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

Mu Zhang, Yue Duan, Heng Yin, Zhirou Zhao
Department of EECS, Syracuse University, Syracuse, NY, USA

ACM CCS'14

Presented by:
Chih-Wen Ou
ADLab
2014/10/21
Outline

- Introduction
- Overview
- Weighted Contextual API Dependency Graph
- Android Malware Classification
- Evaluation
- Discussion
- Related Work
- Conclusion
Introduction

• Signature-based detection
  – Look for specific patterns in the bytecode and API calls
  – Easily evaded by bytecode-level transformation attacks

• Machine learning-based detection
  – Extract features from an application's behavior
  – Associated with application syntax, rather than program semantics.

• Graph representation
  – Control-flow graph, Data dependency graph, permission event graph
  – Potentially evaded by polymorphic variants.
  – Cannot be used to battle zero-days
Introduction

- Semantics-based approach
  - Weighted contextual API dependency graph
  - Use program semantics to construct feature sets
  - Semantic-level behavior rather than program syntax
  - Graph similarity metrics
    - Homogeneous essential application behaviors while tolerating minor implementation differences.
    - Malware variants and zero-day malware.
Introduction

- DroidSIFT
  - Soot: dependency graph generation
  - Similarity query: a graph matching toolkit to compute graph edit distance
  - 2200 malware samples
  - 13500 benign samples
  - 93% detection rate
  - False negative rate 2%
  - False positive rate 5.15%
Overview

• Problem statement

• Architecture overview
  – Behavior graph generation
  – Scalable graph similarity query
  – Graph-based feature vector extraction
  – Anomaly & signature detection
Problem statement

- Official Google Play vetting system: Bouncer
  - Evaded by emulator detection
  - Evaded by bytecode-level transformation

- Design goals
  - Semantic-based detection
  - High scalability
    - Scalable to cope with millions of app samples
  - Variant resiliency
    - Similarity of app behaviors
    - Implementation variants toleration
    - Similarity scores

Figure 1: Deployment of DroidSIFT
Architecture Overview

Figure 2: Overview of DroidSIFT
Architecture overview

- Behavior graph generation
  - Consider graph similarity as the feature vector
  - Static program analysis
  - Bytecode programs => graph representations
    - Entry point discovery and Call graph analysis => API calling context
    - Forward and backward dataflow analysis => API dependencies
  - Weighted Contextual API Dependency Graph
Architecture overview

- **Scalable graph similarity query**
  - Able to query the graph database for the one graph most similar to a given graph

- **Graph-based feature vector extraction**
  - Attempting to find the best match for each of its graphs from the database
  - Each element of the vector associates with an existing graph in the database

- **Anomaly & signature detection**
  - A signature classifier and an anomaly detector
  - Feature vectors used for training the classifier for signature detection
  - The anomaly detector discovers zero-day malware.
  - The signature uncovers the type (family) of the malware
Weighted Contextual API Dependency Graph

• Key Behavioral Aspects
• Formal Definition
• A Real Example
• Graph Generation
Key Behavioral Aspects

- **API Dependency**
  - API calls indicate how an app interacts with the Android framework

- **Context**
  - An entry point of an API call is a program entry point that direct or indirectly triggers the call.
  - User interfaces and background callbacks.

- **Constant**
  - Conveying semantic information by revealing the values of critical parameters and uncovering fine-grained API semantics.
  - Analysis disclose the data dependencies of some certain security sensitive APIs whose benignness is dependent upon whether an input is constant.
Formal Definition

- **WC-ADG**
  - Two kinds of labeling
  - API operations
    - API prototype
    - Entry point
    - Constant parameter
  - Weights
    - Real numbers
  - Both all labeling on vertexes.

**Definition 1.** A Weighted Contextual API Dependency Graph is a directed graph \( G = (V, E, \alpha, \beta) \) over a set of API operations \( \Sigma \) and a weight space \( W \), where:
  - The set of vertices \( V \) corresponds to the contextual API operations in \( \Sigma \);
  - 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.
A Real Example

- Zitmo: A class of banking Trojan malware that steals a user's SMS messages to discover banking information.

Figure 3: WC-ADG of Zitmo
Graph Generation

- Entry point discovery
  - Entry point discovery is essential to revealing whether the user is **aware** that a certain API call has been made
  - The prior work CHEX does not consider this below:

> Figure 4: Callgraph for asynchronously sending an SMS message. “e” and “a” stand for “event handler” and “action” respectively.
Graph Generation

- Entry point discovery

<table>
<thead>
<tr>
<th>Top-level Class</th>
<th>Run Method</th>
<th>Start Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runnable</td>
<td>run()</td>
<td>start()</td>
</tr>
<tr>
<td>AsyncTask</td>
<td>onPreExecute()</td>
<td>execute()</td>
</tr>
<tr>
<td>AsyncTask</td>
<td>doInBackground()</td>
<td>onPreExecute()</td>
</tr>
<tr>
<td>AsyncTask</td>
<td>onPostExecute()</td>
<td>doInBackground()</td>
</tr>
<tr>
<td>Message</td>
<td>handleMessage()</td>
<td>sendMessage()</td>
</tr>
</tbody>
</table>

