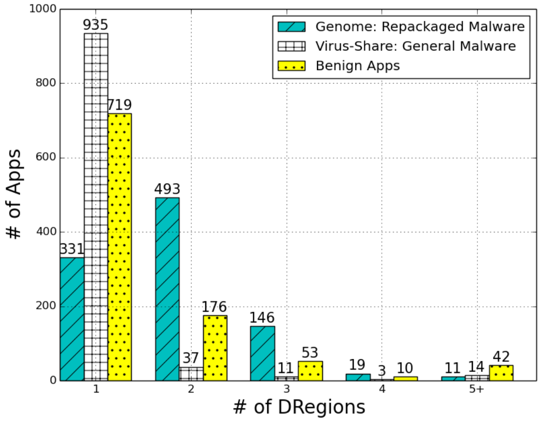


Fig. 7: The distribution of the number of DRegions in different datasets, where X axis represents the number of DRegions in an app and Y axis represents the count of apps with a certain number of DRegions. Repackaged malware tends to have more DRegions.

5.4 False Positive Analysis with Popular Apps

The purpose of this evaluation is to experimentally assess how likely our detection generates false positives (i.e, false alerts). We collect 1,617 free popular apps from Google Play market, the selection covers a wide range of categories. We evaluate a subset of apps (158 out of 1,617) that have multiple large DRegions. Each app contains 2 or more class-level DRegions with at least 20 classes in the DRegion. In the 158 apps, Virus Total identifies 135 of them as true benign apps, that apps raise no security warnings.

The most common cause of multi-DRegions is the use of ad libraries. A majority of multiple DRegion apps have at least one ad library (e.g., admob). The ad library acquires sensitive permissions, access information and monitor users' behaviors to send the related ads for profit. Some aggressive ad libraries, e.g., Adlantis, results in a false alarm in our detection. Adlantis acquires multiple sensitive permissions, and it tries to read user private information. The ad package involves no user interactions. We identify ad libraries by matching the package name in a whitelist. More effort is needed to automatically identify and separate ad libraries. Table 5 presents the false positive rate with and without ads libraries. The normal ad libraries do not affect our detection accuracy, while the aggressive ads libraries dilute our classification results and introduce false alerts into our detection. When excluding aggressive ad libraries, our detection misclassifies 4 out of 135 benign apps. To answer **Q4**, our approach raises a false positive rate (FPR) of 2.96% when classifying free popular apps and a false negative rate (FNR) of 0.35% when classifying repackaged malware.

5.5 Performance Evaluation

We evaluate the performance of our approach based on the execution time. The detection of a repackaged malware includes graph generation, feature extraction and classification of DRegions. We measure each step separately and evaluate the runtime performance.

	w/o Ads	w/ Group 1 Ads	w/ Group 2 Ads
% of Alerts	2.96%	2.96%	5.18%

TABLE 5: For 135 benign apps, how the percentage of alerts changes with the inclusion of ad libraries. Group 1 Ads are benign ad libraries, namely *admob* and *google.ads*. Group 2 Ads refer to the known aggressive ad library *Adlantis*. Group 1 does not affect our detection accuracy, whereas Group 2 increases the number of alerts.

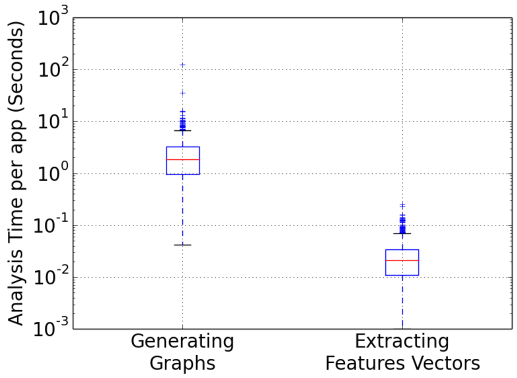


Fig. 8: Time distribution for generating graphs and extracting features.

