DroidMiner: Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android Applications

Chao Yang[1], Zhaoyan Xu[1], Guofei Gu[1], Vinod Yegneswaran[2], Phillip Porras[2]

[1]Texas A&M University, College Station, TX, USA
{yangchao, z0x0427, guofei}@cse.tamu.edu
[2]SRI International, Menlo Park, CA, USA
{vinod, porras}@csl.sri.com

Abstract Most existing malicious Android app detection approaches rely on manually selected detection heuristics, features, and models. In this paper, we describe a new, complementary system, called *DroidMiner*, which uses static analysis to *automatically* mine malicious program logic from known Android malware, abstracts this logic into a sequence of threat modalities, and then seeks out these threat modality patterns in other unknown (or newly published) Android apps. We formalize a two-level behavioral graph representation used to capture Android app program logic, and design new techniques to identify and label elements of the graph that capture malicious behavioral patterns (or malicious modalities). After the automatic learning of these malicious behavioral models, DroidMiner can scan a new Android app to (i) determine whether it contains malicious modalities, (ii) diagnose the malware family to which it is most closely associated, (iii) and provide further evidence as to why the app is considered to be malicious by including a concise description of identified malicious behaviors. We evaluate DroidMiner using 2,466 malicious apps, identified from a corpus of over 67,000 third-party market Android apps, plus an additional set of over 10,000 official market Android apps. Using this set of real-world apps, we demonstrate that DroidMiner achieves a 95.3% detection rate, with only a 0.4% false positive rate. We further evaluate DroidMiner's ability to classify malicious apps under their proper family labels, and measure its label accuracy at 92%.

Keywords: Mobile Security, Android Malware Analysis and Detection

1 Introduction

Analysis of Android applications (apps) is complicated by the nature of the interaction between the various entities in its component-based framework. Existing static analysis approaches for detecting Android malware rely on either matching against *manually-selected* heuristics and *pre-defined* programming patterns [41,18] or designing detection models that use *coarse-grained* features such as permissions registered in the apps [29]. Some studies [36,14] design detection models by calculating the frequencies of isolated framework API calls, which still miss capturing the important programming logic of Android malware.

In this work, we introduce **DroidMiner**, a new approach to detect and characterize Android malware through robust and automated learning of fine-grained programming logic and patterns in known malware. Specifically, DroidMiner extends traditional static analysis techniques to map the functionalities of an Android app into a two-tiered behavior graph. This two-tiered behavior graph is specialized for modeling the complex, multi-entity interactions that are typical for Android applications. Within this behavior graph, DroidMiner automatically identifies *modalities*, i.e., programming logic segments in the graph that correspond to known suspicious behavior. The set of identified modalities is then used to define a modality vector. DroidMiner then uses common modality vectors to offer a more robust classification scheme, in which variant applications can be grouped together based on their shared patterns of suspicious logic. While DroidMiner also relies on analyzing Framework API calls, different from existing approaches that merely analyze the isolated usage of Framework APIs, DroidMiner relies on the modalities that robustly capture the semantic relationships across multiple APIs and proposes new techniques to automatically extract them. Rather than simply examining whether or not the target app is malicious (a binary answer), DroidMiner also provides specific app behavior traits (modalities) to support detection decisions.

We present DroidMiner's algorithm for discovering and automatically extracting malware modalities. We evaluate DroidMiner using 2,466 malicious apps, identified from a corpus of over 67,000 third-party market apps, plus an additional set of over 10,000 official market apps from *GooglePlay*. We measure the utility of DroidMiner modalities with respect to three specific use cases: (i) malware detection, (ii) malware family classification, and (iii) malware behavioral characterization. Our results validate that DroidMiner modalities are useful for classification and capable of isolating a wide range of suspicious behavioral traits embedded within parasitic Android applications. Furthermore, the composite of these traits enables a unique means by which Android malware can be identified with a high degree of accuracy. We anticipate that programs identified as sharing common modalities with known malicious apps would then be subject to more in-depth scrutiny through, potentially more expensive, dynamic analysis tools.

The contributions of our paper include the following:

- A description of our new two-tiered behavioral graph model for characterizing Android application behavior, and labeling its logical paths within known malicious apps as malicious modalities.
- The design and implementation of DroidMiner, a novel system for *automated* extraction of robust and fine-grained Android app program behaviors into modalities, as well as *automated* characterization of such behaviors to support detection decisions.
- An in-depth evaluation of DroidMiner including its run-time performance and efficacy in malware detection, family classification, and behavioral characterization.

2 Motivation and System Goals

2.1 Motivations

We motivate our system design by introducing the inner working of a real-world Android malware (MD5: c05c25b 769919fd7f1b12b4800e374b5). It attempts to perform the following malicious behaviors in the background after the phone is booted: stealing users' personal sensitive information (e.g., IMEI and IMSI) and sending them to remote servers, sending and deleting SMS messages, downloading unsolicited apps, and issuing HTTP search requests to increase websites' search rankings on the search engine.

As illustrated in Figure 1, once the phone is booted, the receiver will send out an alarm every two minutes and trigger another receiver (named "MyAlarmReceiver") by using three API calls: AlarmManager(), getServiceSystem(), and getBroadcast(). Then, MyAlarmReceiver starts a background service (named "MyService") by calling startService() in its lifecycle call onReceive(). Once the service is triggered, it will read the device ID (getDeviceId()) and subscriber ID (getSubscriberId()) in the phone, and register an object handler to access the short message database (content://sms/). Meanwhile, the service monitors changes to the SMS Inbox database (content://sms/inbox/) by calling ContentObserver.onChange() and deleting particular messages using delete(), and also attempts to download unsolicited APK files (e.g., "myupdate.apk"). More details can be found in our extended technical report [38].

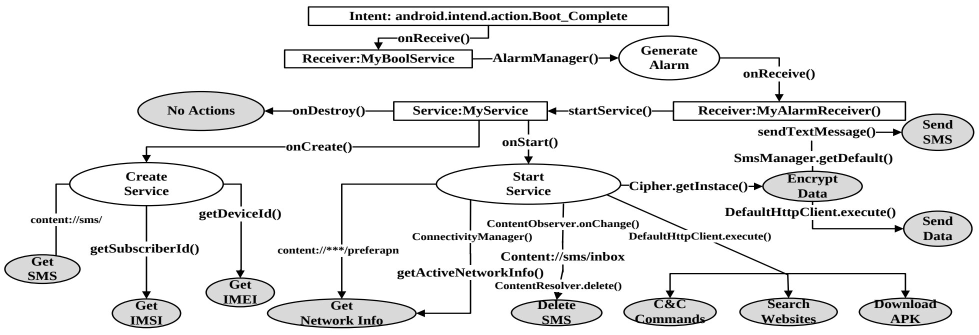

Figure 1. Capabilities embedded in malware from the ADRD family. The sample achieves its malicious functionalities by mainly invoking a series of framework APIs in order.

