DroidClone: Detecting Android Malware Variants by Exposing Code Clones

Shahid Alam, Ryan Riley
Department of Computer Science and Engineering
Qatar University, Doha, Qatar
Email: {salam,ryan.riley}@qu.edu.qa

Ibrahim Sogukpinar, Necmeddin Carkaci
Department of Computer Engineering
Gebze Technical University, Gebze, Turkey
Email: {ispinar,ncarkaci}@bilmuh.gyte.edu.tr

***Abstract*—According to the Symantec threat report the total number of new malware variants added in 2013 and 2014 were 252 millions and 317 millions (a 26% increase from 2013) respectively. Mobile malware development in 2013 and 2014 continues to focus exclusively (∼99%) on the Android platform. For detecting malware, if parts of a malware family match parts of a program then this provides us a strong evidence that the program is/contain a malware. Based on this hypothesis, we propose *DroidClone* that exposes code clones (segments of code that are similar) in Android applications to help detect malware variants. *DroidClone* uses a new Malware Analysis and Intermediate Language (MAIL) for finding code clones in Android applications. MAIL helps *DroidClone* to use specific control flow patterns for reducing the effect of obfuscations and provides automation and platform independence. Unlike other works *DroidClone* is able to detect both bytecode and native code Android malware variants. When tested with traditional malware variants it achieves a detection rate (DR) of 97.85%, compared to the other two works *DroidSim* and *NiCad* that achieved a DR of 89.62% and 83.11% respectively.**

I. Introduction

Clones are segments of code that are similar according to some definition of similarity [1]. There are various reasons of finding code clones in software. Some of them are plagiarism detection, copyright infringement investigation, code compaction (e.g., for mobile devices), and detecting bugs and malicious software (malware) [2]. All these tasks require the extraction of syntactically or semantically similar code fragments. For detecting malware, if parts of a malware family match parts of a software then this provides us a strong evidence that the software is/contain a malware. Mobile malware development in 2013 [3] and in 2014 [4] continues to focus exclusively (∼99%) on the Android platform. Based on these motivations in this paper, we develop techniques for detecting Android malware variants by exposing code clones.

One of the basic techniques used by a malware writer is obfuscation [5]. Such a technique obscure a code to make it difficult to understand, analyze and detect malware embedded in the code. This technique is also used to create **variants** of the same malware. According to the Symantec threat report [6] the total number of new malware variants added in 2013 and 2014 were 252 millions and 317 millions (a 26% increase from 2013) respectively. Some of the previous studies [7, 8, 9] have evaluated the resistance of state of the art commercial antimalware products to Android malware variants of known malware samples. In the study on 6 malware samples performed in [7] none of the transformed malware samples were detected by the products tested. The detection rate results of the products tested in the study on 222 malware samples performed in [8] range from 50.95% – 76.67%. The results reported in [9] also show that most of the antimalware products were not able to detect transformed malware.

There are several researches [10, 11, 12, 13, 14] that have focused on Android malware variants detection. Some of these, such as [10] and [11] use API call graphs, [13] uses components based API calls to find code reuse, [12] uses byte sequence features and [14] uses a text-based near-miss source code clone detector. We believe using control flow patterns is a more general technique for malware analysis and detection than using API call patterns, and it is difficult for a malware to change control flow patterns than changing API call patterns of a program, for evading detection. Byte sequence features and other such techniques has the ability to detect closely similar data objects, but there ability to detect malware variants is much lower.

In this paper, we expose code clones for detecting Android malware variants. For this purpose, we utilize the new intermediate language MAIL (Malware Analysis Intermediate Language) [15] that helps us use specific control flow patterns to reduce the effect of obfuscations and unlike [13] can detect malware variants even with smaller graphs. Our technique, unlike [14], can detect malware variants that are syntactically different but semantically similar upto a certain threshold. A malware writer has to employ an excessive control flow obfuscation to create a variant to evade detection by such an anti-malware. Excessive here means changing the control flow of a program beyond a certain percentage (threshold). In general such threshold is difficult to find, but is calculated for each different dataset.