We focus on two aspects. One aspect is the graph generation time and features extraction time as the preprocess for detecting malware. The other aspect is the machine learning training and prediction time. The performance time can vary a lot in different sizes of apps. We concentrate on the average time for processing an app. We conducted our experiment on an x86_64 system with 3GB memory.

From Figure 8, our average time for generating CDG and MCG graphs is around 2.5 seconds. Our average time for extracting features is 27.84 milliseconds. We felt the average time is considerably short given a large number of apps we examined. Our graph construction algorithm is efficient by parsing instructions of an app. Table 6 shows the runtime of our machine learning operations. We split machine learning operations into the training operation and the prediction operation. Our training operation is used to identify machine learning classifier parameters. The extracted feature vectors are feed into different classifiers during the training operation. In the prediction, the classifier predicts each DRegion separately. We measure the runtime for classifying each DRegion. Our training time is very short from Table 6. R.Forest takes 0.539 seconds for training, and SVM takes 3.583 seconds for training. Our prediction time is also negotiable. On average, D.Tree takes 0.51 seconds for prediction, and R.Forest takes 1.38 seconds for prediction.

Time	Training/Second	Prediction/Millisecond		
		mean	max	min
R.Forest	0.539	1.38	9.66	0.94
KNN	0.534	1.09	13.81	0.67
D.Ttree	0.588	0.51	2.91	0.36
SVM	3.583	0.85	5.24	0.60

TABLE 6: Execution time analysis for machine learning

5.6 Discover New Suspicious Apps

We evaluate a total of 1,979 newly released (2015) apps. Our approach raises a total of 127 alarms. Because of the lack of ground truth in the evaluation of new apps, computing FP requires substantial manual efforts on all these flagged apps. We performed several manual studies on the flagged apps. The apps are randomly selected from different categories.

Manual Verification Our rule for labeling an app as malware in our manual analysis is that the malware collects privacy information and sends it out without user notification. We identify an app as malware based on a two-step validation: *1)* Statistics of permissions and APIs. We compare the frequency of permissions and APIs in the app towards its description. If an app contains critical APIs that do not match its description, e.g., a weather app contains reading and sending SMS APIs, we regard the app is potentially malicious. *2)* Sensitive APIs that are not triggered by user inputs. If there exists a secret and sensitive data flow and the data flow path does not include user interaction functions, we regard the app is malicious. In the manual verification, we utilize static analysis and manual inspection to verify the flagged apps. We decompile each app into Smali intermediate representations (IR). We extract permissions and APIs from Smali IR. We compare permissions with app descriptions to find potentially malicious apps. We manually analyze the methods that invoke sensitive APIs to detect sensitive data flows. We report an app as malware if it is confirmed by our manual analysis.

We list four of them which are identified as malicious by our manual analysis. The first two suspicious apps are verified by our manual analysis, but are missed by Virus Total. Virus Total does detect the latter two apps. To answer **Q5**, our approach is capable of detecting new single-DRegion and multiple-DRegions malware.

1) za.co.tuluntulu is a video app providing streaming TV programs. However, it invokes multiple sensitive APIs to perform surreptitious operations on the background, such as accessing contacts, gathering phone state information, and collecting geometric information.

2) com.herbertlaw.MortagageCaculator is an app for calculating mortgage payments. It contains a benign DRegion by the usage of the admob ad library. It also contains an aggressive library called appflood in the malicious DRegion, which collects privacy information by accessing the phone state and then stores it in a temporary storage file.

3) com.goodyes.vpn.cn is a VPN support app with in-app purchase and contains multiple DRegions. A malicious package Lcom/ccit/mmwlan is integrated with a payment service Lcom/alipay/* in one malicious DRegion. It collects the user name, password, and device information. It exfiltrates information to a constant phone number.

4) longbin.helloworld is a simple calculator app with one DRegion. However, it requests 10 critical permissions. It modifies the SharedPreferences to affect the phone's storage, records the device ID and sends it out through executeHttpPost without any users' involvement.

