Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs

Mu Zhang Yue Duan Heng Yin Zhiruo Zhao
Department of EECS, Syracuse University, Syracuse, NY, USA
{muzhang,yuduan,heyin,zzhao11}@syr.edu

ABSTRACT

The drastic increase of Android malware has led to a strong interest in developing methods to automate the malware analysis process. Existing automated Android malware detection and classification methods fall into two general categories: 1) signature-based and 2) machine learning-based. Signature-based approaches can be easily evaded by bytecode-level transformation attacks. Prior learning-based works extract features from application syntax, rather than program semantics, and are also subject to evasion. In this paper, we propose a novel semantic-based approach that classifies Android malware via dependency graphs. To battle transformation attacks, we extract a *weighted contextual API dependency graph* as program semantics to construct feature sets. To fight against malware variants and zero-day malware, we introduce graph similarity metrics to uncover homogeneous application behaviors while tolerating minor implementation differences. We implement a prototype system, *DroidSIFT*, in 23 thousand lines of Java code. We evaluate our system using 2200 malware samples and 13500 benign samples. Experiments show that our signature detection can correctly label 93% of malware instances; our anomaly detector is capable of detecting zero-day malware with a low false negative rate (2%) and an acceptable false positive rate (5.15%) for a vetting purpose.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*; D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*

General Terms

Security

Keywords

Android; Malware classification; Semantics-aware; Graph similarity; Signature detection; Anomaly detection

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.

http://dx.doi.org/10.1145/2660267.2660359.

1. INTRODUCTION

The number of new Android malware instances has grown exponentially in recent years. McAfee reports [3] that 2.47 million new mobile malware samples were collected in 2013, which represents a 197% increase over 2012. Greater and greater amounts of manual effort are required to analyze the increasing number of new malware instances. This has led to a strong interest in developing methods to automate the malware analysis process.

Existing automated Android malware detection and classification methods fall into two general categories: 1) signature-based and 2) machine learning-based. Signature-based approaches [18, 37] look for specific patterns in the bytecode and API calls, but they are easily evaded by bytecode-level transformation attacks [28]. Machine learning-based approaches [5, 6, 27] extract features from an application's behavior (such as permission requests and critical API calls) and apply standard machine learning algorithms to perform binary classification. Because the extracted features are associated with application syntax, rather than program semantics, these detectors are also susceptible to evasion.

To directly address malware that evades automated detection, prior works distill program semantics into graph representations, such as control-flow graphs [11], data dependency graphs [16, 21] and permission event graphs [10]. These graphs are checked against manually-crafted specifications to detect malware. However, these detectors tend to seek an exact match for a given specification and therefore can potentially be evaded by polymorphic variants. Furthermore, the specifications used for detection are produced from known malware families and cannot be used to battle zero-day malware.

In this paper, we propose a novel semantic-based approach that classifies Android malware via dependency graphs. To battle transformation attacks [28], we extract a weighted contextual API dependency graph as program semantics to construct feature sets. The subsequent classification then depends on more robust semantic-level behavior rather than program syntax. It is much harder for an adversary to use an elaborate bytecode-level transformation to evade such a training system. To fight against malware variants and zero-day malware, we introduce graph similarity metrics to uncover homogeneous essential application behaviors while tolerating minor implementation differences. Consequently, new or polymorphic malware that has a unique implementation, but performs common malicious behaviors, cannot evade detection.

To our knowledge, when compared to traditional semantics-aware approaches for desktop malware detection, we are the first to examine program semantics within the context of Android malware classification. We also take a step further to defeat malware variants and zero-day malware by comparing the similarity of these programs to that of known malware at the behavioral level.

We build a database of behavior graphs for a collection of Android apps. Each graph models the API usage scenario and pro-

gram semantics of the app that it represents. Given a new app, a query is made for the app's behavior graphs to search for the most similar counterpart in the database. The query result is a similarity score which sets the corresponding element in the feature vector of the app. Every element of this feature vector is associated with an individual graph in the database.

We build graph databases for two sets of behaviors: benign and malicious. Feature vectors extracted from these two sets are then used to train two separate classifiers for anomaly detection and signature detection. The former is capable of discovering zero-day malware, and the latter is used to identify malware variants.

We implement a prototype system, *DroidSIFT*, in 23 thousand lines of Java code. Our dependency graph generation is built on top of Soot [2], while our graph similarity query leverages a graph matching toolkit [29] to compute graph edit distance. We evaluate our system using 2200 malware samples and 13500 benign samples. Experiments show that our signature detection can correctly label 93% malware instances; our anomaly detector is capable of detecting zero-day malware with a low false negative rate (2%) and an acceptable false positive rate (5.15%) for vetting purpose.

In summary, this paper makes the following contributions:

- We propose a semantic-based malware classification to accurately detect both zero-day malware and unknown variants of known malware families. We model program semantics with weighted contextual API dependency graphs, introduce graph similarity metrics to capture homogeneous semantics in malware variants or zero-day malware samples, and encode similarity scores into graph-based feature vectors to enable classifiers for anomaly and signature detections.

- We implement a prototype system, DroidSIFT, that performs static program analysis to generate dependency graphs. DroidSIFT builds graph databases and produces graph-based feature vectors by performing graph similarity queries. It then uses the feature vectors of benign and malware apps to construct two separate classifiers that enable anomaly and signature detections, respectively.

- We evaluate the effectiveness of DroidSIFT using 2200 malware samples and 13500 benign samples. Experiments show that our signature detection can correctly label 93% malware instances; our anomaly detector is capable of detecting zero-day malware with a low false negative rate (2%) and an acceptable false positive rate (5.15%) for vetting purpose.

Paper Organization. The rest of the paper is organized as follows. Section 2 describes the deployment of our learning-based detection mechanism and gives an overview of our malware classification technique. Section 3 defines our graph representation of Android program semantics and explains our graph generation methodology. Section 4 elaborates graph similarity query, the extraction of graph-based feature vectors and learning-based anomaly & signature detection. Section 5 shows the experimental results of our approach. Discussion and related work are presented in Section 6 and 7, respectively. Finally, the paper concludes with Section 8.

2. OVERVIEW

In this section, we present an overview of the problem and the deployment and architecture of our proposed solution.

2.1 Problem Statement

An effective vetting process for discovering malicious software is essential for maintaining a healthy ecosystem in the Android app markets. Unfortunately, existing vetting processes are still fairly

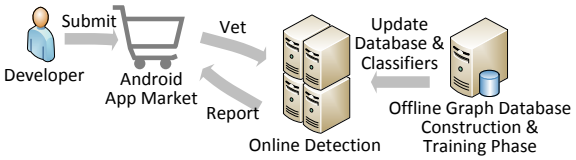


Figure 1: Deployment of DroidSIFT

rudimentary. As an example, consider the Bouncer [22] vetting system that is used by the official Google Play Android market. Though the technical details of Bouncer are not publicly available, experiments by Oberheide and Miller [24] show that Bouncer performs dynamic analysis to examine apps within an emulated Android environment for a limited period of time. This method of analysis can be easily evaded by apps that perform emulator detection, contain hidden behaviors that are timer-based, or otherwise avoid triggering malicious behavior during the short time period when the app is being vetted. Signature detection techniques adopted by the current anti-malware products have also been shown to be trivially evaded by simple bytecode-level transformations [28].

We propose a new technique, *DroidSIFT*, illustrated in Figure 1, that addresses these shortcomings and can be deployed as a replacement for existing vetting techniques currently in use by the Android app markets. This technique is based on static analysis, which is immune to emulation detection and is capable of analyzing the entirety of an app's code. Further, to defeat bytecode-level transformations, our static analysis is semantics-aware and extracts program behaviors at the semantic level. More specifically, we achieve the following design goals:

- **Semantic-based Detection.** Our approach detects malware instances based on their program semantics. It does not rely on malicious code patterns, external symptoms, or heuristics. Our approach is able to perform program analysis for both the interpretation and demonstration of inherent program dependencies and execution contexts.