We also provide Android native code malware detection by finding Android native code clones. Native code referred to in this paper is the machine code that is directly executed by a machine, such as ARM or Intel processors, as opposed to bytecode, which is executed in a virtual machine such as JVM or Dalvik-VM. While the standard programming model for Android applications is to write the application in Java and distribute the bytecode, the model also allows for developers to include native code binaries as part of

their applications. This is frequently done in order to make use of a legacy library or to write code that is hardware dependent, such as for a game graphics engine or video player.

The contributions of this work are as follows:

- This paper proposes DroidClone that exposes code clones to find Android malware variants.
- DroidClone designs cross-platform signatures for Android and operates at the native code level, allowing it to detect malware embedded in either bytecode or native code.
- DroidClone achieves far better results than the other two techniques compared and reviewed in this paper.

The remainder of this paper is organized as follows. Section II presents a brief background. We present our approach, its design and implementation in Section IV. Section V presents the evaluation and comparison of our approach with two other such works. We present related work in Section III and finally conclude in Section VI.

II. Background

Android applications are available in the form of APKs (Android application packages). An APK file contains the application code as Android bytecode and precompiled native binaries (if present). Traditionally, those applications are executed on the phone using a customized Java virtual machine called Dalvik [16]. Dalvik employs just-in-time compilation (JIT), similar to the standard JVM.

Starting from Android 5.0, the Dalvik-VM was replaced by the ART [17]. ART uses ahead of time (AOT) compilation to transform the Android bytecode into native binaries when the application is first installed on the device. ART's AOT compiler can perform complex and advanced code optimizations that are not possible in a virtual machine using a JIT compiler. ART improves the overall execution efficiency and reduces the power consumption of an Android application, but increases the installation time and may increase the code size of an application. In this work, we use ART in order to translate bytecode into native code that can be analyzed using our techniques.

We build specific control flow patterns of Android native binaries and compare them with the previous such control flow patterns of malware of this kind to find code clones.

A. MAIL CFG

Malware Analysis Intermediate Language (MAIL) is a new language first introduced in [15]. MAIL provides platform independence and automation for malware analysis and detection. It provides a higher level representation as well as patterns for optimized analysis of a native program.

A CFG (control flow graph) is a directed graph G = (V, E), where V is the set of basic blocks [18] and E is the set of control flow edges [18]. The CFG of a program represents all the paths that can be taken during program execution. A MAIL CFG is a CFG such that each statement of the CFG is assigned a MAIL Pattern. In this paper MAIL CFG is used to build signatures of Android native binaries for finding code clones and malware.

B. Patterns for Annotation

There are a total of eight basic statements (e.g, assignment, control and conditional, etc) in MAIL that can be used to represent the structural and behavioral information of an assembly program. There are 21 Patterns in the MAIL language and every MAIL statement can be tagged by one of these patterns. A pattern represents the type of a MAIL statement and can be used for easy comparison and matching of MAIL programs. For example, an assignment statement with a constant value and an assignment statement without a constant value are two different patterns. The details of the patterns can be found in [15].

III. Related Works

A very detailed survey of the research done on code clones is presented in [2]. In this paper, we only describe two of the recent efforts on detecting Android malware using code clones.

Chen et al. [14] presents a technique that uses NiCad [19], a near-miss clone detector, to detect Android malware. First they develop signatures from a subset of malware applications by finding clone classes in these applications, and then use these signatures to find similar malware applications in the rest of the malware applications. Their clone detector works at the Java source code level, and hence is not able to detect native Android clones. NiCad compares the source code linewise using an optimized *longest common subsequence* algorithm to detect similar clones. This is a good text-based technique, but may not be efficient for detecting malware variants. For example it may be defeated just by changing the names of the functions and variables, and hence may not be able to detect clones that are syntactically different but semantically similar.

