OpSeq: Android Malware Fingerprinting

Aisha Ali-Gombe[1], Irfan Ahmed[1], Vassil Roussev[1], and Golden Richard III[1]

University of New Orleans LA, 70148

Abstract. Often Android malware are created by injecting payload into a benign application, then redistributing the app to different markets. These malware tend to hide their presence using various form of code and string obfuscation. Recent study has shown, most antivirus software and static analysis tools are not resilient to such transformations. To solve this problem we develop a more resilient algorithm that can detect known malware with minimal footprints and can deal with most complex obfuscation with a high degree of accuracy. Our approach (OpSeq) scores similarity as a function of normalized opcode-sequence in susceptible functional modules and permission requests. The combination of structural and behavioral features gives a distinctive identification to a malware sample, thereby improving our model's overall recall rate. We tested our prototype on 1192 known malware samples, 359 benign and 167 new obfuscated variants, and the results showed OpSeq can correctly detect known malware with an F-Score of 98.9%.

1 Introduction

Most signature based malware detection systems are created to perform comparison check and seldom deal with any level of obfuscation. They search for common signatures developed using API calls, permissions request, simple opcode-sequences and strings. However, trivial obfuscation can render such simple classification attempts ineffectual. Reflection, encryption, code reordering and junk insertion are some of the ways in which malware can disguise its functionality and evade detection. In Android domain, repacking benign app with obfuscated payload is the commonest way of propagating malware. Rastogi et al have shown how trivial transformations of android malware can pose a tremendous setback on the detection system of most antivirus products and by extension many research tools [13].

Our goal is to develop a more resilient approach with various level of obfuscation in mind. Rather than detecting overall application similarity, we aim to contently isolate possible malware functionality out of whole application and create signature based on it. We shift the paradigm to the combine analyses of structural and behavioral features of malware. Structural features are extracted from the most basic logical unit of code; function body (Java method), whereas an app's possible behavior is gathered from its set of permissions in the manifest.

Our system differ from the existing opcode-sequence based similarity DroidMoss and Juxtapp[10,22] in both objective and methodology of feature extraction.

While these two techniques treat an application as a jumbo stream of bytes, in which sub-streams of fixed size "n" are generated, our opcode-sequences follows a more logical pattern. We extract opcodes from individual java methods after first decompiling an app into respective classes and functions. This allows us to get group of opcode that together define certain functionality.
Out of all the functions available in an application code, we chose only those that exhibit certain characteristics like invoking important permission−guarded API, reflection API, source and sink of data etc which we term as susceptible functional modules. We denote the full list of such API's as Sensitive API list. Keeping in mind that our major goal is to isolate malicious code from a whole application code, the susceptible functional module give us hint of what most likely constitute malware payload. In Android system, access to device resources are only available through the framework by invoking certain important API's. Thus the code areas where such calls are made serves as a vital source of information on a malware functionality.
Before signature generation, we normalized all the extracted opcode-sequences and then group each sequence into n−gram opcode. Our 3-step signature matching employs an algorithm that determine inclusion rather than mere comparison. We use Targeted Overlap Coefficient to find the n−gram opcodes similarity within two functions. Next we use Dice Coefficient to calculate the ratio of matched functions. And lastly the final−similarity index is determine as a function of the Dice Coefficient and permission's Targeted overlap Coefficient
Our method of signature generation and matching allows detecting even complex obfuscated malware with different level of transformations. Our work is evaluated on malware variants from 25 different families. We also examine the effect of different obfuscation techniques, comparing our work with some state of the art research tool and antivirus products. Our experiments with OpSeq indicate the system is capable of detecting variants of known malware with an F-Score of 98.9%, 98% recall and 99% precision.
Contribution:
The main contribution of this work is development of an Android malware signature-based detection tool OpSeq. This system is:

1. Effective: detect known malware payload both in standalone or repackaged apps using the combination of small structured signature.
2. Resilient: efficiently identify distinct features of malware footprints that are repackaged in a complexly obfuscated applications.
3. Accurate: maintain a high level of correctness in classifying malicious and benign app and in sub categorizing malware into distinctive individual families.

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

2 Related Work

The first large-scale study of Android malware (Genome Project) was carried out by [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 other research including ours. Their work identified that 86% of Android malware are found as repackaged apps.