- **High Scalability.** Our approach is scalable to cope with millions of unique benign and malicious Android app samples. It addresses the complexity of static program analysis, which can be considerably expensive in terms of both time and memory resources, to perform precisely.

- **Variant Resiliency.** Our approach is resilient to polymorphic variants. It is common for attackers to implement malicious functionalities in slightly different manners and still be able to perform the expected malicious behaviors. This malware polymorphism can defeat detection methods that are based on exact behavior matching, which is the method prevalently adopted by existing signature-based detection and graph-based model checking. To address this, our technique is able to measure the similarity of app behaviors and tolerate such implementation variants by using similarity scores.

Consequently, we are able to conduct two kinds of classifications: anomaly detection and signature detection. Upon receiving a new app submission, our vetting process will conduct anomaly detection to determine whether it contains behaviors that significantly deviate from the benign apps within our database. If such a deviation is discovered, a potential malware instance is identified. Then, we conduct further signature detection on it to determine if this app falls into any malware family within our signature database. If so, the app is flagged as malicious and bounced back to the developer immediately.

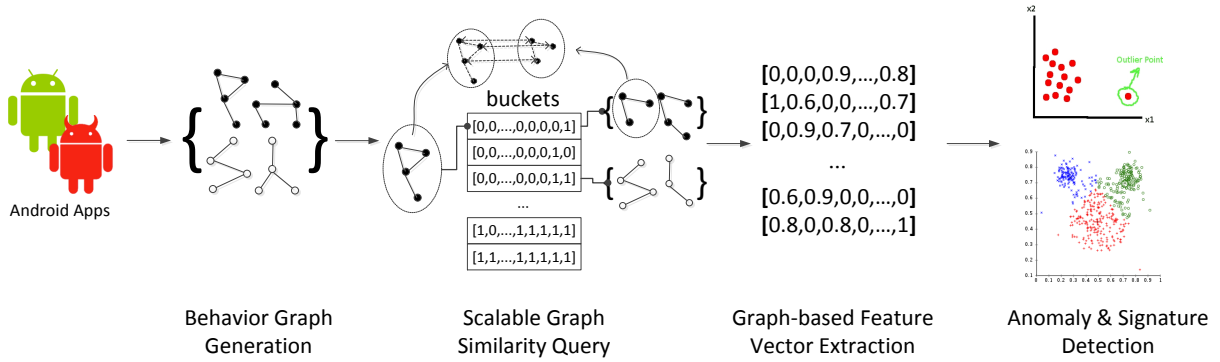


Figure 2: Overview of DroidSIFT

If the app passes this hurdle, it is still possible that a new malware species has been found. We bounce the app back to the developer with a detailed report when suspicious behaviors that deviate from benign behaviors are discovered, and we request justification for the deviation. The app is approved only after the developer makes a convincing justification for the deviation. Otherwise, after further investigation, we may confirm it to indeed be a new malware species. By placing this information into our malware database, we further improve our signature detection to detect this new malware species in the future.

It is also possible to deploy our technique via a more ad-hoc scheme. For example, our detection mechanism can be deployed as a public service that allows a cautious app user to examine an app prior to its installation. An enterprise that maintains its own private app repository could utilize such a security service. The enterprise service conducts vetting prior to adding an app to the internal app pool, thereby protecting employees from apps that contain malware behaviors.

2.2 Architecture Overview

Figure 2 depicts the workflow of our graph-based Android malware classification. This takes the following steps:

(1) **Behavior Graph Generation.** Our malware classification considers graph similarity as the feature vector. To this end, we first perform static program analysis to transform Android bytecode programs into their corresponding graph representations. Our program analysis includes entry point discovery and call graph analysis to better understand the API calling contexts, and it leverages both forward and backward dataflow analysis to explore API dependencies and uncover any constant parameters. The result of this analysis is expressed via *Weighted Contextual API Dependency Graphs* that expose security-related behaviors of Android apps.

(2) **Scalable Graph Similarity Query.** Having generated graphs for both benign and malicious apps, we then query the graph database for the one graph most similar to a given graph. To address the scalability challenge, we utilize a bucket-based indexing scheme to improve search efficiency. Each bucket contains graphs bearing APIs from the same Android packages, and it is indexed with a bitvector that indicates the presence of such packages. Given a graph query, we can quickly seek to the corresponding bucket index by matching the package's vector to the bucket's bitvector. Once a matching bucket is located, we further iterate this bucket to find the best-matching graph. Finding the best-matching graph, instead of an exact match, is necessary to identify polymorphic malware.

(3) **Graph-based Feature Vector Extraction.** Given an app, we attempt to find the best match for each of its graphs from the database. This produces a similarity feature vector. Each element of the vector is associated with an existing graph in the database. This vector bears a non-zero similarity score in one element only if the corresponding graph is the best match to one of the graphs for the given app.

(4) **Anomaly & Signature Detection.** We have implemented a signature classifier and an anomaly detector. We have produced feature vectors for malicious apps, and these vectors are used to train the classifier for signature detection. The anomaly detection discovers zero-day Android malware, and the signature detector uncovers the type (family) of the malware.

3. WEIGHTED CONTEXTUAL API DEPENDENCY GRAPH

In this section, we describe how we capture the semantic-level behaviors of Android malware in the form of graphs. We start by identifying the key behavioral aspects that must be captured, present a formal definition, and then present a real example to demonstrate these aspects.

3.1 Key Behavioral Aspects

We consider the following aspects as essential when describing the semantic-level behaviors of an Android malware sample:

1) **API Dependency.** API calls (including reflective calls to the private framework functions) indicate how an app interacts with the Android framework. It is essential to capture what API calls an app makes and the dependencies among those calls. Prior works on semantic- and behavior-based malware detection and classification for desktop environments all make use of API dependency information [16, 21]. Android malware shares the same characteristics.

2) **Context.** An entry point of an API call is a program entry point that directly or indirectly triggers the call. From a user-awareness point of view, there are two kinds of entry points: user interfaces and background callbacks. Malware authors commonly exploit background callbacks to enable malicious functionalities without the user's knowledge. From a security analyst's perspective, it is a suspicious behavior when a typical user interactive API (e.g., `AudioRecord.startRecording()`) is called stealthily [10]. As a result, we must pay special attention to APIs activated from background callbacks.

3) **Constant.** Constants convey semantic information by revealing the values of critical parameters and uncovering fine-grained API semantics. For instance, `Runtime.exec()` may execute

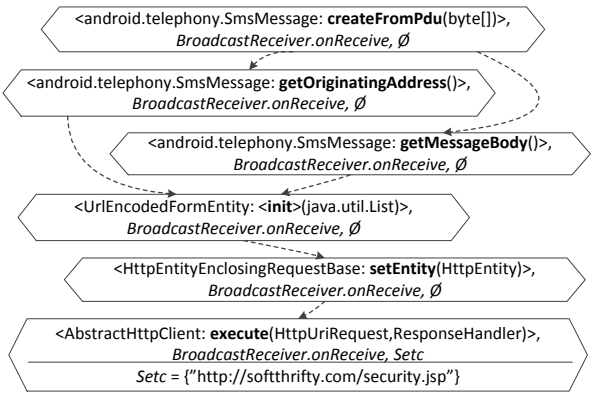


Figure 3: WC−ADG of Zitmo

varied shell commands, such as `ps` or `chmod`, depending upon the input string constant. Constant analysis also discloses the data dependencies of certain security-sensitive APIs whose benignness is dependent upon whether an input is constant. For example, a `sendTextMessage()` call taking a constant premium-rate phone number as a parameter is a more suspicious behavior than the call to the same API receiving the phone number from user input via `getText()`. Consequently, it is crucial to extract information about the usage of constants for security analysis.

Once we look at app behaviors using these three perspectives, we perform similarity checking, rather than seeking an exact match, on the behavioral graphs. Since each individual API node plays a distinctive role in an app, it contributes differently to the graph similarity. With regards to malware detection, we emphasize security-sensitive APIs combined with critical contexts or constant parameters. We assign weights to different API nodes, giving greater weights to the nodes containing critical calls, to improve the "quality" of behavior graphs when measuring similarity. Moreover, the weight generation is automated. Thus, similar graphs have higher similarity scores by design.

