OpSeq: Android Malware Fingerprinting

Aisha Ali-Gombe
aaligomb@uno.edu

Irfan Ahmed
irfan@cs.uno.edu

Golden G. Richard III
golden@cs.uno.edu

Vassil Roussev
vassil@cs.uno.edu

Dept. of Computer Science
University of New Orleans
New Orleans LA 70148

ABSTRACT

Android malware are often created by injecting malicious payloads into benign applications. They employ code and string obfuscation techniques to hide their presence from antivirus scanners. Recent studies have shown that common antivirus software and static analysis tools are not resilient to such obfuscation techniques. To address this problem, we develop a robust fingerprinting approach that can deal with complex obfuscation with a high degree of accuracy.

Our approach, called *OpSeq*, scores similarity as a function of normalized opcode sequences found in *sensitive functional modules* as well as app permission requests. This combination of structural and behavioral features results in a distinctive fingerprint for a malware sample, thereby improving our model's overall recall rate. We tested our prototype on 1,192 known malware samples belonging to 25 different families, 359 benign apps, and 207 new obfuscated malware variants. The empirical results show that *OpSeq* can correctly detect known malware with an F-Score of 98%.

CCS Concepts

•**Security and privacy** → **Malware and its mitigation;** •**Software and its engineering** → *Automated static analysis;* •**Information systems** → *Similarity measures;*

Keywords

Android, Malware, Fingerprinting, Static Analysis, Opcode-Sequence, Obfuscation

1. INTRODUCTION

Signature-based malware detection systems have long been used in identifying known malicious samples. On Android systems, malware is often introduced by repackaging benign applications with obfuscated malicious payloads, via a variety of transformations. These forms of obfuscation have been shown to trick COTS antivirus products [13, 19] and by extension the methodologies of many other Android research tools. Most of these common tools use algorithms that search application code for strings or other signatures, which can easily be subverted. Our aim is to develop a better system that can detect known malware, which have been obfuscated and repackaged within a new application.

In this work we present *OpSeq*–a new malware-variant detection approach that is resilient against common obfuscation techniques, including reflection, encryption, code reordering and junk insertion. Given a malicious application, *OpSeq* performs static analysis of the application code and identifies functional level components (Java methods) referred to as *sensitive functional modules.* These modules invoke vital APIs such as reflective, permission-guarded, resource/data access and network/file system activities. Based on the characterization of Android malware in well-known existing work [23], these sensitive APIs open a channel by which malicious apps can manipulate a victim's device.

OpSeq extracts the components from a known sample and creates corresponding signatures, which are used to search for similar components in target applications. Target applications are then classified as malicious or benign based on this evaluation. This approach is a significant improvement over existing work that targets opcode-sequence similarity [10, 22, 14], in that it filters out irrelevant application code (reducing noise in signatures) and focuses only on a small portion of code that has high potential to contain malicious code (making signatures more accurate). As a secondary feature, we use the list of requested permissions to improve detection accuracy, as app variants tend to have very similar permission sets.

Contribution: The main contribution of this work is the development of *OpSeq*, which achieves the following specific goals by design:

1. Accuracy: *OpSeq* achieves a high-level of accuracy both in classifying malicious and benign apps and in further sub-categorizing malicious apps into their distinctive individual families.
2. Resilience: *OpSeq* is effective against simple and complex obfuscation techniques commonly found in malicious apps.
3. Efficiency: *OpSeq* takes only few seconds (on average) to process an app, making it an efficient and scalable tool.

PPREW-5, December 08 2015, Los Angeles, CA, USA
© 2015 ACM. ISBN 978-1-4503-3642-0/15/12. . . $15.00
DOI: http://dx.doi.org/10.1145/2843859.2843860

We use two datasets to evaluate *OpSeq* thoroughly. The first dataset has 1,551 Android applications in total, consisting of 1,192 malicious apps from the *Android Malware Genome Project* [23] and 359 benign apps downloaded from *Google Play*. Our experimental results (on the dataset) show that *OpSeq* detects variants of known malware with an F-Score of 97.5%, 98% recall, and 97% precision. The second dataset has 207 malicious apps employing four obfuscation techniques: reflection, encryption, code reordering and junk insertion. We use two Java bytecode obfuscators to create the dataset, i.e., DroidChameleon [13] and SandMark [11]. Fourteen COTS antivirus tools, *DroidLegacy* (a state-of the art Android malware detection tool), and *OpSeq* are tested with the dataset. The results show that *OpSeq* outperforms the other tools by an average margin of 35%.

The rest of the paper is organized as follows: Section 2 presents a summary of related work; Section 3 provides an overview of our design and algorithms; Section 4 presents the implementation and evaluation of the proposed approach; Section 5 contains a discussion of our results and also discusses limitation of our work; Section 6 summarizes our findings and conclusions.

2. RELATED WORK

The first large-scale study of Android malware–the *Android Malware Genome Project*–was carried out by Zhou and Jiang [23]. Their work was aimed at characterizing existing Android malware but they did not detail their analysis and classification methodology. However, the corpus and characterization information they provided became the basis for a lot of follow up research, including ours. Their work identified that 86% of the malware is found as repackaged apps.

2.1 Opcode-sequence similarity

Santos et al. [14] developed a system of detecting malware using opcode-sequence frequencies on the *Intel x86* platform. On Android, Hanna and Zhou [10, 22] developed methodologies for using opcode-sequences to detect repackaged applications in both primary and secondary app markets. Our work differs from theirs in terms of goals and approach: theirs are focused on detecting application repackaging in general, whereas we are interested in detecting malware, in particular. Their opcode-sequence-based detection can be disturbed with relative ease by injecting small amounts of noise; in contrast, we explicitly designed our approach to deal with different obfuscation techniques. Our system normalizes an opcode-sequence both on the known and target samples before comparison, which significantly reduces the effects of dead and junk code on our similarity measures.

For example, using the same target code snipped illustrated by [12] in their evaluation of Android repackaging detection algorithms, the original sequence is altered with a series of junk instructions to form an obfuscated version as shown with the mnemonics in Listing 1. In the worst case, DroidMoss [22] can take the hash of the entire sequence (i.e., a 4-gram). In such cases, detection can be evaded completely. Conversely, in its best case, using a 2-gram rest point, DroidMoss can attain a maximum of ≈15% similarity.