The above description motivates an important design premise that when malware authors design malicious apps to achieve specific malicious behaviors, they typically require the use of sets of framework API calls and specific resources (e.g., content providers). More specifically, although attackers may attempt to launch malicious behaviors in a more surreptitious way, they would still have to use those framework APIs or access those important resources.

2.2 Goals and Assumptions

The goal of DroidMiner is to automatically, effectively and efficiently mine Android apps and interrogate them for potentially malicious behaviors. Given an unknown app, DroidMiner should be able to determine whether or not it is malicious. Going beyond just providing a yes or no answer, our system should be able to provide further evidence as to why the app is considered as malicious by including a concise description of identified malicious behaviors. This kind of information is typically considered the hallmark of a good malware detection system. For example, DroidMiner can inform us that a given app is malicious, and that it contains behaviors such as sending SMS messages and blocking certain incoming SMS messages.

Currently, we do not analyze native Android apps implemented using the Android Native Develop Kit (Android NDK). According to our observations, an overwhelming majority of Android apps today are developed using the Android SDK. Furthermore, the vast majority of malicious behaviors in Android apps are achieved by using Android SDK rather than Android NDK. Even for those malicious apps that use the NDK to achieve some malicious behaviors, they typically also use certain Android Framework APIs to obtain some auxiliary information. For example, "rooting" malware (e.g., samples in the family of DroidKungFu), which utilizes native code to achieve privilege escalation, still needs to use specific Framework APIs to obtain auxiliary information (e.g., the version of the operating system) to successfully root the phone. Hence, the presence of such APIs in the Dalvik bytecode could still provide hints for detecting such malware. Extending our system to include complete analysis of native code in Android apps is future work and outside the scope of this paper.

3 System Design

DroidMiner contains two phases: Mining and Identification. As illustrated in Figure 2, in the mining phase, DroidMiner takes both benign and malicious Android apps as input data and automatically mines malicious behavior patterns or models, which we call *modalities*. In the identification phase, our system takes an unknown app as input, extracts a Modality Vector (MV) based on our trained modalities, and outputs whether or not it is malicious, and which family it belongs to. In addition to a simple yes/no answer, our system can also characterize the behaviors of the app given the Modality Vector representation.

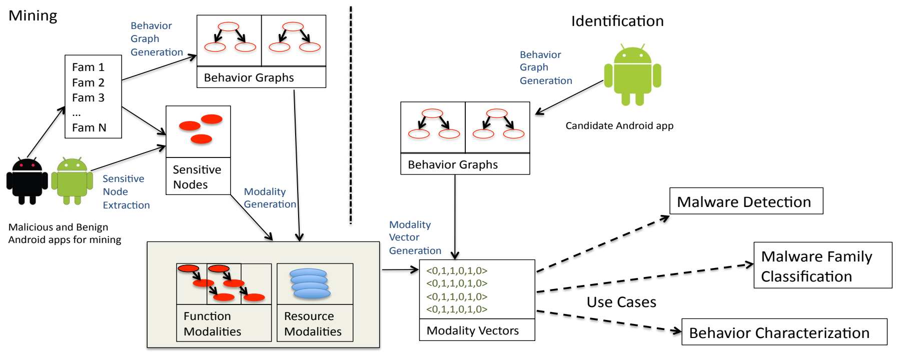

Figure 2. DroidMiner System Architecture

An important component in our system is the Behavior Graph Generator, which takes an app as input and outputs a behavior graph representation. As illustrated in Figure 1, although Android malware authors have significant flexibility in constructing malicious code, they must obey certain specific rules, pre-defined by the Android platform, to realize malware functionality (e.g., using particular Android/Java framework APIs and accessing particular content providers). These framework APIs and content providers capture the interactions of Android apps with Android framework software or phone hardware, which could be used to model Android apps' behaviors. With this intuition, DroidMiner builds a behavior graph based on the analysis of Android framework APIs and content providers used in apps' bytecode.

In the Mining phase, DroidMiner will attempt to automatically learn the malicious behaviors/patterns from a training set of malicious applications. The basic intuition is that malicious apps in the same family will typically share similar functionalities and behaviors. DroidMiner will examine the similarities from the behavior graphs of these malicious apps and automatically extract common subsets of suspicious behavior specifications, which we call *modalities*. From an intrusion detection perspective, these modalities are essentially micro detection models that characterize various suspicious behaviors found in malicious apps (in Section 3.1).

In the Identification phase, DroidMiner transforms an unknown malicious app into its behavior graph representation (using Behavior Graph Extractor) and extract a Modality Vector (based on all trained modalities), described in Section 3.3. Then, DroidMiner applies machine-learning techniques to detect whether or not the app is malicious. DroidMiner also has a data-mining module that implements Association Rule Mining to automatically learn the behavior characterization (in Section 3.4).

3.1 Behavior Graph and Modalities

Behavior Graph. DroidMiner detects malware by analyzing the program logic of sensitive Android and Java framework API functions and sensitive Android resources. To represent such logic, we use a two-tiered graphical model. As shown in Figure 3, at upper tier, the behaviors (functionalities) of each Android app could be viewed as the interaction among four types of components (Activities, Services, Broadcast Receivers, and Content Observers). We represent this tier using a **Component Dependency Graph (CDG)**. At the lower tier, each component has its own semantic functionalities and a relatively independent behavior logic during its lifetime. Here, we represent this independent logic using **Component Behavior Graphs (CBG)**.

Component Dependency Graph (CDG) (upper tier of Figure 3) represents the interaction relationships among all components in an app. Each node in the CDG is a component (Activity, Service, or Broadcast Receiver). (Note that multiple nodes could belong to the same type of component.) There is an edge from one node v_i to another node v_j, if the component v_i could activate the start of component v_j's lifecycle.