Summary.

Our results validate the effectiveness of code heterogeneity analysis in detecting Android malware. We summarize major experimental findings as follows.

- Our prototype is able to identify malicious and benign code by distinguishing heterogeneous DRegions.
- Our partition-based detection reduces false negatives (i.e., missed detection) by 30-fold, when compared to the non-partition-based method.
- Our prototype achieves low false negative rate (0.35%) and low false positive rate (2.96%).

Our tool can also be used to identify ad libraries and separate them from the main app. These components can be confined at runtime in a new restricted environment, as proposed recently in [19].

6 Discussion and Limitations

Graph Accuracy. Our current prototype is built on the Smali code intermediate representation [20] for a low overhead. Machine-learning based approaches require a large number of apps for training. This graph generation is based on analyzing patterns on the instructions of Smali code. Our approach may miss detection of some data-dependence edges (e.g. implicit ICCs [21] and onBind functions), because of a lack of flow sensitivity [22] [23]. Our analysis under-approximates the dependence-related graph because of the missing edges. Context- and flow-sensitive program analysis improves the graph accuracy and increases analysis overhead. To balance the performance and the accuracy in constructing the graphs is one of our future directions. We plan to extend our prototype to an advanced program analysis technique without compromising the performance.

Our graph construction is based on the static code analysis. The current static analysis is not sound because it cannot represent the full app logic [24]. Advanced evasion techniques (e.g., dynamic loading, code obfuscation, and drive-by downloading) may result in the missing graph edges. This poses challenges for detecting repackaged malware, as our approach could not identify these dependence relations. Drive-by download attacks, for example, could be carried out by inserting simple logic to retrieve a malicious payload; this payload could ultimately be isolated from app's main code, and thus, not easily detectable. A potential solution would be to combine our static analysis with dynamic monitoring. By extracting DRegions from the aggregation of both the main app and downloaded payload, we can construct dependence graphs depicting the apps full logic, and partition accordingly.

To reduce under-approximation of dependence-related graphs, we further plan to broaden the definitions of dependence relations. Our current approach identifies three types (call, data and ICC dependencies). Additional edge dependences can include reflection-based call relations and call relations from dynamically-loaded code. For each additional dependence relation, we aim to extend our approach with more specific analysis techniques (e.g. dynamic monitoring and string analysis) to achieve more robust graph construction. Our future work is to extend the dependence relations for a more sound graph construction.

Dynamic Code. As our prototype employs static analysis, the examining dynamic code is outside its scope. Static analysis cannot precisely approximate dependence relations only identifiable dynamically (e.g. calling through Java's

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2745575, IEEE Transactions on Dependable and Secure Computing

native interface, and native code [25]). Ignoring dynamic analysis results in missing graph edges and additional DRegions, which may skew classification results because of imprecision. Still, we believe the impact of dynamic obfuscation bears little impact; AndroidLeaks [26] found only 7% of apps containing native code. In our future work, we plan to hybridize static and dynamic analysis to mitigate these risks.

Code Obfuscation. Malware writers may utilize code obfuscation to evade detection [27] through using tools such as Google's ProGuard[5], which simply renames app classes and methods. Our approach is resilient against renaming-based obfuscation, since it does not modify data dependencies or call relations among our graphs. However, our static analysis approach will fall short when presented with more advanced techniques such as reflection. Specifically, reflection does away with direct call invocation, leaving the callee defined as a string and thus, not identifiable in the invocation instruction. The obfuscation introduces implicit dependence relations, which cannot be directly resolved in our analysis. For the future work, we aim to extend our approach with advanced reflection-targeted analysis techniques [28].