Our algorithm on the other hand first normalizes both sequences, then group them into 2-gram pattern, as shown in Listing 2. The target-overlap coefficient, which gives emphasis to the known profile will be ≈ 67%.

Furthermore, our small structured signatures target only sensitive functions, which can help eliminate most GUI code. This is useful because GUI code is almost always irrelevant for malware detection.

Listing 1: Original and obfuscated sequences

```
//Original sequence
invoke-static, move-result-object, const-string,
    invoke-interface
//Obfuscated sequence
invoke-static, move-result-object, move-object,
    const-string, move, invoke-interface, move,
    move-object
```

Listing 2: Original and obfuscated sequences normalized and grouped into 2-gram pattern by *OpSeq*

```
//Original sequence normalized
const-string, invoke-interface,invoke-static,
    move-result-object

//Original sequence grouped into 2-gram pattern
const-string:invoke-interface,
    invoke-interface:invoke-static,
    invoke-static:move-result-object

//Obfuscated sequence normalized
const-string, invoke-interface, invoke-static, move,
    move, move-object, move-object, move-result-object

//Obfuscated sequence grouped into 2-gram pattern
const-string:invoke-interface,
    invoke-interface:invoke-static,
    invoke-static:move, move:move, move:move-object,
    move-object:move-object,
    move-object:move-result-object
```

2.2 Semantic-based detection

Semantic-based detection uses information flows as features to detect similarity between Android applications [3, 4, 7, 15, 17, 20, 21]. PiggyApps [21] first identifies the code containing the main functionality (the primary module) in legitimate apps. Then, it extracts and organizes this semantic information from the module as a vantage point tree. The resulting signatures are used to search for "piggybacked" apps in Android markets. Apposcopy [7] provides a specification language that allows the manual creation of signatures for known malware. To find similarities, it extracts semantic features of a new app using inter-component call graphs and performs static taint analysis.

DroidLegacy [4] is an API-based static malware detection system that breaks an app into sub-modules. The set of API calls made in these modules are compared against the signatures of known samples. DroidAnalytics [20] uses a three-level signature that represents API calls made from within apps. API call sequences form signatures for methods, and the collection of all method signatures forms a signature for each class. The collection of class signatures then forms the signature for the entire application. All of the techniques discussed above can be easily circumvented with simple obfuscation. Encryption alone can hinder data flow analyses while the combination of encryption and reflection will make it difficult to extract any meaningful information from the application code.

2.3 Permission-based certification

Kirin [5] detects dangerous behavior in applications by analyzing their permission requests. It uses a set of rules that defines which permissions combination might be dangerous. Another permission-based behavioral fingerprinting is DroidRanger [24]. However, as detailed in [6], most Android apps are over-privileged in general and even benign apps have a tendency to request combinations of permissions that could be considered dangerous. SCanDroid [8] is a security certification tool that determines if specifications in the application manifest match what is requested within the app's components. RiskRanker [9] provides a systematic approach that measures the risk of dangerous behavior associated with an application based on native code, dynamic class loading, and callback handlers. VetDroid [18] uses dynamic analyses to reconstruct how permissions are used to access resources. All these techniques attempt to discover if dangerous behavior is present, while *OpSeq*'s primary goal is to measure similarity of unknown apps against known malware.

3. DESIGN

The *OpSeq* architecture consists of two major components as shown in Figure 1: feature extraction, and signature generation. Feature extraction identifies code in *sensitive functional modules* and extracts the corresponding opcode-sequences. It also extracts the list of permissions used by the app to gain access to system resources. The signature generation step normalizes the sequences from feature extraction, and then slices each into a small chunk of n-gram opcodes, which constitute of the signature used for similarity matching.

OpSeq's signature matching is a 3-step process, each illustrated in the algorithms 1, 2, and 3 respectively. We use a bottom-up approach consisting of three levels for similarity detection. First, we determine matches at the opcode level, and then their aggregate gives the similarity at the functional level. Finally, the result of functional-level and permission-overlap determines the final index. A similarity score is computed at each level where a subsequent level takes into account the score of its immediate last level, achieving substantial improvement in the overall accuracy of our system.

3.1 Feature extraction

In this phase, permission requests and functional opcodes are extracted from the app's manifest and `classes.dex` files, respectively. The `classes.dex` file, denoted by cd, is a set of m Java classes, jc:

$$cd = \{jc^1, jc^2, \ldots, jc^m\}.$$

Each Java class $jc^k, 1 \leq k \leq m$ is made up of n functions.

$$jc^k = \{f_1^k, f_2^k, \ldots, f_n^k\}.$$

We can simplify the notation by aggregating the set of functions in a dex file as:

$$cd = \{\{f_1^1, f_2^1, .., f_n^1\}, \{f_1^2, f_2^2, .., f_n^2\}, .., \{f_1^m, f_2^m, .., f_n^m\}\}$$

An individual function f_i consists of set of instructions I: $f_i = \{I_1, I_2, \ldots, I_k\}$. Instructions are tuples containing an *opcode* o and a (potentially empty) list of operands. For the purposes of our analysis, we focus solely on the opcodes and we disregard the operands. In other words, we view a function f as a sequence of i opcodes:

$$f = o_1, o_2, \ldots, o_k.$$

To be included in the feature set, we filter the list of functions based on two criteria: a) the function must invoke at least one method from a *sensitive* API (a manually compiled list of selected system APIs); and b) its opcode sequence is not on the list of the most commonly found opcode sequences FS (determined empirically).

3.2 Signature generation

Our signatures are formed by taking into account the type of obfuscation that can affect opcode sequences; these include both junk code insertion and code reordering. Junk code insertion is an obfuscation technique which embeds pools of instructions that never execute at run time, such as instructions in an artificial `if-else` branch that never triggers. Code reordering permutes instructions that have no ordering dependencies; the main point is to subvert hash-based detection schemes.

The next step of the process is to normalize the extracted opcodes in each sequence f. This distorts the order of opcode arrangement and groups similar opcodes in the same cluster. Next, for each normalized opcode sequence, we generate subsequences of n-gram opcodes. In choosing the best value for n, we run our system with uni-gram, 2-gram and 3-gram. Empirical results (as shown in section 4) indicate that 2-gram gives the best accuracy. From now on, we refer to each sequence (representing one function) containing k number of 2-gram opcodes as *pattern P*.

$$S = \{P^1, P^2, \ldots, P^m\} \tag{1}$$

where each

$$P^m = \{os_1^m, os_2^m, \ldots, os_i^m\} \tag{2}$$

Depending on number of sensitive functions found, a signature for known profile S is a set of m patterns P, where each pattern is a set of i 2-gram opcodes denoted as os. This forms the structural features for the familial malware.