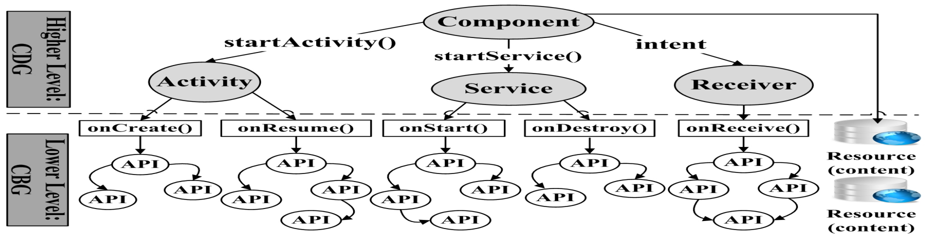

Figure 3. Two-tier behavior graph.

The **Component Behavior Graphs (CBG)** (lower tier of Figure 3) represents each component's *lifetime* [1] behavior logic (functionalities), i.e., each CBG represents the control-flow logic of those permission-related Android and Java API functions, and actions performed on particular resources of each component. Specifically, as illustrated in Figure 3, a CBG contains four types of node:

- A *root* note (v_{root}), denoting the component itself (e.g., an Activity).
- *Lifecycle functions* (V_{lcf}), used to achieve the runtime programming logic (e.g., onCreate() in activities, onReceive() in receivers, and onStart() in services).

[1] Lifetime, as defined by Android, is time between the moment when the OS considers a component to be constructed and the moment when the it considers the component to be destroyed.

– *Permission-related API functions* (V_{pf}), representing those permission-related (Android SDK or Java SDK) API functions (e.g., Java API Runtime.execute() or Android API sendTextMessage()). For simplicity, in the rest of paper, we refer both lifecycle functions and API functions as *framework API functions*.
– *Sensitive resource* (V_{res}), i.e., sensitive data (files or databases) that are accessed by the component. In this work, we consider resources as content providers (e.g., content://sms/inbox/), which could be extended to any other type of sensitive data. The usage of framework API functions and sensitive resources in an app essentially captures the interactions of an app with the Android platform hardware and sensitive data. Hence, the control-flow logic of framework API functions and the actions performed on those sensitive resources reflect an application's range of capabilities.

The edges in CBG represent the control-flow logic of framework API functions and sensitive resources. In terms of framework API functions, we consider that there is a direct edge from function node v_i to v_j in the CBG, if (1) when v_i and v_j are in the same control-flow block, v_j is executed just after v_i with no other functions executed between them; or (2) when v_i and v_j are in two continuous control-flow blocks B_i and B_j respectively (i.e., B_j follows B_i), v_i is the last function node in B_i and v_j is the first node in B_j. Then, we call v_j "is a successor of" v_i. For example, in terms of the malware sample illustrated in Figure 1, there is an edge from smsManager.getDefault() to sendTextMessage(). In terms of sensitive resources, since our work mainly focuses on analyzing the control-flow of sensitive functions rather than the data flow of sensitive data, we simply consider that there is an edge from the root to the resource v_r, if the component uses that sensitive resource[2].

Modality. We use the term, *modalities* to refer to malicious behavior patterns that are mined from behavior graphs of Android malware. More specifically, each modality is an ordered sequence (reserving the control-flow order) of framework API functions (function modality) or a set of sensitive resources (resource modality) in commonly shared in malicious apps' behavior graphs[3], which could be used to implement suspicious activities (e.g., sending SMS messages to premium-rate numbers or stealing sensitive information). As an example, the malware sample illustrated in Figure 1 relies on a function modality with an ordered sequence of two framework functions (onChange() $\rightarrow$ ContentResolver.delete()), and a resource modality (content://sms/inbox/) to partially achieve the malicious behavior of deleting messages in the SMS inbox.

3.2 Mining Modalities

Based on previous concepts, DroidMiner's approach to efficient mining of modalities from large malware corpora involves the following three steps: Behavior Graph Generation, Sensitive Node Extraction, and Modality Generation.

Behavior Graph Generation The generation of the behavior graph of an app contains two phases: generating CDG and generating CBG. Due to the page limitation, we mainly introduce the details of generating CBG (Details of generating CDG can refer [38].) Since Android is component driven, and each component has its own lifetime execution logic, the extraction of control-flow logic of framework API functions

[2] We could also choose to build an edge from a framework API function (that uses that resource) to the resource, which relies on the data flow analysis.

[3] Although modalities described in this paper are localized within a CBG, our work could be extended to include cross CBG modalities with the usage of CDG.

is more complex than traditional program analysis. DroidMiner generates the behavior graph by using the following three steps.

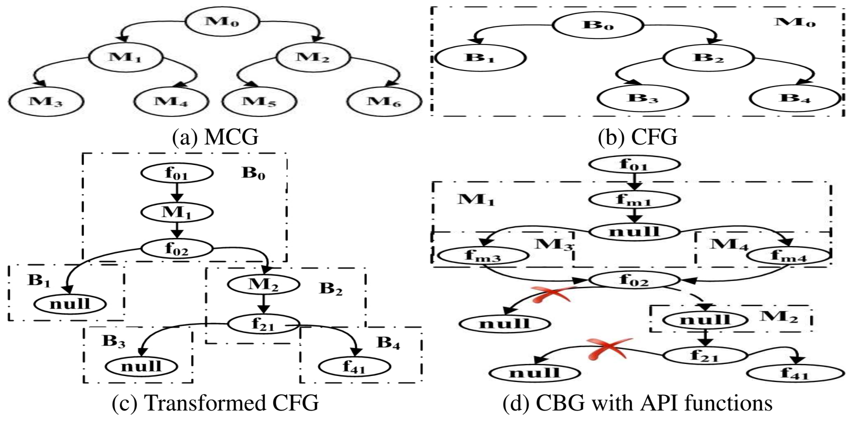

Figure 4. Illustration of generating a CBG with framework API functions.

Step 1: Generate Method Call Graph. For each component, our system generates a method call graph (MCG) containing two types of nodes: Android lifecycle functions and user-defined methods. Since each type of component has fixed lifecycle functions (e.g., onCreate() in an Activity), DroidMiner extracts lifecycle functions by analyzing method names in the component. Those user-defined methods are identified by using a static analysis tool. As illustrated in Figure 4(a), the directed edge from method M_0 to M_1 implies that M_0 calls M_1.