2.1 Opcode-Sequence Similarity

[14] developed a system of detecting malware using opcode−sequence frequencies in traditional x86 systems. On android platform, [10,22] came up with the methodology of using opcode−sequences in detecting repackaged Android applications both in primary and secondary app markets. As mentioned earlier, our work differs from theirs in terms of goals and approach. The focal point of their work is specifically to detect application repackaging rather than malware. If for example a benign application is repackaged with a small payload of malware, their system will most likely detect the similarity between the original benign app and the new app than between the new app and a malware. Their opcode−sequences can easily be distorted with little noise, whereas our work is distinguished in its attempt to deal with different obfuscation techniques that can be employed to hamper similarity metrics. More so, our methodology examines the combination of code and permissions in creating signatures, theirs only code sequences are analyzed.

2.2 Semantic-Based Detection

This group uses semantic information flow as features in detecting similarity between Android applications [3,4,7,15,17,20,21]. PiggyApps [21] first identified the code containing main functionality (primary module) in legitimate apps. Then they extract and organize this semantic information from the module as a vantage point tree. These signatures are then use to scalably search for piggybacked apps in Android markets. Apposcopy [7] on the other hand, manually creates signatures for known malware using a specification language. To find similarity, it extracts semantic features of the new app using inter-component call graphs and static taint analysis. DroidLegacy [4] determines malicious modules, and then creates malware signatures from the cluster of the shared code found within malware class variant. DroidAnalytics [20] uses a three-level signature that represents API calls made from within apps. API calls sequences forms signature for methods, and the collection of all method signatures form signature for class. And finally in turn class signatures are arranged to form application signatures. All these techniques can easily be 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. They created a set of rules that defines which permissions combination might be dangerous. Another permission−based behavioral fingerprinting is DroidRanger [24]. However, [6] explains how most Android apps are over privileged in general and even benign apps do request the combination of some dangerous permissions. SCanDroid [8] is a security certification tool that determines if specifications in manifest matches 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 technique attempt to discover if dangerous behavior is present, while OpSeq's main goal is to measure similarity of unknown apps against known malware.

3 Design

Simple to complex obfuscation have been shown to trick antivirus products [13,19] and by extension the methodology of many other android research tools. Most of these algorithms traverse application code in search of strings and names, which can easily be subverted. Our aim is to develop a better system that can detect footprint of a known malware obfuscated and repackaged within a new application.
The workflow of our system Op-Seq as illustrated in Figure 1 constitutes three major steps, feature extraction, signature generation and signature matching. The two important features we extract are; permissions request and functional opcodes. Android permissions are written in an xml file, which by standard has to follow certain rules, thus its content cannot be obfuscated. The importance and un-obscure nature of android permissions made it a vital feature for our algorithm. However, the content of this file can only give a faint idea of the behavior of an application. More so, having two unrelated applications with same permission request is not very unlikely, therefore permission alone cannot be a reliable measure for similarity. Thus the need for our second feature; function opcode. OpSeq generates profile for every malware sample using list of permissions request and normalized opcode-sequences extraction from the application one per each functional module. And finally to determine similarity with a new sample, its signature is also generated and compare with our stored profile based on a 3-level process which are; Pattern−level similarity, Function−level similarity and Final−similarity index.

3.1 Feature Extraction

In this phase, permission request and functional opcodes are extracted from android xml file and classes.dex file respectively. A classes.dex file *"cd"* is a set

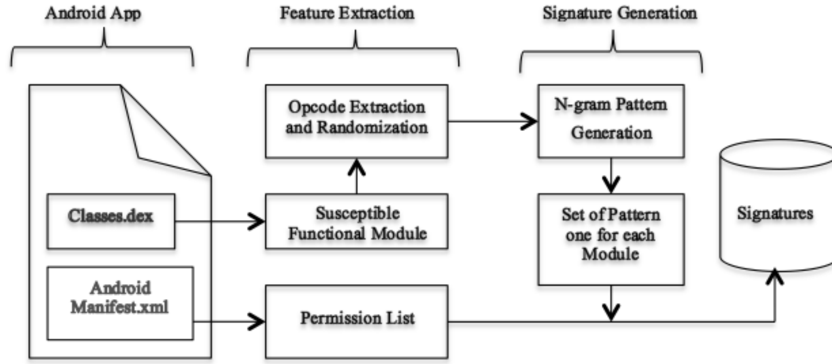

Fig. 1. OpSeq Signature Generation Workflow

of java classes *"jc"* where $cd = \{jc1, jc2, \ldots, jcm\}$. Each java class *"jc"* is made up of functions *"f"* where $jc = \{f1, f2, \ldots, fn\}$.Thus we can say *"cd"*: is set of functions *f*, $cd = \{f1, f2, \ldots, fm\}$. Each *"f"* in *"cd"* is a sequence of instruction *"I"*, where $f = \{I1, I2, \ldots, Ip\}$. Every instruction *"I"* is a tuple containing the opcode and operand, $I = \{opcode{:}operand\}$. For our feature set we concentrate only on the opcode portion of the Dalvik byte instruction. Each *"I"* represent one opcode. Thus every function *"f"* in *"cd"* is made up of a sequence of opcodes *"o"*.