Our technique allows similarity to be measured from the basic unit of code upwards. Similarity between apps becomes an aggregate of individual functional similarities and as such the likelihood of determining relationships between two related codes increases.

We know that malware variants will not be exact copies of one another, but our assumption is that most of their malicious functionality and code structure remains similar. Malware can add, remove or substitute code within a function. However, for it to retain vital key behaviors, some part of the code has to be consistent across variants. Thus by carefully analyzing each function as a single unit and normalizing its opcodes, our algorithm can ascertain whether a relationship exists between two functions of different applications.

3.3 Similarity Matching

Our similarity matching is a 3-step process that begins with Pattern-level similarity, then Function-level similarity to determine a score for all the matched functions. Lastly, the Final-similarity index scores similarity as a function of the Function-level similarity and permissions overlap.

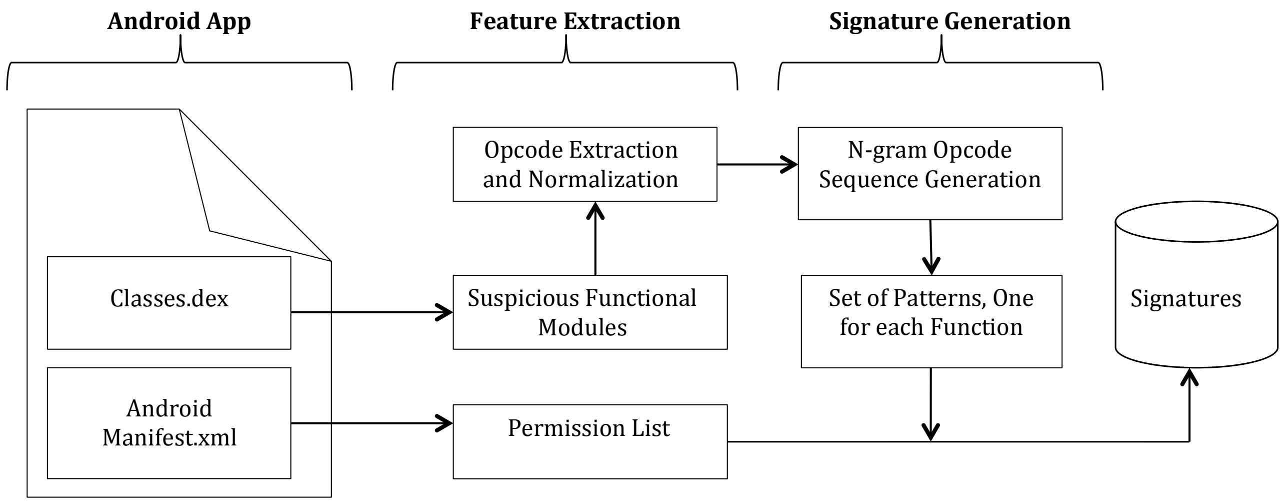
Android App
Feature Extraction
Signature Generation
Classes.dex
Android Manifest.xml
Opcode Extraction and Normalization
Suspicious Functional Modules
Permission List
N-gram Opcode Sequence Generation
Set of Patterns, One for each Function
Signatures

Figure 1: *OpSeq* Signature Generation Workflow

3.3.1 Pattern-level similarity

Given a reference piece of malware A with signature S_A and sample application B with signature S_B:

$$\begin{aligned} S_A &= \{P^1, P^2, \ldots, P^m\} \\ P^m &= \{os_1^m, os_2^m, \ldots, os_i^m\} \\ S_B &= \{P^1, P^2, \ldots, P^n\} \\ P^n &= \{os_1^n, os_2^n, \ldots, os_j^n\} \end{aligned} \tag{3}$$

For each P^m in S_A , we determine its best match in S_B using the **Targeted Overlap Coefficient** (TOC) technique[16]. TOC is derived from the Overlap Coefficient or Szymkiewicz-Simpson coefficient, which is defined as the ratio of the size of the intersection of two sets to the size of a *target* set.

In pattern-level similarity, the TOC measures the ratio of common 2-gram opcodes found in the intersection of P^n and P^m to the size of P^m, given P^m as the target set. The result indicates the power of inclusion of P^m in P^n. The TOC, denoted by $R(P^n, P^m)$ is:

$$R(P^m, P^n) = |P^n \cap P^m| / |P^m| \tag{4}$$

Since our algorithm specifically leverages finding the relationship between a new pattern and a known target pattern, the TOC is our preferred similarity metric. To determine how P^m relates to P^n, we require a threshold value T defined as the *pattern-level threshold* (PLT). This denotes the minimum acceptable similarity ratio. In this step, If R $(P^m, P^n) \geq T$ then we write R to a buffer BUF and we eliminate both P^m and P^n. While if R $(P^m, P^n) < T$ then P^m is eliminated while P^n remains. This loop continues until all the patterns in S_A are compared to patterns in S_B (Algorithm 1).

The pattern-level similarity is measured by the value of R, which lies between 0 and 1. As R approaches 1, it means most 2-gram opcodes found in P^m are also present in P^n, hence P^m is similar to P^n. However as R approaches 0, the similarity between P^m and P^n diminishes. The similarity score calculated is not transitive–P^m is a pattern from our known profiles while P^n is a pattern in a test sample and the idea is to calculate how close P^m is to P^n and not vice versa. A positive outcome at this level of matching can be attributed to one of the following reasons. If P^m and P^n are modules with the same functionality then R will be close to 1. It is also possible P^n is a disguised version of P^m that has been obfuscated but still retains most of its original opcodes. In this situation, we can also derive a match. It is also possible for P^m and P^n to match to a certain degree even though neither is derived from same functional module, which will create a false positive. Our evaluation results have shown this is quite rare in practice.

This pattern level-matching algorithm has proven effective in overcoming the effect of junk insertion and reordering obfuscation techniques.