Step 2: Generate Control-Flow Graph. To extract the program logic corresponding to the usage of framework APIs, DroidMiner extracts each method's control-flow graph (CFG) by identifying branch-jump instructions in the method's bytecode (e.g., if-nez or packed-switch). Each node is a block of Dalvik bytecode without any jump-branch instructions. For example, M_0 with five blocks is illustrated in Figure 4(b). The directed edge from block B_0 to B_1 implies that B_1 is a successor block of B_0. Then, each block is represented as an ordered sequence of framework API functions and user-defined methods, which are extracted from the Dalvik bytecode with function call instructions (e.g., invoke-direct). We label a block as "null", if it does not contain any function call instructions . For example, in the method M_0, if (1) B_0 contains two API functions and user-defined method M_1, with the execution order of f_{01}, M_1 and f_{02}; (2) B_1 and B_3 do not contain any function calls; (3)B_2 contains method M_2 and one API function f_{21}; (4) B_3 contains one API function f_{41}, then the control-flow graph of M_0 is formed as Figure 4(c).

Step 3: Replace User-Defined Methods. As illustrated in Figure 4(c), since each leaf in the method-call graph does not call any other user-defined method, the leaf either contains a subgraph of framework API functions or is "null". Then, our approach replaces its position in its parents' control-flow graphs with that subgraph. This process is recursively performed, until all user-defined methods are replaced with framework API functions. For example, if (1) M_1 contains three framework API functions (f_{m1}, f_{m3}, and f_{m4}) and one "null" node after replacing its children methods M_3 and M_4 as illustrated in the middle of Figure 4(d), and M_2 does not contain any function nodes, then after replacing its children methods M_5 and M_6, the graph will be transformed to Figure 4(d). Finally, the CBG will be generated by removing those leaves that are "null". After the above three steps, each app's CBG could be generated that represents the control flow of its framework API calls.

Sensitive Node Extraction A modality is an ordered sequence of framework API functions and a set of sensitive resources that are commonly observed in malware's behavioral graphs. We denote those framework API functions and sensitive resources as sensitive nodes (the former are called *sensitive function nodes*, and the latter are called *sensitive resource nodes*).

We use two strategies to *automatically* extract sensitive nodes. The first strategy is based on the observation that malware samples belonging to the same family tend to share similar malicious logic. Such an observation has been validated by a recent study, which reports that Android malware in the same family tends to hide in multiple categories of fake versions of popular apps [1]. Based on this intuition, we group known malware samples according to their families. Then, for each malware family, we extract function nodes and resource nodes that are commonly shared by at least $\theta\%$ members in this family. Our second strategy is based on the observation that malware samples hosted on third-party market websites tend to be parasitic, i.e., they masquerade as popular benign apps by injecting malicious payloads into original benign apps. Based on this intuition, we automatically extract sensitive nodes by calculating the additional bytecode between the known malicious app and official Android apps sharing similar app names. More details/discussions of the two strategies are in our technical report [38].

Modality Generation Intuitively, our system generates function modalities by mining an ordered sequence (path) of sensitive function nodes from known malware samples' behavior graphs, as illustrated in Figure 2. In particular, for each path of each known malware's CBG, we denote a subpath of it as a sensitive path, if it starts from one sensitive function node and ends with another sensitive function node. Then, after *removing those non-sensitive nodes* sitting in the middle of the sensitive path, we generate function modalities from the transformed sensitive path by extracting all of its subsequences. Generating function modalities involves the two steps: Extract Sensitive Path and Extract All Subsequences. (Due to the page limit, we leave the detailed algorithm in [38].)
Step 1: Extract Sensitive Path. For each pair of sensitive nodes S_i and S_j, we extract sensitive paths P_{ij} of framework API functions from all known malware samples' CBGs, if P_{ij} starts from S_i and ends with S_j. In particular, for each path in the malware's CBG, we generate modalities from the longest sensitive path, which will cover the results extracted from those shorter sensitive paths. As an illustrative example in Figure 4(d), if f_{01}, f_{m4} and f_{02} are sensitive nodes, the longest sensitive path could be illustrated as Figure 5(a). Then, we could generate a transformed path of function nodes, through removing non-sensitive nodes in the middle. In the previous example, a transformed sensitive path $f_{01} \to f_{m4} \to f_{02}$ can be extracted by removing two non-sensitive nodes f_{m1} and "null" in the middle.
Step 2: Extract All Subsequences. We generate function modalities by extracting all *order-preserving*[4] subsequences of the transformed path of sensitive function nodes. Accordingly, we could mine four function modalities from the previous example (see

[4] This implies that the order of two function nodes in the subsequece remains the same as in the original path.

Figure 5(b)). Since DroidMiner utilizes *all subsequences* to generate the modalities instead of using the original single long sequence/path, DroidMiner is resilient to many evasion attempts by malware, e.g., insertion of loop framework API calls in the middle that serve no purpose other than adding noise. Hence, our modalities are a more robust representation of specific malware programming logic than using simple call sequences or frequencies.

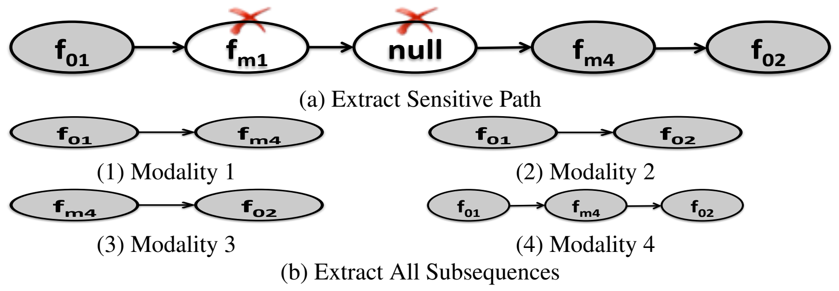

Figure 5. An illustration of function modality generation.

3.3 Identification of Modalities

After mining modalities, the second phase of DroidMiner involves the identification of modalities in unknown apps (i.e., determine which modalities are contained in unknown apps). As illustrated in Figure 2, for each unknown app, DroidMiner identifies its modalities by extracting its behavior graph and generating a Modality Vector, specifying the presence of mined modalities.

More specifically, for each unknown app, DroidMiner generates its behavior graph and extracts sensitive paths from the graph. Then, DroidMiner obtains all potential sub-paths by generalizing those sensitive paths. For each sub-path, if it is a modality (belonging to the mined modality set), we consider this app to contain this modality. This process of modality extraction is highly efficient due to the limited number of sensitive nodes present in each app. In this way, once M different modalities are mined from known malware samples, each app could be transformed into a boolean vector $(X_1, X_2, \ldots, X_M)$, denoted as a "Modality Vector": $X_i = 1$, if the app contains the modality M_i; otherwise, $X_i = 0$. In this way, an app's Modality Vector could represent its spectrum of potentially malicious behaviors.