3.2 Formal Definition

To address all of the aforementioned factors, we describe app behaviors using *Weighted Contextual API Dependency Graphs* (WC−ADG). At a high level, a WC−ADG consists of API operations where some of the operations have data dependencies. A formal definition is presented as follows.

Definition 1. A *Weighted Contextual API Dependency Graph* is a directed graph $G = (V, E, \alpha, \beta)$ over a set of API operations Σ and a weight space W, where:

• The set of vertices V corresponds to the contextual API operations in Σ;

• The set of edges $E \subseteq V \times V$ corresponds to the *data dependencies* between operations;

• The labeling function $\alpha : V \rightarrow \Sigma$ associates nodes with the labels of corresponding contextual API operations, where each label is comprised of 3 elements: API prototype, entry point and constant parameter;

• The labeling function $\beta : V \rightarrow W$ associates nodes with their corresponding weights, where $\forall w \in W$, $w \in R$, and R represents the space of real numbers.

3.3 A Real Example

Zitmo is a class of banking trojan malware that steals a user's SMS messages to discover banking information (e.g., mTANs). Figure 3 presents an example WC−ADG that depicts the malicious behavior of a Zitmo malware sample in a concise, yet complete, manner. This graph contains five API call nodes. Each node contains the call's prototype, a set of any constant parameters, and the entry points of the call. Dashed arrows that connect a pair of nodes indicates that a data dependency exists between the two calls in those nodes.

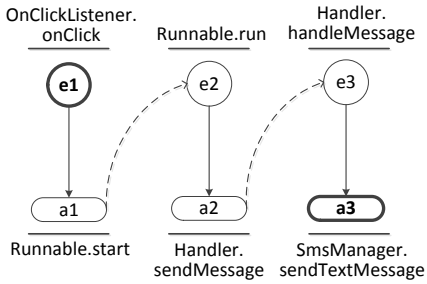


Figure 4: Callgraph for asynchronously sending an SMS message. "e" and "a" stand for "event handler" and "action" respectively.

By combining the knowledge of API prototypes with the data dependency information shown in the graph, we know that the app is forwarding an incoming SMS to the network. Once an SMS is received by the mobile phone, Zitmo creates an SMS object from the raw Protocol Data Unit by calling `createFromPdu(byte[])`. It extracts the sender's phone number and message content by calling `getOriginatingAddress()` and `getMessageBody()`. Both strings are encoded into an `UrlEncodedFormEntity` object and enclosed into `HttpEntityEnclosingRequestBase` by using the `setEntity()` call. Finally, this HTTP request is sent to the network via `AbstractHttpClient.execute()`.

Zitmo variants may also exploit various other communication-related API calls for the sending purpose. Another Zitmo instance uses `SmsManager.sendTextMessage()` to deliver the stolen information as a text message to the attacker's phone. Such variations motivate us to consider graph similarity metrics, rather than an exact matching of API call behavior, when determining whether a sample app is benign or malicious.

The context provided by the entry points of these API calls informs us that the user is not aware of this SMS forwarding behavior. These consecutive API invocations start within the entry point method `onReceive()` with a call to `createFromPdu(byte[])`. `onReceive()` is a broadcast receiver registered by the app to receive incoming SMS messages in the background. Therefore, the `createFromPdu(byte[])` and subsequent API calls are activated from a non-user-interactive entry point and are hidden from the user.

Constant analysis of the graph further indicates that the forwarding destination is suspicious. The parameter of `execute()` is neither the sender (i.e., the bank) nor any familiar parties from the contacts. It is a constant URL belonging to an unknown third-party.

3.4 Graph Generation

We have implemented a graph generation tool on top of Soot [2] in 20k lines of code. This tool examines an Android app to conduct entry point discovery and perform context-sensitive, flow-sensitive, and interprocedural dataflow analyses. These analyses locate API call parameters and return values of interest, extract constant parameters, and determine the data dependencies among the API calls.

Entry Point Discovery.

Entry point discovery is essential to revealing whether the user is aware that a certain API call has been made. However, this identification is not straightforward. Consider the callgraph seen in Figure 4. This graph describes a code snippet that registers a

Algorithm 1 Entry Point Reduction for Asynchronous Callbacks

$M_{entry} \leftarrow$ {Possible entry point callback methods}
$CM_{async} \leftarrow$ {Pairs of ($BaseClass$, $RunMethod$) for asynchronous calls in framework}
$RS_{async} \leftarrow$ {Map from $RunMethod$ to $StartMethod$ for asynchronous calls in framework}
for $m_{entry} \in M_{entry}$ **do**
  $c \leftarrow$ the class declaring m_{entry}
  $base \leftarrow$ the base class of c
  if $(base, m_{entry}) \in CM_{async}$ **then**
    $m_{start} \leftarrow$ Lookup(m_{entry}) in RS_{async}
    for $\forall$ *call* to m_{start} **do**
      $r \leftarrow$ "this" reference of *call*
      $PointsToSet \leftarrow$ PointsToAnalysis(r)
      if $c \in PointsToSet$ **then**
        $M_{entry} = M_{entry} - \{m_{entry}\}$
        BuildDependencyStub(m_{start}, m_{entry})
      end if
    end for
  end if
end for
output M_{entry} as reduced entry point set

`onClick()` event handler for a button. From within the event handler, the code starts a thread instance by calling `Thread.start()`, which invokes the `run()` method implementing `Runnable.run()`. The `run()` method passes an `android.os.Message` object to the message queue of the hosting thread via `Handler.sendMessage()`. A `Handler` object created in the same thread is then bound to this message queue and its `Handler.handleMessage()` callback will process the message and later execute `sendTextMessage()`.

The sole entry point to the graph is the user-interactive callback `onClick()`. However, prior work [23] on the identification of program entry points does not consider asynchronous calls and recognizes all three callbacks in the program as individual entry points. This confuses the determination of whether the user is aware that an API call has been made in response to a user-interactive callback. To address this limitation, we propose Algorithm 1 to remove any potential entry points that are actually part of an asynchronous call chain with only a single entry point.

Algorithm 1 accepts three inputs and provides one output. The first input is M_{entry}, which is a set of possible entry points. The second is CM_{async}, which is a set of ($BaseClass$, $RunMethod$) pairs. $BaseClass$ represents a top-level asynchronous base class (e.g., `Runnable`) in the Android framework and $RunMethod$ is the asynchronous call target (e.g., `Runnable.run()`) declared in this class. The third input is RS_{async}, which maps $RunMethod$ to $StartMethod$. $RunMethod$ and $StartMethod$ are the callee and caller in an asynchronous call (e.g., `Runnable.run()` and `Runnable.start()`). The output is a reduced M_{entry} set.

We compute the M_{entry} input by applying the algorithm proposed by Lu et al. [23], which discovers all reachable callback methods defined by the app that are intended to be called only by the Android framework. To further consider the logical order between `Intent` senders and receivers, we leverage Epicc [25] to resolve the inter-component communications and then remove the `Intent` receivers from M_{entry}.

Through examination of the Android framework code, we generate a list of 3-tuples consisting of $BaseClass$, $RunMethod$ and $StartMethod$. For example, we capture the Android-specific calling convention of `AsyncTask` with `AsyncTask.onPreExecute()` being triggered by `AsyncTask.execute()`. When a new asynchronous call is introduced into the framework code, this list is updated to include the change. Table 1 presents our current model for the calling convention of top-level base asynchronous classes in Android framework.