3.3.2 Function-level similarity

This step analyzes all the results generated in pattern-level matching. As shown above, for each matched pattern, the derived coefficient is stored in a buffer BUF.

$$BUF = \{R_1, R_2, \ldots, R_k\} \tag{5}$$

The set of ratios represents all the matched functions between S_A and S_B. Function-level matching calculates a score between two samples as an aggregate of their pattern-level matching. As preliminary testing, we tried 4 different similarity coefficients (Cosine, Jaccard, Edit Distance and Sorensen (Dice) Coefficient) on sample sets and measured the results. The *Dice Coefficient* gave us the best result. Briefly, the Dice coefficient is a measure of the intersection between two given sets scaled by their size. Although the choice of Dice Coefficient is basically empirical, in general it gives more weight when there is an intersection than the Jaccard, thus strengthening similarity. The Dice Coefficient D is defined as follows:

$$D(S_A,\ S_B) = 2{*}\ |S_A \cap S_b| / |S_A + S_B| \tag{6}$$

A value of T (pattern-level threshold) that is close to 1 means each R in BUF is also close to 1. Thus:

$$|S_A \cap S_B| = \sum BUF \tag{7}$$

And therefore:

$$D = 2^*(\sum BUF) \;/\; |S_A + S_B| \tag{8}$$

The coefficient D (Algorithm 2) denotes structural similarity between extracted functions found in two applications.

3.3.3 Final-similarity index

The final similarity score is calculated based on the result of function-level matching and permissions overlap. We first need to calculate the permissions overlap using the Targeted Overlap Coefficient:

$$\begin{aligned} permA &= permission\ list\ in\ A \\ permB &= permission\ list\ in\ B \end{aligned} \tag{9}$$

Thus the permission overlap from A to B, called PO, is given as:

$$PO\ (A,B) = |permA \cap PermB|/|permA| \tag{10}$$

This gives the ratio of similar permissions found in A and B against the length of set of permissions for A . The permissions overlap is a weight that strengthens the result of the function-level matching and indicates (at a coarse-grained level) what behavior is common between A and B . If two apps contain the same malware footprint, they normally should have some common permissions. The overlap coefficient is a value between 0 and 1.

The final similarity index given as SS and is a function of function-level similarity D and permissions overlap PO. This is defined as:

$$SS = D * PO \tag{11}$$

The value of SS is a coefficient that indicates the strength of similarity between two applications based on our extracted features. A minimum similarity index MSI is required to determine if the coefficient SS is good enough. Thus when $SS \geq MSI$, we report that a known footprint for malicious code is present in the target app.

Algorithm 1 Pattern-Level Matching

```
function :(PLM($S_A$, $S_B$, T))
    for $P^m$ in $S_A$ do:
        for $P^n$ in $S_B$ do
            $inter=multi\_intersect(P^m, P^n)$
            $coef = len(inter) / len(P^m)$
            if $coef \geq T$ then:
                $append(coef, BUF)$
                $remove(P^n, S_B)$
            end if
        end for
    end for
    return $BUF$
end function
```

The summary of all the design notation is shown in Table 1.

Algorithm 2 Function-Level Matching - Dice coefficient

```
function :(FLM($S_A$, $S_B$, BUF))
    $suM=sum(BUF)$
    $funAvg= 2/ (len(S_A)+len(S_B)$
    $D = funcAvg * suM$
    return $D$
end function
```

Algorithm 3 Final-Similarity Index

```
function :(FSI($perm_A$, $perm_B$, D))
    $inter= multi\_intersect (perm_A, perm_B)$
    $perm = inter/len(perm_A)$
    if perm $> 0$ then
        $SS=perm*D$
    end if
    return $SS$
end function
```

4. EVALUATION

We have implemented a prototype of *OpSeq* in Python to test its efficacy on obfuscation techniques typically employed by Android malware. This section presents our empirical results.

4.1 Focus of the Evaluation

The focus of the evaluation is twofold: 1) detection of known malware, and 2) further categorization of detected malware into their respective malware families. In particular, the evaluation targets the following two research questions:

1. **Malware/Benign apps detection (Mal/Ben):** Can *OpSeq* accurately detect a variant of repackaged malicious code without confusion with benign applications?
    - True Positive (TP): known malware code is correctly detected.
    - False Positive (FP): benign code wrongly detected as containing malware code.
    - True Negative (TN): benign apps are not flagged as having malware code.
    - False Negative (FN): known malware code is not detected.
2. **Malware class detection (Mal/Class):** Can *OpSeq* categorize a known repackaged malware code into its respective family?
    - True Positive (TP): all malware code that are correctly categorized.
    - False Positive (FP): all malware code that are wrongly categorized as variants of different families.
    - False Negative (FN): all malware code that were not detected.
    - True Negative (TN): true negative is eliminated in this test because all the malware samples belong to at least one known class. The samples used in this test were derived from the true positives in Mal/Ben above.

Table 1: Design Notation Table

Notation	Definition
cd	classes.dex
jc	Java Class
f	Java Function
o	Instruction Opcode
S	Signature
P	Pattern from a Signature
$TOC - R(P^m, P^n)$	Targeted Overlap
BUF	Set of Targeted Overlap Coefficients
T	Threshold
$D(S_A, S_B)$	Dice Cofficient
$PermA$	Permission List from Sample A
$P.O(A, B)$	Targeted Permission Overlap
SS	Final Similarity Index
MSI	Minimum Similarity Index
PLT	Pattern Level Threshold

Table 2: Malware sample distibution as per four obfuscation techniques: reflection, encryption, code reordering, and junk insertion

Obfuscation Type	Obfuscation Techniques	Number of Malware Samples
	DroidChameleon	
Simple Obfuscation	Encryption (E)	24
	Reflection (R)	25
	Reordering (O)	24
	Junk (J)	24
Complex Obfuscation	E & R	23
	E & R & O	24
	E & R & J	23
	SandMark	
	Transparent Branches	20
	Random DeadCode	20
Total		207

4.2 Dataset

We use two datasets for experiments. The first dataset has 1,551 Android applications consisting of 359 benign applications downloaded from *Google Play*, and 1,192 malware samples (of 25 families) from the well-known *Android Genome project* [23]. The second dataset has 207 malware samples employing four obfuscation techniques: reflection, encryption, code reordering and junk insertion. DroidChameleon [13], and SandMark [11] are used to generate the samples. Specifically, malware sample distribution of DroidChameleon and SandMark are 167 variants (of 25 malware classes), and 40 variants (of 20 malware classes) respectively. If only one obfuscation technique is used by a malware, we refer it as *simple obfuscation*, otherwise, it is referred as *complex obfuscation.* Table 2 presents the distribution of malware in the dataset in accordance with obfuscation tool and type.

