ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/351970319

BLADE: Robust malware detection against obfuscation in android

Article *in* Forensic Science International Digital Investigation · September 2021

DOI: 10.1016/j.fsidi.2021.301176

CITATIONS
34

READS
501

3 authors:

Vikas Sihag
Sardar Patel University of Police, Security and Criminal Justice, Jodhpur
46 PUBLICATIONS 317 CITATIONS

SEE PROFILE

Manu Vardhan
National Institute of Technology Raipur
93 PUBLICATIONS 1,345 CITATIONS

SEE PROFILE

Pradeep Singh
National Institute of Technology Raipur
87 PUBLICATIONS 1,389 CITATIONS

SEE PROFILE

All content following this page was uploaded by Vikas Sihag on 31 May 2021.

The user has requested enhancement of the downloaded file.

BLADE: Robust Malware Detection against Obfuscation in Android.

Vikas Sihag[a,b,*], Manu Vardhan[b], Pradeep Singh[b]

[a]*Sardar Patel University of Police, Security and Criminal Justice, Jodhpur, India*
[b]*National Institute of Technology, Raipur, India*

Abstract

Android OS popularity has given significant rise to malicious apps targeting it. Malware use state of the art obfuscation methods to hide their functionality and evade anti-malware engines. We present BLADE, a novel obfuscation resilient system based on Opcode Segments for detection. It makes three contributions: Firstly, a novel Opcode Segment Document results in feature characterization resilient to obfuscation techniques. Secondly, we perform semantics based simplification of dalvik opcodes to enhance the resilience. Thirdly, we evaluate effectiveness of BLADE against different obfuscation techniques such as trivial obfuscation, string encryption, class encryption, reflection and their combinations. Our approach is found effective, accurate and resilient, when tested against benchmark datasets for malware detection, familial classification, malware type detection, obfuscation type detection and obfuscation resilient familial classification.

Keywords: Android, Malware detection, Code obfuscation, Familial classification

1. Introduction

Android OS since its release in 2008, has grown as the most preferred choice in the market with 72.26% share of 3.8 billion smartphone users worldwide

*Corresponding author

Email addresses: `vikas.sihag@policeuniversity.ac.in` (Vikas Sihag), `mvardhan.cs@nitrr.ac.in` (Manu Vardhan), `psingh.cs@nitrr.ac.in` (Pradeep Singh)

in July 2020 [1]. Android's popularity and its application distribution model tenders to new attack surfaces targeting user's privacy and security [2]. Recently among top 5000 Android apps on Play Store, 655 were found having zero-days and 983 with known vulnerabilities [3]. Mobile attacks by cyber criminals have increased from backdoors and crypto mining to click farming, ad fraud and fake reviews using malicious applications (aka Apps). Malicious activities comprises of information leakage, device failure or data corruption with selfish or harmful motives.

Malware researchers are adopting state of the art application stealth techniques such as advanced code obfuscation and protection mechanisms to evade anti-malwares [4, 5, 6]. Current malwares are enhanced with code obfuscation, encryption, dynamic loading and/or native code execution techniques to prevent app reversal [7, 8].

The process of understanding the functionality and infection of a malware is popularly known as *Malware Analysis*. It is generally classified into static (code) analysis and dynamic (behavioral) analysis. Static approach analyzes code sequences without executing them, whereas dynamic approaches the run time execution [9, 10]. *Static analysis* is light weighted and has high code coverage as compared to dynamic analysis [7, 11]. *Dynamic analysis* executes and monitors an application, to track its behaviour, understand features and identify technical indicators that can be used as detection signatures [12, 13, 14]. Malware analysis is generally tasked to detect an executable sample as malicious (i.e. malware detection) or to identify which malware family does it belong to (i.e. familial classification). App stealth techniques poses challenge towards efficient malware detection and familial classification [15].

Obfuscation comprises of actions that modifies an App code without changing its functionality or semantics [16]. Obfuscation techniques can be classified into trivial and non-trivial. Trivial techniques do not perform code level changes, where as the non-trivial does. Trivial obfuscation methods such as repackaging are used to attach malicious code(s) in legitimate apps. 86% of malware samples were found to be using these methods [17]. Identification of malicious component in repackaged app is a challenge for malware analysis. Non-trivial obfuscation methods such as class encryption, string encryption, identifier renaming, code reordering, reflection etc. modifies the code semantics thus preventing analysis and evading detection systems. For instance, Listing 1 shows code fragment from DroidDream and its corresponding code 2 after identifier renaming. Semantic changes induced by obfuscation methods can easily evade signature based classification.

```
const-string v15, "profile"
const-string v16, "mount -o remount rw system\nexit\m"
invoke-static {v15, v16}, Lcom/android/root/Setting;->runRootCommand(Ljava/
    lang/String;Ljava/lang/String;)Ljava/lang/String;
move-result-object v10
```

Listing 1: A bytecode fragment from DroidDream malware.

```
const-string v15, "profile"
const-string v16, "mount -o remount rw system\nexit\m"
invoke-static {v15, v16}, Lcom/hxbvgH/IWNcZs/jFAbKo;->axDnBL(Ljava/lang/
    String;Ljava/lang/String;)Ljava/lang/String;
move-result-object v10
```

Listing 2: The bytecode after performing identifier renaming on listing 1.

To address above challenges we propose `BLADE` (roBust maLwAre DEtection system), a novel obfuscation resilient approach based on opcode segments. We first generate `.smali` files of an input `APK` (an Android executable), followed by Dalvik opcode [18] sequences from `.smali` files. As Dalvik opcodes represents behavioral pattern of an application, it is then used to generate opcode sequences using simplification. Opcode sequences are then segmented to represent an APK as an Opcode Segments Document (OSD). Furthermore, OSD is used for malware detection and familial classification.

In short, the main contributions are summarized below:

– Opcode Segment Document: We analyzed Android applications from a different perspective and proposed an Opcode Segments Document (OSD) based novel approach for malware characterization.

– `BLADE`: We propose `BLADE`, an efficient and effective malware detection and familial classification system based on OSD.

– Obfuscation Resilient Evaluation: We evaluated effectiveness of `BLADE` against popular obfuscation techniques such as trivial obfuscation, string encryption, reflection, class encryption and their combinations.

– Typically Android apps contain single DEX file, but some may comprise of multiple DEX files. `BLADE` is able to handle these complex apps, by extracting features from multiple DEX files.

– Scalable Detection: We evaluated and compared `BLADE` over bench mark datasets. It is effective and accurate for malware detection, familial classification and is obfuscation resilient. Overall, it achieves better performance when compared with other state of the art approaches based on several aspects.