3.4 Modality Use Cases

We introduce how to use an Android app's Modality Vector to address the following three use-case scenarios: Malware Detection, Malware Family Classification, and Malicious Behavior Characterization.

Malware Detection. The first use case involves simply determining whether or not an Android app is malicious. In fact, it is challenging to make a confirmative decision. For example, although some sensitive behaviors (e.g., sending network packets or SMS messages to remote identities) are commonly seen in malware, without a deep analysis about such behaviors (e.g., the analysis of the reputation of those remote identities), we cannot blindly declare all apps with such behaviors to be malware. However, Android malware typically needs to use multiple sensitive functions (or modalities) to achieve its objectives: e.g., (i) sending SMS AND blocking notifications or (ii) rooting the phone AND installing new apps. According to this observation, DroidMiner considers an app to be malicious only if the cumulative malware indication from all of its modalities exceed a sufficient threshold. That is, the single usage of one modality in a benign app will not cause it to be labeled as malware. We use machine learning techniques to learn the indication of each modality used in the cumulative scoring process. More specifically, we consider each of mined modalities as one detection feature in the machine-learning model. Thus, the number of detection features is equal to the dimensionality of the *Modality Vector*. By feeding modality vectors extracted from known malware and benign apps into the applied machine-learning classifier, the indication of those modalities that are highly correlated with malicious apps are up-weighted in judging an app to be malicious; those modalities that are also commonly used in benign apps are down-weighted.

DroidMiner could also be designed to detect malware using pre-defined (strict) detection rules, like policy-based detection systems discussed in Section 5, which may lead to a lower false positive rate. However, such a policy-based design requires considerable domain knowledge and comprehensive manual investigations of malware samples, which can limit overall scalability and thus is more suitable to be applied to detect specific attacks. Our goal of designing a fully automated approach motivated us to use the learning-based approach instead of policy-based ones.

Malware Family Classification. Besides detecting malware from a corpus of apps, another use case is automatically determining the family that an identified malware sample may belong to, given sufficient knowledge from existing known malware families. This problem is also important for understanding and analyzing malware families. In fact, many antivirus vendors still rely on common code extraction techniques, which typically manually extract signatures after gathering a large collection of malware samples belonging to the same malware family.

Different malware samples in the same family tend to share similar malicious behaviors, which could be depicted by *Modality Vectors*. Thus, the degree of similarity between the Modality Vectors of two malware samples provides an indication of whether they belong to the same family. Hence, with the knowledge of Modality Vectors mined from malware samples belonging to existing malware families, we could also build a malware family classifier for unknown malicious apps.

Malicious Behavior Characterization The final use case involves characterizing the specific malicious functionality embedded within a candidate app. To solve this problem, we essentially need to know which modalities could be used to achieve specific malicious behaviors. Then, if an app contains those modalities, we could claim with high confidence that the app is malicious. To realize this goal, we use a data mining technique, called "Association Rule Mining [6]". Due to the page limit, we only introduce the basic intuition here, and recommend interested readers to read our extended version [38]. Intuitively, we mine relationships (association rules) from modalities to malicious behaviors. More specifically, DroidMiner derives association rules by analyzing the relationship between the modality usage in existing known malware families and their corresponding malicious behaviors. For example, Zsone has two known malicious behaviors: (i) sending SMS and (ii) blocking SMS. DroidMiner associates modalities generated from this family to these two behaviors.

4 Evaluation

We present our evaluation results by implementing a prototype of DroidMiner and applying it to apps collected from existing third-party Android markets and from the official Android market (*GooglePlay*).

4.1 Prototype Implementation

We implement a prototype of DroidMiner on top of a popular static analysis tool (Androguard [2]). In our experience, comparing with other public Android app decompilers (e.g., Dex2Jar [7] or Smali [10]), Androguard produces more accurate decompilation results, especially in terms of handling exceptions. The prototype decompiles an Android app into Dalvik bytecode, further builds its behavior graph and mines its modalities based on the bytecode.

The method call graph in an app is built by analyzing the caller-callee relationships of all methods used in the app. For each method, DroidMiner extracts its callee methods by analyzing the *invoke-kind* instructions (e.g., *invoke-virtual* and *invoke-direct*) used in the method. Since Android is event-driven, the entrance of an app could also be UI event methods (e.g., onClick). However, such UI event methods could only be executed after the corresponding UI event listeners are registered (e,g., setOnClickListener). Thus, to make the program logic more complete, DroidMiner adds an edge from UI events listeners to corresponding UI event methods, although there is no such caller-callee relationships in the bytecode. We use a similar strategy to address registered event handlers and threads. DroidMiner generate the control-flow graph in each method by analyzing branch jump instructions (e.g., if-eq). In our implementation, all behavior graphs are stored in XGMML [8] format, a highly efficient format for graph representation and matching.

4.2 Data Collection

We crawled four representative marketplaces, including GooglePlay, and three alternative markets (SlideMe [9], AppDH [5], and Anzhi [4]). The collection from the alternative markets occurred during a 13-day period. GooglePlay collection was harvested during a two-months period. As described in Table 1, in total, we collected 67,797 free apps, where 17% of the apps (11,529) were collected from GooglePlay, and the remaining 83% (56,268) were harvested from the alternative markets.

Table 1. Summary of Android App Collection

	Official Market	SlideMe	AppDH	Anzhi
Location	U.S.A	U.S.A	China	China
Number of Apps	11,529	15,129	2,349	38,790
Total Apps	11,529 (17%)	56,268 (83%)		
	67,797			

Next, we isolate the set of malicious apps from our corpus by submitting the set of apps from the alternative markets to "VirusTotal.com", which is a free antivirus (AV) service that scans each uploaded Android app using over 40 different AV products [11]. For each app, if it has been scanned earlier by an AV tool, we can obtain the full VirusTotal report, which includes the first and last time the app was seen, as well as the results from the individual AV scans. For example, BitDefender has a report for a malicious app (*MD5: 7acb7c624d7a19ad4fa92cacfddd9257*) as Droid.Trojan.KungFu.C. Thus, we obtained 1,247 malicious apps identified by at least one AV product. For each malicious app, we extract its associated malware family name, and when AV reports disagree, we derive a consensus label using the label that dominates the responses from the AV tools. In addition, we obtain another set of malware samples from Genome Project [3,40]. This dataset contains the family label for each malware sample. After excluding those already appeared in our crawled malware set, there are 1,219 different malware apps. Thus, in total, our malware dataset consists of *2,466* (1,247+1,219) unique malicious apps that belong to *68* malware families.