4.3 Optimum Variables

As mentioned in the previous section, we chose to slice our normalized sequences using 2-gram sub-sequences, based on results from empirical studies. The average accuracy of our approach tested with unigram, 2-gram and 3-gram was 91%, 98.9% and 96% respectively. More so, we have identified two variables necessary for our algorithm: 1) Pattern-level threshold (PLT) - the minimum overlap ratio required to assume similarity between two 2-gram patterns from different apps (in Pattern-level similarity), and 2) Minimum similarity index (MSI) - the minimum score that determines if a malware footprint is present in an application (in Final-similarity index). To get the optimal values of PLT and MSI, we choose arbitrary values and plugged them into our algorithm. The chosen values for PLT are 70, 80 and 90, all expressed as a percentage. MSI values are 3, 4, 5, 6, 7, again expressed as a percentage. Our variables are chosen based on the statistics of Mal/Ben detection.

In choosing the optimal values, we use F-Score statistics. The F-Score measures the overall accuracy of a test, which depends on precision and recall. Recall is the measure of accuracy that a specific class has been detected (% of correct malware families detected out of all malware samples), whereas precision is the percentage of positive prediction (% of all malware detected out of all sample applications).

The combination of PLT and MSI that gives the highest F-score is the optimal solution for the test data. We ran *OpSeq* on our dataset using the above combination of PLT and MSI and the results of our execution is shown in Table 3. Each row (a model) represents one combination. We then calculate the statistics (false positive, false negative, true positive, true negative, precision, recall and F-score) for each model. The equation for F-Score statistics is:

$$F = 2 * ((Precision * Recall)/(Precision + Recall)) \quad (12)$$

The results in Table 3 above indicate the highest F-Score is attained with PLT equal to 80% and MSI equal to 3%. This model gives us 99.3% precision, 98.5% recalls and an F-Score of 98.9% for Mal/Ben detection. Furthermore, the false negative rate (given as FN/TP+FN) for this model was ≈1.5%. This indicates our system has a very small probability of "miss" detection for known malicious code. The false positive rate for the same model is ≈2.2%.

Using the same metrics on Mal/Class, our F-Score accuracy was 97.5%, 98% recall and 97% precision. Thus overall, our system is capable of accurately detecting malware 98.9% of the time and can categorize the malware into its correct family with 97.5% accuracy. The confusion matrix for Mal/Ben and Mal/Class based on the optimal values is shown in Table 4 and Table 5, respectively.

The rationale for choosing our optimal values is to further buttress the precision/recall curve of our solution as shown in Figure 2. With each point representing one model, the points on the curve where precision and recall are around their maximum and are nearly equal indicate the point of maximum accuracy (interpolated precision is used to smooth the PR curve).

4.4 Empirical Results

Accuracy: Given the F-Score for Mal/Ben, our system can detect malware footprints in an app with 98.9% accuracy. However due to some reasons identified below, the Mal/Class test (i.e., identifying the class category) is slightly lower (97.5%). In comparison with some recent known malware detection tools, *OpSeq*'s Mal/Class accuracy (97.5%) did better than Apposcopy [7] with 90% accuracy. On the other hand DroidLegacy [4] recorded a membership test accuracy of 98%, slightly higher than *OpSeq*'s Mal/Class figure of 97.5%.