Paper organization: In Section 2, we describe Dalvik bytecode and obfuscation methods in Android apps as the background required for the proposed work. Section 3 elaborates working and design principles of BLADE. Section 4 defines research questions and evaluates the performance of BLADE against them. Section 5 contrasts the proposed work with the existing state of the solutions. Furthermore, related works is discussed in 6, followed by conclusion and future direction in section 7.

2. Background

In this section we discuss the preliminary background knowledge required for our work. We discuss Dalvik bytecode (Section 2.1) and popular obfuscation techniques (Section 2.2).

2.1. Dalvik Bytecode

Android has a distinct executable machine code format called Dalvik Bytecode. Source code java `.class` files along with other `.jar` library files are converted into dalvik executable `classes.dex` file. It along with compiled resources and shared object (`.so`) files is then compressed into an Android PacKage (APK) file. This APK file is downloaded for installation, when requested from Google Play Store. A `classes.dex` file contains definitions of multiple classes, with each comprising of multiple methods. While `classes.dex` is a non-readable binary file, it can be disassembled into smali files, which are intermediate human readable format. Smali code generated from Dalvik bytecode comprises of classes and its methods in each smali file. Each method contains register based instructions and each instruction consists of an operation code and its operand(s). For instance, the instruction `move-wide/from16 vBB, vAAAA` has `move` as the base opcode, `wide` (64-bit data) as the name suffix, `from16` (16-bit register reference) as the opcode suffix, `vBB` as the destination register and `vAAAA` as the source register. Dalvik opcode constant lists 237 opcodes of which only 217 are used in practice in APKs [19]. Being human readable Dalvik bytecode is easier than machine code. Tools such as Androguard [20], APKTool [21], and Dexdump are popular reverse engineering tools to extract APK dex code.

2.2. Android Application Obfuscation

In our context, the term *obfuscation* refers to transformation of an application executable (`APK`) without altering its original functionality. Obfuscation techniques employed by Android applications is a double-edged sword for analysts as it protects legit developers against code cloning as well the malware authors against a range of analysis engines [22]. Following popular obfuscation techniques pose challenge to malware analysis.

Trivial Obfuscation: It defines obfuscation methods which affects the strings, but the executable instructions in bytecode. It comprises of renaming files, fields, classes, methods and packages with random or predefined nomenclature. It also includes repackaging of the APK.

Repackaging: In repackaging, an APK is unpacked, re-packed and signed with a new key to generate repackaged app. Popular applications are inserted

Table 1: Comparative analysis of Android application obfuscation tools

Tool	Repackaging	Flow Obfuscation	String Encryption	Class Encryption	Resource Encryption	Reflection
Allatori [26]	✓		✓			
APK Protect [27]	✓		✓	✓		
Arxan	✓		✓	✓	✓	
DexGuard [28]	✓	✓	✓	✓	✓	✓
DashO [29]	✓	✓	✓		✓	
DexProtector [30]	✓	✓	✓	✓	✓	✓
Ijiami	✓	✓	✓	✓		
Mobile Protector [31]	✓	✓	✓	✓	✓	
ProGuard [32]	✓	✓	✓	✓	✓	✓
Promon Shield [33]	✓	✓	✓	✓	✓	✓
Stringer [34]	✓		✓	✓	✓	

with malicious code and repackaged to be hosted on market places posing challenge for user to verify its authenticity.

Control Flow Obfuscation: It is the process of rearrangement of instructions in a method, to evade control flow analysis of instructions. This include instruction patterns used by reverse-engineering tools to decompile the source code.

String Encryption: Strings often reveal malware identifiable information such as names or URLs. String encryption could obstruct hard-coded string based searching by rendering strings unreadable [22, 23]. In it original string is stored in an encrypted form and requires an additional decryption function.

Class Encryption: Its an advanced code obfuscation technique which encrypts a class. The encrypted class is decrypted and loaded at runtime by a separate function. The computational overhead of class encryption is high along with its resilience against static analysis [24].

Reflection: Reflection is a popular feature in Java to allow object interaction at runtime. It is popular among developers to obfuscate sensitive library and API calls [25]. It transfers execution flow to the desired code segment implicitly.

Resource Encryption: Resources and assets are used by malware for payload or code hiding. This technique encrypts the application resources and are decrypted during execution. For example, Rootnik malware encrypted its resource file to `secData0.jar` file [5].

A comprehensive analysis of Android application obfuscation tools with ref-

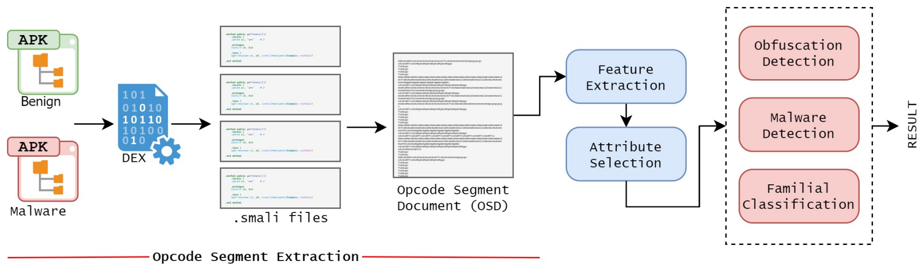

Figure 1: Architecture of the proposed approach.

erence to their features and techniques is illustrated in table 1. Tools listed are popular among developers used for applications hardening [5].

3. Design of BLADE

Overview

We convert the problem of malware detection and familial classification to a document classification problem. For a text document, characters are its basic building blocks. Ordered set of characters form words, sentences and paragraphs. We develop an Android malware detection system BLADE, which represents an application as a document with opcode characters as its building blocks. BLADE is resilient to obfuscation and has high accuracy on malware detection and familial classification. Proposed approach includes two procedures. Prior is to create the detection and classification model. It follows with prediction of an application for malware detection, familial classification and obfuscation detection. Its overall architecture is illustrated in figure 1.

Malware detection training set comprises of malware apps and benign apps. Training set for familial classification includes malware samples of different family subsets. Training set for obfuscation detection comprises of malware samples into different obfuscation types. For obfuscation detection training set, we have considered trivial obfuscation (T), string encryption (S), reflection (R), class encryption (C), trivial + string encryption (TS), trivial + string encryption +

reflection (TSR) and trivial + string encryption + reflection + class encryption (TSRC).

As shown in the architectural diagram, `APK` sample to be predicted is preprocessed to extract its DEX bytecode file, which is then used to extract `.smali` files. Each smali file specifies methods and fields. Intermediate opcode sequences are generated from `.smali` files. opcode sequences are simplified and segmented to give opcode segments. An application thus is represented as an Opcode Segment Document (OSD). Each OSD is a collection of opcode segments, which are then reduced and selected for detection. Furthermore, this model is used for obfuscation detection and familial classification.