Table 1: Calling Convention of Asynchronous Calls

Top-level Class	Run Method	Start Method
`Runnable`	`run()`	`start()`
`AsyncTask`	`onPreExecute()`	`execute()`
`AsyncTask`	`doInBackground()`	`onPreExecute()`
`AsyncTask`	`onPostExecute()`	`doInBackground()`
`Message`	`handleMessage()`	`sendMessage()`

```
public class AsyncTask{
  public AsyncTask execute(Params... params){
    executeStub(params);
  }
  public AsyncTask executeStub(Params...params){
    onPreExecute();
    Result result = doInBackground(params);
    onPostExecuteStub(result);
  }
  public void onPostExecuteStub(Result result){
    onPostExecute(result);
  }
}
```

Figure 5: Stub code for dataflow of `AsyncTask.execute`

Given these inputs, our algorithm iterates over M_{entry}. For every method m_{entry} in this set, it finds the class c that declares this method and the top-level base class $base$ that c inherits from. Then, it searches the pair of $base$ and m_{entry} in the CM_{async} set. If a match is found, the method m_{entry} is a "callee" by convention. The algorithm thus looks up m_{entry} in the map SR_{async} to find the corresponding "caller" m_{start}. Each call to m_{start} is further examined and a points-to analysis is performed on the "this" reference making the call. If class c of method m_{entry} belongs to the points-to set, we can ensure the calling relationship between the caller m_{start} and the callee m_{entry} and remove the callee from the entry point set.

To indicate the data dependency between these two methods, we introduce a stub which connects the parameters of the asynchronous call to the corresponding parameters of its callee. Figure 5 depicts the example stub code for `AsyncTask`, where the parameter of `execute()` is first passed to `doInBackground()` through the stub `executeStub()`, and then the return from this asynchronous execution is further transferred to `onPostExecute()` via `onPostExecuteStub()`.

Once the algorithm has reduced the number of entry point methods in M_{entry}, we explore all code reachable from those entry points, including both synchronous and asynchronous calls. We further determine the user interactivity of an entry point by examining its top-level base class. If the entry point callback overrides a counterpart declared in one of the three top-level UI-related interfaces (i.e., `android.graphics.drawable.Drawable.Callback`, `android.view.accessibility.AccessibilityEventSource`, and `android.view.KeyEvent.Callback`), we then consider the derived entry point method as a user interface.

Constant Analysis.

We conduct constant analysis for any critical parameters of security sensitive API calls. These calls may expose security-related behaviors depending upon the values of their constant parameters. For example, `Runtime.exec()` can directly execute shell commands, and file or database operations can interact with distinctive targets by providing the proper URIs as input parameters.

To understand these semantic-level differences, we perform backward dataflow analysis on selected parameters and collect all possible constant values on the backward trace. We generate a constant set for each critical API argument and mark the parameter as "Constant" in the corresponding node on the `WC-ADG`. While a more complete string constant analysis is also possible, the computation of regular expressions is fairly expensive for static analysis. The

substring set currently generated effectively reflects the semantics of a critical API call and is sufficient for further feature extraction.

API Dependency Construction.

We perform global dataflow analysis to discover data dependencies between API nodes and build the edges on WC−ADG. However, it is very expensive to analyze every single API call made by an app. To address computational efficiency and our interests on security analysis, we choose to analyze only the security-related API calls. Permissions are strong indicators of security sensitivity in Android systems, so we leverage the API-permission mapping from PScout [8] to focus on permission-related API calls.

Our static dataflow analysis is similar to the "split"-based approach used by CHEX [23]. Each program split includes all code reachable from a single entry point. Dataflow analysis is performed on each split, and then cross-split dataflows are examined. The difference between our analysis and that of CHEX lies in the fact that we compute larger splits due to the consideration of asynchronous calling conventions.

We make a special consideration for reflective calls within our analysis. In Android programs, reflection is realized by calling the method `java.lang.reflect.Method.invoke()`. The "this" reference of this API call is a `Method` object, which is usually obtained by invoking either `getMethod()` or `getDeclaredMethod()` from `java.lang.Class`. The class is often acquired in a reflective manner too, through `Class.forName()`. This API call resolves a string input and retrieves the associated `Class` object.

We consider any reflective `invoke()` call as a sink and conduct backward dataflow analysis to find any prior data dependencies. If such an analysis reaches string constants, we are able to statically resolve the class and method information. Otherwise, the reflective call is not statically resolvable. However, statically unresolvable behavior is still represented within the WC−ADG as nodes which contain no constant parameters. Instead, this reflective call may have several preceding APIs, from a dataflow perspective, which are the sources of its metadata.

4. ANDROID MALWARE CLASSIFICATION

We generate WC−ADGs for both benign and malicious apps. Each unique graph is associated with a feature that we use to classify Android malware and benign applications.

4.1 Graph Matching Score

To quantify the similarity of two graphs, we first compute a graph edit distance. To our knowledge, all existing graph edit distance algorithms treat node and edge uniformly. However, in our case, our graph edit distance calculation must take into account the different weights of different API nodes. At present, we do not consider assigning different weights on edges because this would lead to prohibitively high complexity in graph matching. Moreover, to emphasize the differences between two nodes in different labels, we do not seek to relabel them. Instead, we delete the old node and insert the new one subsequently. This is because node "relabeling" cost, in our context, is not the string edit distance between the API labels of two nodes. It is the cost of deleting the old node plus that of adding the new node.

Definition 2. The *Weighted Graph Edit Distance* (WGED) of two Weighted Contextual API Dependency Graphs *G* and *G'*, with a uniform weight function β, is the minimum cost to transform *G* to *G'*:

$$wged(G, G', \beta) = min(\sum_{v_I \in \{V' - V\}} \beta(v_I) + \sum_{v_D \in \{V - V'\}} \beta(v_D) + |E_I| + |E_D|) \quad (1)$$

, where V and V' are respectively the vertices of two graphs, v_I and v_D are individual vertices inserted to and deleted from *G*, while E_I and E_D are the edges added to and removed from *G*.

WGED presents the absolute difference between two graphs. This implies that *wged(G,G')* is roughly proportional to the sum of graph sizes and therefore two larger graphs are likely to be more distant to one another. To eliminate this bias, we normalize the resulting distance and further define *Weighted Graph Similarity* based upon it.

Definition 3. The *Weighted Graph Similarity* of two Weighted Contextual API Dependency Graphs *G* and *G'*, with a weight function β, is,

$$wgs(G, G', \beta) = 1 - \frac{wged(G, G', \beta)}{wged(G, \emptyset, \beta) + wged(\emptyset, G', \beta)} \quad (2)$$

, where $\emptyset$ is an empty graph. $wged(G, \emptyset, \beta) + wged(\emptyset, G', \beta)$ then equates the maximum possible edit cost to transform *G* to *G'*.

4.2 Weight Assignment

Instead of manually specifying the weights on different APIs (in combination of their attributes), we wish to see a near-optimal weight assignment.

Selection of Critical API Labels.

Given a large number of API labels (unique combinations of API names and attributes), it is unrealistic to automatically assign weights for every one of them. Our goal is malware classification, so we concentrate on assigning weights to labels for the security-sensitive APIs and critical combinations of their attributes. To this end, we perform *concept learning* to discover critical API labels. Given a positive example set (PES) containing malware graphs and a negative example set (NES) containing benign graphs, we seek a critical API label (CA) based on two requirements: 1) frequency(CA,PES) > frequency(CA,NES) and 2) frequency(CA,NES) is less than the median frequency of all critical API labels in NES. The first requirement guarantees that a critical API label is more sensitive to a malware sample than a benign one, while the second requirement ensures the infrequent presence of such an API label in the benign set. Consequently, we have selected 108 critical API labels. Our goal becomes the assignment of appropriate weights to these 108 labels while assigning a default weight of 1 to all remaining API labels.

Weight Assignment.

Intuitively, if two graphs come from the same malware family and share one or more critical API labels, we must maximize the similarity between the two. We call such a pair of graphs a "homogeneous pair". Conversely, if one graph is malicious and the other is benign, even if they share one or more critical API labels, we must minimize the similarity between the two. We call such a pair of graphs a "heterogeneous pair". Therefore, we cast the problem of weight assignment to be an optimization problem.

