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

Chao Yang, Zhaoyan Xu, Guofei Gu
Texas A&M University
{yangchao, z0x0427, guofei}@cse.tamu.edu

Vinod Yegneswaran, Phillip Porras
SRI International
{vinod, porras}@csl.sri.com

***Abstract*—DroidMiner is a new malicious Android app detection system that uses static analysis to automatically mine malicious program logic from known Android malware. DroidMiner uses a behavioral graph to abstract malware program logic into a sequence of threat modalities, and then applies machine-learning techniques to identify and label elements of the graph that match harvested threat modalities. Once trained on a mobile malware corpus, DroidMiner can automatically 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, and ($iii$) precisely characterize behaviors found within the analyzed app. While DroidMiner is not the first to attempt automated classification of Android applications based on Framework API calls, it is distinguished by its development of modalities that are resistant to noise insertions and its use of associative rule mining that enables automated association of malicious behaviors with modalities. 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, DroidMiner achieves a 95.3% detection rate, with 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%.**

## I. 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 programming patterns [56, 23] or designing detection models that use coarse-grained features such as permissions registered in the apps [45]. In this work, we introduce **DroidMiner**, a new approach to scalably 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.

DroidMiner is intended to be a fast first-level filter and not a complete malware analysis system. 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 [29, 46]. Our work formalizes and extends prior research efforts that have employed simple permission- and heuristic-based filtering to perform automated malware detection in Android markets [56]. We demonstrate that it is indeed possible to automatically perform significantly finer characterization of malicious program patterns without incurring significant additional performance costs.

We present DroidMiner's algorithm for discovering and automatically extracting malware modalities. While our efforts are primarily focused on identifying and then characterizing malware behavior, aspects of our methodology are also directly applicable to automated characterization of a broad class of Android application behaviors, including the detection of shared security vulnerabilities [24]. 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 from *GooglePlay [7]*. Specifically, 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.

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 Android app modalities: in particular use of $\delta$-analysis in sensitive node extraction and use of associative rule mining in automated mining of associations between malicious behaviors and modalities.
- An in-depth evaluation of DroidMiner with respect to its run-time performance and efficacy in malware detection, family classification, and behavioral characterization.

## II. Motivation and System Goals

Program analysis techniques (e.g., data flow analysis and control flow analysis) have been widely used to analyze and detect traditional malware. Kolbitsch et al. proposed to detect host-based malware by extracting malware's behavior graph through analyzing the function-call flow [38]. Fredrikson et al. proposed to utilize control flow to extract discriminating specifications to identify a class of malware [34]. Christodorescu et al. proposed to mine specific malicious behavioral patterns (such as decryption loops) from tracking the data flow and control flow of malware [26, 25].

Unlike traditional desktop-based malware, Android malware is composed of several components and has a complex and event-driven programming paradigm involving multiple entry points. Android defines a component-based framework for developing mobile apps. Android apps comprise four types of components: Activities, Services, Broadcast Receivers, and Content Providers. Each component in an app works as a unit performing certain tasks:

- **Activities** support basic functionalities such as interacting with end-users through graphical user interfaces (GUIs); each GUI (screen) is controlled by one Activity.
- **Services** are designed to provide interfaces in the background for communicating with other components and applications. Thus, unlike activities, services do not represent any GUI and cannot be activated/stopped by users. They will run as background processes forever until they are stopped by some certain application components.
- **Broadcast Receivers** are designed to achieve the mechanism of incident response in Android. A receiver will continuously listen to system-wide broadcast messages. When it receives relevant messages, it will automatically trigger corresponding registered events/operations.
- **Content Providers** act as database management systems, from where other components/apps could query or store an app's data without the requirement of knowing how the data is stored.

Android application authors could implement Android components in an app as Java classes by inheriting corresponding super classes defined in the Android SDK (e.g., Activity, Service, BroadcastReceiver or ContentProvider). Android components are identified by other components through registration in the applications' manifest file ("AndroidManifest.XML"). This enables these components to interact with each other by using specific intents and framework API calls defined in the Android Framework. For example, an activity could activate a service by invoking the `startService()` Framework API call. In addition, unlike traditional software, the lifetimes of Android components are controlled by a series of lifecycle API functions defined by the Android platform (e.g., `onStart()` and `onDestroy()` used in a service will start and stop the service, respectively). Moreover, the (data and control) sub-flows in an app are typically loosely connected. All these differences make Android program analysis uniquely challenging and different from traditional malware analysis.

We motivate our system design by introducing the inner working of a real-world malicious Android application. This malware sample (*MD5: c05c25b769919f d7f1b12b4800e374b5*) belongs to the family of ADRD (a.k.a HongTouTou). 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, HongTouTou registers a receiver (named "MyBoolService") to receive the boot intent `BOOT_COMPLETED` message. 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 through `onCreate()` or `onStart()`[1], 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/)`). Before sending out sensitive information and communicating with the C&C server, the service obtains network information (e.g., network types such as "CMWAP", "UNIWAP" and "wifi") by invoking two Framework API calls: `ConnectivityManager()` and `getActiveNetworkInfo()`, and reading the content provider `content://telephony/carriers-/preferapn`. It then encrypts personal information by using `Cipher.getInstance()`, `Cipher.init()` and `Cipher.doFinal()`, and exfiltrates encrypted data through SMS by using `SmsManager.getDefault()` and `sendTextMessage()`, and issuing HTTP requests by using `DefaultHttpClient.execute()`. Meanwhile, the service monitors changes to the SMS Inbox database (`content://sms/inbox/`) by calling `ContentObserver.onChange()` and deleting particular messages using `delete()`. Finally, it also attempts to download unsolicited APK files (e.g., "myupdate.apk"), to receive C&C commands and data, and to visit search engines by issuing HTTP requests.

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.

## A. 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 Android 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

[1]If the service is triggered as the first time, it will call onCreate and onStart; otherwise, it will only call onStart.

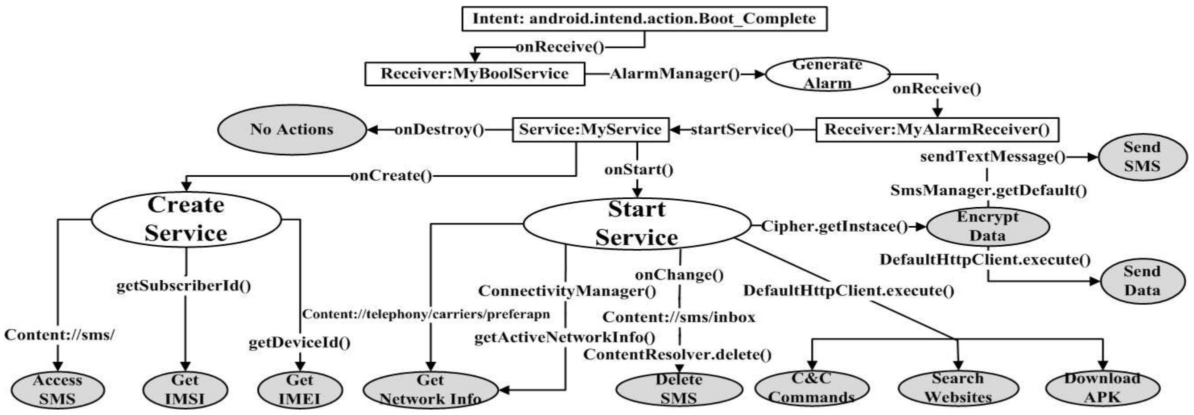


Figure 1. Capabilities embedded in malware from the ADRD family. The sample achieves its malicious functionality by invoking a series of framework APIs in order and accessing specific content providers.

incoming SMS messages. With such information, an informed analyst could further infer that this is probably a money-stealing app that uses SMS to register for a premium service, spends money, and then suppresses the end-user notification.