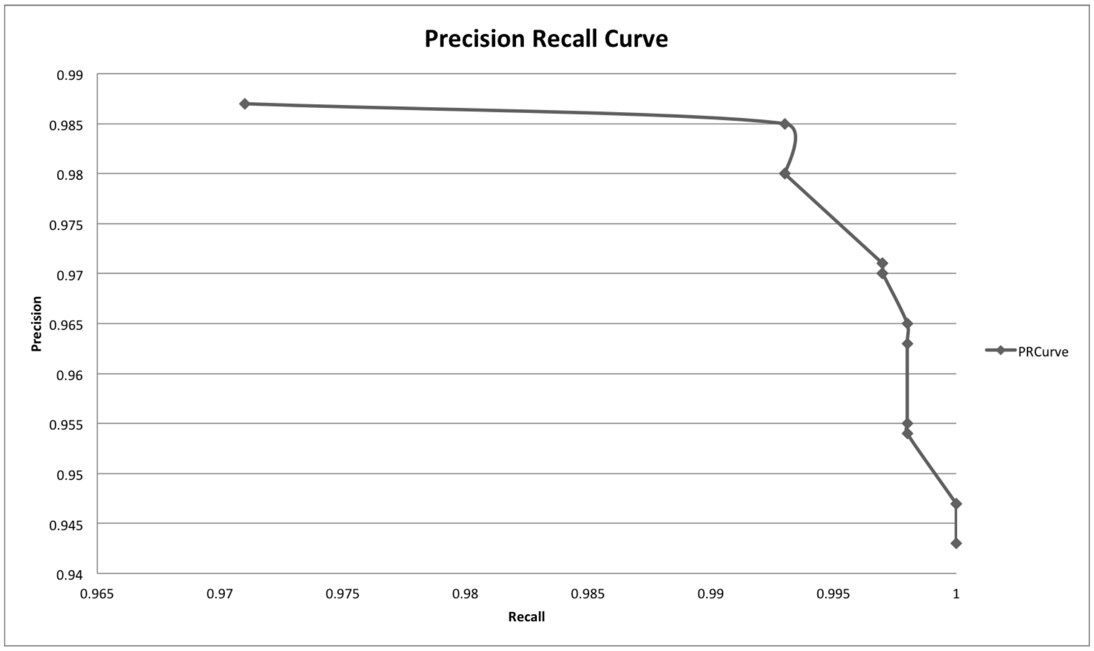

Figure 2: Precision & Recall Curve

Table 3: Model Selection Result: F-Score, Precision, Recall

Model	False Positive	False Negative	True Positive	True Negative	Precision	Recall	Interpolated Precision	F-Score
T_90M_7	0	68	1124	359	1	0.943	1	0.971
T_90M_6	1	63	1129	358	0.999	0.947	1	0.973
T_80M_7	0	63	1129	359	1	0.947	1	0.973
T_90M_5	2	55	1137	357	0.998	0.954	0.998	0.976
T_80M_6	2	54	1138	357	0.998	0.955	0.998	0.976
T_90M_4	2	44	1148	357	0.998	0.963	0.998	0.98
T_80M_5	2	42	1150	357	0.998	0.965	0.998	0.981
T_70M_6	25	31	1161	334	0.98	0.97	0.997	0.983
T_70M_7	10	35	1157	349	0.99	0.97	0.997	0.983
T_80M_4	5	34	1158	354	0.996	0.971	0.997	0.984
T_90M_3	4	34	1158	355	0.997	0.971	0.997	0.984
T_70M_4	108	22	1170	251	0.92	0.98	0.993	0.986
T_70M_5	52	27	1165	307	0.96	0.98	0.993	0.986
T_80M_3	8	18	1174	351	0.993	0.985	0.993	0.989
T_70M_3	192	15	1177	167	0.86	0.987	0.971	0.979

However, comparing DroidLegacy's recall rate of 94%, which is the true positive rate, versus our figure of 98%, indicates our system is capable of better detection. Furthermore, our system has better coverage in terms of malware families processed (they processed 11 families against ours with 25 families) and resiliency to different obfuscators as shown in the next subsection.

False Negative: Eighteen (18) malware were incorrectly categorize as benign apps. Samples from DroidKungFu, DroidDreamLight and Anserverbot constitute the bulk of our false negative predictions as shown in Table 5. One reason for the false negatives can be traced to our signature generation process. This process randomly picks only one sample from a set to create a class signature; it is possible that the sample might be the oldest, newest or even a variant that has more inclusion or exclusion of instructions within the malicious code. For instance, the sample we use to generate the signature for DroidDreamLight is a newer version than the rest of its variants. This sample has about 98 functions that were extracted for the signature against 18 for the older version. Thus, since our similarity calculates overlap based on the target known profile, this sample was a miss.

Another reason for some false negatives can be attributed to native code exploits. Malware samples that have most of their malicious code written in native code will often result in a miss detection (e.g., some few variants of DroidKungFu and Anserverbot). Currently, *OpSeq* is only designed to handle the dex file (containing Java bytecode) and handling native code is the subject of ongoing research.

False Positives: For Mal/Ben, 8 applications out of 359 downloaded from Google Play were detected as malware by

Table 4: Mal/Ben Best Model Confusion Matrix

	(Positive) Malware	(Negative) Benign	Total
(Positive) Malware	1174	8	1182
(Negative) Benign	18	351	369
Total	1192	359	

Table 5: Percentage of Mal/Class Prediction Result

Malware Family	False Negative	False Positive	True Positive	Total
ADRD	0	0	22	22
Anserverbot	7	12	286	305
BeanBot	0	0	8	8
Bgserv	0	0	9	9
CruseWin	0	0	2	2
DroidDream	1	1	14	16
DroidDreamLight	4	0	43	47
DroidKungFu	5	19	418	442
Geinimi	0	0	69	69
GingerMaster	0	0	4	4
GoldDream	0	0	47	47
Gone60	0	0	9	9
GPSSMSSpy	0	0	6	6
HippoSMS	0	0	4	4
jSMSHider	0	0	13	13
KMin	0	0	52	52
Pjapps	1	2	55	58
Plankton	0	1	10	11
RogueLemon	0	0	2	2
RogueSPPush	0	0	9	9
SndApps	0	0	10	10
Tapsnake	0	0	2	2
YZHC	0	0	22	22
zHash	0	0	11	11
Zsone	0	0	12	12
Total	18	35	1139	1192

OpSeq, as shown in Table 4. In order to confirm the true nature of these 8 applications, we ran them through VirusTotal [1]. The output flagged 3 out the 8 apps as malware, reducing our true false positives to only 5 apps.

In Mal/Class, the false positives recorded were largely due to the use of common code snippets. This code ranges from 3rd party libraries to adware. Most of the malware families originate from the same location and have common targets, so it is not uncommon to find similar libraries and/or adware packaged within the applications. For instance, we analyzed one of the Anserverbot samples which *OpSeq* miscategorized as DroidKungFu and it contains the Adware.waps, while the sample used to generate the signature for the Anserverbot family does not. However, the adware on its own collects various user data from a device and invokes some sensitive APIs, like getRunningTask. Thus our system retrieved more features than were available for its family' s signature. On the other hand, the DroidKungFu sample for signature generation contains the same adware. Since *OpSeq* assigned the malware class based on the strength of the final similarity index, that Anserverbot sample was flagged as DroidKungFu.

4.5 Evaluating Resiliency

Our approach can detect known malicious code obfuscated and repackaged within another app. In order to measure its resiliency, we employ two different open source obfuscators: *DroidChameleon* - a newer system specifically designed to obfuscate Android apps, and *SandMark* - a well known Java obfuscator. In both systems, we gave special emphasis to those modules/algorithms that can severely hinder opcode-sequence-based detection.

4.5.1 Test against DroidChameleon

Using DroidChameleon, we generated 167 obfuscated variants of the malware in our sample set with varying degrees of transformations, ranging from simple (one method) to complex (combination of obfuscators). In this segment of our evaluation we choose 4 common obfuscation techniques employed by Android malware:

1. `Encryption`: names, strings and field encryption
2. `Reflection`: permission based API calls are changed from direct invocation to using reflection API in helper classes
3. `Junk Insertion`: addition of "noise" instructions
4. `Code Reordering`: altering the flow of program execution by changing the positions of unrelated instructions

In simple obfuscation, an application is transformed with just one technique, e.g., *getDeviceId* is transformed from a direct call to invoking a helper class that calls *class.getMethod()* and then *method.invoke()*. In complex obfuscations, applications get obfuscated with two or three techniques, e.g., all strings, fields and names are encrypted and then all permission-based API calls are invoked using a reflection API.

For simple obfuscation, we introduced encryption into 24 samples, 25 samples had added reflection, junk instructions were added in 24 samples, and code reordering in 24 samples. For complex obfuscation our samples were transformed with a minimum of two different techniques. We modified DroidChameleon such that when an app is unpackaged, it is first encrypted, then the bytecode is run through more obfuscator modules before repackaging. We successfully repackaged 23 samples with encryption and reflection, 24 had (encryption, reflection and reordering) combined and finally 23 had (encryption, reflection and junk) combined. Our results for simple and complex obfuscation as shown in Table 6 and Table 7, respectively. *OpSeq* scores 100% average detection rate in simple obfuscation and 88% for complex transformations.