Table 1: Calling Convention of Asynchronous Calls

```java
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
Graph Generation

- Constant analysis
  - These calls may expose security-related behaviors depending upon the values of their constant parameters.
  - Performing backward dataflow analysis on selected parameters and collect all possible constant values on the backward trace
  - Generating a constant set.
Graph Generation

• API Dependency Construction
  - Performing global dataflow analysis to discover data dependencies between API nodes
  - Building edges on WC-ADG
  - Analyzing only security-related API calls
  - Split-based approach
    • Each program split includes all code reachable from a single entry point.
    • Performing dataflow analysis on each split
Android Malware Classification

• Graph Matching Score
• Weight Assignment
• Implementation
• Graph Database Query
• Malware Classification
Graph Matching Score

- Graph Edit Distance
  - The dissimilarity of two graphs
- Weighted Graph Edit Distance (WGED)

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

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

(1)
Graph Matching Score

• Weighted Graph Similarity
  – Normalization of WEDG

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

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

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'.
Weight Assignment

• Selection of Critical API Labels
  – API labels: unique combinations of APIs and attributes.
  – Only assign weights on security-sensitive APIs and critical combinations of their attributes
  – Performing concept learning
    • Concept learning: the search for and listing of attributes that can be used to distinguish exemplars from non exemplars of various categories
  – 108 critical API labels selected consequently
Weight Assignment

- **Weight Assignment**
  - Maximize the similarity: homogeneous pair
    - Same malware family
    - Sharing one or more critical API labels
  - Minimize the similarity: heterogeneous pair
    - One is benign and the other is malicious
    - Sharing one or more critical API labels
  - An optimization problem to maximize the result of an objective function for a given set of graph pairs \( \{<G,G'>\} \)
Weight Assignment

- Beta is the weight function that requires optimization
- Theta is the upper bound of a weight, set to 20.
- Use Hill Climbing Algorithm implement a feedback loop that gradually improves the quality of weight assignment

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

\[
\max f(\{G, G'\}, \beta) = \sum_{G, G'} wgs(G, G', \beta) - \sum_{G, G'} wgs(G, G', \beta)
\]

\( s.t. \)

\[
1 \leq \beta(v) \leq \theta, \text{if } v \text{ is a critical node;}
\]

\[
\beta(v) = 1, \text{otherwise.}
\]

(3)

Figure 6: A Feedback Loop to Solve the Optimization Problem
Malware Classification

• Anomaly detection
  – Given an app, providing a binary result that indicates whether the app is abnormal or not.
  – Matching the WC-ADGs of the given app against the ones in the database.
  – Report anomaly if no similar one found
Evaluation

- Data set & Experiment Setup
- Summary of Graph Generation
- Classification Results
- Runtime Performance
- Effectiveness of Weight Generation and Weighted Graph Matching
Data Set & Experiment Setup

- 2200 malware samples from AMGP and McAfee
- Benign dataset consisted of 13500 samples
  - Download from Google Play
  - Inspected by VirusTotal
- Ubuntu 64 bit
  - Intel Xeon CPU(20M cache, 2GHz)
  - 128GB physical memory
  - Behavior graph generation, graph database creation, graph similarity query and feature vector extraction.
Summary of Graph Generation

- 9a. 9b. Illustrating the number of graphs generated from sample apps
  - On average, 7.8 graphs from each benign app. 9.8 graphs from malicious one.
  - 92% of benign samples and 98% of malicious ones, no more than 20 graphs are produced from an individual app.

- 9c. 9d. Present the number of nodes of benign and malicious behavior graphs
  - On average, 15 nodes for benign graph. 16.4 nodes for malicious graph.
  - 94% of benign graphs and 91% of malicious graphs carry less than 50 nodes

Figure 9: Graph Generation Summary.
Classification Results

- Anomaly Detection
  - Different benign apps may share the same behaviors.
  - Unique graphs are generated and the curve begins to flatten.
  - 10420 unique graphs from 11400 benign apps.
  - False positive rate is 2% when using 2200 malware samples against the benign classifier.
  - False negative rate is 5.15% when using 2100 benign samples against anomaly detector.
Classification Results

Figure 10: Convergence of Unique Graphs in Benign Apps
Classification Results

• Detection of Transformation Attack
  – 23 DroidDream samples
  – 2 benign apps applying the same technique
  – All identified correctly.
  – Comparing to AVs

![Image showing detection ratio for obfuscated malware](Figure 11: Detection Ratio for Obfuscated Malware)
Runtime Performance

- Graph generation, anomaly detection signature detection for 3000 apps.
- The average for an app is 175.8 seconds.

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

• Native Code & HTML5-based Apps
  – Not handled

• Evasion
  – Learn-based detection is subject to poisoning attack
Related Work

• Android malware classification
  – Juxtapp:
    • feature hashing on opcode sequence
    • Detecting malicious code use
  – DroidAPIMiner:
    • API level malware feature
  – DREBIN
    • Permissions and sensitive APIs as features
Related Work

- Android malware detection & program analysis
  - DroidRanger, Kirin, WHYPER
  - TaintDroid, DroidScope, VetDroid
  - Ded, CHEX, AppSealer, Capper, PEG, FlowDroid

- Graph-based code analysis
  - Function call graph
  - CPG (my last presentation)
  - HI-CFG
  - Embedded call graph
Conclusion

- Weighted contextual API dependency graph
- DroidSIFT
- 13500 benign and 2200 malicious samples
- Signature detection correctly label 93% malware instance
- Anomaly detector's false negative rate is 2%, and the false positive rate is 5.15%