$$\forall\, f \in cd{:}\ f = \{o1,\ o2, \ldots,\ op\} \tag{1}$$

Before extracting each opcode−sequence, OpSeq requires that certain criteria must be met. The function must at least invoke one of he APIs in the Sensitive API list and it is not in Freq−sequence. Freq−Sequence is the list most commonly found opcode sequences. In other to reduce noise and improve the effectiveness of our signatures, it imperative to eliminate such common sequences from the signatures.
Thus, a function *"f"* is a member of extracted opcode-sequence list *"S"* if and only if one or more of its instructions invokes a Sensitive API and the function *"f"* is not an item in the Freq−sequence list.

$$f \in S \ iff \ (\ (p \in A\ \&\ I\ invokes\ p)\ \&\ I \in f)\ \&\ f \notin fs \tag{2}$$

Where:
f = function (javamethod)
S = set of extracted opcode-sequences
A = Sensitive API list
p = API call
I = Instruction
fs = list of frequently used opcode-sequences
We also decode AndroidManifest.xml file into set of permission

$$perm = \{\ M1,\ M2, \ldots,\ Mn\ \} \tag{3}$$

We now have set of extracted opcode-sequences *"os" and "perm"* that satisfied our condition above.

3.2 Signature Generation

Our signatures are formed by carefully taking into account the type of obfuscation that can affect opcode−sequence based malware detection in particular and all other forms of obfuscations in general. Code junk insertion is an obfuscation technique that can embed pool of unused instruction, i.e., a branch of code that may not be traversed. More so, malware variants can also remove or substitute instructions. Code reordering, on the hand, moves around non−dependent instructions. These techniques are mostly aimed at breaking hash−based detection. From S above, all the extracted opcode−sequences are normalized. This distorts the order of opcode arrangement and group similar opcode in same cluster. Next for each normalized opcode sequence, we generate sub−sequences of n−gram opcodes. In choosing the best value for n, we run our system with 1−gram, 2−gram and 3−gram. Our result indicates 2−gram gives the best accuracy. This step is essential because, if we do a comparison of the entire stream, the system will likely be affected by simple trick. However breaking down the sequence into small streams has shown to tremendously reduce the effect of such code-level obfuscation. Thus for naming convention, each sequence (representing one function) containing k number of 2−gram opcodes is referred to as *Pattern*.

$$S = \{P1, P2, \ldots, Py\} \tag{4}$$

where each;

$$P = \{os1,\ os2, \ldots,\ osk\} \tag{5}$$

Depending on number of susceptible functions found, a signature for known profile *"S"* is made up of set of patterns *"P"*, where each pattern contains set of 2−gram opcodes. This denotes structural features for the familial malware.
Our technique allowed for similarity to be measured from the basic unit of code upward. Program similarity becomes an aggregate of individual functional similarity and as such the likelihood of determining relationship between two related of code increases.
We know variants of malware may not be exact copy of one another, but our assumption is most of their malicious functionality and code structure remains similar. Malware can add, remove or substitute code within a function. But in other to retain it's key behavior, some part of the code has to be maintained. Thus by carefully analyzing each function as a single unit and normalizing its opcode, our algorithm can ascertain if relationship exist between two functions in different applications.

3.3 Similarity Matching

Our similarity matching is a 3-step process that begins with the Pattern−level similarity, then Function−level similarity that determine a score for all the

matched functions. Lastly the Final−similarity index scores similarity as function of the Function−level similarity and permission overlap.

Pattern−level similarity Given a signature "S_a" generated for a malware "*A*", let a test sample application be denoted as "*B*". We use the process above to generate signatures for "*B*" as "S_b". Thus:

$$\begin{aligned} S_a &= \{P_{a1},\ P_{a2},\ldots,P_{ay}\} \\ P_a &= \{os_{a1},\ os_{a2},\ldots,os_{ai}\} \\ S_b &= \{P_{b1},P_{b2},\ldots,P_{bz}\} \\ P_b &= \{\ os_b1,\ os_b2,\ldots,os_bj\} \end{aligned} \tag{6}$$