Sun et al. [13] presents a technique using component-based control flow graph (CBCFG). CBCFG is a graph of Android APIs as nodes and there control flow precedence relationship as edges. These CBCFGs are then used to detect code reuse in Android repackaged applications and malware variants. CBCFG may not be able to detect malware applications that obfuscate by using fake API calls, hiding API calls (e.g, using class loading to load an API at runtime), inlining APIs (e.g, instead of calling an external API, include the complete API code inside the application), and reflection (i.e, creation of programmatic class instances and/or method invocation using the literal strings, and a subsequent encryption of the class/method name can make it impossible for any static analysis to recover the call).

In general, changing (obfuscating) control flow patterns is more difficult (i.e, it needs a comprehensive change in the program) than changing just API call patterns of a program to evade detection. API based techniques look for specific API call patterns (including call sequences) in Android

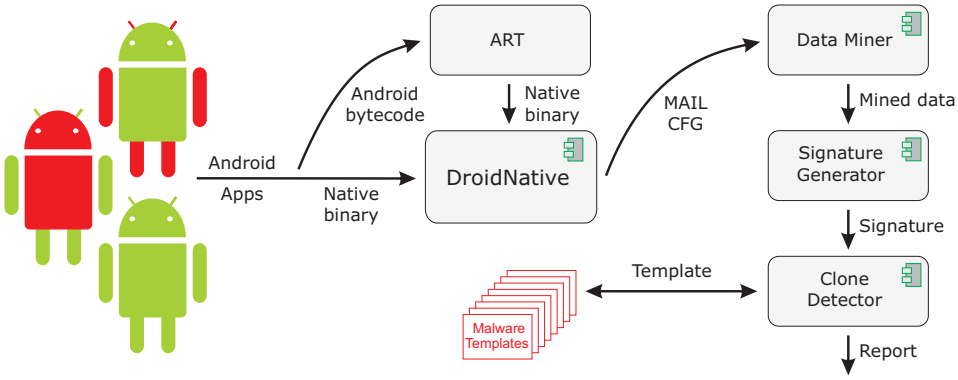

Fig. 1: Overview of DroidClone.

malware programs for there detection, which may also be present in Android benign applications that are protected against reverse engineering attacks. These API call patterns can be packing/unpacking, calling a remote server for encryption/decryption, dynamic loading, and system calls, etc.

IV. Overview of Our Approach

Given the sheer number of Android applications available, detecting malware among them needs to be automated. As part of accomplishing this, we make use of a new intermediate language MAIL [15] to provide an abstract representation of an Android native program. This representation is used for finding code clones that helps in malware detection. By translating native binaries compiled for different platforms (Android supports both x86 and ARM architectures) to MAIL, we achieve platform independence and use MAIL Patterns to optimize malware analysis and detection.

Figure 1 provides an overview of the *DroidClone* architecture. As explained before Android applications are present either in bytecode or native code. The Android applications in bytecode are translated to native code using the ART. Then we use the tool DroidNative [20] to translate the native code to a MAIL program which is transformed to a CFG, called MAIL CFG as explained in Section II-A.

Next Sections discuss why and how we use and adapt, MAIL for malware analysis and detection of Android code clones and malware variants.

A. Signature Generation of a MAIL Program

This Section explains how MAIL Patterns are used to generate the signature of a MAIL program.

DroidNative [20] as shown in Figure 1 is used to generate CFG of a MAIL program. Each block in the CFG contains MAIL statements and a MAIL CFG of an Android binary (benign or malware) contains these blocks. One of the examples of such a MAIL CFG (of a malware sample) is listed in Table I.