The input into our system is an Android application developed with the Android SDK. Currently, we do not analyze native Android applications implemented using the Android Native Develop Kit (Android NDK). According to our observations, an overwhelming majority of Android applications 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 applications is future work and outside the scope of this paper.

## III. 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 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.

An important component in our system is the Behavior Graph Generator, which takes an app as input and outputs a behavior graph representation. As the analysis of a real-world malicious app shown 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 framework APIs and accessing particular content providers). These framework APIs and sensitive 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. We provide more detailed descriptions in Section III-B.

In the Identification phase, DroidMiner will transform an unknown malicious application into its behavior graph representation (using Behavior Graph Extractor) and extract a Modality Vector (based on all trained modalities), described in Section III-C. Then, DroidMiner can apply 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 of a given Modality Vector, described in Section III-D.

### A. Behavior Graph and Modality

**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

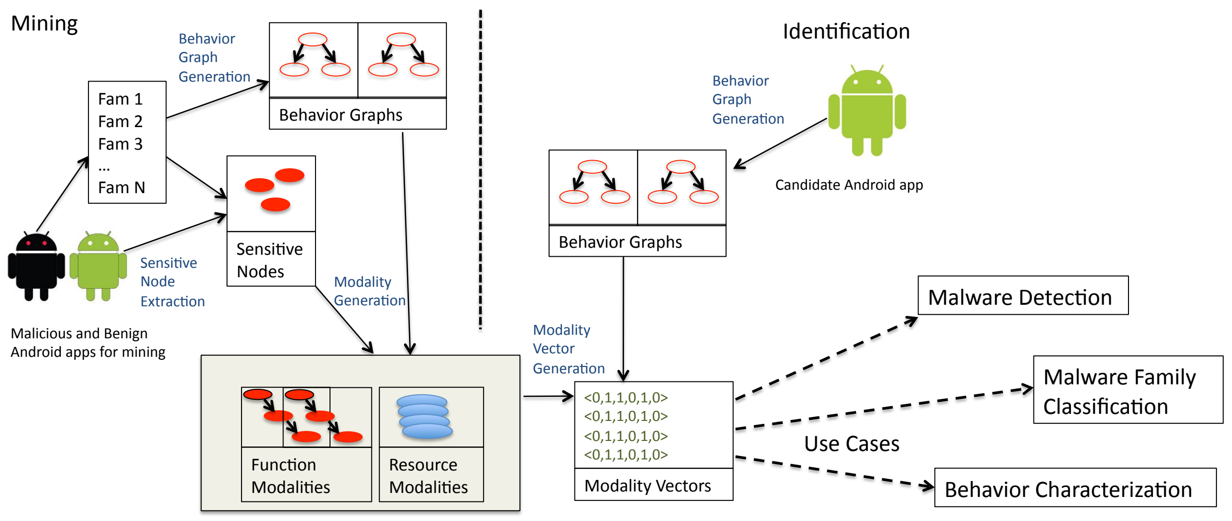


Figure 2. DroidMiner System Architecture

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)**.

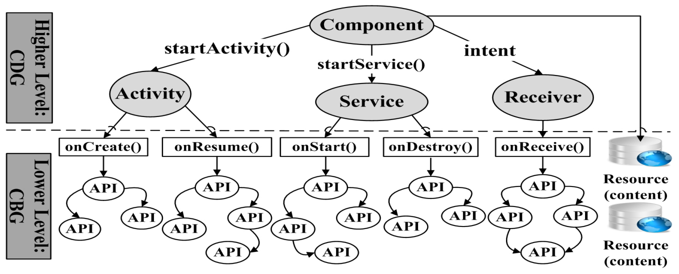


Figure 3. Two-tier behavior graph.

The **Component Dependency Graph (CDG)** (upper tier of Figure 3) represents the interaction relationships among all components in an app. In particular, 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. For example, in terms of the malware sample illustrated in Figure 1, since *MyAlarmReceiver* could activate *MyService* by using `startService()`, its CDG has an edge from a broadcast receiver node *MyAlarmReceiver* to a service node *MyService*.

The **Component Behavior Graphs (CBG)** (lower tier of Figure 3) represents each component's *lifetime* [2] 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., one Activity or one Service).
- *Lifecycle functions* ($V_{lcf}$), used to achieve the runtime logic of specific type of component (e.g., `onCreate()` in an activity, `onReceive()` in a receiver, and `onStart()` in a service).
- *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

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

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[3].

**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 of framework API functions (function modality) or a set of sensitive resources (resource modality) in commonly shared in malicious apps' behavior graphs[4], 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()` $\to$ `ContentResolver.delete()`), and a resource modality (`content://sms/inbox/`) to partially achieve the malicious behavior of deleting messages in the SMS inbox.

## B. Mining Modalities

Our desire to conduct efficient mining of modalities from large malware corpora calls for an automated approach to mining malicious patterns. We now describe the details of our modality mining process, which involves the following three steps: Behavior Graph Generation, Sensitive Node Extraction, and Modality Generation.

*1) Behavior Graph Generation:* The generation of the behavior graph of an app contains two phases: generating CDG and generating CBG. The generation of CDG is relatively straightforward. The nodes in an app's CDG are acquired by analyzing activities, receivers, and services registered in its manifest file ("AndroidManifest.xml"). As a special case, DroidMiner extracts runtime the Broadcast Receiver by analyzing instances of `Context .registerReceiver()` instead of parsing the manifest file. Much like [54], DroidMiner acquires the edges of an app's CDG by analyzing the usage of intents in each component. For example, an intent used in `startActivity(Intent)` can activate an activity; an intent used in `startService(Intent)` can start a background service.

Since Android is component driven, and each component has its own lifetime execution logic, the extraction of control-flow logic of framework API functions (rather than the control-flow logic of methods in traditional program analysis) described in the model of our CBG is more complex, which involves the following three steps: Generate Method Call Graph, Generate Control-Flow Graph, and Replace User-Defined Methods.

**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 according to the type. Those user-defined methods could be identified by using a static analysis tool. As illustrated in Figure 4(a), there is a directed edge from method $M_0$ to $M_1$, which implies $M_0$ calls $M_1$.

**Step 2: Generate Control-Flow Graph.** To extract the programming logical usage of framework API calls, DroidMiner first extract each method's control-flow graph (CFG) via 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). There is a directed edge from block $B_0$ to $B_1$, if $B_1$ is a successor block of $B_0$. Then, each block is represented as 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 $M_0$ could be formed as Figure 4(c).