Integrated Malware. We plan to generalize our heterogeneity analysis by supporting complex code structures containing unclear boundaries between code segments. Our prototype is not designed to detect malicious DRegions semantically connected and integrated with the remaining app code. Integrated repackaged malware may be produced through code rewriting techniques where malicious code is triggered by hijacked execution [29]. In such cases, partitioning dependence graphs into DRegions would be challenging because of their connectedness. However, to be successful, attackers would need in-depth knowledge of the apps original execution. For example, careful manipulation of the dependence graphs would be needed to isolate (superficially) connected componentss [24] [30] based on semantics and functionality. Given the burden of expertise, we view this scenario as unlikely.

Advanced Malware. There is a trend that malware writers tend to abuse packing services to evade malware screening [31] [32]. Malware adopts code packing techniques to prevent the analyst from acquiring app bytecode. Typical anti-analysis techniques include metadata modification and DEX encryption. This advanced malware poses challenges for our approach to construct CDG and MCG. Our approach cannot construct the dependence graph without obtaining the complete DEX code of an app. For instance, the DEX file is encrypted and obfuscated to evade static analysis. An efficient code unpacking system is needed as the pre-process to extract DEX files for code heterogeneity analysis. The code unpacking system needs to reconstruct the DEX file in memory at runtime. DexHunter [31] instrumented both ART and DVM virtual machines to recover the DEX files from packed apps. DexHunter identified the location of DEX files by instrumenting key functions in the virtual machines and dumped DEX files in memory. AppSpear [32] proposed another bytecode decrypting and DEX reassembling method to recover protected bytecode without the knowledge of packer techniques. AppSpear instrumented the DVM and collected the Dalvik Data Struct (DDS) information to reassemble a normal DEX file. The extracted DEX file can be applied for general program analysis, e.g., our code heterogeneity analysis. These two approaches demonstrated promising results for practical DEX code extraction from packed apps.

To mitigate our approach's limitation of analyzing packed apps, one possible solution is to extend our approach with DexHunter or AppSpear to increase the resilience towards the advanced malware. We could apply AppSpear as the pre-process to extract the legitimate DEX code. The DEX files are dumped from memory at runtime. The DEX code is then used as the input in our approach to construct CDG and MCG for partition. Our future work will generalize our heterogeneity analysis by supporting anti-packing code extraction.

7 Related Work

Repackaged Malware Detection. DroidMOSS [7] applied a fuzzy hashing technique to generate a fingerprint to detect app repackaging, the fingerprint is computed by hashing each subset of the entire opcode sequences. Juxtapp [6] examined code similarity through features of k-grams of opcode sequences. ResDroid [33] combined the activity layout resources with the relationship among activities to detect repackaged malware. Zhou *et al.* [12] detected the piggybacked code based on the signature comparison.

However, the code level similarity comparisons are vulnerable to obfuscation technique, which is largely used in app repackaging. To improve obfuscation resilience, Potharaju et al. [34] provided three-level detection of plagiarized apps, which is based on the bytecode-level symbol table and method-level abstract syntax tree (AST). MassVet [5] utilizes UI structures to compute centroid metrics for comparing the code similarity among apps.

Solutions have been proposed on the similarity comparison of apps based on graph representations. DNADroid [4] compared the program dependence graphs of apps to examine the code reuse. AnDarwin [35] speeds up DNADroid by deploying semantic blocks in program dependence graphs, and then deployed locality hashing to find code clones. DroidSim [36] used component-based control flow graph to measure the similarity of apps. ViewDroid [3] focused on how apps define and encode user's navigation behaviors by using UI transition graph. DroidLegacy [37] detected a family of apps based on the extracted signature.

Instead of finding pairs of similar apps, our approach explores the code heterogeneity for detecting malicious code and benign code. Our approach avoids the expensive and often error-prone whole-app comparisons. It complements existing similarity-based repackage detection approaches.