For each "P_a" in "S_a" , we determine its best match in "S_b" using **Targeted Overlap Coefficient of the "**P_a . This calculates the ratio of common 2−gram opcode found in the intersection of "P_a" and "P_b" to the size of "*Pa*" . This is vital in determining how "P_a" relates "P_b". The Targeted Overlap Coefficient from $P_a \rightarrow P_b$ is define as:

$$R(P_a \rightarrow P_b) = |P_a \cap P_b|/|P_a| \tag{7}$$

This Target overlap coefficient denoted by "*R*" is ideal in our case because we specifically target "P_a". Since our goal is to determine how "*Pa*" relates to "*Pb*", we then require a threshold value "*T*" (define as Pattern-Level threshold **PLT**) that connotes minimum acceptable similarity ratio. In this step, If R $(P_a \rightarrow P_b) \geq T$ then we write "*R*" to a buffer "*BUF*" and we eliminate both "P_a" and "P_b" . While if R $(P_a \rightarrow P_b) < T$ then "P_a" is eliminated while "P_b" remains. This loop continues until all the patterns in "S_a" are compared to patterns in "S_a" (Algorithm 1).
The pattern−level similarity is measured by the power of the "*R*". The coefficient "*R*" lies between 0 and 1. As "*R*" tends to 1, it means most 2−gram opcode found in "P_a" are also present in "P_b", hence "P_a" is similar to "P_b". However as "*R*" tends towards 0, the similarity from "P_a" to "P_b" diminishes. Note that the similarity score calculated is not transitive. "P_a" is a pattern from our known profiles while "P_b" is a pattern in a test sample; the idea is to calculate how close "P_a" is to "P_b" and not vice versa. A positive outcome at this level of matching can be attributed to one of the following reasons. If "P_a" and "P_b" are modules with the same functionality then "*R*" will be close to 1. It is also possible "P_b" is a disguised version of "P_a" that have been polluted with junk but still retain most of its opcodes. In this situation too we can get a match. More so, "P_a" and "P_b" can match to a certain degree even though "P_a" and "P_b" are not derived from same functional module, this will create a false positive. However our analysis results have shown this to be quite rare.
This pattern level−matching algorithm is effective in overcoming the effect of junk insertion and reordering obfuscation techniques.

Function−level similarity The step analyze all the results generated in pattern−level matching. As shown above, for each matched pattern, the derived coefficient is stored in a buffer *"BUF"*.

$$BUF = \{R1,R2,\ldots,Rk\} \tag{8}$$

This 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 some sample sets and measured the results. The *Sorensen-Dice Coefficient* gave us the best result. Briefly, the Dice coefficient is a measure of intersection between two given sets scaled by their size. In particular, it gives more weight when there is an intersection than Jaccard coefficient.

$$Dice\ Coefficient\ D = 2 * |S_a \cap S_b| / |S_a + S_b| \tag{9}$$

Given *"T"* (pattern−level threshold) as a value close to 1, this means each *"R"* in *"BUF* is also close to 1. Thus;

$$|S_a \cap S_b| \approx \sum BUF \tag{10}$$

Thus;

$$D = 2 * (\sum BUF)\ /\ len\ |S_a| + len|S_b| \tag{11}$$

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

Final−Similarity Index The final similarity score is calculated based on the result of function−level matching and the permission overlap. We first need to calculate the permission overlap using the Targeted Overlap Coefficient.

$$\begin{aligned} &permA = permission\ list\ in\ A \\ &permB = permission\ list\ in\ B \\ &PO\ (A \rightarrow B) = |permA \cap PermB| / |permA| \end{aligned} \tag{12}$$

Where PO = Permission overlap from A to B
Again this gives us the ratio of similar permissions used in both known sample *"A"* and *"B"* against the length of permissions in *"A"* . The permission overlap is a weight that strengthens the result of the function−level matching. It tells us what behavior is common between *"A"* and *"A"* . If two apps contain the same malware footprint, they normally should have some common permissions. The overlap is a value between 0 and 1.
Given functional-level coefficient as *"D"*, the final Similarity score (Algorithm 3) is given as:

$$SS = D * PO \tag{13}$$

Algorithm 1 Pattern-Level Matching

```
function :(PLM(S_a, S_b, T))
    for P_a in S_a do:
        for P_b in S_b do
            inter=multi_intersect(P_a, P_b)
            coef = len(inter) / len(P_a)
            if coef ≥ T then:
                append(coef, BUF)          ▷ append coefficient to buffer
                remove(P_b, S_b)           ▷ remove pattern P_b from S_b
            end if
        end for
    end for
    return BUF
end function
```

Algorithm 2 Function-Level Matching-Dice coefficient algorithm

```
function :(FLM(S_a, S_b, BUF))
    suM=sum(BUF)
    funAvg= 2/ (len(S_a)+len(S_a)
    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)               ▷ Permissions targeted overlap coefficient
    if perm ¿ 0 then
        SS=perm*D
    end if
    return SS                              ▷ SS returns Final similarity Index
end function
```

As *"PO"* get closer to 0, *"SS"* becomes low and that means app *"A"* and app *"B"* have less similar behavior. However *"PO"* close to 1 means app *"A"* and *"B"* contain common permissions, which strengthen the functional similarity index *"D"*. The final similarity score *"SS"* indicate how much two applications match based on our extracted features. A minimum similarity index MSI is required to determine if the coefficient SS is good enough. Thus when $SS \geq MSI$ implies footprint of particular malware is present in the test app.