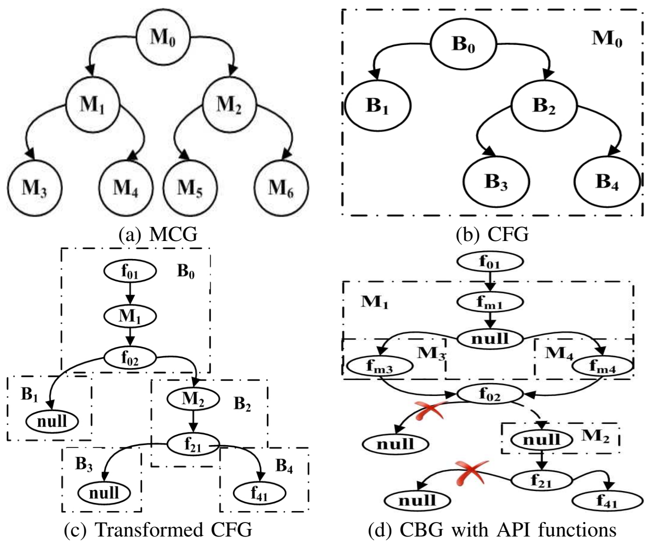
**Step 3: Replace User-Defined Methods.** As illustrated in Figure 4(c), since each leaf in the method-call graph does not call any other user-defined method, the leaf either contains a subgraph of framework API functions or is "null". Then, our approach replaces its position in its parents' control-flow 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.

*2) Sensitive Node Extraction:* A modality is essentially an ordered sequence of framework API functions and a set of sensitive resources that are commonly observed in malicious apps behavioral graphs. We denote those framework API functions and sensitive resources as sensitive nodes (the former are called *sensitive function nodes*, while 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. (Note that the process of deriving the family label for known malware is only used in the mining phase and depends on the way of collecting malware. DroidMiner automatically acquires the malware's family label by parsing antivirus reports. More details are provided in Section IV-B).

[3]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. More discussion of our strategy could be found in Section VI.

[4]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.



(a) MCG (b) CFG (c) Transformed CFG (d) CBG with API functions

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

Then, for each malware family, we extract function nodes and resource nodes that are commonly shared by at least $\theta\%$ members in this family [5].

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 ($\delta$), i.e., additional bytecode between the known malicious app and official Android apps sharing similar application names. The official apps are acquired by automatically searching for known malicious app names in GooglePlay. (We skip this process for known malware whose names are not registered in GooglePlay.)

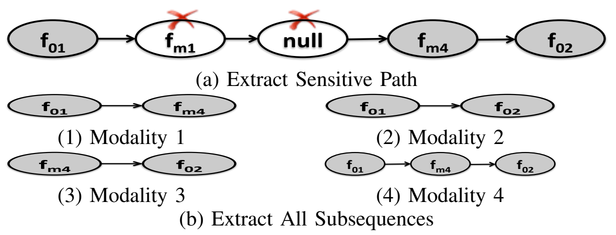
In practice, our two strategies can be complementary. To detect malicious apps, our approach relies on the control-flow logic of these sensitive nodes. Also, the effect of those false positive sensitive nodes could be further decreased when we add benign apps in training the detection model to decrease the weight of those benign patterns. In terms of the false negatives induced by the second strategy, although not all apps from the *GooglePlay* are benign, this market is still the only official one with the best reputation for Android apps. Also, if the known malware family set contains sufficient malware samples, those missed patterns through the comparison with official apps could still be found by using the first strategy.

*3) Modality Generation:* As defined in Section III-A, we now detail how we automatically generate function modalities and resource modalities.

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 following two steps: Extract Sensitive Path and Extract All Subsequences.

**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 $f_{01} \rightarrow f_{m4} \rightarrow f_{02}$ 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*[6] subsequences of the transformed path of sensitive function nodes. Accordingly, we could mine four function modalities from the previous example (see 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.



(a) Extract Sensitive Path

(1) Modality 1 (2) Modality 2 (3) Modality 3 (4) Modality 4

(b) Extract All Subsequences

Figure 5. An illustration of function modality generation.

The detailed algorithm for generating function modalities can be seen in Algorithm 1. We take a sensitive node set $V$ and a behavior graph set $G$ as inputs. For each graph $g \in G$, we extract the set $P$ containing all of its paths. Then, for each path $p \in P$, we generate an order-preserving array $S$ of all of its sensitive nodes. Next, we add all unique subsequences of $S$ to the function modality set. Finally, we output $M_f$, which is the composite modality set generate by the analysis of all malware samples. As sensitive resources comprise of content providers that typically do not have strong control-flow logic (unlike framework API functions), we consider each content provider as a resource modality in isolation.

[5]In our preliminary experiments, we set the threshold as 30%. Empirically, we found that the number of modalities does not change significantly between 30% and 90%.

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

**Algorithm 1** Generating Function Modalities

$p_i$: $i$th ($0 \leq i \leq n$) API function node in path $p$
$S$: An ordered sequence of API functions
**Input:** Sensitive Node Set $V$ and Behavior Graph Set $G$
**Output:** Function Modality Set $M_f$

```
1: $M_f \leftarrow \{\}$
2: for $g \in G$ do
3:     $P \leftarrow$ getPathSet(g)
4:     for $p \in P$ do
5:         if $p_0 \in V$ and $p_n \in V$ then
6:             $S \leftarrow \{p_0\}$
7:             for $1 \leq i \leq n-1$ do
8:                 if $p_i \in V$ then
9:                     $S \leftarrow \{p_i\}$
10:                end if
11:            end for
12:            $S \leftarrow \{p_n\}$
13:            for $m \in$ getsubSequenceSet(S) do
14:                if $m \notin M_f$ then
15:                    $M_f \leftarrow M_f \cup m$
16:                end if
17:            end for
18:        end if
19:    end for
20: end for
21: Return $M_f$
```

## *C. 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.

## *D. 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, as seen in Table VI, 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 exceeds 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 (described in Section IV) 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 V, 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.** Another use case is automatically determining which malware family an malicious app that is determined to belong to. 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 essentially be depicted by *Modality Vectors*. Thus, the degree of similarity between the Modality Vectors of two malware samples provides an indication of whether these two samples belong to the same family. Hence, with the knowledge of Modality Vectors mined from malware samples belonging to existing malware families, we could build a malware family classifier for unknown malicious apps by using machine learning techniques.

**Malicious Behavior Characterization** The final use case involves characterizing the specific malicious functionality that is 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 well-known data mining technique, called "Association Rule Mining". The problem of association rule mining is defined as follows: Let $A = \{a_1, a_2, \ldots, a_n\}$ be a set of binary attributes. Let

$B = \{B_1, B_2, \ldots, B_m\}$ be a set of items, where $B_i = \{a_{i1}, a_{i2}, \ldots, a_{in}\}$. A rule is defined as an implication of the form $X \Rightarrow Y$, where $X, Y \subseteq A$ and $X \cap Y = \phi$. The attribute sets $X$ and $Y$ are called antecedent and consequent of the rule, respectively. It represents the scenario that if the attributes in $X$ are true, then the attributes in $Y$ are also true. The support $supp(X)$ of an attribute set $X$ is defined as the proportion of items in the item set whose attributes in $X$ are all true. The confidence of a rule is defined as $conf(X \Rightarrow Y) = supp(X \bigcup Y)/supp(X)$, which could be interpreted as an estimate of the probability $P(Y|X)$.