Definition 4. The *Weight Assignment* is an optimization problem to maximize the result of an objective function for a given set of graph pairs {<G,G'>}:

$$\max\ f(\{< G, G' >\}, \beta) = \sum_{\substack{<G,G'> \text{ is a} \\ \text{homogeneous pair}}} wgs(G, G', \beta) - \sum_{\substack{<G,G'> \text{ is a} \\ \text{heterogeneous pair}}} wgs(G, G', \beta)$$

$$s.t.$$
$$1 \le \beta(v) \le \theta, if\ v\ is\ a\ critical\ node;$$
$$\beta(v) = 1, otherwise. \quad (3)$$

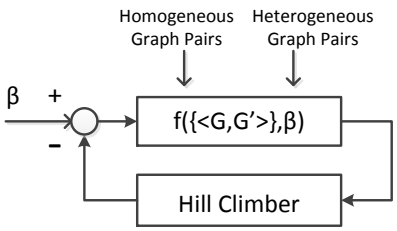


Figure 6: A Feedback Loop to Solve the Optimization Problem

, where β is the weight function that requires optimization; θ is the upper bound of a weight. Empirically, we set θ to be 20.

To achieve the optimization of Equation 3, we use the *Hill Climbing* algorithm [30] to implement a feedback loop that gradually improves the quality of weight assignment. Figure 6 presents such a system, which takes two sets of graph pairs and an initial weight function β as inputs. β is a discrete function which is represented as a weight vector. At each iteration, Hill Climbing adjusts a single element in the weight vector and determines whether the change improves the value of objective function $f(\{<G,G'>\},\beta)$. Any change that improves $f(\{<G,G'>\},\beta)$ is accepted, and the process continues until no change can be found that further improves the value.

4.3 Implementation

To compute the weighted graph similarity, we use a bipartite graph matching tool [29]. We cannot directly use this graph matching tool because it does not support assigning different weights on different nodes in a graph. To work around this limitation, we enhanced the bipartite algorithm to support weights on individual nodes.

4.4 Graph Database Query

Given an app, we match its `WC-ADG`s against all existing graphs in the database. The number of graphs in the database can be fairly large, so the design of the graph query must be scalable.

Intuitively, we could insert graphs into individual buckets, with each bucket labeled according to the presence of critical APIs. Instead of comparing a new graph against every existing graph in the database, we limit the comparison to only the graphs within a particular bucket that possesses graphs containing a corresponding set of critical APIs. Critical APIs generally have higher weights than regular APIs, so graphs in other buckets will not be very similar to the input graph and are safe to ignore. However, API-based bucket indexing may be overly strict because APIs from the same package usually share similar functionality. For instance, both `getDeviceId()` and `getSubscriberId()` are located in `TelephonyManager` package, and both retrieve identity-related information. Therefore, we instead index buckets based on the package names of critical APIs.

More specifically, to build a graph database, we must first build an API package bitvector for all existing graphs within the database. Such a bitvector has n elements, each of which indicates the presence of a particular Android API package. For example, a graph that calls `sendTextMessage()` and `getDeviceId()` will set the corresponding bits for the `android.telephony.SmsManager` and `android.telephony.TelephonyManager` packages. Graphs that share the same bitvector (i.e., the same API package combination) are then placed into the same bucket. When querying a new graph against the database, we encode its API package combination into a bitvector and compare that bitvector against each database index. Notice that, to ensure the scalability, we implement the bucket-based indexing with a hash map where the key is the API package bitvector and the value is a corresponding graph set.

Empirically, we found this one-level indexing efficient enough for our problem. If the database grows much larger, we can transition to a hierarchical database structure, such as vantage point tree [20], under each bucket.

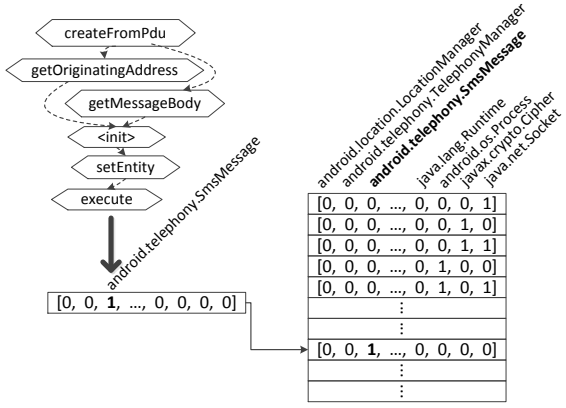


Figure 7: Bucket-based Indexing of Graph Database

Figure 7 demonstrates the bucket query for the `WC-ADG` of Zitmo shown in Figure 3. This graph contains six API calls, three of which belong to a critical package: `android.telephony.SmsManager`. The generated bitvector for the Zitmo graph indicates the presence of this API package, and an exact match for the bitvector is performed against the bucket index. Notice that the presence of a single critical package is different from that of a combination of multiple critical packages. Thus, the result bucket used by this search contains graphs that include `android.telephony.SmsManager` as the only critical package in use. `SmsManager`, being a critical package, helps capture the SMS retrieval behavior and narrow down the search range. Because HTTP-related API packages are not considered as critical, such an exact match over index will not exclude Zitmo variants that use other I/O packages, such as SMS.

4.5 Malware Classification

Anomaly Detection.

We have implemented a detector to conduct anomaly detection. Given an app, the detector provides a binary result that indicates whether the app is abnormal or not. To achieve this goal, we build a graph database for benign apps. The detector then attempts to match the `WC-ADG`s of the given app against the ones in database. If a sufficiently similar one for any of the behavior graphs is not found, an anomaly is reported by the detector. We have set the similarity threshold to be 70% per our empirical studies.

Signature Detection.

We next use a classifier to perform signature detection. Our signature detector is a multi-label classifier designed to identify the malware families of unknown malware instances.

To enable classification, we first build a malware graph database. To this end, we conduct static analysis on the malware samples from the Android Malware Genome Project [1,36] to extract `WC-ADG`s. In order to consider only the unique graphs, we remove any graphs that have a high level of similarity to existing ones. With experimental study, we consider a high similarity to be greater than 80%. Further, to guarantee the distinctiveness of malware behaviors, we compare these malware graphs against our benign graph set and remove the common ones.

Next, given an app, we generate its feature vector for classification purpose. In such a vector, each element is associated with a graph in our database. And, in turn, all the existing graphs are projected to a feature vector. In other words, there exists a one-to-one correspondence between the elements in a feature vector and

	G1	G2	G3	G4	G5	G6	G7	G8	...	G861	G862
ADRD	0	0	0	0	0	0.8	0.9	0	...	0	0
DroidDream	0.9	0	0	0	0.8	0.7	0.7	0	...	0	0
DroidKungFu	0	0.7	0	0	0.6	0	0.6	0	...	0	0.9

Figure 8: An Example of Feature Vectors

the existing graphs in the database. To construct the feature vector of the given app, we produce its WC−ADGs and then query the graph database for all the generated graphs. For each query, a best matching graph is found. The similarity score is then put into the feature vector at the position corresponding to this best matching graph. Specifically, the feature vector of a known malware sample is attached with its family label so that the classifier can understand the discrepancy between different malware families.

Figure 8 gives an example of feature vectors. In our malware graph database of 862 graphs, a feature vector of 862 elements is constructed for each app. The two behavior graphs of ADRD are most similar to graph G6 and G7, respectively, from the database. The corresponding elements of the feature vector are set to the similarity scores of theose features. The rest of the elements remain set to zero.

Once we have produced the feature vectors for the training samples, we can next use them to train a classifier. We select Naïve Bayes algorithm for the classification. In fact, we can choose different algorithms for the same purpose. However, since our graph-based features are fairly strong, even Naïve Bayes can produce satisfying results. Naïve Bayes also has several advantages: it requires only a small amount of training data; parameter adjustment is straightforward and simple; and runtime performance is favorable.