4.5.2 AntiVirus Results

Using the same obfuscated samples mentioned above, we assessed the detection ratio of other antivirus products using the VirusTotal website. For simple obfuscation, out of the top 14 antivirus products, AVG recorded the highest detection rate with average detection of 65.83%, followed by Dr. Web, F-Secure, Kaspersky, AhnLab-V3 with slightly above 50%, and Panda, with the lowest detection rate of 4%. In the complex obfuscation cases, AhnLab-V3 led other antivirus

Table 6: Evaluation results for DroidChameleon's simple obfuscation.

	Encryption	**Reflection**	**Reordering**	**Junk**	**Average Detection Rate**
Total No. of Sample	24	25	24	24	
Research Tools					
OpSeq	24	25	24	24	100
DroidLegacy	6	0	15	15	37.5
AntiVirus Software					
AVG	8	20	18	18	65.98
DrWeb	17	5	17	17	57.73
F-Secure	6	16	17	16	56.7
Kaspersky	6	16	16	15	54.64
AhnLab-V3	14	14	10	12	51.55
Avast	6	15	13	14	49.48
Avira	5	7	12	12	37.11
Symantec	4	12	7	11	35.05
Ad-Aware	6	6	7	7	26.8
BitDefender	6	6	7	7	26.8
AVware	5	6	6	6	23.71
McAfee	5	5	5	5	20.62
Panda	2	2	2	2	8.25
Baidu-International	1	1	1	1	4.12

Table 7: Evaluation results for DroidChameleon's complex obfuscation.

	Encryption & Reflection	**Encryption Reflection & Reordering**	**Encryption Reflection & Junk**	**Average Detection Rate**
Total No. of Sample	23	24	23	
Research Tools				
OpSeq	23	24	15	88.57
DroidLegacy	0	0	0	0
AntiVirus Software				
AhnLab-V3	12	13	12	49.33
AVG	6	7	6	27.14
Kaspersky	6	6	4	22.86
Ad-Aware	6	7	3	22.86
BitDefender	6	7	3	22.86
F-Secure	6	7	3	21.33
Avast	6	6	3	21.43
Avira	5	5	3	18.57
AVware	5	5	3	18.57
DrWeb	5	5	3	18.57
McAfee	5	5	3	18.57
Symantec	4	4	2	14.29
Panda	2	1	2	7.14
Baidu-International	1	2	2	7.14

products with a detection rate of 49.33%, then AVG with 27%. All the rest recorded less than 25%.

4.5.3 DroidLegacy Results

We also used the same obfuscated variants described above to evaluate the performance of DroidLegacy. We set the optimum threshold of 0.7 as specified in their paper. For simple obfuscation, DroidLegacy had an average detection rate of 37% and 0% for complex obfuscation. Like many common malware detection tools, DroidLegacy depends on API names to create signatures for malware. In situations where the name is obfuscated using reflection, chances of detection become very low. This explains why it recorded 0% for all apps that were repackaged using reflective method invocation. Furthermore, it performed poorly with encryption alone, detecting only 6 out of 24 encrypted apps.

4.5.4 Tests against Sandmark

To further evaluate the resilience of *OpSeq* against major obfuscation techniques, we tested it against SandMark. SandMark is well known and highly documented tool used for watermarking, tamper-proofing and code obfuscation of Java bytecode [11]. For the purpose of our analysis, we used only some specific modules within *Obfuscation Executive* to transform the application bytecode. We generated 40 new variants by inserting *Transparent Branches and Random DeadCode.* The resulting new jar files were repackaged, aligned and signed before analysis.

In this experiment, we tested *OpSeq*'s similarity detection capability by performing a one to one comparison between each of the 40 repackaged malware and its original sample. The results indicates *OpSeq* can detect these obfuscations with 100% accuracy.

4.6 Measure of Performance

Our experimental system an Ubuntu Linux 64-bit system running on an Intel Xeon CPU at 2.5 GHz, with 16 GB RAM. We leverage an open source Android reverse engineering tool called *apktool* for the conversion of the Android dex file into an intermediate bytecode representation, called *smali*. Given a target application with 205 Java functions (28624 lines of Dalvik bytecode), for which 46 of these functions invokes one or more of the sensitive APIs, it takes an average of 4.5 seconds on our test machine to perform a one-to-one similarity matching with a known sample that has 118 Java methods (19307 lines of Dalvik instructions).

Furthermore, for detection purposes, it takes an average of 11.6 seconds to analyze the same target app against known profiles of the 25 malware families (classification). This results outperforms Apposcopy, which took an average of 346 seconds per analysis of an app with 26786 lines of Dalvik bytecode.

The fact that *OpSeq* is designed to fingerprint apps based on small structured signatures extracted from vital points within the program code base helps to eliminate unnecessary noise in the matching process and thus improves our system's overall performance.

5. DISCUSSION

The experiments in the previous section illustrate that for a large class of Android malware, *OpSeq* significantly outperforms state of the art commercial and research products that rely on signature-based detection algorithms. Our tests indicate that *OpSeq* is effective in detecting malware in the presence of both simple and complex obfuscation techniques, many of which compromise the accuracy of existing detection techniques.

In the malware families we analyzed, the largest stored signature has 118 patterns (slices) to be matched while the smallest has 3 patterns. Theoretically, if we compare *OpSeq* with other opcode-sequence based detection like [22, 10] which slice the entire app's opcode-sequence into n slices, our app's signature sizes are guaranteed to be smaller and therefore much more efficient. Furthermore, our average processing time of 11.6 seconds per 25 family comparison indicates our algorithm is fairly scalable.

During the course of our analysis, our findings indicate extensive code reuse amongst some of the malware families. For instance, Zhou et al has categorized DroidKungFu into 5 major families [23]. However we found malware from these classes to contain a considerable amount of common code segments. Thus we categorize them as one class.