A block in a CFG starts with the symbol `START` and ends with the symbol `END`. There are total 6 blocks in the MAIL program listed in Table I that are parsed using MAIL patterns into block strings as follows:

```
block 1: ACACSSASAAACACACJC, block 2: ACJC,
block 3: ACJS, block 4: ACJ, block 5: JC,
block 6: AC
where:
ASSIGN_C ⇒ AC, ASSIGN ⇒ A,  JUMP  ⇒ J,
JUMP_C   ⇒ JC, JUMP_S ⇒ JS, STACK ⇒ S
```

A MAIL program consists of a set of functions. To build the signature of a MAIL program, we first build the signature of each function in the set of functions. A function ends with a symbol `EOF` and consists of a set of blocks. The algorithm we use to build the MAIL signature of a function is as follows: We first fetch the pattern of the first and the last statement of each block in the set of blocks in the function and reduce the pattern string as explained above (`ASSIGN_C` ⇒ `AC`). We also build the block string for each block in the function as explained above. Each function's MAIL signature contains a function signature (string) and a set of block signatures (strings).

There are 2 functions in the MAIL program listed in Table I. The signature of these functions are:

```
Function 1:
{ACJCACJCACJS, [ ACACSSASAAACACACJC, ACJC, ACJS ]}
Function 2:
{ACJJCAC, [ ACJ, JC, AC ]}
```

For the function 1, `ACJCACJCACJS` is the function string and `{ACACSSASAAACACACJC, ACJC, ACJS}` is the set of it's three block signatures. For the function 2, `ACJJCAC` is the function string and `{ACJ, JC, AC}` is the set of it's three block signatures.

These two function signatures together become the signature of the MAIL program listed in Table I. This signature contains part of the CFG of a MAIL program and hence represents upto a certain level the semantics of the program. This greatly helps us in detecting variants of the same program (malware/benign). Signature of each function is considered as a segment of a code in a MAIL program and is used to find code clones. As stated before, for detecting malware if parts of a malware family match parts of a software then this provides us a strong evidence that the software is/contain a malware. Based on this hypothesis, we expose these code clones in Android applications to help us in detecting malware variants.

TABLE I: CFG (control flow graph) of a MAIL program generated by DroidNative [20].

Num	Offset	MAIL Statement	Pattern	Block/ Function	Jump To
0	1018	r12 = sp - 8192;	[ASSIGN_C]	START	
1	101c	r12 = [r12, 0];	[ASSIGN_C]		
1	101c	r12 = [r12, 0];	[ASSIGN_C]		
2	1020	[sp=sp+0x1] = r5;[sp=sp+0x1] = r6;[sp=sp+0x1] = lr;	[STACK]		
3	1024	sp = sp - 20;	[STACK]		
4	1026	r6 = r0;	[ASSIGN]		
5	1028	[sp, 0] = r0;sp = sp - 0x1;	[STACK]		
6	102a	r5 = r1;	[ASSIGN]		
7	102c	r0 = r6;	[ASSIGN]		
8	102e	r0 = [r0, 12];	[ASSIGN_C]		
9	1030	r1 = r5;	[ASSIGN]		
10	1032	r0 = [r0, 12];	[ASSIGN_C]		
11	1034	lr = [r0, 40];	[ASSIGN_C]		
12	1038	jmp 0x1046;	[JUMP_C]	END	
13	103a	r4 = r4 - 1;	[ASSIGN_C]	START	
14	103c	jmp 0x1046;	[JUMP_C]	END	1046
15	1040	sp = sp + 20;	[ASSIGN_C]	START	
16	1042	r5 = [sp=sp-0x1];r6 = [sp=sp-0x1];pc = [sp=sp-0x1];	[JUMP_S]	END **EOF**	
17	1046	lr = [r9, 560];	[ASSIGN_C]	START	
18	104a	jmp lr;	[JUMP]	END	
19	104c	jmp 0x1040;	[JUMP_C]	START END	1040
20	104e	r0 = r0 << 0;	[ASSIGN_C]	START END **EOF**	

B. Training

For malware detection, we train our classifier on the known malware dataset. During training we build a list of malware signatures to be used latter for classification. To optimize the performance and runtime of the classification model, we store only the distinguish functions' signature of malware samples for classification. Algorithm 1 describes how we build our list of signatures during training.