4 Implementation and Evaluation

We implemented a prototype of our approach in Python, using apktool [1] as our disassembler. Apktool is an open-source Android reverse engineering tool that disassembles classes.dex into an intermediate representation (smali) and decodes

apk resources. Our system comprises modules for feature extraction, signature generation and similarity index.

4.1 System Requirement

Before executing OpSeq, the system requires a Sensitive API and the Freq−Sequence list. The processing of these list is a one time step needed before the analysis of our corpus. As mentioned earlier in section 3, the Sensitive API list serves as roadmap to susceptible functional modules. The list is developed based on careful and extensive analysis of predominant android malware attack vectors. This APIs includes permission−protected, reflection, sources and sink of data, dynamic and native class loading APIs.
The Freq−Sequence on the other hand, is the list of most commonly found opcode−sequences. The accuracy of our system dependent on the uniqueness of our generated sequences. This list is created by frequency analysis of functional opcode−sequence extracted from 2800 applications containing both malware and benign samples. We define noise as opcode−sequence that appear at least once in every application. Sequences that constitute noise are eliminated from our feature set.

4.2 DataSet

Our dataset contains 1718 applications; 1192 known malware from the Android Genome project [23] classified into 25 families, 359 applications downloaded from Google Play and 167 new variants of our malware classes that are obfuscated using DroidChameleon [13]. DroidChameleon is an open source Android application transformation/obfuscation research tool. Our newly created variants were developed using four important modules in DroidChameleon; encryption, code reordering, junk-code insertion and reflection.
Out of the 49 families in the malware genome project, we choose 25 that have at least 2 or more variants. The performance of our system is measured based on its ability to detect footprint of known malware in an application and categorize it to its family. We also measure the effectiveness of our approach on the obfuscated sample viz−a−viz antivirus products and other research tools. The experiment addresses several concerns:

1. **Malware/Benign apps detection (Mal/Ben):** Based on 1192 Genome project samples and 359 Benign google play samples, can the system accurately flag an application repackaged with known malware?
   True Positive (TP): all malware are correctly detected.
   False Positive (FP): all benign apps that are wrongly detected as malware.
   True Negative (TN): all benign apps that are not flagged as malware.
   False Negative (FN): all malware that are not detected.
2. **Malware class detection (Mal_Class):** Can the system categorize the malware detected from *"a"* above into their respective families?
   True Positive (TP): all malware that are correctly categorized.

Table 1. Model Selection Result: F−Score, Precision, Recall

Model	False Positive	False Negetive	True Positive	True Negetive	Precision	Recall	Interpolated Precsion	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

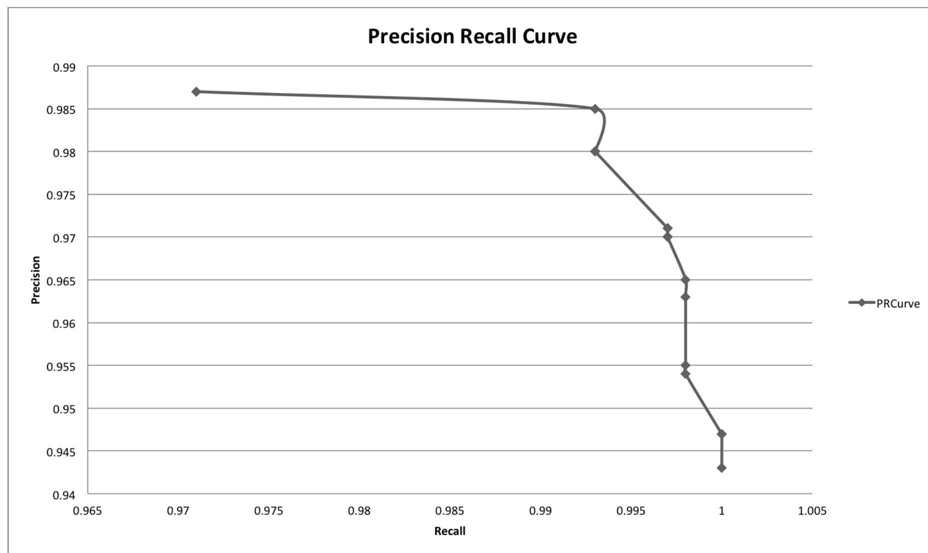

Fig. 2. Precision & Recall Curve

False Positive (FP): all malware that are wrongly categorized as variants of different family.
False Negative (FN): all malware that were not detected.