5. EVALUATION

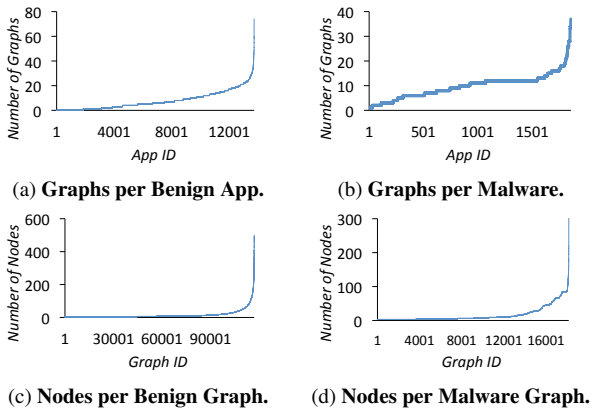
5.1 Dataset & Experiment Setup

We collected 2200 malware samples from the Android Malware Genome Project [1] and McAfee, Inc, a leading antivirus company. To build a benign dataset, we received a number of benign samples from McAfee, and we downloaded a variety of popular apps having a high ranking from Google Play. To further sanitize this benign dataset, we sent these apps to the VirusTotal service for inspection. The final benign dataset consisted of 13500 samples. We performed the behavior graph generation, graph database creation, graph similarity query and feature vector extraction using this dataset. We conducted the experiment on a test machine equipped with Intel(R) Xeon(R) E5-2650 CPU (20M Cache, 2GHz) and 128GB of physical memory. The operating system is Ubuntu 12.04.3 (64bit).

5.2 Summary of Graph Generation

Figure 9 summarizes the characteristics of the behavior graphs generated from both benign and malicious apps. Among them, Figure 9a and Figure 9b illustrate the number of graphs generated from benign and malicious apps. On average, 7.8 graphs are computed from each benign app, while 9.8 graphs are generated from each malware instance. Most apps focus on limited functionalities and do not produce a large number of behavior graphs. In 92% of benign samples and 98% of malicious ones, no more than 20 graphs are produced from an individual app.

Figure 9c and Figure 9d present the number of nodes of benign and malicious behavior graphs. A benign graph, on average, has 15 nodes, while a malicious graph carries 16.4. Again, most of the activities are not intensive, so the majority of these graphs have a small number of nodes. Statistics show that 94% of the benign graphs and 91% of the malicious ones carry less than 50 nodes.



(a) **Graphs per Benign App.** (b) **Graphs per Malware.**

(c) **Nodes per Benign Graph.** (d) **Nodes per Malware Graph.**

Figure 9: **Graph Generation Summary.**

These facts serve as the basic requirements for the scalability of our approach, since the runtime performance of graph matching and query is largely dependent upon the number of nodes and graphs, respectively.

5.3 Classification Results

Signature Detection.

We use a multi-label classification to identify the malware family of the unrecognized malicious samples. Therefore, we expect to only include those malware behavior graphs, that are well labeled with family information, into the database. To this end, we rely on the malware samples from the Android Malware Genome Project and use them to construct the malware graph database. Consequently, we built such a database of 862 unique behavior graphs, with each graph labeled with a specific malware family.

We then selected 1050 malware samples from the Android Malware Genome Project and used them as a training set. Next, we would like to collect testing samples from the rest of our collection. However, a majority of malware samples from McAfee are not strongly labeled. Over 90% of the samples are coarsely labeled as "Trojan" or "Downloader", but in fact belong to a specific malware family (e.g., DroidDream). Moreover, even VirusTotal cannot provide reliable malware family information for a given sample because the antivirus products used by VirusTotal seldom reach a consensus. This fact tells us two things: 1) It is a non-trivial task to collect evident samples as the ground truth in the context of multi-label classification; 2) multi-label malware detection or classification is, in general, a challenging real-world problem.

Despite the difficulty, we obtained 193 samples, each of which is detected as the same malware by major AVs. We then used those samples as testing data. The experiment result shows that our classifier can correctly label 93% of these malware instances.

Among the successfully labeled malware samples are two types of Zitmo variants. One uses HTTP for communication, and the other uses SMS. While the former one is present in our malware database, the latter one was not. Nevertheless, our signature detector is still able to capture this variant. This indicates that our similarity metrics effectively tolerate variations in behavior graphs.

We further examined the 7% of the samples that were mislabeled. It turns out that the mislabeled cases can be roughly put into two categories. First, DroidDream samples are labeled as DroidKungFu. DroidDream and DroidKungFu share multiple malicious behaviors such as gathering privacy-related information and hidden network I/O. Consequently, there exists a significant overlap between their WC−ADGs. Second, Zitmo, Zsone and YZHC instances are labeled as one another. These three families are SMS Tro-

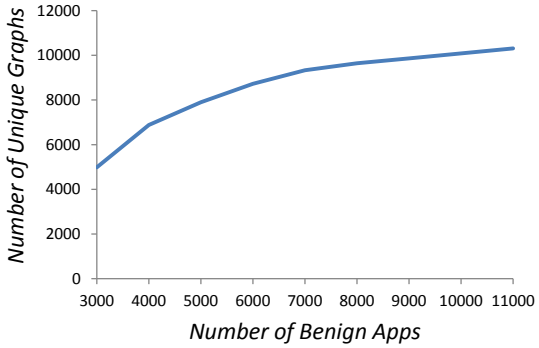


Figure 10: Convergence of Unique Graphs in Benign Apps

jans. Though their behaviors are slightly different from each other, they all exploit `sendTextMessage()` to deliver the user's information to an attacker-specified phone number. Despite the mislabeled cases, we still manage to successfully label 93% of the malware samples with a Naïve Bayes classifier. Applying a more advanced classification algorithm will further improve the accuracy.

Anomaly Detection.

Since we wish to perform anomaly detection using our benign graph database, the coverage of this database is essential. In theory, the more benign apps that the database collects, the more benign behaviors it covers. However, in practice, it is extremely difficult to retrieve benign apps exhaustively. Luckily, different benign apps may share the same behaviors. Therefore, we can focus on unique behaviors (rather than unique apps). Moreover, with more and more apps being fed into the benign database, the database size grows slower and slower. Figure 10 depicts our discovery. When the number of apps increases from 3000 to 4000, there is a sharp increase (2087) of unique graphs. However, when the number of apps grows from 10000 to 11000, only 220 new, unique graphs are generated, and the curve begins to flatten.

We built a database of 10420 unique graphs from 11400 benign apps. Then, we tested 2200 malware samples against the benign classifier. The false negative rate was 2%, which indicates that 42 malware instances were not detected. However, we noted that most of the missed samples are exploits or Downloaders. In these cases, their bytecode programs do not bear significant API-level behaviors, and therefore generated `WC-ADGs` do not necessarily look abnormal when compared to benign ones. At this point, we have only considered the presence of constant parameters in an API call. We did not further differentiate API behaviors based upon constant values. Therefore, we cannot distinguish the behaviors of `Runtime.exec()` calls or network I/O APIs with varied string inputs. Nevertheless, if we create a custom filter for these string constants, we will then be able to identify these malware samples and the false negative rate will drop to 0.

Next, we used the remaining 2100 benign apps as test samples to evaluate the false positive rate of our anomaly detector. The result shows that 5.15% of clean apps are mistakenly recognized as suspicious ones during anomaly detection. This means, if our anomaly detector is applied to Google Play, among the approximately 1200 new apps per day [4], around 60 apps will be mislabeled as containing anomalies and be bounced back to the developers. We believe that this is an acceptable ratio for vetting purpose. Moreover, since we do not reject the suspicious apps immediately, but rather ask the developers for justifications instead, we can further eliminate these false positives during this interactive process. In addition, as we add more benign samples into the dataset, the false positive rate will further decrease.