We abstract this as an Association Rule Mining problem, i.e., we need to 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. e.g., Zsone has two known malicious behaviors: ($i$) sending SMS and ($ii$) blocking SMS. Hence, we attempt to associate modalities generated from this family to these two behaviors.

Our assumption is that in most cases malware samples belonging to particular malware families tend to express similar malicious behaviors. (While our ground truth may not be perfect, we believe that this assumption will be valid for most cases.) More specifically, given a set of modalities $M = \{M_1, M_2, \ldots, M_n\}$ and a set of malware samples $S = \{S_1, S_2, \ldots, S_n\}$ with their malware family names, for each malware sample $S_i$, we extract its Modality Vector $SM_i = \{M_{i1}, M_{i2}, \ldots, M_{ip}\}$. Given a set of malicious behaviors $B = \{B_1, B_2, \ldots, B_q\}$, we generate a behavior vector for $S_i$, $B_i = \{B_{i1}, B_{i2}, \ldots, B_{iq}\}$, where $B_{ik} = 1$, if $S_i$'s family contains malicious behavior $B_k$; otherwise $B_{ik} = 0$. Accordingly, as illustrated in Table I, we build a behavior matrix $BM_{n \times (p+q)}$ by setting: (1) $BM_{i,j} = (S_i, M_j)$ denotes whether $ith$ malware sample in the malware set contains $jth$ modality in the modality set, where $1 \le i \le n$ and $1 \le j \le p$; (2) $BM_{i,p+k} = B_{ik}$, where $1 \le i \le n$ and $1 \le k \le q$.

Table I. EXAMPLE BEHAVIOR MATRIX.

| | $M_1$ | $M_2$ | ... | $M_p$ | $B_1$ | ... | $B_q$ |
|---|---|---|---|---|---|---|---|
| $S_1$ | 0 | 1 | ... | 1 | 0 | ... | 1 |
| $S_2$ | 1 | 0 | ... | 0 | 1 | ... | 0 |
| $S_3$ | 0 | 1 | ... | 0 | 0 | ... | 1 |
| ... | ... | ... | ... | ... | ... | ... | ... |
| $S_n$ | 0 | 0 | ... | 1 | 1 | ... | 1 |

Thus, the problem of *identifying which modalities could be used to achieve the malicious behavior* $B_k$ could be transformed to the following problem:

*Finding a set of modalities,* $M_k = \{M_i | M_i \in M, 1 \le i \le m\}$,

s.t., $C(M_k) = conf(M_k \Rightarrow B_{p+k}) = supp(M_k \bigcup B_{p+k}) / supp(M_k) \ge T_{conf}$, *where* $T_{conf}$ *is a pre-defined threshold.*

Then, we consider the set of modalities $M_k$ that could be used to achieve malicious behavior $B_k$ with a confidence score of $C(M_k)$. Accordingly, we mine association rules from modalities to malicious behaviors with high confidence scores and sufficient support scores, and apply them to candidate malicious apps to characterize their malicious behaviors.

## IV. 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*).

### A. 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 [6] 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 an event-driven system, the entrance of an app could be UI event methods (e.g., `onClick`) instead of lifetime cycle methods. 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 relationship in the bytecode. We use a similar strategy to address registered event handlers by linking the handle method (e.g., `handleMessage`) to its corresponding construction method (e.g., Landroid/os/Handler.init). We also modify Androguard to generate the control-flow graph in each method by analyzing branch jump instructions (e.g., `if-eq`).

```
1  invoke-virtual v8, v3, Lcom/xxx/yyy/MyService;->getSystemService(Ljava/lang/String;)
2  move-result-object v2
3  check-cast v2, Landroid/telephony/TelephonyManager;
4  invoke-virtual v2, Landroid/telephony/TelephonyManager;->getDeviceId()
5  move-result-object v3
6  iput-object v3, v8, Lcom/xxx/yyy/MyService;->imei Ljava/lang/String;
7  invoke-virtual v2, Landroid/telephony/TelephonyManager;->getSubscriberId()
8  iget-object v3, v8, Lcom/xxx/yyy/MyService;->smsObserver Lcom/xxx/yyy/SMSObserver;
9  if-nez v3, +1e
10 new-instance v3, Lcom/xxx/yyy/SMSObserver;
11 new-instance v4, Landroid/os/Handler;
12 invoke-direct v4, Landroid/os/Handler;-><init>()V
13 invoke-direct v3, v4, v8, Lcom/xxx/yyy/SMSObserver;-><init>
14 iput-object v3, v8, Lcom/xxx/yyy/MyService;->smsObserver Lcom/xxx/yyy/SMSObserver;
15 invoke-virtual v8, Lcom/xxx/yyy/MyService;->getContentResolver()
16 move-result-object v3
17 const-string v4, 'content://sms/'
18 invoke-static v4, Landroid/net/Uri;->parse(Ljava/lang/String;)Landroid/net/Uri;
```

Figure 6. The Dalvik bytecode of the method `Myservice.onCreat()` used in a real-world malware with capabilities of reading device ID and accessing SMS.

As an illustrative example, Figure 6 shows part of Dalvik code for the method Myservice.onCreate() used in the malware sample described in Section II. From Line 1, DroidMiner will build an edge from `Myservice.onCreat()` to `Myservice.getSystemService()` in its method call graph. Frome Line 9, which contains a branch-jump instruction (`if-nez`), DroidMiner will generate a new code block, while generating the control-flow logic. Two sensitive framework API functions will be recorded from Line 4 and Line 7, and one sensitive resource (content provider) will be recorded from

Line 17. Thus each application's modalities could be mined through examination of its usage of framework API functions and content providers.

## B. Data Collection

We crawled four representative marketplaces, including GooglePlay, and three alternative Android marketplaces (SlideMe [9], AppDH [5], and Anzhi [4]). The collection from the alternative Android markets occurred during a 13-day period, from June 3 through June 15, 2012. The GooglePlay collection was harvested during a two-months period, from August 23 through October 23. Our resulting app corpus is described in Table II. In total, we collected 67,822 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 II. 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 attempt to 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 [12]. 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 application (*MD5: 7acb7c624d7a19ad4fa92cacfddd9257*) as `Droid.Trojan.KungFu.C`. In this way, 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, 55]. 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* different malware families.

In addition to the malware dataset, we also 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 [45].

## C. Evaluation Result

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

*1) Malware Detection.:* As introduced in Section III-D, 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*. *NaiveBayes* is a probabilistic-based classifier. It is fast, easy to understand, and has been widely used in spam detection studies. Since this classifier relies on the assumption that each individual feature is distributed independently of other features, its main disadvantage is that it could not learn interactions between features. Accordingly, it is not very powerful when the feature set is complex and the training set is big with high variance. *SVM* is a kernel-function-based classifier, very popular for text classification problems. It could achieve a relatively high accuracy regarding over-fitting, especially when the number of the feature dimensions is very high. However, its performance is sensitive to the choice of the kernel functions and parameters.