The method `generateSig(sample)` shown in Algorithm 1 generates the signature of a MAIL program (sample) as described in Section IV-A. The `simScore(s1, s2)` method finds the largest common substring (LCS) in the two signatures (strings) and returns the percentage of the length of the LCS in relation to the larger signature (string).

The method `buildSigs()` shown in Algorithm 1 builds the list of signatures (`sigs`) that is used during malware/clone detection. This list contains all the distinguished functions found in the malware samples. This method first generates the signature for the sample. Then it iterates through all the function signatures of the sample and append the signature to the list (`sigs`) if the signature is not found in the list.

The method `sigMatch()` shown in Algorithm 1 only compares the function signatures during training when the list (`sigs`) is build, but compares both the function and block signatures during malware/clone detection. We believe function signatures are enough to build our training model in an efficient way. This also optimizes the training time.

Time complexity of Algorithm 1 in worst case is $\frac{n(n-1)}{2}$, where n is the number of signatures in the list of *sigs*.

Algorithm 1 Building signatures for classification

```
buildSigs(malwareSamples, threshold)
sigs = []            /* list of sigs to be built */
foreach sample in malwareSamples
   sig = generateSig(sample)
   foreach f in sig
      if sigs.isEmpty()
         sigs.append(f)
      elif sigMatch(f,sigs,True,threshold) = False
         sigs.append(f)
return sigs

sigMatch(sampleSig, sigs, isTraining, threshold)
foreach sig in sigs
   f1 = sig[0]                  /* function sig */
   f2 = sampleSig[0]            /* function sig */
   if simScore(f1, f2) ≥ threshold
      if isTraining
         return True
      b1 = sig[1]          /* set of block sigs */
      b2 = sampleSig[1]    /* set of block sigs */
      len = min(b1, b2)
      match = True
      bn = 0
      while bn < len
         if simScore(b1[bn],b2[bn]) < threshold
            match = False
            break
         bn++
   if match = True
      return True
return False
```

C. Malware/Clone Detection

For malware detection, we classify a sample as either malware or benign. We first build a list of malware signatures that consists of distinguished functions based on a threshold as described in Section IV-B. This list is used for malware classification. We compare an Android application's signature with each of the signature in the list of malware signatures, comparing the function as well as the block signatures of the function. For comparing signatures (stored as strings), we use

TABLE II: Class distribution of the 166 malware samples

Class / Family	Number of samples
DroidKungFu1	30
DroidKungFu2	30
DroidKungFu3	30
DroidKungFu4	30
DroidDream	16
DroidDreamLight	30

an optimized version of *largest common substring* comparison based on a threshold. If a certain number (threshold) of functions in the application's signature are found (clones) in the list of malware signatures, then the application is classified as malware as shown in Algorithm 2.

Algorithm 2 Malware/Clone detection

```
detectClone(sampleSig, sigs, desiredSim, threshold)
m = 0
lenSigs = len(sigs)
foreach f in sampleSig
   if sigMatch(f,sigs,False,desiredSim) = True
      m++
      simScore = m / lenSigs
      if simScore ≥ threshold
         return True
return False
```

Time complexity of Algorithm 2 in worst case is n^2, where n is the number of signatures in the list of *sigs*.

V. Experimental Evaluation

We carried out an empirical study to analyse the correctness and the efficiency of our approach. We present in this section the evaluation metrics, the empirical study, obtained results and analysis.