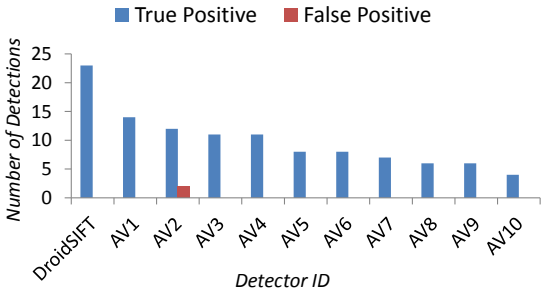


Figure 11: Detection Ratio for Obfuscated Malware

Further, we would like to evaluate our anomaly detector with malicious samples from new malware families. To this end, we retrieve a new piece of malware called Android.HeHe, which was first found and reported in January 2014 [12]. Android.HeHe exercises a variety of malicious functionalities such as SMS interception, information stealing and command-and-control. This new malware family does not appear in the samples from the Android Malware Genome Project, which were collected from August 2010 to October 2011, and therefore cannot be labeled via signature detection. DroidSIFT generates 49 `WC-ADGs` for this sample and, once these graphs are presented to the anomaly detector, a warning is raised indicating the abnormal behaviors expressed by these graphs.

Detection of Transformation Attacks.

We collected 23 DroidDream samples, which are all intentionally obfuscated using a transformation technique [28], and 2 benign apps that are deliberately disguised as malware instances by applying the same technique. We ran these samples through our anomaly detection engine and then sent the detected abnormal ones through the signature detector. The result shows that while 23 true malware instances are flagged as abnormal ones in anomaly detection, the 2 clean ones also correctly pass the detection without raising any warnings. We then compared our signature detection results with antivirus products. To obtain detection results of antivirus software, we sent these samples to VirusTotal and selected 10 anti-virus (AV) products (i.e., AegisLab, F-Prot, ESET-NOD32, DrWeb, AntiVir, CAT-QuickHeal, Sophos, F-Secure, Avast, and Ad-Aware) that bear the *highest* detection rates. Notice that we consider the detection to be successful only if the AV can correctly flag a piece of malware as DroidDream or its variant. In fact, to our observation, many AV products can provide partial detection results based upon the native exploit code included in the app package or common network I/O behaviors. As a result, they usually recognize these DroidDream samples as "exploits" or "Downloaders" while missing many other important malicious behaviors. Figure 11 presents the detection ratios of "DroidDream" across different detectors. While none of the antivirus products can achieve a detection rate higher than 61%, DroidSIFT can successfully flag all the obfuscated samples as DroidDream instances. In addition, we also notice that AV2 produces a relatively high detection ratio (52.17%), but it also mistakenly flags those two clean samples as malicious apps. Since the disguising technique simply renames the benign app package to the one commonly used by DroidDream (and thus confuses this AV detector), such false positives again explain that external symptoms are not robust and reliable features for malware detection.

5.4 Runtime Performance

Figure 12 illustrates the runtime performance of DroidSIFT. Specifically, it demonstrates the cumulative time consumption of graph

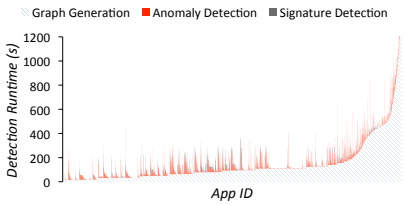


Figure 12: Detection Runtime (s) for 3000 Benign and Malicious Apps

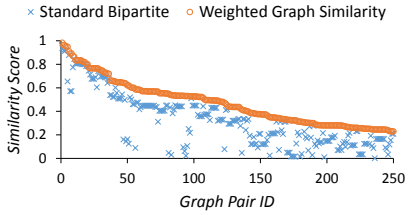


Figure 13: Similarity between Malicious Graph Pairs.

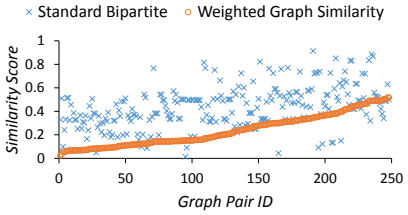


Figure 14: Similarity between Benign and Malicious Graphs.

generation, anomaly detection, and signature detection for 3000 apps.

The average detection runtime of 3000 apps is 175.8 seconds, while the detection for a majority (86%) of apps is completed within 5 minutes. Further, most of the apps (96%) are processed within 10 minutes. The time cost of graph generation dominates the overall runtime, taking up at least 50% of total runtime for 83.5% of the apps. On the other hand, the signature and anomaly detectors are usually (i.e., in 98% of the cases) able to finish running in 3 minutes and 1 minute, respectively.

5.5 Effectiveness of Weight Generation and Weighted Graph Matching

Finally, we evaluated the effectiveness of the generated weights and weighted graph matching.

Our weight generation automatically assign weights to the critical API labels, based on a training set of *homogeneous graph pairs* and *heterogeneous graph pairs*. Consequently, `killProcess()`, `getMemoryInfo()` and `sendTextMessage()` with a constant phone number, for example, are assigned with fairly high weights.

Then, given a graph pair sharing the same critical API labels, other than the pairs used for training, we want to compare their weighted graph similarity with the similarity score calculated by the standard bipartite algorithm. To this end, we randomly picked 250 homogeneous pairs and 250 heterogeneous pairs.

The results of these comparisons, presented in Figure 13 and Figure 14, conform to our expectation. Figure 13 shows that for every homogeneous pair, the similarity score generated by weighted graph matching is almost always higher than the corresponding one computed using standard bipartite algorithm. In addition, the bipartite algorithm sometimes produces an extremely low similarity (i.e., near zero) between two malicious graphs of the same family, while weighted graph matching manages to improve the similarity score significantly for these cases.

Similarly, Figure 14 reveals that between a heterogeneous pair, the weighted similarity score is usually lower than the one from bipartite computation. Again, the bipartite algorithm occasionally considers a benign graph considerably similar to a malicious one, provided that they share the same API nodes. Such results can confuse a training system and the latter one thus fails to tell the differences between malicious and benign behaviors. On the other hand, weighted graph matching can effectively distinguish a malicious graph from a benign one, even if they both have the same critical API nodes.

We further attempted to implement the standard bipartite algorithm and apply it to our detectors. We then compared the consequent detection results with those of the detectors with weighted graph matching enabled. The results show that weighted graph matching significantly outperforms the bipartite one. While the signature detector using the former one correctly labels 93% of malware samples, the detector with the latter one is able to only label 73% of them. On the other hand, anomaly detection with the bipartite algorithm incurs a false negative rate of 10%, which is 5 times greater than that introduced by the same detector using weighted matching.

The result indicates that our algorithm is more sensitive to critical API-level semantics than the standard bipartite graph matching, and thus can produce more reasonable similarity scores for the feature extraction.

6. DISCUSSION

In this section, we discuss the limitation and potential evasions of our proposed technique.

6.1 Native Code & HTML5-based Apps

We perform static analysis on Dalvik bytecode to generate the behavior graphs. In general, bytecode-level static program analysis cannot handle native code or HTML5-based applications. This is because neither the ARM binary running on the underlying Linux nor the JavaScript code executed in WebView are visible from a bytecode perspective. Therefore, an alternative mechanism is necessary to defeat malware hidden from the Dalvik bytecode.

6.2 Evasion

Learning-based detection is subject to poisoning attacks. To confuse a training system, an adversary can poison the benign dataset by introducing clean apps bearing malicious features. For example, she can inject harmless code intensively making sensitive API calls that are rarely observed in clean apps. Once such samples are accepted by the benign dataset, these APIs are therefore no longer the distinctive features to detect related malware instances.

However, our detectors are slightly different from prior works. First of all, the features are associated with behavior graphs, rather than individual APIs. Therefore, it is much harder for an attacker to engineer confusing samples at the behavioral-level. Second, our anomaly detection serves as a sanitizer for new benign samples. Any abnormal behavior will be detected, and the developer is requested to provide justifications for the anomalies.

On the other hand, in theory, it is possible for adversaries to launch mimicry attacks and embed malicious code into seemingly benign graphs to evade our detection mechanism. This, by itself, is an interesting research topic and deserves serious consideration. Nevertheless, we note that it is non-trivial to evade detections based upon high-level program semantics, and automating such evasion attacks does not appear to be an easy task. In contrast, existing low-level transformation attacks can be easily automated to generate many malware variants to bypass the AV scanners. DroidSIFT certainly defeats such evasion attempts.

7. RELATED WORK

In this section, we discuss the previous work related to Android malware classification, Android malware detection, and graph-based program analysis.

