Juxtapp
A Scalable System for Detecting Code Reuse Among Android Applications

Steve Hanna
sch@eecs.berkeley.edu
Saung Li
shadowcwal@berkeley.edu

Ling Huang
ling.huang@intel.com
Charles Chen
charleschen@berkeley.edu

Edward Wu
edwardxwu@berkeley.edu
Dawn Song
dawnsong@cs.berkeley.edu

Steve Hanna – July 2012
Android Mobile Markets

- Android operating system serves as 48% of mobile market
- Android App Store, Amazon Application Store
  - Central repositories to obtain applications
  - Users have an expectation of safety
- Markets largely rely on a reactive approach to removing items
  - User policing and reporting
  - User ratings as indicators
  - Bouncer, the Android scanner, leaves much to be desired
Markets, not so safe

- Piracy
  - Games currently the largest target of piracy
  - Paid games made free by pirates
    - Repackaged, removing validation code

- Code Reuse & Bugs
  - Copy paste errors introduce security vulnerabilities

- Known Malware
  - As of August 2011, users are 2.5 times more likely to encounter malware than 6 months
  - Estimated that high as 1 million users exposed to malware

Problem Statement - A need for detection

- Reactive approach not enough

- Detecting application similarity as a first defense shows promise in mitigating threats to users
  - Significantly raises the bar for pirates and malware authors
  - Early detection of known bugs
    - Reject applications upon submission

- Provides a first chance detection scheme for programs with well known bugs
Applications and Goals

Architecture for systematic analysis of Android applications to detect:
- Code reuse and bug discovery
- Piracy
- Software Containment
- Repackaged known malware

Design Goals
- High performance
- Accurately and efficiently represent the applications under analysis
- Efficiently incrementally update application repository
- Extendable to many features
Methodology

- Feature Hashing
  - Collect static code features and represent them as a bitvector

- Agglomerative Hierarchical Clustering
  - Cluster based on a similarity threshold

- Similarity Containment
  - Determine what portions of A’s code exist in B.
Feature Hashing

- Reduces dimensionality of data being analyzed
- Feature representation is compact, efficient
  - Pairwise comparison efficient
  - Comes at the cost of potential collisions
- Given an efficient bit vector representation of size $n$ (prime) and a window size of $k$
  - Able to store presence or absence of a feature with 1 or 0
Unique Problem Domain

- Android applications written in DEX
  - Executes on Dalvik, Android’s virtual machine
- Application packages (.APK), archive of:
  - Application Code
  - Android Manifest (permissions and exports)
  - Resources (images, text, raw data)
  - Certificate Information
- Contains structured information about the application
  - DEX format fully describes the Java application
Given an APK

For each class we extract

Basic block with instructions

Each instruction’s op code

If a `const`, we record this constant data

Package, class and method name
A metric for Similarity

Jaccard Similarity
- Logical representation, not set representation
- Gives a percentage in common
- *Jaccard Dissimilarity* measured as $1 - J(A,B)$
  - Both have ranges $[0,1]$

Containment
- Defined as the percentage of features in Application B that exist within Application A

$$J(\hat{A}, \hat{B}) = \frac{|\hat{A} \cap \hat{B}|}{|\hat{A} \cup \hat{B}|}$$

$$C(\hat{B}|\hat{A}) = \frac{|\hat{A} \cap \hat{B}|}{|\hat{A}|}$$
Agglomerative Hierarchical Clustering

- Each application begins in its own cluster
  - Applications under analysis represented as a matrix of vectors

- Clusters are merged \textit{iff} the distance between any two applications in the cluster is less than some threshold ($T$)
  - For instance, 90% similar allows for additional code up to 10% of the body.

- Resulting clusters show applications with threshold ($T$) similarity in common
Architecture
LEFT: Computed overhead of entire workflow on 100,000 applications with varying numbers of slave nodes.

RIGHT: The cost of running the workflow and updating the application repository for new applications.
Result Refinement

- Exclude Popular Features and Idioms
  - Including const-data makes this less sensitive

- Clustering
  - Determine applications that are similar within a threshold
  - Pare down search space

- Exclusion Lists
  - **Problem**: Common packages dominate clustering and similarity
  - Class/Package Frequency Analysis
    - First attempt was excluding most commonly used packages
    - Led to a very long tail, with clusters
  - Core functionality
    - If we can differentiate between classes that are indirectly invoked from those that are required for functionality,
Defining Core Functionality

Core Functionality

Key Intuition: Android applications have many entry points. Some are invoked from implicit edges in the application, we only consider direct edges.