Obfuscation techniques mentioned in 2.2 are a challenge towards malware detection. Our proposed solution mitigates some of these threats.

Opcode simplification and OSD generation

Proposed approach represents each malware sample with Opcode Segment Document (OSD) generated from its DEX code. As outlined in 2.1, DEX code represents instruction level operation code. We decompile and extract `.smali` files from DEX code using APKTool [21]. We analyze `.smali` files and have grouped multiple instructions from them based on their usage. Instruction performing same operation but on different register indices are considered similar. For example, both Dalvik instructions `"move vA, vB"` and `"move/from16 vAA, vBBBB"`, move contents from one register to another; the difference is number of bits of registers to move. All instructions based on their semantics are attributed into 19 symbolic groups. Table 2 establishes symbols attributed to 224 dalvik instructions. For example, symbol `A` represents all instruction of arithmetic operations like `add-int`, `add-int/2addr` or `sub-int`. Instruction `nop` is responsible for no operation are not allotted any symbol, thus if encountered are skipped. This grouping of similar opcodes (dalvik instructions) based on semantics is defined as *Opcode Simplification*. Opcode simplification results into an application represented as a collection of opcode sequences.

Table 2: Symbolic representation of Dalvik instruction set

Semantics	Opcode prefixes	Number	Symbol
Arithmetic	add-double | add-int | add-float | add-long | div-double | div-int | div-float | div-long | mul-double | mul-int | mul-float | mul-long | rem-double | rem-int | rem-float | rem-long | sub-double | sub-int | sub-float | sub-long | rsub-int	50	A
Bitwise	shl-int | shl-long | shr-int | shr-long | ushr-int | ushr-long	15	B
Casting	check-cast	1	H
Comparison	cmp-long | cmpg-double | cmpg-float | cmpl-double | cmpl-float	5	C
Definition	const | const-class | const-string | const-wide	11	D
If conditional	if-eq | if-eqz | if-ge | if-gez | if-gt | if-gtz | if-le | if-lez | if-lt | if-ltz | if-ne | if-nez	12	I
Inline	execute-inline	1	U
Invoke	invoke-direct | invoke-direct-empty | invoke-interface | invoke-static | invoke-super | invoke-super-quick | invoke-virtual | invoke-virtual-quick	15	V
Instance	fill-array-data | filled-new-array | instance-of | new-array | new-instance	6	F
Jump	goto	3	J
Logical	and-int | and-long | neg-double | neg-float | neg-int | neg-long | not-int | not-long | or-int | or-long | xor-int | xor-long	24	L
Monitor	monitor-enter | monitor-exit	2	E
Move	move | move-exception | move-object | move-result | move-result-object | move-result-wide | move-wide	13	M
Read	aget | aget-boolean | aget-byte | aget-char | aget-object | aget-short | aget-wide | iget | iget-boolean | iget-byte | iget-char | iget-object | iget-object-quick | iget-quick | iget-short | iget-wide | iget-wide-quick | sget | sget-boolean | sget-byte | sget-char | sget-object | sget-short | sget-wide	24	G
Return	return | return-object | return-void | return-wide	4	R
Switch	packed-switch | sparse-switch	2	S
Throw	throw	1	O
Type Change	double-to-float | double-to-int | double-to-long | float-to-double | float-to-int | float-to-long | int-to-byte | int-to-char | int-to-double | int-to-float | int-to-long | int-to-short | long-to-double | long-to-float | long-to-int	15	T
Write	aput | aput-boolean | aput-byte | aput-char | aput-object | aput-short | aput-wide | iput | iput-boolean | iput-byte | iput-char | iput-object | iput-object-quick | iput-quick | iput-short | iput-wide | iput-wide-quick | sput | sput-boolean | sput-byte | sput-char | sput-object | sput-short | sput-wide	24	P

Furthermore, an opcode sequence is divided into opcode segments. An opcode segment is an functional block of opcode instructions in succession. A new segment is created by breaking opcode sequence at locations where there exists a diversion of flow control. For example, a block of opcode sequence `DFFPDJGDGVM` is divided into `DFFPD` and `GDGVM` based on pivot opcode `J` corresponding to a jump. Furthermore, `nop` instructions are skipped during symbol mapping as they do not add functional value to the code. Working of OSD generation is described in Algorithm 1.

Algorithm 1: OSD Generation

```
Input: sample.APK
Output: Opcode Segment Document of the sample
Initialize OSD file
Extract DEX files from sample.APK
foreach DEX file do
    Extract .smali files
end
foreach .smali file do
    Initialize OpcodeSegment
    Extract instructions
    Ignore instruction operands
    foreach instruction do
        if instruction is nop then
            continue
        end
        if instruction is for control diversion then
            Create new OpcodeSegment
            continue
        else
            Map instruction to Symbol using Symbol Table
            Append the Symbol to OpcodeSequence
        end
    end
    Append OpcodeSegment to OSD
end
```

Feature Extraction

To make an OSD document classifiable, we perform feature extraction that is to convert the document into a set of features. Each opcode segment word in the OSD is treated as a feature with its frequency as a feature value. We generate a feature vector representation of opcode segment words, quantified with number of occurrences of each in an OSD.

Attribute Selection

Feature extraction discussed above output features, of which many are irrelevant. We use attribute selection to choose significant features from the extracted ones. During attribute selection we evaluate the worth of each feature by calculating its information gain. Information gain depicts the entropy reduction due to a classification, thus capturing feature effectiveness with reference to the class. Formally, let F be a set of features to be classified into M classes and F_m denote the m-th subclass. Then, the entropy of F is:

$$E(F) = -\sum_{m \in M} \frac{|F_m|}{|F|} \times \log_2 \frac{|F_m|}{|F|}$$

For a feature f with V as the set of its possible values, let F_v denote the sample subset with feature value v for A [35]. Thus information gain of the feature f can be calculated as:

$$IG(F, f) = E(F) - \sum_{v \in V(f)} \frac{|F_v|}{|F|} \times E(F_v)$$

Features are then ranked based on correlation to class by calculating information gain value.

Classification Model

We implement classification and detection phase in BLADE by implementing machine learning approaches. The representation of sensitive behaviors enables us to detect and classify malware samples effectively using learning techniques.

Table 3: Description of different datasets

Dataset	# benign	# malwares	# families	Year of release
AndroAutopsy	109193	9990	30	2015
AndroTracker	51179	4554	20	2015
Drebin	-	5560	179	2014
PRAGuard (Malgenome)	-	8750	23	2015
PRAGuard (Contagio)	-	1652	-	2015

We select J48, k-NN, Random Forest (RF) and Sequential Minimal Optimization (SMO) for unsupervised learning. Our system is trained on labeled data and then evaluated on testing data.