A. Dataset

Our dataset for the experiments consists of total 246 Android applications. Of these, 166 are Android malware programs collected from [21] and the other 70 are benign programs containing Android 5.0 standard applications. Table II lists the class/family distribution of the 166 malware samples. The 4 *DroidKungFu* variants make use of obfuscations to evade detection, such as different ways of storing the command and control server addresses and change of class names, etc. *DroidDream* and *DroidDreamLight* also contain number of variants through obfuscations. These malware samples were chosen to test the ability of *DroidClone* to detect variants of malware and also compare the proposed technique in this paper with the other two techniques discussed in Section III.

B. Evaluation Metrics

Before evaluating the proposed technique, we first define the following evaluation metrics. We define two metrics TP rate and FP rate. The TP rate also called detection rate (**DR**) corresponds to the percentage of samples correctly recognized as malware out of the total malware dataset. The FP rate (**FPR**) metric corresponds to the percentage of samples incorrectly recognized as malware out of the total benign dataset. These two metrics are defined as follows:

$$DR = \frac{TP}{P} \qquad FPR = \frac{FP}{N} \tag{1}$$

where TP is the number of malware that are classified as malware, FP is the number of benign samples that are classified as malware and, P and N are the total number of malware and benign samples respectively.

Accuracy is the fraction of samples, including malware and benign, that are correctly detected as either malware or benign. This metric is defined as follows:

$$Accuracy = \frac{TP + TN}{P + N} \tag{2}$$

where TN is the number of benign samples that are classified as benign.

C. Experiments and Results

We carried out two experiments to evaluate DroidClone and compare it with the other two works discussed in Section III. All experiments were run on an Intel® Xeon® CPU E5-2670 @ 2.60GHz with 128 GB of RAM, running Ubuntu 14.04.1. The first experiment was carried out to evaluate the performance of DroidClone based on the evaluation metrics described in Section V-B.

For the first experiment, out of the 166 malware samples 136 were used for training and the rest of the 30 were used for testing. We also tested 70 benign samples for false positives. DroidClone was able to achieve a DR of 90% and a FPR of 8.5%. Out of the 30 malware samples 27 were detected as malware, and only 6 out of 70 benign samples were detected as malware. DroidClone achieved an accuracy of 91%. This shows that DroidClone was able to detect correctly (both malware and benign samples) 91% of the time.

We favor DR over precision, because we do not want to miss on any malware and expect (in real-life) that all applications labeled as malware will go through further analysis, such as a human analyst or a cloud based malware detector, where it is possible to carry out more deep (both static and dynamic) analysis of these applications. Currently we are working on improving this aspect of *DroidClone* to make it a completely practical and scalable malware detection system.

1) Comparison with Other Techniques: The second experiment was carried out to compare DroidClone with the other two works discussed in Section III. These works tested there detectors only with malware samples and not with benign samples. Therefore to present a fair comparison with them, we also tested our detector with only the available malware samples (166) and not the benign samples from the dataset. The results of this experiment as detection rate per malware family are presented in Table III. Except the two malware families (DroidKungFu3 and DroidKungFu4) the number of samples in the other malware families tested by DroidSim and NiCad are almost the same as tested by DroidClone.

TABLE III: Comparison with two other recent Android malware clone detection techniques discussed in Section III.

Malware Family	Detection Rate (%)		
	DroidClone	DroidSim [13]	NiCad [14]
DroidKungFu1	100.0	100.0	95.83
DroidKungFu2	96.66	53.57	100.0
DroidKungFu3	100.0	99.66	99.66
DroidKungFu4	100.0	91.66	90.69
DroidDream	93.75	92.85	87.50
DroidDreamLight	96.66	100.0	25.00
All	97.85	89.62	83.11

The results as shown in Table III of *DroidClone* are superior compared to the other two works. Unlike *DroidClone*, none of the other two techniques *DroidSim* or *NiCad* can handle Android native code. The other two techniques did not test their detectors for FPR, which in addition to DR is a major criteria for a detector to be used in real life.