*Decision Tree* and *Random Forest* are two rule-based classifiers. They are non-parametric and could easily handle feature interactions. Thus, they could achieve high performance, even when the data is not linearly separable. *Random Forest* considers the problem of over-fitting, which could perform better than *Decision Tree*. Specifically, *Random Forest* is fast, scalable and often the winner for many problems in classification. Also, *Random Forest* does not require developers to excessively tune parameters as *SVM* does.

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 [45]), which uses registered permissions as major detection features, based on our collected dataset.[7] Although [45] is mainly designed to rank apps' risks based on apps' registered permissions and categories, it also reports the true positive rate and false positive rate by choosing a particular risk value as indicative of malicious apps.

Table III 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 [45]. In particular, 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

[7]We are unable to provide a direct corpus comparative evaluation with other detection systems discussed in related work [56, 23], because they are not publicly available and it is generally difficult to completely reproduce similar systems and parameter selections.

Table III. Effectiveness of malware detection (DR denotes detection rate, FP denotes false positive).

| **Classifier** | NaiveBayes | | SVM | |
|---|---|---|---|---|
| **Method** | Permission | DroidMiner | Permission[45] | DroidMiner |
| **DR** | 75.1% | 82.2% | 78.8% | 86.7% |
| **FP Rate** | 7.2% | 4.4% | 3.5% | 1.1% |
| **Classifier** | Decision Tree | | Random Forest | |
| **Method** | Permission[45] | DroidMiner | Permission[45] | DroidMiner |
| **DR** | 85.7% | 92.4% | 87.0% | 95.3% |
| **FP Rate** | 2.2% | 1.0% | 2.0% | 0.4% |

Table IV. Training time (in seconds).

| **Classifier** | NaiveBayes | SVM | Decision Tree | Random Forest |
|---|---|---|---|---|
| **Time (s)** | 0.15 | 141.21 | 76.08 | 8.15 |

approach. Also, DroidMiner could maintain the detection rate higher than 86% for all four classifiers. In addition, we can see that *Decision Tree*, *Random Forest* and *SVM* could achieve better performance than *NaiveBayes* by using both permissions and modalities as inputs, mainly because the features (both permissions and modalities) are not totally linear separable. In terms of permission, particular permissions with semantic coordination are often granted together (e.g., `SEND_SMS` and `RECEIVE_SMS`). In terms of modalities, a shorter (more general) modality may be a part of a longer (more specific) modality. Also, since *Random Forest* could solve over-fitting without the need of tuning parameters, its performance could beat *Decision Tree* and *SVM*.

**Training Time.** In this experiment, we compare the average training time used for each classifier with DroidMiner. As seen in Table IV, we find that the training time used for all four classifiers could be maintained lower than 150 seconds. Particularly, although *NaiveBayes* could not achieve an accuracy as high as other classifiers, it is the fastest one (taking only 0.15 seconds) to train the model, which validates what have we discussed about this classifier. We see that *Random Forest* is both fast (taking only 8.15 seconds) and accurate.

**Analysis of False Positives and False Negatives.** To understand false positives/negatives, we randomly choose 20 false negatives and 15 false positives generated in the case of *Random Forest* for further investigation that were induced in the first two rounds of our ten-fold cross validation experiment. Through manually analyzing these apps, we find four possible reasons that induce those false negatives: (*i*) Adware: we find DroidMiner missed identifying 11 instances of adware (seven belong to *Leadbolt* and four belong to *Airpush*), due to the diverse implementation of those adware examples. (*ii*) Native code: Since DroidMiner relies on the static analysis on the Dalvik code, it generated four false negatives that utilize native code to achieve malicious goals (e.g., rooting the phone). (*iii*) Dynamic payload: we also find four malware instances that will dynamically launch malicious payloads by either downloading from the remote servers (e.g., *Plankton*) or modifying local files ( *AnserverBot*). Since such malware initially does not contain (or activate) malicious payloads, DroidMiner could not detect them through statically analyzing Dalvik code. (*iv*) False label: we also found 1 false negative, which is labeled as malware belonging to the family of *Pjapps* by Sophos in our data collection phase. However, our manual analysis does not find any malicious payload from the app that could be seen in other apps belonging to this family. Then,

Table V. Malware samples used for classification

| **Ind** | **Family** | **Num** | **Ind** | **Family** | **Num** |
|---|---|---|---|---|---|
| 1 | GingerMaster | 166 | 7 | KMin | 52 |
| 2 | GoldDream | 57 | 8 | BaseBridge | 122 |
| 3 | Airpush | 568 | 9 | Geinimi | 69 |
| 4 | AnserverBot | 187 | 10 | DroidKungFu3 | 327 |
| 5 | DroidKungFu | 70 | 11 | DroidKungFu4 | 104 |
| 6 | Leadbolt | 52 | 12 | Plankton | 194 |

we re-submited this app to VirusTotal again and found that Sophos has changed its description on this app and identified it as benign.

Similarly, we find that our false positives could be classified into four categories: (*i*) Eight apps from GooglePlay are identified as malicious because they could send out sensitive information. In particular, four apps (three Game apps and one Shopping app) sent out phone information (e.g., IMSI) or account information[8]; three apps sent out Geo-location information; the other app could send out the contact information. (*ii*) Three apps could achieve sensitive functionalities as malware. Two of them could automatically monitor and send out the phone state, and even unlock the phone without using the password. The other one, named as "Task Manger", could kill the background process, start/restart an app, clean web browsing history, and so on. (*iii*) One app is essentially adware, which belongs to both *Leadbolt* and *Airpush*. (*iv*) Three other benign apps are falsely identified as malware.

*2) 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 V.

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%.

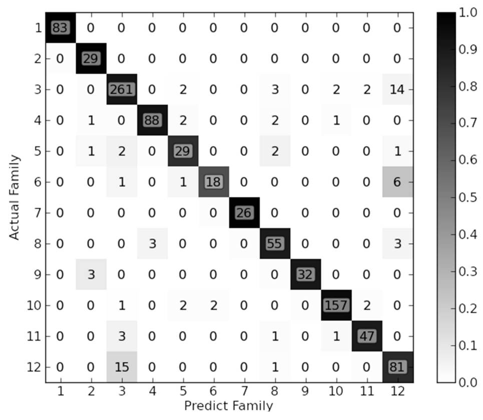


Figure 7. The confusion matrix of malware classification for multiple malware families.

[8]That could be because some Game or Shopping apps tend to use such information as the unique identifier to distinguish registered accounts.

Figure 7 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.

This experiment suggests that Modality Vectors also have a potential applicability to assist in the classification of malware family labels.

*3) Behavior Characterization:* As described in Section III-D, to characterize malicious apps' behaviors, we first construct a behavior matrix based on malicious behaviors observed within an existing training set of known malware applications. To decrease sampling bias, we produce our *training* dataset using malware samples from 29 different malware families, each contributing a minimum of 5 members. Next, for each selected 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 [11]. 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). Table VI summarizes malicious behaviors observed within those 29 malware families.

Using an Association Rule Mining system, DroidMiner automatically learned 439 behavior association rules. In Table VII, we summarize the number of association rules mined for each malicious behavior. Applying these learned rules to test new malware samples (not in the training set) with ground truth information, we find that DroidMiner could generate correct behavior characterizations. Due to page limit, we skip details of those learned rules (some representative rules could be found in Table VIII). Table IX shows the characterization results on 10 sample malware apps by applying those mined rules.