Machine-Learning-based Malware Detection. Peng *et al.* [8] used the requested permissions to construct different probabilistic generative models. Wolfe *et al.* [38] used the frequencies of n-grams decompiled Java bytecode as features. DroidAPIMiner [10] and DroidMiner [39] extracted features from API calls invoked in the app. Drebin [14] gathered as many features including APIs, permissions, components to represent an app, and then uses the collected information

[5] https://developer.android.com/studio/build/shrink-code.html

for classification. Gascon *et al*. [18] transformed the function call graph into features to conduct the classification. STILO [40] and CMarkov [41] applied hidden Markov models to detect anomaly behaviors. AppContext [42] adopted context factors such as events and conditions that lead to a sensitive call as features for classifying malicious and benign method calls. DIALDroid [43] and MR-Droid [44] detected inter-component communication vulnerabilities for a large scale of Android apps. Crowdroid [45] used low-level kernel system call traces as features.

These solutions cannot recognize code heterogeneity in apps, as they do not partition a program into regions. In comparison, features in our approach are extracted from each DRegion to profile both benign and malicious DRegion behaviors.

8 Conclusions and Future Work

We addressed the problem of detecting repackaged malware through code heterogeneity analysis. We demonstrated its application in classifying semantically disjoint code regions. Our preliminary experimental results showed that our prototype is very effective in detecting repackaged malware and Android malware in general. For future work, we plan to improve our code heterogeneity techniques by enhancing dependence graphs with context and flow sensitivities.

Acknowledgement

The authors would like to thank the anonymous reviewers for their insightful comments and suggestions on the work. This work was supported in part by DARPA APAC award FA8750-15-2-0076.

References

[1] K. Tian, D. D. Yao, B. G. Ryder, and G. Tan, "Analysis of code heterogeneity for high-precision classification of repackaged malware," in *Proc. MoST, in conjunction with the IEEE Symposium on Security and Privacy*, 2016.

[2] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE (S&P)*, 2012.

[3] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "ViewDroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proc. WiSec*, 2014.

[4] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on Android markets," in *Proc. ESORICS*, 2012.

[5] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale," in *Proc. USENIX Security*, 2015.

[6] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A scalable system for detecting code reuse among Android applications," in *Proc. DIMVA*, 2013.

[7] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party Android marketplaces," in *Proc. CODASPY*, 2012.

[8] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Using probabilistic generative models for ranking risks of Android apps," in *Proc. CCS*, 2012.

[9] B. Wolfe, K. Elish, and D. Yao, "Comprehensive behavior profiling for proactive Android malware detection," in *Proc. ISC*, 2014.

[10] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-level features for robust malware detection in Android," in *Proc. SecureComm*, 2013.

[11] K. O. Elish, X. Shu, D. Yao, B. G. Ryder, and X. Jiang, "Profiling user-trigger dependence for Android malware detection," *Computers & Security*, 2014.

[12] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of piggybacked mobile applications," in *Proc. CODASPY*, 2013.

[13] W. Hu, J. Tao, X. Ma, W. Zhou, S. Zhao, and T. Han, "MIGDroid: Detecting app-repackaging Android malware via method invocation graph," in *Proc. ICCCN*, 2014.

[14] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of Android malware in your pocket," in *Proc. NDSS*, 2014.

[15] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proc. MobiSys*, 2011.

[16] H. Zhang, D. Yao, N. Ramakrishnanvt, and Z. Zhang, "Causality reasoning about network events for detecting stealthy malware activities," *Computers & Security*, 2016.

[17] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the Android permission specification," in *Proc. CCS*, 2012.

[18] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of Android malware using embedded call graphs," in *Proc. AISec*, 2013.

[19] M. Sun and G. Tan, "NativeGuard: Protecting Android applications from third-party native libraries," in *Proc. WiSec*, 2014.

[20] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth, "Slicing-droids: program slicing for smali code," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 2013.

[21] K. O. Elish, D. Yao, and B. G. Ryder, "On the need of precise inter-app ICC classification for detecting Android malware collusions," in *Proc. MoST*, 2015.