We construct a benign dataset using popular apps collected from GooglePlay. To further clean this dataset, we submit our candidate set of 11,529 free GooglePlay apps to VirusTotal, of which 1,126 apps were labeled as malicious by one AV product. We discarded those apps and constructed our benign dataset using the remaining *10,403* free GooglePlay Android apps. Clearly, the benign app dataset may still contain some malicious apps, but this set has at least been vetted by the GooglePlay anti-malware analysis and by more than 40 AV products from VirusTotal. The problem of producing a perfect benign app corpus remains a hard challenge, and we note that a similar approach to construct a benign app dataset has been used in prior related work [29].

4.3 Evaluation Result

Below, we summarize our system evaluation results for malware detection, malware family classification, behavior characterization, and efficiency.

Malware Detection. As introduced in Section 3.4, we utilize machine learning techniques to conduct malicious app detection. To better evaluate the effectiveness of DroidMiner, we utilize four widely used machine learning (ML) classifiers: *NaiveBayes*, *Support Vector Machine (SVM)*, *DecisionTree* and *Random Forest*.

For each classifier, we conduct a series of experiments using a *ten-fold cross validation* to compute three performance metrics: *False Positive Rate*, *Detection Rate*, and *Accuracy*. Specifically, we divide both malicious and benign datasets randomly into 10 groups, respectively. In each of the 10 rounds, we choose the combination of one group of benign apps and malicious apps as the testing dataset, and the remaining 9 groups as the training dataset. We further compare the performance of DroidMiner with another classifier (used in [29]), which uses registered permissions as major detection features, based on our collected dataset.[5]

Table 2 shows the results of using permission versus DroidMiner based on different classifiers. We see that for all four classifiers, the usage of modalities as the input feature set (DroidMiner) produces a higher detection rate and lower false positive rate than the approach of using permission features [29]. Particularly, using *Random Forest* DroidMiner achieved a detection rate of 95.3%, roughly 10% higher than the that of using permission. Furthermore, DroidMiner produced a lower false positive rate of (0.4%), or around 1/5th of the compared approach. Also, DroidMiner could maintain the detection rate higher than 86% for all four classifiers. Due to space limit, we leave a more detailed analysis of false positives and negatives in [38].

[5] We are unable to provide a direct corpus comparative evaluation with other detection systems discussed in related work [41,18], because they are not publicly available and it is generally difficult to completely reproduce similar systems and parameter selections.

Table 2. Detection Results (DR denotes detection rate, FP denotes false positive).

Classifier	NaiveBayes		SVM		Decision Tree		Random Forest	
Method	Permission	DroidMiner	Permission[29]	DroidMiner	Permission[29]	DroidMiner	Permission[29]	DroidMiner
DR	75.1%	82.2%	78.8%	86.7%	85.7%	92.4%	87.0%	95.3
FP Rate	7.2%	4.4%	3.5%	1.1%	2.2%	1.0%	2.0%	0.4%

Family Classification The purpose of this experiment is to measure the accuracy of using Modality Vectors to correctly assign apps that are classified as malicious to their correct corresponding malware family. To conduct the malware family classification, we use samples from 12 families, each of which has more than 50 samples. The number of samples of each family is shown in Table 3.

Table 3. Malware samples used for classification

Ind	Family	Num	Ind	Family	Num	Ind	Family	Num	Ind	Family	Num
1	GingerMaster	166	4	AnserverBot	187	7	KMin	52	10	DroidKungFu3	327
2	GoldDream	57	5	DroidKungFu	70	8	BaseBridge	122	11	DroidKungFu4	10
3	Airpush	568	6	Leadbolt	52	9	Geinimi	69	12	Plankton	194

For each family, we use half of the samples as training dataset, and the other half as the testing dataset. In this case, the classification accuracy represents the ratio of the number of correctly classified samples to the total number of samples in the test dataset. Here, we use *Random Forest* for classifying both the training and testing datasets. The classifier produces a relatively high classification accuracy of 92.07%.

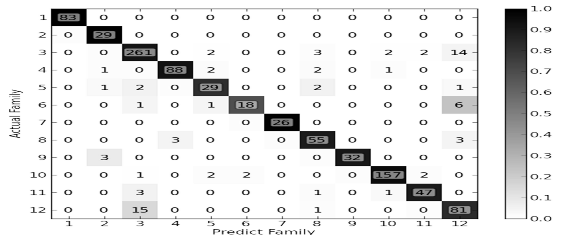

Figure 6. The confusion matrix of malware classification for multiple malware families.

Figure 6 shows the confusion matrix produced from our classification of the dataset into the malware family label set. The value of the cell (i, j) in the matrix shows the number of samples in family i, which are classified as being family j. Thus, the central diagonal in the matrix shows the number of *correctly* predicted samples per malware family. The darker the cell color is, the higher the classification accuracy is. With the exception of *Leadbolt* (index is 6), most of the other families achieve an accuracy higher than 90%. *Leadbolt* is an adware family, and thus its implementation may be influenced by the campaign it is serving, and thus producing a behavior that has a wide variability, leading its samples to appear to match a wider range of potential families.

Behavior Characterization As described in Section 3.4, to characterize malicious behaviors, we construct a behavior matrix based on malicious behaviors observed within an existing training set of known malware apps. To decrease sampling bias, we produce our *training* dataset using malware samples from families which have a minimum of 5 members. Next, for each family, we manually extract a malicious behavior description for this family using documentation describing the malware family from sites that contain malware analysis reports, such as threat reports from various AV companies (e.g., Symantec.com). There are many detailed public sources of information regarding malicious behavior description for many existing Android malware families. For this experiment, we focus on the following six malicious behaviors commonly observed within many malware families: stealing phone information (GetPho), Sending SMS (SdSMS), blocking SMS (BkSMS), communicating with a C&C (C&C), escalating root privilege (Root) and accessing geographical information (GetGeo). We refer interested readers to [38] for more details.

Table 4. Characterizations on 10 malware samples.