*4) Efficiency:* We now consider the performance overhead of DroidMiner in identifing modalities. As described in Section III-C, modality identification involves three steps: 1) decompilation, 2) behavior graph generation and 3) modality vector generation. Table X 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 X illustrates that DroidMiner expended an average of 19.8 seconds and a median of 5.4 seconds to identify modalities in an app. We further find that the vast majority of this time is spent on behavior graph generation.

Table VI. MALICIOUS BEHAVIORS IN DIFFERENT FAMILIES.

| Family | GetPho | SdSMS | BkSMS | C&C | Root | GetGeo |
|---|---|---|---|---|---|---|
| **ADRD** | | √ | √ | √ | | √ |
| **AnserverBot** | | √ | | √ | | |
| **Asroot** | | | | | √ | |
| **BaseBridge** | | √ | | √ | √ | |
| **BeanBot** | √ | √ | √ | √ | | |
| **Bgserv** | √ | √ | √ | √ | | |
| **DroidDream** | | | | √ | √ | |
| **DDLight** | | | | √ | | |
| **DroidKungFu1** | √ | | | √ | √ | |
| **DroidKungFu2** | √ | | | √ | √ | |
| **DroidKungFu3** | √ | | | √ | √ | |
| **DroidKungFu4** | | | | √ | | |
| **DroidKungFu5** | √ | | | √ | √ | |
| **FakePlayer** | | √ | | | | |
| **Geinimi** | √ | √ | | √ | | |
| **GingerMaster** | | | | √ | √ | |
| **GoldDream** | √ | | | | | |
| **Gone60** | √ | √ | | | | |
| **GPSSMSSpy** | | √ | | | | |
| **jSMSHider** | √ | √ | √ | | | |
| **KMin** | √ | √ | | | | |
| **Pjapps** | √ | √ | | | | |
| **Plankton** | | | | √ | | |
| **RogueSPPush** | | √ | | | | |
| **SmsSend** | | √ | | | | |
| **SndApps** | √ | | | | | √ |
| **YZHC** | √ | √ | √ | | | |
| **zHash** | | | | | √ | |
| **Zsone** | | √ | √ | | | |

Table VII. NUMBER OF ASSOCIATION RULES MINED FOR COMMON MALICIOUS BEHAVIORS

| GetPho | SdSMS | BkSMS | C&C | Root | GetGeo |
|---|---|---|---|---|---|
| 157 | 144 | 11 | 71 | 37 | 19 |

Table IX. BEHAVIORS CHARACTERIZATIONS ON 10 SAMPLE MALWARE APPS.

| 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 |

Table X. PROCESSING TIME FOR IDENTIFYING MODALITIES.

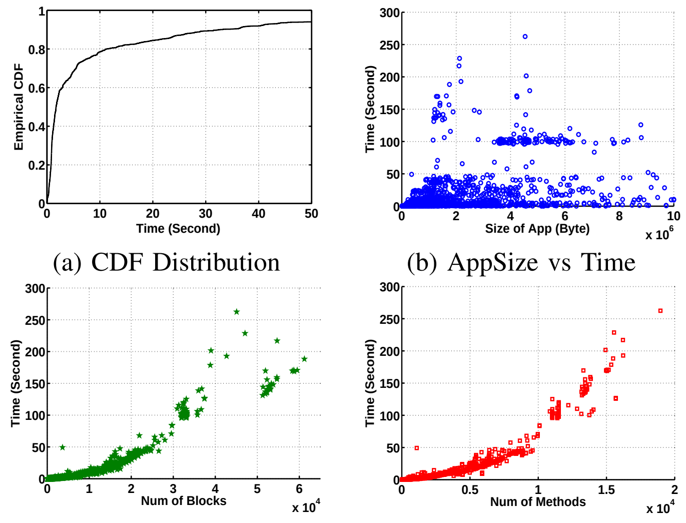
| Step | Decompile | Behavior Graph | Modality Vector | Overall |
|---|---|---|---|---|
| **Mean** | 3.87 | 15.19 | 1.10 | 19.83 |
| **Median** | 1.65 | 3.08 | 0.56 | 5.35 |

For a more fine-grained performance analysis of this step, Figure 8(a) shows the cumulative distribution of time used to generate behavior graphs for our collected apps. For approximately 80% of the apps, our system generates their behavior graphs within 10 seconds. As seen in Figure 8(c) and 8(d), the values of time spent generating behavior graphs typically rise with the increased number of control-flow blocks and programmer-defined methods found in the app. This occurs because the behavior graphs of apps are extracted through analyzing the control-flow logic of API functions with the consideration of their located control-flow blocks and

Table VIII. REPRESENTATIVE RULES FOR MALICIOUS BEHAVIOR CHARACTERIZATION.

| Index | Behavior | Rule |
|---|---|---|
| 1 | **GetPho** | HttpURLConnection.openConnection()→TelephonyManager.getSimSerialNumber() |
| 2 | **GetPho** | URL.openConnection()→TelephonyManager.getDeviceId()→HttpURLConnection.connect() |
| 3 | **GetPho** | TelephonyManager.getLine1Number()→Socket() |
| 4 | **SdSMS** | SmsManager.getDefault()→SmsManager.sendTextMessage() |
| 5 | **SdSMS** | SmsManager.getDefault()→SmsManager.sendTextMessage(); content://sms |
| 6 | **SdSMS** | SmsManager.getDefault()→SmsManager.sendTextMessage(); TelephonyManager.getSubscriberId()→DefaultHttpClient() |
| 7 | **BkSMS** | ContentObserver.onChange()→ContentResolver.delete(); content://sms/inbox |
| 8 | **BkSMS** | BroadCastReceiver.onReceive()→BroadCastReceiver.abortBroadCast() |
| 9 | **BkSMS** | Notification.PendingIntent() →NotificationManager.cancel(); content://sms/inbox |
| 10 | **C&C** | ConnectivityManager()→ConnectivityManager.getNetworkInfo() |
| 11 | **C&C** | ConnectivityManager.getActiveNetworkInfo()→TelephonyManager.getDeviceId()→DefaultHttpClient.execute() |
| 12 | **C&C** | WifiManager.getConnectionInfo()→URL.openConnection(); content://telephony/carriers/preferapn |
| 13 | **Root** | Runtime.exec()→Process.killProcess() |
| 14 | **Root** | Runtime.exe()→WifiManager.setWifiEnabled()→WifiManager.getWifiState() |
| 15 | **Root** | DefaultHttpClient.execute()→Runtime.exec() |
| 16 | **GetGeo** | LocationManager.isProviderEnabled()→LocationManager.requestLocationUpdates() |
| 17 | **GetGeo** | LocationManager.isProviderEnabled()→WifiManager.setWifiEnabled() |
| 18 | **GetGeo** | LocationManager.getBestProvider()→LocationManager.getLastKnownLocation() |

programmer-defined methods. Thus, the numbers of control-flow blocks and programmer-defined methods will affect the time used to generate the graphs. However, as shown in Figure 8(b), the time spent in generating behavior graphs does not increase due to increase in the app size. That is, a bigger app size does not necessarily contain more complex control-flow logic.