Android Malware Classification.

Many prior efforts have been made to automatically classify Android malware via machine learning. H. Peng et al. [27] proposed a permission-based classification approach and introduced probabilistic generative models for ranking risks for Android apps. Juxtapp [19] performed feature hashing on the opcode sequence to detect malicious code reuse. DroidAPIMiner [5] extracted Android malware features at the API level and provided light-weight classifiers to defend against malware installations. DREBIN [6] took a hybrid approach and considered both Android permissions and sensitive APIs as malware features. To this end, it performed broad static analysis to extract feature sets from both manifest files and bytecode programs. It further embedded all feature sets into a joint vector space. As a result, the features contributing to malware detection can be analyzed geometrically and used to explain the detection results. Despite the effectiveness and computational efficiency, these machine learning based approaches extract features from solely external symptoms and do not seek an accurate and complete interpretation of app behaviors. In contrast, we produce weighted contextual API dependency graphs as more robust features to reflect essential behaviors.

Android Malware Detection & Program Analysis.

Previous studies were focused on large-scale and light-weight detection of malicious or dangerous Android apps. DroidRanger [37] proposed permission-based footprinting and heuristics-based schemes to detect new samples of known malware families and identify certain behaviors of unknown malicious families, respectively. RiskRanker [18] developed an automated system to uncover dangerous app behaviors, such as root exploits, and assess potential security risks. Kirin [15] proposed a security service to certify apps based upon predefined security specifications. WHYPER [26] leveraged Natural Language Processing and automated risk assessment of mobile apps by revealing discrepancies between application descriptions and their true functionalities. Efforts were also made to pursue in-depth analysis of malware and application behaviors. TaintDroid [13], DroidScope [32] and VetDroid [35] conducted dynamic taint analysis to detect suspicious behaviors during runtime. Ded [14], CHEX [23], AppSealer [33], Capper [34], PEG [10], and FlowDroid [7] exercised static dataflow analysis to identify dangerous code in Android apps. The effectiveness of these approaches depends upon the quality of human crafted detection patterns specific to certain dangerous or vulnerable behaviors.

Graph-based Code Analysis.

Graph-based code analysis has been well-studied for traditional client-server programs. Hu et al. [20] proposed two-level malware indexing to address the scalability of querying malware function-call graphs in databases. Kolbitsch et al. [21] performed dynamic analysis to extract program slices responsible for malicious information flow between system calls, and then conducted model checking by matching the generated slices against unknown programs. Fredrikson et al. [16] presented an automated technique for extracting optimally discriminative specifications which uniquely identify a class of program, such as a malware family. Yamaguchi et al. [31] introduced a novel representation of source code, called a "code property graph", that merges concepts of classic program analysis (abstract syntax trees, control flow graphs, and program dependence graphs) into a joint data structure. Such a graph representation enables elegant modeling of common vulnerabilities. HI-CFG [9] inferred a hybrid information and control-flow graph from a binary instruction trace, and based on the graph, enabled attack polymorphism. Compared to these approaches, the novelty of our work lies in the fact that our dependency graph generation needs to cope with Android programming paradigms. Gascon et al. [17] also extended graph-based malware analysis to the Android environment. However, their malware detection was based upon the structural similarity of callgraphs, while DroidSIFT relies upon more robust, high-level (API) program semantics.

8. CONCLUSION

In this paper, we propose a novel, semantic-based approach that classifies Android malware via dependency graphs. To battle transformation attacks, we extract a *weighted contextual API dependency graph* as program semantics to construct feature sets. To fight against malware variants and zero-day malware, we introduce graph similarity metrics to uncover homogeneous application behaviors while tolerating minor implementation differences. We implement a prototype system, DroidSIFT, in 23 thousand lines of Java code. We evaluate our system using 2200 malware samples and 13500 benign samples. Experiments show that our signature detection can correctly label 93% malware instances; our anomaly detector is capable of detecting zero-day malware with relatively low false negative rate (2%) and false positive rate (5.15%).

9. ACKNOWLEDGMENT

We would like to thank anonymous reviewers for their comments. This research was supported in part by NSF Grant #1018217, NSF Grant #1054605 and McAfee Inc. Any opinions, findings, and conclusions made in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

10. REFERENCES

[1] Android Malware Genome Project. `http://www.malgenomeproject.org/`.

[2] Soot: a Java Optimization Framework. `http://www.sable.mcgill.ca/soot/`.

[3] McAfee Labs Threats report Fourth Quarter 2013. `http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q4-2013.pdf`, 2013.

[4] Number of Android Applications. `http://www.appbrain.com/stats/number-of-android-apps`, 2014.

[5] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In *Proceedings of the 9th International Conference on Security and Privacy in Communication Networks (SecureComm'13)*, September 2013.

[6] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck. Drebin: Efficient and Explainable Detection of Android Malware in Your Pocket. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)*, February 2014.

[7] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, June 2014.

[8] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*, October 2012.

[9] D. Caselden, A. Bazhanyuk, M. Payer, S. McCamant, and D. Song. HI-CFG: Construction by Binary Analysis, and Application to Attack Polymorphism. In *Proceedings of 18th*

European Symposium on Research in Computer Security (ESORICS'13), September 2013.

[10] K. Z. Chen, N. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. Magrino, E. X. Wu, M. Rinard, and D. Song. Contextual Policy Enforcement in Android Applications with Permission Event Graphs. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'13)*, February 2013.

[11] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-Aware Malware Detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland'05)*, May 2005.

[12] H. Dharmdasani. Android.HeHe: Malware Now Disconnects Phone Calls. `http://www.fireeye.com/blog/technical/2014/01/android-hehe-malware-now-disconnects-phone-calls.html`, 2014.

[13] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, October 2010.

[14] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th Usenix Security Symposium*, August 2011.

[15] W. Enck, M. Ongtang, and P. McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*, November 2009.

[16] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (Oakland'10)*, May 2010.

[17] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural Detection of Android Malware Using Embedded Call Graphs. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security (AISec'13)*, November 2013.

[18] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications and Services (MobiSys'12)*, June 2012.

[19] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'12)*, July 2012.

[20] X. Hu, T.-c. Chiueh, and K. G. Shin. Large-scale Malware Indexing Using Function-call Graphs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*, November 2009.

[21] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and Efficient Malware Detection at the End Host. In *Proceedings of the 18th Conference on USENIX Security Symposium*, August 2009.

[22] H. Lockheimer. Android and Security. `http://googlemobile.blogspot.com/2012/02/android-and-security.html`, 2012.

[23] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*, October 2012.

[24] J. Oberheide and C. Miller. Dissecting the Android Bouncer. SummerCon, 2012.

[25] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In *Proceedings of the 22nd USENIX Security Symposium*, August 2013.

[26] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *Proceedings of the 22nd USENIX Conference on Security*, August 2013.

[27] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using Probabilistic Generative Models for Ranking Risks of Android Apps. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*, October 2012.

[28] V. Rastogi, Y. Chen, and X. Jiang. DroidChameleon: Evaluating Android Anti-malware against Transformation Attacks. In *Proceedings of the 8th ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS'13)*, May 2013.

[29] K. Riesen, S. Emmenegger, and H. Bunke. A Novel Software Toolkit for Graph Edit Distance Computation. In *Proceedings of the 9th International Workshop on Graph Based Representations in Pattern Recognition*, May 2013.

[30] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. 2003.

[31] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland'14)*, May 2014.

[32] L.-K. Yan and H. Yin. DroidScope: Seamlessly Reconstructing OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21st USENIX Security Symposium*, August 2012.

[33] M. Zhang and H. Yin. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)*, San Diego, CA, February 2014.

[34] M. Zhang and H. Yin. Efficient, Context-aware Privacy Leakage Confinement for Android Applications Without Firmware Modding. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS'14)*, 2014.

[35] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)*, November 2013.

[36] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland'12)*, May 2012.

[37] 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 19th Annual Network and Distributed System Security Symposium (NDSS'12)*, February 2012.