4. Performance Evaluation

In this section, we first introduce datasets and evaluation parameters. It follows with the evaluation of our proposed approach against the following Research questions.

RQ1: Can `BLADE` detect malware samples with high accuracy? (*Malware detection*)

RQ2: Can `BLADE` effectively classify malware samples into their respective families? (*Familial Classification*)

RQ3: Can `BLADE` classify malware samples into their classes with high `TPR` and low `FPR`? (*Malware Class/Type Detection*)

RQ4: Can `BLADE` effectively detect obfuscation type used by a malware? (*Obfuscation Detection*)

RQ5: Can `BLADE` be resilient to obfuscation methods while classifying malware samples? (*Familial Classification*)

4.1. Datasets and Evaluation Metrics

In order to answer above mentioned research questions we evaluate BLADE against different benchmark datasets. We selected four Android application datasets namely: AndroAutosy [36], AndroTracker [37], Drebin [38] and Android PRAGuard [23]. Table 3 describes the datasets used.

Table 4: Malware detection and classification evaluation metrics.

Term	Abbreviation	Definition
True Positive	TP	No. of samples correctly detected as malware or correctly classified into family f.
True Negative	TN	No. of samples correctly detected as benign or correctly not classified into family f.
False Positive	FP	No of sample incorrectly detected as malware or incorrectly classified into family f.
False Negative	FN	No of sample incorrectly detected as benign or incorrectly not classified into family f.
Precision	p	$TP/(TP + FP)$
Recall	r	$TP/(TP + FN)$
F-measure	F_1	$2rp/(r + p)$
ROC Area	AUC	Area under ROC curve
Accuracy	Acc	Percentage of malwares correctly detected or classified

AndroAutopsy contains 109193 benign and 9990 malware samples classified into 30 families [36]. AndroTracker contains 51179 benign and 4554 malware samples classified into 20 families [37]. Malware samples in AndroTracker includes four categories, which are Adware, Downloader, Riskware and Trojan. Whereas, Drebin contains only malicious samples (5560) in 179 families [38]. These three datasets are used to answer RQs pertaining to malware detection, familial classification and malware class detection.

To evaluate obfuscation resilience of BLADE, we selected Android PRAGuard dataset, which is a collection of obfuscated malware samples. It contains 10479 obfuscated malware samples, generated by applying different obfuscation methods on Malgenome [17] and Contagio MiniDump [39]. It employed trivial obfuscation, string encryption, reflection, class encryption obfuscation methods and their combinations. Obfuscated malwares in Android PRAGuard generated from Malgenome are classified into 23 family labels. We use Android PRAGuard to answer RQs related to obfuscation resilience and classification of obfuscated malwares.

Table 4 lists the evaluation parameters employed to evaluate BLADE.

4.2. Methods for Performance Comparison

We selected four machine learning algorithms as appropriate classifiers for our approach, namely: J48 decision tree (number of folds = 3; confidence factor

Table 5: Results: Malware detection by BLADE on AndroAutopsy and AndroTracker datasets

Method	*TPR*	*FPR*	*AUC*	*Acc*(%)	Method	*TPR*	*FPR*	*AUC*	*Acc*(%)
		AndroAutopsy					*AndroTracker*		
J48	0.972	0.030	0.973	97.21	J48	0.984	0.016	0.986	98.39
k-NN	0.978	0.025	0.985	97.75	k-NN	0.985	**0.015**	0.993	98.54
RF	**0.982**	**0.023**	**0.997**	**98.18**	RF	**0.988**	0.016	**0.999**	**98.78**
SMO	0.974	0.027	0.973	97.37	SMO	0.977	0.022	0.977	97.70

= 0.25), k-nearest neighbors (k=1), Random Forest (number of trees = 100) and SMO (complexity parameter=1; tolerance parameter=0.001). We do not abandon any features in the experiments. We use above algorithms for training and testing. We selected 10-fold cross validation for testing.

4.3. RQ1: Can BLADE detect malware samples with high accuracy?

Malware detection problem deals with identification of malicious samples amongst benign ones. We considered AndroAutopsy (*benign=109193* & *malware=9990*) and AndroTracker (*benign=51179* & *malware=4554*) datasets to evaluate malware detection performance of BLADE equipped with four different classifiers. detection accuracy of our approach. Table 5 shows the results of BLADE against *TPR*, *FPR*, *AUC* and *Acc* parameters. Following conclusions are drawn from it:

- All classifiers perform satisfactorily on both datasets with accuracy (greater than 97%).
- Random Forest outperforms other classifiers in almost all parameters. k-NN (*FPR*=0.015) slightly outperforms Random Forest (*FPR*=0.016) in terms of false positive rate when evaluated on AndroTracker.

> ⇒***RQ1 Answer:*** *BLADE can effectively detect malware samples with high accuracy.*

4.4. RQ2: Can BLADE effectively classify malware samples into their respective families?

The problem of classifying malicious samples into respective malware families is popularly known as familial classification. For performance evaluation of BLADE we considered three benchmark datasets, which are AndroAutopsy, AndroTracker and Drebin. Malware samples in AndroAutopsy (9990 samples) and AndroTracker (4554) dataset are categorized into 30 and 20 families respectively. We selected top 20 families from Drebin dataset for evaluation. All four classifiers are tested against above three datasets for familial classification. Table 6 shows the results of BLADE against TPR, FPR, AUC and ACC parameters. Following conclusions are drawn from it:

- All classifiers perform satisfactorily on AndroAutopsy, AndroTracker and Drebin with accuracy greater than 94% and AUC greater than 0.993.
- SMO classifier is more effective than J48, k-NN and RF in terms of TPR, FPR and accuracy.
- Performance of Random Forest is better in term of AUC parameter. Weighted average AUC of Random Forest on AndroTracker is 1.

Table 7 illustrates detailed familial classification performance analysis of BLADE with SMO when applied on top 20 families in Drebin. Dataset comprised of 4664 malware samples categorized into 20 families. Since family datasets are imbalanced, F_1 measure is a preferred choice for comparison. BLADE with SMO classifier is effective with weighted average F_1 measure of 0.985, accuracy of 98.47% and $\boldsymbol{FPR}$ of 0.002. However, F_1 measure of only LinuxLotoor and Glodream families are between 0.88 and 0.90. This behavior is due to fewer samples in a family and inter-family similarity.