3. **Obfuscation**: Can the system detect the footprint of a known malware after it has been obfuscated and repackaged using DroidChameleon?

4.3 Optimum Variables

In section 3, we have identified two variables for OpSeq's algorithm; Pattern−Level threshold (PLT) − this is minimum overlap ratio required to assume similarity between two 2−gram patterns from different apps(in Pattern−level similarity). Minimum similarity index (MSI) − is the minimum score that determines if malware footprint is present in an application (in Final−similarity index). To get the optimum values of PLT and MSI, we choose some arbitrary values and plug 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 variable are chosen based on the statistics of Mal/Ben detection.
In choosing the best values, we use F−Score statistics. F−Score measures the overall accuracy of a test, which depends on the precision and recall. Recall is the measure of accuracy that a specific class has been detected (% of correct malware family is 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 that highest F−score is the optimum solution. We run OpSeq on our dataset using the above combination of PLT and MSI. The result of our execution is shown in Table 1. Each row (a model) represent one combination. We then calculate the statistics (false positive, false negative, true positive, true negative, precision, recall and F-score) for each model. The result indicates highest F−Score is attained at PLT of 80% and 3% MSI, and hence our optimum solution. This model gives us 99.3% precision, 98.5% recalls and F-Score of 98.9% for Mal/Ben detection. Using same metric on Mal/Class, we get F-Score of 97.5%, 98% recall and 97% precision. Thus our system is capable of accurately detecting malware 98.9% of the time and can categorize the malware into its correct Family/Class with 97.5% accuracy. The confusion matrix for Mal/Ben and Mal/Class based on the optimum values is shown in Table 2 and Table 3 respectively.
More the rationale for choosing our optimum values is further buttress in the precision/recall curve of our solution as shown in 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 point of maximum accuracy.

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 Mal/Class of 97.5%.
However, comparing DroidLegacy's recall rate of 94%, which is the true positive rate versus ours of 98% indicate our system is capable of better detection. More

Table 2. Mal/Ben Best Model Confusion Matrix

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

Table 3. Percentage of Mal/Class Prediction Result

Malware Family	False_Negetive	False_Postive	True_Positive	Total
ADRD	0	0	1	22
Anserverbot	0.02	0.04	0.94	305
BeanBot	0	0	1	8
Bgserv	0	0	1	9
CruseWin	0	0	1	2
DroidDream	0.06	0.06	0.88	16
DroidDreamLight	0.09	0	0.91	47
DroidKungFu	0.01	0.04	0.95	442
Geinimi	0	0	1	69
GingerMaster	0	0	1	4
GoldDream	0	0	1	47
Gone60	0	0	1	9
GPSSMSSpy	0	0	1	6
HippoSMS	0	0	1	4
jSMSHider	0	0	1	13
KMin	0	0	1	52
Pjapps	0.02	0.03	0.95	58
Plankton	0	0.09	0.91	11
RogueLemon	0	0	1	2
RogueSPPush	0	0	1	9
SndApps	0	0	1	10
Tapsnake	0	0	1	2
YZHC	0	0	1	22
zHash	0	0	1	11
Zsone	0	0	1	12
Total	18	35	1139	1192

so, our system has better coverage in terms of malware families processed; they processed 11 families against ours with 25 families.

False Negative: Malware from Anserverbot, DroidKungFu and DroidDreamLight constitute the bulk of our false negative prediction as shown in Table 3. One reason for the false negatives can be traced to our signature generation. 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 malicious behavior. For instance, the sample we use to generate signature for DroidDreamLight is a newer version than the rest of its variants. The newer sample has about 98 functions that were extracted for the signature against 18 for the older version, which means more sensitive API calls are been made in the former than the latter. Since our similarity calculates overlap based on the known profile, this sample was a miss.
Another reason for some false negative can be attributed to native code exploit. Sample that have most its malicious code written in native code will often result in a miss detection e.g. some few variants of DroidKungFu and Anserverbot. Reason been that OpSeq is only design to handle dex file.
False Positives: For Mal/Ben, 8 applications out of 359 downloaded from the Google Play were detected as malware by OpSeq as shown in Table 2. In order to confirm true nature of these 8 applications, we run them in Virus Total [16]. The output flagged 3 out the 8 apps as malware. Thus 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 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 miss-categorize as DroidKungFu. That sample contains the Adware.waps, while the sample use to generate the signature for the Anserverbot family does not. However, the adware on its own collects numerous user data from a devices and invoke 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 same adware. Since OpSeq assigned class based on the strength of the final similarity index, that Anserverbot sample was flagged as DroidKungFu.