The FPR of DroidClone, while reasonable, is higher than we would like. In the future, we will investigate reducing the FPR by training the current classifier using commonalities not only among the malware samples, but also among the benign samples, and between the malware and benign samples. We are also investigating alternative classifiers such as Naive Bayes and Support Vector Machine, etc. The challenge here will be to update DroidClone without increasing its run time per sample.

VI. Conclusion

In this paper, we have proposed *DroidClone* that exposes code clones and detect both bytecode and native code Android malware variants. DroidClone uses the language MAIL to provide automation and platform independence; and uses control flow with patterns to reduce the effect of obfuscations. DroidClone shows superior results for the detection of Android malware variants compared to the other research efforts.

In future work, we will make DroidClone more resilient to other control flow obfuscations through normalization and improve it's runtime through parallelization. We will also explore the possibility of using DroidClone as part of a complete system, such as a cloud-based malware analysis and detection system. We would also like to increase the training database, i.e, number of malware and benign samples, and test against various Android obfuscation tools.

Acknowledgment

This paper was made possible by NPRP grant 6-1014-2-414 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

References

[1] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Software Maintenance, 1998. Proceedings., International Conference on*, pp. 368–377, IEEE, 1998.

[2] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," tech. rep., 541, Queens University at Kingston, ON, Canada, 2007.

[3] *Symantec cyber and Internet security threat report*, 2014.

[4] *F-Secure mobile threat report*, Q1 2014.

[5] C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations," tech. rep., University of Auckland, 1997.

[6] *Symantec cyber and Internet security threat report*, 2015.

[7] V. Rastogi, Y. Chen, and X. Jiang, "DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks," in *ASIA CCS*, pp. 329–334, ACM, 2013.

[8] M. Zheng, P. P. Lee, and J. C. Lui, "ADAM: an automatic and extensible platform to stress test android anti-virus systems," in *DIMVA*, pp. 82–101, Springer, 2013.

[9] P. Faruki, A. Bharmal, V. Laxmi, M. Gaur, M. Conti, and M. Rajarajan, "Evaluation of Android Anti-malware Techniques against Dalvik Bytecode Obfuscation," in *TrustCom*, pp. 414–421, 2014.

[10] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs," in *CCS*, pp. 1105–1116, ACM, 2014.

[11] L. Deshotels, V. Notani, and A. Lakhotia, "DroidLegacy: Automated Familial Classification of Android Malware," in *SIGPLAN*, pp. 1–12, ACM, 2014.

[12] P. Faruki, V. Laxmi, A. Bharmal, M. Gaur, and V. Ganmoor, "Androsimilar: Robust signature for detecting variants of android malware," *Journal of Information Security and Applications*, vol. 22, pp. 66–80, 2015.

[13] X. Sun, Y. Zhongyang, Z. Xin, B. Mao, and L. Xie, "Detecting Code Reuse in Android Applications Using Component-Based Control Flow Graph," in *ICT*, pp. 142–155, Springer, 2014.

[14] J. Chen, M. H. Alalfi, T. R. Dean, and Y. Zou, "Detecting android malware using clone detection," *Journal of Computer Science and Technology*, vol. 30, no. 5, pp. 942–956, 2015.

[15] S. Alam, R. N. Horspool, and I. Traore, "MAIL: Malware Analysis Intermediate Language - A Step Towards Automating and Optimizing Malware Detection," in *SIN*, pp. 233–240, ACM SIGSAC, 2013.

[16] *Dalvik*. http://en.wikipedia.org/wiki/Android_Runtime.

[17] *ART*. http://en.wikipedia.org/wiki/Android_Runtime.

[18] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley, Inc., 2006.

[19] J. R. Cordy and C. K. Roy, "The nicad clone detector," in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pp. 219–220, IEEE, 2011.

[20] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi, *DroidNative: Semantic-Based Detection of Android Native Code Malware*. http://arxiv.org/abs/1602.04693, 2016.

[21] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Security and Privacy*, pp. 95–109, IEEE, 2012.