(a) CDF Distribution (b) AppSize vs Time

(c) Block Number vs Time (d) Method Number vs Time

Figure 8. Processing time for generating behavior graphs.

In summary, we find that our system does have the great potential for daily use in an Android marketplace seeking to vet newly posted applications.

## V. RELATED WORK

We broadly classify related work as belonging to three major topic areas: Mobile Malware Detection, Android Platform Security Defense and Android Platform Security Analysis.

### A. Mobile Malware Detection

The growing threat of malicious mobile applications, particularly on the smartphone platform, has attracted considerable research attention. We group proposed detection approaches for mobile malware into the following four subcategories, based on the inputs that each algorithm consumes.

**Hardware/System Performance Monitoring.** Early research on monitoring mobile malware rely on detecting abnormal hardware usage patterns (e.g., CPU and Battery) [37, 36, 40]. Shabtai et al. proposed "Andromaly", monitoring hardware information (e.g., CPU and network usage) and the number of incoming/outgoing phone calls [14]. They also designed another detection approach that analyzed suspicious temporal patterns in system usage [13]. However, as Android malware often share system usage with many competing apps, such approaches tend to suffer in accuracy, and also do not provide the fine-grained logic-focused characterization that we seek.

**System Call Monitoring.** Systems such as Crowdroid[21], Paranoid Android[46] and [48, 49] detect malware through monitoring and analysis of system calls. A fundamental shortcoming of such approaches is the semantic gap between the system calls and specific behaviors (e.g., it is exceedingly difficult to know whether an app sends an SMS to a premium number by analyzing a sequence of Android kernel-level system calls). DroidScope [53] is designed to reconstruct both OS-level and Java-level semantics. Their dynamic analysis approach is limited by path exploration challenges, but is a useful complement to DroidMiner's static-based approach.

**Android Permission Monitoring.** Enck et al. studied the security of Android apps by analyzing the permissions registered in the top 1,100 free Android apps in the official Android Market [30]. Stowaway [32] and COPES [18] are designed to find those apps that request more permissions than they need. PScout [17] analyzes the usage trend of permissions in Android apps. Kirin [31] detects 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 Naive Bayes classifiers, whose features were built primarily on the application categories and permissions [45]. 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 overclaim their set of required permissions.

**Framework API Monitoring.** Bose et al. detected malware on Symbian OS through analyzing the temporal pattern of the usage of APIs in the DLL files [19]. TaintDroid [29] tracks the data flow and the usage of framework API calls to detect those apps that may leak users' privacy information. However,

Table XI. COMPARISON OF DROIDMINER WITH PROPOSED APPROACHES FOR DETECTING MOBILE MALWARE.

| | [37, 36, 14, 13, 40] | [48, 49, 21, 46] | [31, 45] | [19] | [56] | **DroidMiner** |
|---|---|---|---|---|---|---|
| Inputs | Hardware Performance | System Call | Permission | Framework API | Framework API | Framework API |
| Platform | General | Android | Android | Symbian | Android | Android |
| Environment | Dynamic | Dynamic | Static | Static | Static and Dynamic | Static |
| Efficiency | Lower | Lower | Higher | Higher | Higher | Higher |
| Accuracy | Lower | Lower | Lower | Higher | Higher | Higher |
| Detection Rules | Automatic | Automatic | Automatic | Manual | Manual | Automatic |
| Behavior Prediction | NO | NO | NO | YES | NO | YES |

it is not designed to detect other kinds of malicious behaviors such as stealthily sending SMS. RiskRanker [57] detects malicious apps based on the knowledge of known Android system vulnerabilities, which could be utilized by malicious apps, and several heuristics, e.g., malware intends to charge the victims while blocking notifications to the victims. DroidRanger [56] detects malicious Android apps by statically matching against "pre-defined" signatures (permissions and Android Framework API calls) of well-known malware families. It also includes a heuristic-based approach to detect malicious applications from unknown families that requires semi-manual analysis of suspicious system calls. In [51], the frequencies of API calls were used as detection features, and more recently in [15], the names and parameters of APIs and packages were used as detection features. Both 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. In addition, DroidMiner introduces the use of $\delta$-analysis for sensitive node identification and associative rule mining in identifying malicious modalities. Pegasus [23] is designed to detect Android malware through abstraction of Android apps into permission event graphs, and checking whether such graphs contain pre-defined malicious intents. However, such manual selection of heuristics (or detection patterns) is not systematic and not robust to the evolution of malware.

While DroidMiner also relies on analyzing Framework API calls, it differs from existing approaches in the following ways: (1) it uses a learning-based approach to automatically generate behavior models, which are composed of individual modalities and could be used to detect malware instance from unseen families; (2) rather than simply examining whether or not the target app is malicious, it also reports specific app behavior traits (modalities); (3) instead of focusing on analyzing isolated usage of (or even the number of) Framework APIs, our detection model considers the API usage sequence, enabling DroidMiner to capture the semantic relationships across multiple APIs.

We summarize the characteristics and benefits of these approaches in Table XI. Some approaches, e.g., those using the hardware/system performance or Linux system call as the inputs, rely on the dynamic analysis (i.e., running the apps in a simulated or real environment to extract data). Thus, such approaches typically consume more time to obtain the results, leading to a relatively lower efficiency. Other approaches, e.g., those using the permission or Framework API as the inputs, typically rely on the static analysis. Such approaches could achieve relatively higher performance efficiency, but could be subject to obfuscation attacks. Curiously, obfuscation is not yet a severe problem in the Android environment, as most benign applications do not apply obfuscation for intellectual property protection. Meanwhile, approaches using Framework API calls as input could achieve higher detection accuracy than those that use permission due to the coarse-grained nature of permissions and poor software engineering practices [32].

### B. Android Platform Security Defense

Existing studies have also developed several security extensions to defend against specific types of attacks [28]. Quire [28] prevents confused deputy attacks through the design of a provenance system. As a further extension of Quire, Bugiel et al. [20] proposed a security framework to prevent both confused deputy attacks and collusion attacks with pre-defined security policies. Saint [44], Porscha [43], and CRepE [27] achieve the application isolation and protection with the usage of a fine-grained access control model and policy-oriented security policies. AppFence [35] protects sensitive data by either feeding fake data or blocking the leakage path. Checking at install time, Apex [42] allows for the selection of granted permissions, and Kirin [31] performs lightweight certification of applications. Paranoid Android [46], L4Android [39] and Cells [16] utilize the virtual environment to secure smartphone OS. SELinux [8] on Android presents a prototype implementation of SELinux on an Android device[50]. Aurasium [52] protects the system by repackaging Android apps to hook system APIs and enforcing practical policies.

### C. Android Platform Security Analysis

Our work is informed by existing studies that focus on analyzing the security mechanisms of the Android platform and its applications. Stowaway [32] is designed to find those over-privileged apps. SmartDroid [54] automatically finds UI triggers that result in sensitive information leakage. DroidChameleon [47] demonstrates the vulnerability of existing android anti-malware tools to simple malware transformation techniques (e.g., repackaging and changing names). Other related studies include attempts to detect component-hijacking vulnerabilities [41], inter-app communication vulnerabilities [24], and capability leaks [33, 22].