4.5 Testing Obfuscation

Our approach can detect footprint of familial malware obfuscated and repackaged within another application. Using DroidChameleon, we generated 167 obfuscated variants of the malware in our sample set. This new variants have varying degree of transformations from simple to complex. In this segment of evaluation we chose 4 techniques;
Encryption: names, strings and field encryption
Reflection: normal API calls are change from direct call to using helper classes
Junk Insertion: Addition of noise instructions
Code Reordering: altering the flow of program execution by changing the positions of unrelated instructions.
In simple obfuscation, an app is obfuscated with just one technique, while in complex obfuscations; apps get obfuscated with two or three techniques. For simple obfuscation, 24 samples were encrypton, 25 samples were reflected, junks

were added in 24 samples and code reordering in 24 samples. For complex obfuscation, 23 samples had Encryption and Reflection combined, 24 had (Encryption, Reflection and Reordering) combined and finally 23 had (Encryption, Reflection and Junk) combined.

Results against AntiVirus OpSeq scores 100% average detection rate in simple obfuscation and 88% in complex transformations. Using same sample, we assess the detection ratio of other antivirus products using virus total. Out of the top 15 antivirus products; in the simple class, 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 class, AhnLab-V3 leads other antivirus with a detection rate of 49.33%, then AVG with 27%. All the rest recorded less than 25%. Table 4 and 5 showed the detail scores for the simple and complex obfuscation respectively.

Result against DroidLegacy We also run the same obfuscated variants 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. Like most common detection research tools, DroidLegacy depends on API names to create signatures for malware. In situation were the name is obfuscated using reflection, chances of detection become very low. This explains why it recorded 0% for all apps that were repackaged with reflection. More so, it did not do well with encryption alone too, detecting only 6 out of 24 encrypted apps.

5 Discussion

The results of our experiments show the resiliency of OpSeq in detecting Android malware variants. We illustrate how effectively our system can detect malware footprints, even with complex obfuscation in place. 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, eventhough we generate a signature for each, 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 our system only processes the dex file, thus initial analyses flags one as been variant of the other. Information from Foresafe encyclopedia [2] and analysis result of some antivirus products in Virus Total also affirm their relationship; hence we merge them into one class.
The limitations of our system include parsing native code, dynamic payload and

Table 4. Evaluation result for simple obfuscation

	Encrption	Reflection	Reordering	Junk	Average Detection Rate
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
Total No. of Sample	24	25	24	24	

Table 5. Evaluation result for complex obfuscation

	Encrption& Reflection	Encryption Reflection & Reordering	Encryption Reflection & Junk	Avgerage Detection Rate
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
Total No. of Sample	23	24	23	

complete dex file encryption. OpSeq only extracts features from visibly available classes.dex file. Like most static analysis tools, extra classes that are fully encrypted cannot be processed. And as such, malware that have significant segment of its malicious functionality in native code or loads extra classes at runtime cannot be detected.
Code reordering that can split functions into multiple sub functions may affect the effectiveness of our approach. Excessive junk instructions can introduce noise which may affect the quality of our signatures too. However these 2 obfuscation technique can only pose significant ill effect when more than one susceptible malicious functions are tempered, which in practice will require significant human intervention. More so, since our approach cluster common opcodes together by way of 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. This uniques opcode have to be very different from those used within the functions.

6 Conclusions

In this paper, we developed a more resilient approach for statically detecting Android malware variants. Our system generates signatures for familial malware as a function of the normalized opcode-sequence found in susceptible functional modules and permissions requested. Malware belonging to the same family often reuse considerable parts of their codebase and possess some common behavioral characteristics. Permissions requested by an application give a hint of what its behavior might likely be. Thus the combination of these two distinctive features creates a very unique and robust signature for known malware.
Our thorough approach, which involves finding these functions, processing their instruction sequences and the overall statistics of finding similarity makes this algorithm very distinct and effective.
The result of our analysis showed we can correctly detect and categorize malware variants with F-measure of 98.9% and our system is by far resilient to complex obfuscation like reflection, name and string encryption, code junk insertion and reordering when compared to the state of the art available anti-viruses and research tools..