> ⇒***RQ2 Answer:*** *BLADE can effectively classify malicious samples into their families with high accuracy and F-measure*

Table 6: Results: Familial classification by BLADE on AndroAutopsy, AndroTracker and Drebin datasets

Method	TPR	FPR	AUC	$Acc(\%)$	TPR	FPR	AUC	$Acc(\%)$	TPR	FPR	AUC	$Acc(\%)$
	AndroAutopsy				AndroTracker				Drebin			
J48	0.936	0.005	0.976	93.62	0.980	0.004	0.994	97.96	0.975	0.003	0.989	97.49
k-NN	0.932	0.006	0.985	93.19	0.983	**0.002**	0.998	98.29	0.963	0.004	0.989	96.33
RF	0.944	0.006	**0.996**	94.35	0.984	0.003	**1.000**	98.44	0.980	**0.002**	**0.999**	98.01
SMO	**0.950**	**0.004**	0.993	**94.97**	**0.986**	**0.002**	0.998	**98.59**	**0.985**	**0.002**	0.995	**98.47**

Table 7: Familial classification performance of BLADE with SMO for Drebin dataset (top 20 families)

Family	#	TPR	FPR	p	r	F_1	AUC	Family	#	TPR	FPR	p	r	F_1	AUC
Adrd	91	0.989	0.000	0.989	0.989	0.989	0.998	GinMaster	339	0.991	0.000	0.994	0.991	0.993	1.000
BaseBridge	330	0.976	0.000	0.997	0.976	0.986	0.992	Glodream	69	0.826	0.000	0.983	0.826	0.898	0.960
DroidDream	81	0.951	0.000	0.987	0.951	0.969	0.981	Iconosys	152	1.000	0.000	1.000	1.000	1.000	1.000
DroidKungFu	667	0.991	0.004	0.975	0.991	0.983	0.994	Imlog	43	0.953	0.000	1.000	0.953	0.976	1.000
LinuxLotoor	70	0.855	0.001	0.922	0.855	0.887	0.959	Kmin	147	0.993	0.000	0.993	0.993	0.993	1.000
FakeDoc	132	0.992	0.000	1.000	0.992	0.996	0.998	MobileTx	69	1.000	0.000	1.000	1.000	1.000	1.000
FakeInstaller	925	0.987	0.002	0.990	0.987	0.989	0.996	Opfake	613	0.997	0.006	0.961	0.997	0.978	0.997
FakeRun	61	1.000	0.000	0.984	1.000	0.992	1.000	Plankton	625	0.998	0.001	0.994	0.998	0.996	0.999
Gappusin	58	1.000	0.001	0.951	1.000	0.975	1.000	SendPay	59	0.983	0.000	1.000	0.983	0.991	0.986
Geinimi	92	0.967	0.000	1.000	0.967	0.983	0.995	SMSreg	41	0.902	0.000	1.000	0.902	0.949	0.971
Weighted Avg.		**0.985**	**0.002**	**0.985**	**0.985**	**0.985**	**0.995**								

4.5. RQ3: Can BLADE classify malware samples into their classes with high TPR and low FPR?

Malware based on their behavior are categorized into types or classes such as Adware and Trojan. We test effectiveness of BLADE in detecting malware classes against AndroAutopsy, which categorizes its malware samples into five major classes namely: Adware, Downloader, Riskware, Rooter and Trojan. Table 8 illustrates efficacy of BLADE while while categorizing malicious samples into behavior based classes. Following conclusions are drawn from it.

- All classifiers perform satisfactory with accuracy more than 96.5%.
- SMO classifier is more effective in correctly classifying the samples. With better hit rate and low fall-out rate.
- Random Forest classifier is more capable of distinguishing between the classes with AUC of 0.997.

> ⇒***RQ3 Answer:*** *BLADE can effectively distinguish between malicious samples from different classes.*

Table 8: Results: Malware class detection by BLADE on AndroAutopsy dataset

Method	TPR	FPR	*AUC*	*Acc* (%)
		AndroAutopsy		
J48	0.965	0.028	0.974	96.54
k-NN	0.967	0.029	0.988	96.70
RF	0.967	0.041	**0.997**	96.69
SMO	**0.975**	**0.022**	0.980	**97.53**

4.6. RQ4: Can BLADE effectively detect obfuscation type used by a malware?

As discussed in section 2.2, malware authors enhance their applications with obfuscation techniques to evade detection. We test efficacy of BLADE while dealing with obfuscated samples. In this subsection we try to answer, whether our approach is able to differentiate between malware samples obfuscated with different methods. We chose Android PRAGuard [23] dataset for it. Android PRAGuard comprises of malware samples from Malgenome and Contagio datasets obfuscated with multiple methods such as trivial obfuscation, string encryption, reflection, class encryption and their combinations. We created sub-datasets from Android PRAGuard to have a detailed analysis. PRAGuard Malgenome (T, S, R & C) and PRAGuard Contagio (T, S, R & C) datasets comprise of samples obfuscated either by Trivial, String encryption, Reflection or Class encryption. While PRAGuard Malgenome (T, S, R, C, TS, TSR & TSRC) and PRAGuard Contagio (T, S, R, C, TS, TSR & TSRC) datasets comprise of sample enhanced with multiple methods also. Following conclusions are drawn from results illustrated in Table 9.

- J48, Random Forest and SMO classifiers are effective in obfuscation type detection. k-NN classifier based approach is less effective than others.
- BLADE with J48 classifier is effective to distinguish between samples enhanced using single obfuscation methods with accuracy 99.44% (PRAGuard Malgenome) and 98.83% (PRAGuard Conatagio).
- BLADE is more effective on PRAGuard Malgenome (T, S, R & C) with accuracy 99.44% than PRAGuard Malgenome (T, S, R, C, TS, TSR & TSRC) with accuracy 93.53%. It also is more effective on PRAGuard

Table 9: Results: Obfuscation type detection on PRAGuard dataset

Method	TPR	FPR	*AUC*	*Acc* (%)	Method	TPR	FPR	*AUC*	*Acc* (%)
PRAGuard Malgenome (T, S, R & C)					PRAGuard Contagio (T, S, R & C)				
J48	0.994	0.002	0.999	**99.44**	J48	0.988	0.004	0.996	**98.83**
k-NN	0.922	0.026	0.979	92.24	k-NN	0.863	0.046	0.965	86.33
RF	0.991	0.003	1	99.10	RF	0.978	0.007	0.998	97.78
SMO	0.992	0.003	0.995	99.18	SMO	0.981	0.006	0.991	98.09
PRAGuard Malgenome (T, S, R, C, TS, TSR & TSRC)					PRAGuard Contagio (T, S, R, C, TS, TSR & TSRC)				
J48	0.935	0.011	0.980	93.53	J48	0.921	0.013	0.978	92.09
k-NN	0.852	0.025	0.955	85.19	k-NN	0.857	0.024	0.957	85.68
RF	0.916	0.014	0.993	91.63	RF	0.917	0.014	0.990	91.66
SMO	0.920	0.013	0.983	92.03	SMO	0.923	0.013	0.979	92.27

[T: Trivial; S: String Encryption; R: Reflection; C: Class Encryption; TS: Trivial and String Encryption; TSR: Trivial, String encryption and Reflection; TSRC: Trival, String Encryption, Reflection and Class Encryption]

Contagio (T, S, R & C) with accuracy 98.83% than PRAGuard Contagio (T, S, R, C, TS, TSR & TSRC) with accuracy 92.27%. Thus BLADE performs better on single obfuscated samples than combinatory.