## VI. DISCUSSION, LIMITATIONS AND FUTURE WORK

**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.** As there is an arms race between attackers and defenders, Android malware may evolve to be more evasive. As observed by DroidChameleon [47], common malware transformation techniques (e.g., repackaging, changing field names, and changing control-flow logics) could evade many existing commercial anti-malware tools. However, DroidMiner is resilient to these common malware evasion techniques studied in [47]. 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 or to change specific control-flow logic. However, DroidMiner is designed to extract all subsequences of suspicious control-flow logic commonly seen in malware. 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 and ignore noise API injections.

**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.

Our current behavior graphs and modalities primarily model the control flow information corresponding to malware behavior, i.e., we may miss some important data flow information that could help build better behavior models. Also, the obfuscation of the control-flow logic and the constant-string for content providers in the malicious apps' bytecode may decrease the detection rate of our approach. Attackers could also split the constant-string for content providers (e.g., "content://sms/inbox/") into segments and recombine them later to avoid the identification of the usage of sensitive content providers.

DroidMiner currently employs static analysis, which is a reasonable choice given that current Android apps are relatively easy to reverse engineer statically, unlike notorious malware programs commonly seen in PC-based malware. We acknowledge that dynamic analysis provides an advantage in accurately studying runtime behaviors, and in the future we plan to extend DroidMiner to utilize a combination of static and dynamic analyses. 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.

## VII. 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. This is accomplished through a harvesting phase in which commonly labeled malware samples are abstracted into two-tiered behavioral graphs. The behavioral graphs of commonly labeled malware are then subject to a novel application of heuristics, graph comparisons, and associative rule mining that isolates their common malicious code sequences (or malicious modalities). In its detection phase, DroidMiner then derives the behavioral graph of an unlabeled mobile app sample (e.g., a new candidate App store entry), and computes the alignment of its behavioral graph to the set of known malicious modalities captured from our previously harvested malware apps. We represent this alignment computation as a modality vector, which in addition to its use for malware detection, is also useful for assigning malware family labels. 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. In practice, we believe that this approach to static comparative program analysis offers a viable and efficient vetting scheme for filtering parasitic malware in App stores.

## 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] Dex2jar. https://code.google.com/p/dex2jar/.
[7] Google play. https://play.google.com/store?hl=en.
[8] National security agency. security-enhanced linux. http://www.nsa.gov/research/selinux.
[9] Slideme android market. http://slideme.org/.
[10] Smali. https://code.google.com/p/smali/,.
[11] Symantec enterprise. http://www.symantec.com/security_response/landing/azlisting.jsp.
[12] Virustotal. https://www.virustotal.com/.
[13] *Intrusion detection for mobile devices using the knowledge-based, temporal abstraction method*, 2010.
[14] *Andromaly: a behavioral malware detection framework for android devices*, 2012.
[15] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *Proceedings of the 9th SecureComm*, 2013.
[16] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: A virtual mobile smartphone architecture. In *Proceedings of 23rd SOSP*, 2011.
[17] K. Au, Y. Zhou, Z. Huang, D. Lie, X. Gong, X. Han, and W. Zhou. Pscout: Analyzing the android permission specification. In *Proceedings of the 19th CCS*, 2012.
[18] 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 *Proceedings of the 27th*

*IEEE/ACM International Conference On Automated Software Engineering, 2012*.

[19] A. Bose, X. Hu, K. G. Shin, and T. Park. Behavioral detection of malware on mobile handsets. In *Proceeding of the 6th MobiSys*, 2008.

[20] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *Proceedings of the 19th NDSS*, 2012.

[21] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for android. In *Proceedings of the 1st Workshop on CCSSPSM*, 2011.

[22] P. P. Chan, L. C. Hui, and S. M. Yiu. Droidchecker: analyzing android applications for capability leak. In *Proceedings of the 5th ACM conference on Security and Privacy in Wireless and Mobile Networks*, 2012.

[23] 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 *Proceedings of the 20th NDSS*, 2013.

[24] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th MobiSys*, 2011.

[25] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Proceedings of the 6th ESEC/FSE*, 2007.

[26] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proceedings of 26th IEEE Security and Privacy*, 2005.

[27] M. Conti, V. T. N. Nguyen, and B. Crispo. Crepe: Context-related policy enforcement for android. In *Proceedings of the 13th ISC*, 2010.

[28] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX Security*, 2011.

[29] 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 *Proceedings of the 9th OSDI*, 2010.

[30] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX*, 2011.

[31] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th CCS*, 2009.

[32] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystied. In *Proceedings of the 18th CCS*, 2011.

[33] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security*, 2011.

[34] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *Proceedings 31th of IEEE Security and Privacy.*, 2010.

[35] 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 *Proceedings of the 18th CCS*, 2011.

[36] G. Jacoby and N. Davis. Battery-based intrusion detection. In *Proceedings of GLOBECOM)*, 2004.

[37] H. Kim, J. Smith, and K. G. Shin. Detecting energy-greedy anomalies and mobile malware variants. In *Proceedings of the 6th MobiSys*, 2008.

[38] C. Kolbitsch, P. Milani Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proceedings of Usenix*, 2009.

[39] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4android: A generic operating system frame- work for secure smartphones. In *Proceedings of the 1st Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2011.

[40] L. Liu, G. Yan, X. Zhang, and S. Chen. Virusmeter: Preventing your cellphone from spies. In *Proceedings of the 12th RAID*, 2009.

[41] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: Statically vetting android apps for component hijacking vulnerablilities. In *Proceedings of the 19th CCS*, 2012.

[42] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on ICCS, year = 2010,*.

[43] M. Ongtang, K. Butler, and P. McDaniel. Porscha: Policy oriented secure content handling in android. In *Proceedings of the 26th ACSAC*, 2010.

[44] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. In *Proceedings of the 25th ACSAC*, 2009.

[45] 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 *Proceedings of the 19th CCS*, 2012.

[46] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid android: versatile protection for smartphones. In *Proceedings of the 26th ACSAC*, 2010.

[47] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on ICCS*, 2013.

[48] 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.

[49] A. Schmidt, H. Schmidt, J. Clausen, K. Yuksel, O. Kiraz, A. Sahin, and S. Camtepe. Enhancing security of linux-based android devices. In *Proceedings of 15th International Linux Kongress*, 2008.

[50] A. Shabtai, Y. Fledel, and Y. Elovici. Securing android- powered mobile devices using selinux. In *Proceedings of 31th IEEE Security and Privacy.*, 2010.

[51] D. Wu, C. Mao, T. Wei, H. Lee, and K. Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Proceedings of the 7th Asia JCIS*, 2012.

[52] R. Xu, H. Saidi, and R. Anderson. Aurasium: practical policy enforcement for android applications. In *Proceedings of the 21st USENIX Security*, 2012.

[53] L. Yan and H. Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Security*, 2012.

[54] 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 *Proceedings of the 2ed ACM workshop on security and privacy in smartphones and mobile devices*, 2012.

[55] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 33th IEEE Security and Privacy*, 2012.

[56] 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 *Proceedings of the 19th NDSS*, 2012.

[57] Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th MobiSys*, 2012.