References

1. Apktool. Android reverse engineering tool https://code.google.com/p/android−apktool/. Online; accessed September 21, 2014.
2. Forsafe Mobile Security. Android.Anserver. http://encyclopedia.foresafe.com/2012/12/androidanserver_18.html. Online;accessed September 21, 2014.
3. Crussell, J., Gibler, C., & Chen, H. "Attack of the clones: Detecting cloned applications on android markets." In Computer Security?ESORICS 2012 . Springer Berlin Heidelberg., 2012.

4. Deshotels, L. Notani, V, Lakhotia, A. "DroidLegacy: Automated Familial Classification of Android Malware." PPREW'14 Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014. 2014.
5. Enck, W., Ongtang, M., & McDaniel, P. " On lightweight mobile phone application certification." In Proceedings of the 16th ACM conference on Computer and communications security. ACM, November, 2009. (pp. 235−245).
6. Felt, A. P., Chin, E., Hanna, S., Song, D., & Wagner, D. "Android permissions demystified." In Proceedings of the 18th ACM conference on Computer and communications security . ACM, October, 2011. (pp. 627−638).
7. Feng, Y., Anand, S., Dillig, I., & Aiken, A. "Apposcopy: Semantics−Based Detection of Android Malware through Static Analysis." In SIGSOFT FSE., 2014.
8. Fuchs, A. P., Chaudhuri, A., & Foster, J. S. "SCanDroid: Automated security certification of Android applications ." Technical Report, Univ. 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 . June, 2012. (pp. 281−294).
10. Hanna, S., Huang, L., Wu, E., Li, S., Chen, C., & Song, D. "Juxtapp: A scalable system for detecting code reuse among android applications." Edited by Springer Berlin Heidelberg. In Detection of Intrusions and Malware, and Vulnerability Assessment. 2013. (pp. 62−81).
11. Manning, C., D., Raghavan, P., & Schtze, H. Introduction to Information Retreival. Cambridge University Press, 2008.
12. O'Connor, Brendan. AI and Social Science. April 11, 2012. http://brenocon.com/blog/2012/04/f−scores−dice−and−jaccard−set−similarity/ (accessed 2014).
13. Rastogi, V. Chen, Y. and Xuxian, J. "Catch me if you can: Evaluating android anti−malware against transformation attacks." IEEE Transactions on Information Forensics and Security 9 (2014): 99−108.
14. Santos, I., Brezo, F., Nieves, J., Penya, Y. K., Sanz, B., Laorden, C. and Bringas, P. G. "Idea: Opcode−sequence−based Malware Detection." Engineering Secure Software and Systems − LNCS 5965 (2010): 35−43.
15. Schmidt, A. D., Bye, R., Schmidt, H. G., Clausen, J., Kiraz, O., Yuksel, K. A., ... & Albayrak, S. "Static analysis of executables for collaborative malware detection on android." In Communications, 2009. ICC'09. IEEE International Conference. IEEE, June 2009. (pp. 1−5).
16. VirusTotal. Online malware scan service. https://www.virustotal.com/en/.
17. Wu, D. J., Mao, C. H., Wei, T. E., Lee, H. M., & Wu, K. P. Droidmat: Android malware detection through manifest and API calls tracing. In Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference. IEEE, August, 2012. (pp. 62−69).
18. Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., X., Wang, S.and Zang, B.B. "Vetting undesirable behaviors in android apps with permission use analysis." In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM, November), 2013. (pp. 611−622).
19. Zheng, M., Lee, P. P., & Lui, J. C. "ADAM: an automatic and extensible platform to stress test android anti−virus systems." In Detection of Intrusions and Malware, and Vulnerability Assessment. Springer Berlin Heidelberg., 2013. (pp. 82−101).
20. Zheng, M., Sun, M., Lui J. C. S. "DroidAnalytics: a signature based analytic system to collect, extract, analyze and associate Android malware." Trust, Security and Privacy in Computing and Communications (TrustCom), 12th IEEE International Conference. 161−173, 2013.

21. Zhou, W., Zhou, Y., Grace, M., Jiang, X., & Zou, S. " Fast, scalable detection of piggybacked mobile applications." In Proceedings of the third ACM conference on Data and application security and privacy. ACM., February 2013. (pp. 185−196).
22. Zhou, W., Zhou, Y., Jiang, X., & Ning, P. "Detecting repackaged smartphone applications in third−party android marketplaces." Edited by ACM. In Proceedings of the second ACM conference on Data and Application Security and Privacy. February 2012. (pp 317−326).
23. Zhou, Y. and Jiang, X. "Dissecting Android Malware: Characterization and Evolution." 33rd IEEE Symposium on Security and Privacy. 2012.
24. Zhou, Y., Wang, Z., Zhou, W., & Jiang, X. "Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets." In NDSS. February, 2012.