[22] K. Lu, Z. Li, V. P. Kemerlis, Z. Wu, L. Lu, C. Zheng, Z. Qian, W. Lee, and G. Jiang, "Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting," in *Proc. NDSS*, 2015.

[23] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: statically vetting Android apps for component hijacking vulnerabilities," in *Proc. CCS*, 2012.

[24] 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," in *Proc. PLDI*, 2014.

[25] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic reconstruction of Android malware behaviors," in *Proc. NDSS*, 2015.

[26] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale," in *Proc. TRUST*, 2012.

[27] A. Kovacheva, "Efficient code obfuscation for android," in *Proc. International Conference on Advances in Information Technology*, 2013.

[28] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Droidra: Taming reflection to support whole-program analysis of android apps," in *Proc. ISSTA*, 2016.

[29] B. Davis and H. Chen, "Retroskeleton: Retrofitting Android apps," in *Proc. MobiSys*, 2013.

[30] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "Edgeminer: Automatically detecting implicit control flow transitions through the Android framework," in *Proc. NDSS*, 2015.

[31] Y. Zhang, X. Luo, and H. Yin, "DexHunter: toward extracting hidden code from packed android applications," in *Proc. ESORICS*, 2015.

[32] W. Hu and D. Gu, "AppSpear: Bytecode decrypting and DEX reassembling for packed android malware," in *Proc. RAID*, 2015.

[33] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged Android applications," in *Proc. ACSAC*, 2014.

[34] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang, "Plagiarizing smartphone applications: attack strategies and defense techniques," in *Proc. ESSoS*, 2012.

[35] J. Crussell, C. Gibler, and H. Chen, "Andarwin: Scalable detection of semantically similar Android applications," in *Proc. ESORICS*, 2013.

[36] X. Sun, Y. Zhongyang, Z. Xin, B. Mao, and L. Xie, "Detecting code reuse in Android applications using component-based control flow graph," in *Proc. IFIP SEC*, 2014.

[37] L. Deshotels, V. Notani, and A. Lakhotia, "Droidlegacy: automated familial classification of Android malware," in *Proc. PPREW*, 2014.

[38] B. Wolfe, K. Elish, and D. Yao, “High precision screening for Android malware with dimensionality reduction,” in *Proc. ICMLA*, 2014.

[39] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, “DroidMiner: Automated mining and characterization of fine-grained malicious behaviors in Android applications,” in *Proc. ESORICS*, 2014.

[40] K. Xu, D. D. Yao, B. G. Ryder, and K. Tian, “Probabilistic program modeling for high-precision anomaly classification,” in *Proc. CSF*, 2015.

[41] K. Xu, K. Tian, D. Yao, and B. G. Ryder, “A sharper sense of self: Probabilistic reasoning of program behaviors for anomaly detection with context sensitivity,” in *Proc. DSN*, 2016.

[42] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, “AppContext: Differentiating malicious and benign mobile app behaviors using context,” in *Proc. ICSE*, 2015.

[43] A. Bosu, F. Liu, D. D. Yao, and G. Wang, “Collusive data leak and more: Large-scale threat analysis of inter-app communications,” in *Proc. AsiaCCS*, 2017.

[44] F. Liu, H. Cai, G. Wang, D. D. Yao, K. O. Elish, and B. G. Ryder, “MR-Droid: A scalable and prioritized analysis of inter-app communication risks,” in *Proc. MoST, in conjunction with the IEEE Symposium on Security and Privacy*, 2017.

[45] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, “Crowdroid: behavior-based malware detection system for Android,” in *Proc. SPSM*, 2011.

Ke Tian Ke Tian is a PhD student in Department of Computer Science at Virginia Tech, Blacksburg. He received his bachelor degree majoring information security from University of Science and Technology of China. He received the National Scholarship of China in 2012. His research interests is in cybersecurity, Android security and malware detection.