> ⇒***RQ4 Answer:*** *BLADE can effectively differentiate type of obfuscation used by a malicious sample. It also performs well against samples enhanced with multiple obfuscation techniques.*

4.7. RQ5: Can BLADE be resilient to obfuscation methods while classifying malware samples?

To evaluate the resilience of BLADE against obfuscation methods, we perform familial classification of obfuscated samples from PRAGuard Dataset. We created seven subset from Android PRAGuard (Malgenome) on the basis of obfuscation methods. We then measure how well our approach can identify families amongst each sub-dataset (T, S, R, C, TS, TSR & TSRC). Each sub-dataset comprised of 1250 samples categorized into 23 families. Table 10 shows accuracy of familial classification when applied on above sub-datasets. Following conclusions are drawn from it.

- BLADE is resilient to Trivial, String encryption, Reflection and their combinatory techniques.

Table 10: Results: Familial classification accuracy (%) of obfuscated malware samples from PRAGuard Malgenome dataset.

Method	T	S	R	C	TS	TSR	TSRC
J48	98.60	97.86	98.77	**92.77**	97.87	98.53	86.65
k-NN	97.29	96.72	97.70	83.74	96.97	97.05	90.58
RF	98.69	98.44	98.61	85.97	98.37	98.20	91.32
SMO	**99.02**	**99.02**	**99.18**	91.87	**99.26**	**98.69**	**92.47**

[T: Trivial; S: String Encryption; R: Reflection; C: Class Encryption; TS: Trivial and String Encryption; TSR: Trivial, String encryption and Reflection; TSRC: Trival, String Encryption, Reflection and Class Encryption]

- BLADE is less resilient against Class encryption and its combinatory when compared with other obfuscation methods. But it is still effective in detecting Class encryption with 92.77% accuracy.
- SMO classifier performs better than other classifiers in most cases.

⇒***RQ5 Answer:*** *BLADE is resilient to obfuscation methods while classifying malware sample with high accuracy.*

5. Discussion

In this section, we compare our proposed system against state of the art malware detection systems in Android. Table 11 compares performance of the proposed work with DANDroid [40]. The comparison is with reference to various obfuscation methods and their combination. DANDroid use DexProtector tool to obfuscate Drebin dataset, where as results of BLADE are based on Malgenome dataset obfuscated using PRAGuard tool [30, 23]. DANDroid uses Discriminative Adversarial Network based on neural network for detection. Both the approaches performs well against obfuscation methods apart from class encryption which shows a small dip in the accuracy.

Efficiency and performance of the proposed solution is compared with previous studies in table 12. We have listed features used for malware detection or classification, furthermore the dataset(s) with the technique(s) employed. Few works like, Millar et al. [40] and Garcia et al. [] are evaluating their work on both non-obfuscated and obfuscated samples.

Table 11: Classification accuracy comparison of DANDroid and BLADE (proposed work).

Obfuscation	DANDroid[40]	BLADE
Trivial	-	99.02
String Encryption	98.8	99.02
Reflection	99	99.18
Class Encryption	95.1	92.77
Resource Encryption	98.7	-
All obfuscations applied	95.3	92.47

Table 12: Comparison of BLADE with the existing state of the art solutions. [OD: Performance over obfuscated dataset]

Paper	Year	Features	Techniques	Dataset	Acc (%)
Arp et al. [38]	2014	Hardware, API Calls, App components, Intents, Permissions and Network addresses	SVM	Drebin	93.9
Fereidooni et al. [41]	2016	Intent, API Calls and Permissions	SVM, DT, NB, LR, RF, KNN, Adaboost, DL, XG-boost	Genome, Drebin, Virus Total	97
Karbab et al. [42]	2016	Binary, Assembly, Manifest and APK	Permissions, API calls, Network addresses, APK	Drebin, Genome	87
Mariconti et al. [43]	2017	API Calls	Markov Chain Model	Drebin	87
Feizollah et al. [44]	2017	Intents and Permissions	Bayesian Network	Drebin, Google PlayStore	95.5
Wang et al. [13]	2017	App components, Intents, Permissions, API calls, strings, commands and network information	Dempster-Shafer theory based fusion of KNN, random forest and J48 classifiers	Drebin and Android Malware Genome Project	99.7
Garcia et al. [45]	2018	Permissions, App Components and Intent filters	SVM	Malgenome, Drebin, Virus Share and Virus Total	96
Garcia et al. [45]	2018	Permissions, App Components and Intent filters	SVM	Malgenome, Drebin, Virus Share and Virus Total	86 [OD]
Machiry et al. [46]	2018	Code loops	RF	Malgenome and Virus Share	99.1
Alshahrani et al. [47]	2018	Permissions, system information, system calls, network information	SGD, RMSProp, Adagrad, Adam, Nadam, Adadelta and Adamax	Drebin and MARVIN	95.13
Alazab [48]	2020	API Calls	Naive Bayes, kNN, RF, J48, SMO, Logistic Regressions, Adaboost, JRip, Random committee, Simple logistics	VirusTotal, AndroZoo, MalShare, Contagio and Google PlayStore	98.1
Millar et al. [40]	2020	Opcode instructions, permissions, API calls and commands	DAN, CNN, Neural Nets	Drebin and self obfuscated	97.3
Millar et al. [40]	2020	Opcode instructions, permissions, API calls and commands	DAN, CNN, Neural Nets	Drebin and self obfuscated	59.6 [OD]
Sihag et al. (Proposed Work)	2020	Opcode instructions	k-NN, J48, RF and SMO	Drebin, Contagio, Malgenome, PRAGuard	98.6
Sihag et al. (Proposed Work)	2020	Opcode instructions	k-NN, J48, RF and SMO	Drebin, Contagio, Malgenome, PRAGuard	92.47 [OD]

6. Related Works

Android is a market mover and popular target among malware authors. There are several studies on obfuscation techniques used by Android malware and their evolving detection methods.