This allows us to quantify the classes and packages which are directly invoked versus those which are implicitly invoked. This helps us determine which fragments of code are essential to functionality.

Reflections can cause inaccuracy in this method.
Experimental Results

- Experiments performed on EC2 and a local cluster
  - Hadoop Streaming Implementation, C++/Ruby/Python/Java
- Vulnerable Code Reuse
  - In-Application Billing
  - License Verification Library
- Piracy
  - Detection of pirated games which were repackaged
- Malware
  - Detection of repackaged malware and new variants
Android Application Dataset

- Android Market
  - 30k Free Applications

- Anzhi Market (Chinese 3rd Party)
  - 28,159 Free Applications

- Contagio Malware Dump
  - 72 Malware Samples
In-Application Billing

Google provides IAB code verified purchased on the device.
- Dynamic rewriting of the application allowed purchases for free

Detected 295 applications use at least 70% of the sample code.

174 were vulnerable to the free market attack
- 65 detect the attack off device verification or JNI verification
- 56 remained inoperable.
License Verification Library

Identified potential vulnerability points in sample application

- Detected 182 applications with 90% of code
- 272 total applications with at least 70% of code

Single point of checking potentially allows rewriting to circumvent checks

```java
void checkAccess(...) {
    // If we have a valid recent LICENSED response, we can skip asking Market.
    if (mPolicy.allowAccess()) {
        ...
    }
    // Try to use more data here. ANDROID_ID is a single point of attack.
    String deviceId = Secure.getString(getContentResolver(), Secure.ANDROID_ID);
}
License Verification Library

- Examined all 272 applications from set
- 239 appeared to be vulnerable
  - Contained the vulnerable pattern
- Detected even with obfuscated method names and variation in vulnerable pattern

```xml
<i o="iget-object" vC="Lcom/android/vending/licensing/LicenseChecker; ">
<i o="invoke-interface" Policy;.allowAccess()"/>
<i o="move-result" vA="v1"/>
<i o="if-eqz" vA="v1" vB="0015"/>
<i o="invoke-interface" vC="LicenseCheckerCallback;.allow()" vD="v10"/>
```
Piracy on Third Party Markets

- Guardian article claims that these games have been repackaged by pirates:
  - Chillingo’s *The Wars*
  - Neolithic Software’s *Sinister Planet*

- Evaluated 28,159 applications in the Anzhi market
  - Juxtapp found 3 pirated versions of Chillingo’s *The Wars* marketed by Joy World, the same company accused of piracy in the article. No *Sinister Planet* found.
  - 71% code in common with the original application.
  - 2 are distinctly different, and the third has minor variations (string differences).
  - Pirate left “Chillingo” logo in the repackaged code!

Source: http://www.guardian.co.uk/technology/blog/2011/mar/17/android-market-pirated-games-concerns
Identifying Repackaged Malware

- Examined Anzhi Market for repackaged malware
  - 5 families: GoldDream, DroidKungFu 1 & 2, zsone, and DroidDream

- Found 34 instances of malware in the market
  - 13 Distinct GoldDream Carriers Found
    - Games were repackaged with GoldDream
  - Juxtapp quickly allowed us to identify the contaminated code

<table>
<thead>
<tr>
<th>Malware</th>
<th>Instances Found</th>
<th>Distinct New Carriers Found</th>
<th>Malware BB Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>GoldDream</td>
<td>25</td>
<td>13</td>
<td>1,898</td>
</tr>
<tr>
<td>DroidKungFu</td>
<td>6</td>
<td>0</td>
<td>5,357</td>
</tr>
<tr>
<td>DroidKungFu2</td>
<td>2</td>
<td>0</td>
<td>375</td>
</tr>
<tr>
<td>zsone</td>
<td>1</td>
<td>0</td>
<td>280</td>
</tr>
<tr>
<td>DroidDream</td>
<td>0</td>
<td>0</td>
<td>2,526</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>34</strong></td>
<td><strong>13</strong></td>
<td></td>
</tr>
</tbody>
</table>
Questions?

- Thanks for listening!
- Questions? sch@eecs.berkeley.edu