MD5	Family	Behavior
917a1aa8fafb97cdb91475709ca15cdb	MobileTX	SdSMS, C&C
49ea90de2336dccee188c3078ea64656	Gappusin	SdSMS, BKSMS, C&C, GetGeo
d6aea5963681cf6415cc3f221e4e403b	Cosha	SdSMS, C&C, GetGeo
8ef081ff9fb2dd866bfc6af6749abdcf	Fakeflash	C&C
a835b82de9e15330893ddf2da67a6a49	HippoSMS	SdSMS, BkSMS
bbb6f9a1aad8cc8c38d4441bac4852c0	DroidDeluxe	Root
9b0d331aa9019bfb550f4753aba45d27	RogueLemon	SdSMS, BKSMS, C&C
cfa9edb8c9648ae2757a85e6066f6515	Spitmo	GetPho, SdSMS, BKSMS, C&C
ee0f74897785eb3f7af84a293263c6c5	Gamex	Root
c00e43c563ecadf1e22097124538c24a	Tapsnake	C&C, GetGeo

Efficiency We now consider the performance overhead of DroidMiner in identifying modalities. As described in Section 3.3, modality identification involves three steps: 1) decompilation, 2) behavior graph generation and 3) modality vector generation. Table 5 shows the mean and median value of time spent on each step and the overall time required to identify modalities for all collected apps. Table 5 illustrates that DroidMiner expended an average of 19.8 seconds and a median of 5.4 seconds to identify modalities in an app. We provide a fine-grained analysis of the time used for generating behavior graphs in our extended version [38].

Table 5. Time for identifying modalities.

Step	Decompile	Behavior Graph	Modality Vector	Overall
Mean	3.87	15.19	1.10	19.83
Median	1.65	3.08	0.56	5.35

5 Related Work

5.1 Mobile Malware Detection

System Call Monitoring. Systems such as [30,33,34] detect malware by monitoring and analysis of system calls. A fundamental shortcoming of such approaches is the semantic gap between the system calls and specific behaviors. DroidScope [37] is designed to reconstruct both OS-level and Java-level semantics. Their dynamic analysis approach is limited by path exploration challenges.

Android Permission Monitoring. Enck et al. studied the security of Android apps by analyzing the permissions registered in the top official Market apps [21]. Stowaway [23] and COPES [16] are designed to find those apps that request more permissions than they need. PScout [15] analyzes the usage trend of permissions in Android apps. Kirin [22] detected malicious Android apps by finding permissions declared in Android apps that break "pre-defined" security rules. More recent work also detected malicious Android apps by designing several classifiers, whose features were built primarily on the application categories and permissions [29]. A concern with these approaches is false positives stemming from the coarse-grained nature of permissions and the highly common nature of benign apps to over-claim their set of required permissions. Mario et al. [24] presented their studies of permission request patterns of Android and Facebook applications.

Framework API Monitoring. DroidRanger [41] and Pegasus [18] detect malicious Android apps by statically matching against "pre-defined" signatures (permissions and Android Framework API calls) of well-known malware families. Such approaches requires semi-manual analysis of suspicious system calls and manual selection of heuristics (or detection patterns). Thus, they are not systematic and not robust to the evolution of malware. In [36,14], the frequencies of API calls were used as detection features, and more recently in [12], the names and parameters of APIs and packages were used as detection features. Such studies differ fundamentally from DroidMiner in that our modalities capture the connections of multiple sensitive API functions, not just the frequency or names of APIs.

Online Malware Detection Service. We intend to make DroidMiner available as a public webservice for Android malware analysis and detection. Similar public services include AndroTotal [27] which allows users to submit applications and have them simultaneously analyzed by various mobile antivirus systems and CopperDroid [32] which performs system-call centric dynamic analysis.

Due to space limit, we leave more detailed comparisons and discussions in [38].

5.2 Android Platform Security Defense and Analysis.

Existing studies have also developed several security extensions to defend against specific types of attacks. TaintDroid [20] detects those apps that may leak users' privacy information. However, it is not designed to detect other types of malicious behaviors such as stealthily sending of SMS. RiskRanker [42] detects malicious apps based on the knowledge of known Android system vulnerabilities, which could be utilized by malicious apps, and several heuristics. Dendroid [35] is a static analysis tool which specializes in text mining of android malware code. Quire [19] prevents confused deputy attacks. Bugiel et al. [17] proposed a security framework to prevent both confused deputy attacks and collusion attacks. AppFence [25] protects sensitive data by either feeding fake data or blocking the leakage path. Apex [28] allows for the selection of granted permissions, and Kirin [22] performs lightweight certification of applications. Paranoid Android [30], L4Android [26] and Cells [13] utilize the virtual environment to secure smartphone OS. SmartDroid [39] automatically finds UI triggers that result in sensitive information leakage.

6 Discussion

DroidMiner Against Zero-day Attacks. Emerging malware generally falls into two classes: fundamentally new strain with entirely novel code bases, and malware that improves (evolves) from an existing code base. The latter form arguably represents the dominant case. We believe DroidMiner is well designed to adapt to evolutionary change in existing code bases, and thus useful in detecting most emerging variant strains. As long as new malware launches malicious behaviors through utilizing modalities observed in known malware families, DroidMiner should detect it. For entirely novel malware strains, an additional strength of DroidMinder is that unlike traditional systems that require human expertise, DroidMiner's features (modalities) can be automatically learned and updated by feeding new malware samples.

DroidMiner Against Common Evasion Techniques. We can envision that Android malware may evolve to be more evasive. As observed by DroidChameleon [31], common malware transformation techniques (e.g., repackaging, changing field names, and changing control-flow logic) could evade many existing commercial anti-malware tools. However, DroidMiner is resilient to these common evasion techniques studied in [31]. Specifically, DroidMiner does not rely on specific signing signatures or class/method-/field names to detect malware. The simple program transformation (resigning, repackaging, changing names) will not affect the detection model used in DroidMiner. Another type of evasion technique is to insert noisy code and spurious calls in between malicious sequences, or to change specific control-flow logic. However, DroidMiner is designed to extract *all subsequences* of suspicious control-flow logic commonly seen in malware (instead of relying on the exact matching of one full/long execution path). As long as the malware follows a known programming paradigm to achieve malicious goals (e.g., intercepting short text messages after receiving them, and obtaining the phone number before sending it), DroidMiner could still capture such suspicious logic regardless of the noisy/spurious API injections in the middle of execution paths. Last but not least, malicious apps may include a large number of benign patterns to confuse DroidMiner. As mentioned earlier, our learning procedure typically down-weights modalities commonly used in benign apps and up-weights truly malicious modalities learned before. Thus, DroidMiner still has a good tolerance of such evasion.