Obfuscation and its effectiveness

Obfuscation methods are a new normal for both developers and malware authors. Tam et al. [12], Nigam [49] and Suarez-Tangil [50] have extensively discussed the evolution of Android malware over the last decade. Apvrille and Nigam in [25] explores the practical usage of stealth techniques by Android malware. Faruki et al. in [16] discussed obfuscation methods, application protection and deobfuscation methods specific to Android.

Dong et al. in [22] provided an understanding into Android code obfuscation and carried out a large scale investigation on 114,560 samples for its usage. Various static and dynamic code obfuscation approaches are presented in [22, 51, 52, 53, 54] such as renaming, string encryption, control flow obfuscation and reflections. Effectiveness of these obfuscation are evaluated in [55, 56, 4, 23, 57, 58, 59, 60, 61]. Park et al. in [58], empirically analyzed application similarity between original software and the one transformed by code obfuscation. Furthermore, it tried to question the legality of the obfuscated app. State of art deobfuscation methods are proposed in [62, 63, 64].

Detection using Opcodes

Opcodes which represent application code at instruction level are popularly used static analysis approach. Statistical properties of application opcodes are useful for malware detection. Multiple studies have evaluated its effectiveness for classification. Hang et al. [65] proposes simplification of 218 dalvik opcode and was more effective than anti-malware softwares. Chen et al. [66] also performs simplification but only of 107 representative opcodes. Canfora et al. [67] divided opcodes into n-grams for detection. It used frequency characteristic, which are then fed into SVM and RF classifiers. They concluded that n-gram approach with n=2 was most accurate for malware detection. Hahn et al. [68] included both opcode sequence and opcode frequency for classification using machine learning (Bayesian Network, k-NN and Random Forest). Mclaughlin et al. [69] employed CNN for deep learning based on opcode sequences. They concluded it to be more effective than n-gram approach while considering scalability. Other approaches have also employed similarity measure on opcode sequences or n-grams for classification [70, 71].

7. Conclusion

Malware detection and its classification is a complex problem involving distinct feature identification and selection from malware samples. The task gets more complicated with malware employing obfuscation methods to evade such identification. This paper introduces `BLADE`, a novel system based on Opcode Segment Document (OSD) for malware detection and familial classification. It is effective, accurate and resilient to obfuscation. BLADE relies on opcode segments, which represents sequential instruction. We evaluated it to answer research questions of malware detection, malware familial classification, malware class/type detection, obfuscation type detection and familial classification of obfuscated samples. `BLADE` was tested against benchmark datasets AndroAutopsy, AndroTracker, Drebin and Android PRAGuard. It is found effective in detecting samples using multiple obfuscation techniques.

As part of the future work, we need to explore obfuscation methods where malicious code is located outside the DEX file, such as native code and libraries. Furthermore, we plan to explore the behavioral representation of fine-grained opcode segments against with the behavioral abstraction from dynamic analysis.

References

[1] Number of smartphone users worldwide from 2016 to 2021.
URL https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/

[2] N. Grover, J. Saxena, V. Sihag, Security Analysis of OnlineCabBooking Android Application, 2017, pp. 603–611.

[3] O. Alrawi, C. Zuo, R. Duan, R. P. Kasturi, Z. Lin, B. Saltaformaggio, The betrayal at cloud city: An empirical analysis of cloud-based mobile backends, in: 28th {USENIX} Security Symposium ({USENIX} Security 19), 2019, pp. 551–566.

[4] M. Dalla Preda, F. Maggi, Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology, Journal of Computer Virology and Hacking Techniques 13 (3) (2017) 209–232.

[5] V. Sihag, M. Vardhan, P. Singh, A survey of android application and malware hardening, Computer Science Review 39 (2021) 100365.

[6] A. Bacci, A. Bartoli, F. Martinelli, E. Medvet, F. Mercaldo, Detection of obfuscation techniques in android applications, in: Proceedings of the 13th International Conference on Availability, Reliability and Security, 2018, pp. 1–9.

[7] H.-J. Zhu, Z.-H. You, Z.-X. Zhu, W.-L. Shi, X. Chen, L. Cheng, Droiddet: effective and robust detection of android malware using static analysis along with rotation forest model, Neurocomputing 272 (2018) 638–646.

[8] Y. Feng, S. Anand, I. Dillig, A. Aiken, 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, pp. 576–587.

[9] H. Kang, J.-w. Jang, A. Mohaisen, H. K. Kim, Detecting and classifying android malware using static analysis along with creator information, International Journal of Distributed Sensor Networks 11 (6) (2015) 479174.

[10] J.-w. Jang, H. K. Kim, Function-oriented mobile malware analysis as first aid, Mobile Information Systems 2016.

[11] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-An, H. Ye, Significant permission identification for machine-learning-based android malware detection, IEEE Transactions on Industrial Informatics 14 (7) (2018) 3216–3225.

[12] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, L. Cavallaro, The evolution of android malware and android analysis techniques, ACM Computing Surveys (CSUR) 49 (4) (2017) 1–41.

[13] X. Wang, D. Zhang, X. Su, W. Li, Mlifdect: android malware detection based on parallel machine learning and information fusion, Security and Communication Networks 2017.

[14] V. Sihag, A. Swami, M. Vardhan, P. Singh, Signature based malicious behavior detection in android, in: International Conference on Computing Science, Communication and Security, Springer, 2020, pp. 251–262.

[15] P. Sharma, V. K. Sihag, Hybrid Single Sign-On Protocol for Lightweight Devices, 2016, pp. 679–684.

[16] P. Faruki, H. Fereidooni, V. Laxmi, M. Conti, M. Gaur, Android code protection via obfuscation techniques: past, present and future directions, CoRR abs/1611.10231.

[17] Y. Zhou, X. Jiang, Dissecting android malware: Characterization and evolution, in: 2012 IEEE symposium on security and privacy, IEEE, 2012, pp. 95–109.

[18] J. G. de la Puerta, B. Sanz, I. Santos, P. G. Bringas, Using dalvik opcodes for malware detection on android, in: International Conference on Hybrid Artificial Intelligence Systems, Springer, 2015, pp. 416–426.

[19] A. Bartel, J. Klein, Y. Le Traon, M. Monperrus, Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot, in: Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis, SOAP 12, Association for Computing Machinery, New

York, NY, USA, 2012, p. 2738. doi:10.1145/2259051.2259056.
URL https://doi.org/10.1145/2259051.2259056

[20] A. Desnos, et al., Androguard: Reverse engineering, malware and goodware analysis of android applications... and more (ninja!), Retrieved June 10 (2011) 2014.

[21] R. Winsniewski, Android–apktool: A tool for reverse engineering android apk files, Retrieved February 10 (2012) 2020.

[22] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, K. Zhang, Understanding android obfuscation techniques: A large-scale investigation in the wild, in: International Conference on Security and Privacy in Communication Systems, Springer, 2018, pp. 172–192.