Also, Anserverbot and Basebridge have been found to contain a similar main package com.keji.danti. They differ slightly where BaseBridge loads an extra payload that leads to privilege escalation while some Anserverbot variants do not. But since *OpSeq* only processes the dex file, our analyses flag one as a variant of the other. Information from Foresafe encyclopedia [2] and analysis results of some antivirus products in VirusTotal also affirm their relationship, hence we merge them into one class. Some common adware and external libraries were also found to be present in malware of different families. Such instances have increased the rate of our false positives and affect the overall accuracy for Mal_Class detection, but these issues also raise indicate that *OpSeq* has significant value in better malware classification as well as detection.

6. LIMITATIONS

Like most malware detection schemes based on static analysis, *OpSeq* can be thwarted via whole class encryption and extensive dynamic class loading. This is because *OpSeq* only extracts features from the available classes.dex file. Extra classes that are fully encrypted or loaded at runtime cannot be processed. Furthermore, *OpSeq* is designed to process only the Dalvik instructions and as such cannot handle native code. However as part of our proposed future work, we aim to extend our system to use similar techniques to parse and create signatures from the native code's assembly instructions.

Code-reordering that can split functions into multiple subfunctions may also negatively impact our approach. Also, very large numbers of junk instructions can introduce so much noise that it may affect the quality of our signatures. However these obfuscation techniques will only be problematic when more than one *sensitive functional module* is tampered with, which in practice will require significant human intervention. Furthermore, since our approach clusters common opcodes together by normalizing them before we slice the whole sequence into 2-gram patterns, excessive noise can only affect our signature when unique opcodes are introduced viz-a-viz the normalization pattern. These unique opcodes must vary significantly from those normally used within the functions.

7. CONCLUSIONS

In this paper, we developed a new, resilient approach for statically detecting Android malware variants. Our system generates signatures for known malicious code as a function of the normalized opcode sequence found in *sensitive functional modules* and the permissions an app requests. Malware belonging to the same family often reuses considerable portions of their codebase and possesses common behavioral characteristics. Permissions requested by an application gives a hint of what the resulting behavior might likely be. Thus the combination of these two distinctive features creates a unique and robust signature for known malware.

The result of our analyses illustrated that we can correctly detect and categorize malware variants with an F-measure of 98.9% and our system is resilient to even complex obfuscation schemes, such as reflection, name and string encryption, junk code insertion, and code reordering when compared to the current state of the art in Android malware detection tools, including both commercial antivirus and research tools.

Acknowledgment

This work was partially funded by the NSF grant, CNS #1409534.

8. REFERENCES

[1] VirusTotal online malware scan service.

[2] Forsafe Mobile Security android.anserver. http://encyclopedia.foresafe.com/2012/12/androidanserver_18.html, 2012.

[3] Crussell, J., Gibler, C., and Chen, H. Attack of the clones: Detecting cloned applications on android markets. In *Computer Security âĂŞ ESORICS 2012*, vol. 7459 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 37–54.

[4] Deshotels, L., Notani, V., and Lakhotia, A. Droidlegacy: Automated familial classification of android malware. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014* (2014), PPREW'14, pp. 3:1–3:12.

[5] Enck, W., Ongtang, M., and McDaniel, P. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (2009), CCS '09, pp. 235–245.

[6] Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (2011), CCS '11, pp. 627–638.

[7] Feng, Y., Anand, S., Dillig, I., and Aiken, A. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), FSE 2014, pp. 576–587.

[8] Fuchs, A. P., Chaudhuri, A., and Foster, J. S. Scandroid: Automated security certification of android applications. Tech. rep., University of Maryland, 2009.

[9] Grace, M., Zhou, Y., Zhang, Q., Zou, S., and Jiang, X. Riskranker: Scalable and accurate zero-day android malware detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2012), MobiSys '12, pp. 281–294.

[10] Hanna, S., Huang, L., Wu, E., Li, S., Chen, C., and Song, D. 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* (Berlin, Heidelberg, 2013), DIMVA'12, pp. 62–81.

[11] Heffner, K., and Collberg, C. The obfuscation executive. In *Information Security*, vol. 3225 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004, pp. 428–440.

[12] Huang, H., Zhu, S., Liu, P., and Wu, D. A framework for evaluating mobile app repackaging detection algorithms. In *Trust and Trustworthy Computing*, vol. 7904 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 169–186.

[13] Rastogi, V., Chen, Y., and Jiang, X. Catch me if you can: Evaluating android anti-malware against transformation attacks. *Information Forensics and Security, IEEE Transactions on 9*, 1 (Jan 2014), 99–108.

[14] Santos, I., Brezo, F., Nieves, J., Penya, Y., Sanz, B., Laorden, C., and Bringas, P. Idea: Opcode-sequence-based malware detection. In *Engineering Secure Software and Systems*, vol. 5965 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010, pp. 35–43.

[15] Schmidt, A.-D., Bye, R., Schmidt, H.-G., Clausen, J., Kiraz, O., Yuksel, K., Camtepe, S., and Albayrak, S. Static analysis of executables for collaborative malware detection on android. In *Communications, 2009. ICC '09. IEEE International Conference on* (June 2009), pp. 1–5.

[16] Tustison, N. J., and Gee, J. C. Introducing dice, jaccard, and other label overlap measures to itk. *The Insight Journal* (July-December 2009), 1–4.

[17] Wu, D.-J., Mao, C.-H., Wei, T.-E., Lee, H.-M., and Wu, K.-P. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on* (Aug 2012), pp. 62–69.

[18] Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang, X. S., and Zang, B. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (2013), CCS '13, pp. 611–622.

[19] Zheng, M., Lee, P., and Lui, J. Adam: An automatic and extensible platform to stress test android anti-virus systems. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, vol. 7591 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 82–101.

[20] Zheng, M., Sun, M., and Lui, J. C. S. Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware. In *Proceedings of the 2013 12th IEEE International Conference on Trust, Security and*

Privacy in Computing and Communications (2013), TRUSTCOM '13, pp. 163–171.

[21] Zhou, W., Zhou, Y., Grace, M., Jiang, X., and Zou, S. Fast, scalable detection of "piggybacked" mobile applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy* (2013), CODASPY '13, pp. 185–196.

[22] Zhou, W., Zhou, Y., Jiang, X., and Ning, P. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy* (2012), CODASPY '12, pp. 317–326.

[23] Zhou, Y., and Jiang, X. Dissecting android malware: Characterization and evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy* (2012), Oakland '12.

[24] Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the 19th Network and Distributed System Security Symposium* (2012), NDSS '12.