Danfeng (Daphne) Yao Daphne Yao is an associate professor of computer science at Virginia Tech. In the past decade, she has been working on designing and developing data-driven anomaly detection techniques for securing networked systems against stealthy exploits and attacks. Her expertise also includes mobile security. Dr. Yao received her Ph.D. in Computer Science from Brown University. Dr. Yao is an Elizabeth and James E. Turner Jr. ’56 Faculty Fellow and L-3 Faculty Fellow. She received the NSF CAREER Award in 2010 for her work on human-behavior driven malware detection, and the ARO Young Investigator Award for her semantic reasoning for mission-oriented security work in 2014. She has several Best Paper Awards (e.g., ICNP ’12, CollaborateCom ’09, and ICICS ’06) and Best Poster Awards (e.g., ACM CODASPY ’15). She was given the Award for Technological Innovation from Brown University in 2006. She held multiple U.S. patents for her anomaly detection technologies. Dr. Yao is an associate editor of IEEE Transactions on Dependable and Secure Computing (TDSC). She serves as PC members in numerous computer security conferences, including ACM CCS. She has over 75 peer-reviewed publications in major security and privacy conferences and journals.

Barbara G. Ryder Dr. Barbara G. Ryder is a emerita faculty member in the Department of Computer Science at Virginia Tech, where she held the J. Byron Maupin Professorship in Engineering. She received her A.B. degree in Applied Mathematics from Brown University (1969), her Masters degree in Computer Science from Stanford University (1971) and her Ph.D. degree in Computer Science at Rutgers University (1982). From 2008-2015 she served as Head of the Department of Computer Science at Virginia Tech, and retired on September 1, 2016. Dr. Ryder served on the faculty of Rutgers from 1982-2008. She also worked in the 1970s at AT&T Bell Laboratories in Murray Hill, NJ. Dr. Ryder’s research interests on static/dynamic program analyses for object-oriented and dynamic programming languages and systems, focus on usage in practical software tools for ensuring the quality and security of industrial-strength applications.

Dr. Ryder became a Fellow of the ACM in 1998, and received the ACM SIGSOFT Influential Educator Award (2015), the Virginia AAUW Woman of Achievement Award (2014), and the ACM President’s Award (2008). She received a Rutgers School of Arts and Sciences Computer Science Distinguished Alumni Award (2016), was named a CRA-W Distinguished Professor (2004), and was given the ACM SIGPLAN Distinguished Service Award (2001). Dr. Ryder led the Department of Computer Science team that tied nationally for 2nd place in the 2016 NCWIT NEXT Awards.She has been an active leader in ACM (e.g., Vice President 2010-2012, Secretary-Treasurer 2008-2010; ACM Council 2000-2008; General Chair, FCRC 2003; Chair ACM SIGPLAN (1995-97)). She serves currently as a Member of the Board of Directors of the Computer Research Association (2014-2020,1998-2001). Dr. Ryder is an editorial board member of ACM Transactions on Software Engineering Methodology and has served as an editorial board member of ACM Transactions on Programming Languages and Systems, IEEE Transactions on Software Engineering, Software: Practice and Experience, and Science of Computer Programming.

Gang Tan Dr. Gang Tan is the James F. Will Career Development Associate Professor in the Department of Computer Science and Engineering at the Pennsylvania State University, University Park, PA. He leads the Security of Software (SOS) Lab. His research is at the interface between computer security, programming languages, and formal methods. He received his bachelors degree in Computer Science with honors from Tsinghua University in 1999 and his Ph.D. degree from Princeton University in 2005. He has received an NSF CAREER award, two Google Research Awards, and a Francis Upton Graduate Fellowship. He is a member of IEEE and ACM.

Guojun Peng Guojun Peng is an associate professor in the School of Computer Science at Wuhan University, China. He received BS, MS and PhD degree from Wuhan University. He was a visiting scholar in Virginia Tech from 2015 to 2016. His main research interests include in information system security and malware detection.