Limitations and Future Work. Like any learning-based approach, DroidMiner requires an accurate training dataset to mine its malicious behaviors into modalities. The effectiveness of our approach depends on the quality of the given training data, e.g., labeled malicious Android apps and their families. Fortunately, it was easy for us to obtain such data (thanks to prior research efforts from academia and industry). In fact, one may also recognize DroidMiner's automatic learning approach as a feature rather than a strict liability. Whereas most existing approaches require significant manual labor to generate signature, specifications, and models for detection, DroidMiner offers far more automated model generation.

DroidMiner currently employs static analysis, which is a reasonable choice given that current Android apps are relatively easy to reverse engineer statically, unlike notorious PC-based malware. Like other Java static analysis studies, DroidMiner may fail to identify certain usages of instances/methods, which are encrypted or made by using Java Reflection and native code. This serves as another motivation for us to incorporate dynamic analysis in our future work.

7 Conclusion

DroidMiner is a new static analysis system that automatically mines malicious parasitic code segments from a corpus of malicious mobile applications, and then detects the presence of these code segments within other, previously unlabeled, mobile apps.

We present our DroidMiner prototype and an extensive evaluation of this algorithm on a corpus of over 2,400 malicious apps. From these 2,400 malware apps DroidMiner achieves a 95% accuracy rate in processing over 77,000 samples from real-world app stores. Further, we show that DroidMiner achieves a 92% accuracy in assigning malicious labels to blind test suites.

8 Acknowledgments

This material is based upon work supported in part by the National Science Foundation under Grant CNS-0954096, IIS-0905518 and the Air Force Office of Scientific Research under Grant FA9550-13-1-0077. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF and AFOSR.

References

1. 60 percentage of android malware hide in fake versions of popular apps. `http://thenextweb.com/google/2012/10/05/over-60-percent-of-android-malware-comes-from-one-family-hides-in-fake-versions-of-popular-apps/`.
2. Androguard. `http://code.google.com/p/androguard/`.
3. Android malware genome project. `http://www.malgenomeproject.org/`.
4. Anzhi android market. `http://www.anzhi.com/`.
5. App dh android market. `http://www.appdh.com/`.
6. Association mining rule. `http://en.wikipedia.org/wiki/Association_rule_learning`.
7. Dex2jar. `https://code.google.com/p/dex2jar/`.
8. extensible graph markup and modeling language. `http://www.cs.rpi.edu/research/groups/pb/punin/public_html/XGMML/draft-xgmml-20001006.html`.
9. Slideme android market. `http://slideme.org/`.
10. Smali. `https://code.google.com/p/smali/`,.
11. Virustotal. `https://www.virustotal.com/`.
12. Y. Aafer, W. Du, and H. Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *Proc. of the 9th SecureComm*, 2013.
13. J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: A virtual mobile smartphone architecture. In *Proc. of the 23rd SOSP*, 2011.
14. D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *Proc. of NDSS*, 2014.
15. K. Au, Y. Zhou, Z. Huang, D. Lie, X. Gong, X. Han, and W. Zhou. Pscout: Analyzing the android permission specification. In *Proc. of the 19th CCS*, 2012.
16. A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon. Automatically securing permission-based software by reducing the attack surface: An application to android. In *Proc. of the 27th IEEE/ACM International Conference On Automated Software Engineering, 2012*.
17. S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *Proc. of the 19th NDSS*, 2012.
18. K. Chen, N. Johnson, V. Silva, S. Dai, K. MacNamara, T. Magrino, E. Wu, M. Rinard, and Dawn Song. Contextual policy enforcement in android applications with permission event graphs. In *Proc. of the 20th NDSS*, 2013.
19. M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: lightweight provenance for smart phone operating systems. In *Proc. of the 20th USENIX Security*, 2011.

20. W. Enck, P. Gilbert, B.G. Chun, L. P. Cox, J. Jung, P. Mc-Daniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of the 9th OSDI*, 2010.
21. W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proc. of the 20th USENIX*, 2011.
22. W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proc. of the 16th CCS*, 2009.
23. A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystied. In *Proc. of the 18th CCS*, 2011.
24. M. Frank, B. Dong, A. P. Felt, and D. Song. Mining permission request patterns from android and facebook applications. In *Proc. of ICDM'12*, 2012.
25. P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These arent the droids youre looking for: Retrofitting android to protect data from imperious applications. In *Proc. of the 18th CCS*, 2011.
26. M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4android: A generic operating system frame- work for secure smartphones. In *Proc. of the 1st Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2011.
27. F. Maggi, A. Valdi, and S. Zanero. AndroTotal: a flexible, scalable toolbox and service for testing mobile malware detectors. In *Proc of SPSM'13*, 2013.
28. M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proc. of the 5th ICCS, year = 2010,*.
29. H. Peng, C. Gates, B. Sarm, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of android apps. In *Proc. of the 19th CCS*.
30. G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid android: versatile protection for smartphones. In *Proc. of the 26th ACSAC*, 2010.
31. V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proc. of the 8th ICCS*, 2013.
32. A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *Proc. of EUROSEC'13*, 2013.
33. A. Schmidt, R. Bye, H. Schmidt, J. Clausen, O. Kiraz, K. Yxksel, S. Camtepe, and A. Sahin. Static analysis of executables for collaborative malware detection on android. In *ICC Communication and Information Systems Security Symposium*, 2009.
34. A. Schmidt, H. Schmidt, J. Clausen, K. Yuksel, O. Kiraz, A. Sahin, and S. Camtepe. Enhancing security of linux-based android devices. In *Proc. of 15th International Linux Kongress*.
35. G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. B. Alis. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. 2012.
36. D. Wu, C. Mao, T. Wei, H. Lee, and K. Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Proc. of the 7th Asia JCIS*, 2012.
37. L. Yan and H. Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proc. of the 21st USENIX Security*, 2012.
38. Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. Technical report, Texas A&M University, 2014. `http://faculty.cse.tamu.edu/guofei/paper/DroidMiner_TechReport_2014.pdf`.
39. C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zhou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proc. of the 2ed workshop on security and privacy in smartphones and mobile devices*, 2012.
40. Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proc. of the 33th IEEE Security and Privacy*, 2012.
41. Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proc. of the 19th NDSS*, 2012.
42. Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proc. of the 10th MobiSys*, 2012.