[23] D. Maiorca, D. Ariu, I. Corona, M. Aresu, G. Giacinto, Stealth attacks: An extended insight into the obfuscation effects on android malware, Computers & Security 51 (2015) 16–31.

[24] H. Cho, J. H. Yi, G.-J. Ahn, Dexmonitor: Dynamically analyzing and monitoring obfuscated android applications, IEEE Access 6 (2018) 71229–71240.

[25] A. Apvrille, R. Nigam, Obfuscation in android malware, and how to fight back, Virus Bulletin (2014) 1–10.

[26] B. Saikoa, Allatori java obfuscator, [Accessed: 09-Apr-2020].
URL http://www.allatori.com/

[27] Apk protect: Android apk security protection (2013).
URL https://sourceforge.net/projects/apkprotect/

[28] B. Saikoa, Dexguard.

[29] P. Solutions, Dasho: Java & android obfuscator & runtime protection.

[30] L. Licel, Dexprotector–cutting edge obfuscator for android apps, [Accessed: 09-Apr-2020].
URL https://dexprotector.com/

[31] Mobile protector by gemalto, a thales company.
URL https://thales-protector-oath-sdk.docs.stoplight.io/releases/5.2.0/general/overview

[32] G. Square, Proguard, [Accessed: 09-Apr-2020].
URL https://www.guardsquare.com/en/products/proguard

[33] Promon shield — in-app protection & application shielding.
URL https://promon.co

[34] L. Licel, Stringer java obfuscator, [Accessed: 09-Apr-2020].
URL https://jfxstore.com/stringer/

[35] K. P. Murphy, Machine learning: a probabilistic perspective, MIT press, 2012.

[36] J. wook Jang, H. Kang, J. Woo, A. Mohaisen, H. K. Kim, Andro-autopsy: Anti-malware system based on similarity matching of malware and malware creator-centric information, Digital Investigation 14 (2015) 17 – 35. doi:http://dx.doi.org/10.1016/j.diin.2015.06.002.

[37] H. J. Kang, J.-w. Jang, A. Mohaisen, H. K. Kim, Androtracker: Creator information based android malware classification system, in: Information Security Applications-15th International Workshop, WISA, Vol. 8909, 2014.

[38] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, C. Siemens, Drebin: Effective and explainable detection of android malware in your pocket., in: Ndss, Vol. 14, 2014, pp. 23–26.

[39] M. Parkour, Contagio mobile - mobile malware mini dump (2012).
URL http://contagiominidump.blogspot.com

[40] S. Millar, N. McLaughlin, J. Martinez del Rincon, P. Miller, Z. Zhao, Dan-droid: A multi-view discriminative adversarial network for obfuscated android malware detection, in: Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy, 2020, pp. 353–364.

[41] H. Fereidooni, M. Conti, D. Yao, A. Sperduti, Anastasia: Android malware detection using static analysis of applications, in: 2016 8th IFIP international conference on new technologies, mobility and security (NTMS), IEEE, 2016, pp. 1–5.

[42] E. B. Karbab, M. Debbabi, A. Derhab, D. Mouheb, Cypider: building community-based cyber-defense infrastructure for android malware detection, in: Proceedings of the 32nd Annual Conference on Computer Security Applications, 2016, pp. 348–362.

[43] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, G. Stringhini, Mamadroid: Detecting android malware by building markov chains of behavioral models, arXiv preprint arXiv:1612.04433.

[44] A. Feizollah, N. B. Anuar, R. Salleh, G. Suarez-Tangil, S. Furnell, Androdialysis: Analysis of android intent effectiveness in malware detection, computers & security 65 (2017) 121–134.

[45] J. Garcia, M. Hammad, S. Malek, Lightweight, obfuscation-resilient detection and family identification of android malware, ACM Transactions on Software Engineering and Methodology (TOSEM) 26 (3) (2018) 1–29.

[46] A. Machiry, N. Redini, E. Gustafson, Y. Fratantonio, Y. R. Choe, C. Kruegel, G. Vigna, Using loops for malware classification resilient to feature-unaware perturbations, in: Proceedings of the 34th Annual Computer Security Applications Conference, 2018, pp. 112–123.

[47] H. Alshahrani, H. Mansourt, S. Thorn, A. Alshehri, A. Alzahrani, H. Fu, Ddefender: Android application threat detection using static and dynamic

analysis, in: 2018 IEEE International Conference on Consumer Electronics (ICCE), IEEE, 2018, pp. 1–6.

[48] M. Alazab, Automated malware detection in mobile app stores based on robust feature generation, Electronics 9 (3) (2020) 435.

[49] R. Nigam, A timeline of mobile botnets, Virus Bulletin, March.

[50] G. Suarez-Tangil, G. Stringhini, Eight years of rider measurement in the android malware ecosystem: evolution and lessons learned, arXiv preprint arXiv:1801.08115.

[51] F. C. Freiling, M. Protsenko, Y. Zhuang, An empirical evaluation of software obfuscation techniques applied to android apks, in: International Conference on Security and Privacy in Communication Networks, Springer, 2014, pp. 315–328.

[52] M. Kühnel, M. Smieschek, U. Meyer, Fast identification of obfuscation and mobile advertising in mobile malware, in: 2015 IEEE Trustcom/BigDataSE/ISPA, Vol. 1, IEEE, 2015, pp. 214–221.

[53] V. Rastogi, Y. Chen, X. Jiang, Catch me if you can: Evaluating android anti-malware against transformation attacks, IEEE Transactions on Information Forensics and Security 9 (1) (2013) 99–108.

[54] M. Zheng, P. P. Lee, J. C. Lui, Adam: an automatic and extensible platform to stress test android anti-virus systems, in: International conference on detection of intrusions and malware, and vulnerability assessment, Springer, 2012, pp. 82–101.

[55] T. Cho, H. Kim, J. H. Yi, Security assessment of code obfuscation based on dynamic monitoring in android things, IEEE Access 5 (2017) 6361–6371.

[56] J. Hoffmann, T. Rytilahti, D. Maiorca, M. Winandy, G. Giacinto, T. Holz, Evaluating analysis tools for android apps: Status quo and robustness

against obfuscation, in: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, 2016, pp. 139–141.

[57] D. Maier, T. Müller, M. Protsenko, Divide-and-conquer: Why android malware cannot be stopped, in: 2014 Ninth International Conference on Availability, Reliability and Security, IEEE, 2014, pp. 30–39.

[58] J. Park, H. Kim, Y. Jeong, S.-j. Cho, S. Han, M. Park, Effects of code obfuscation on android app similarity analysis., JoWUA 6 (4) (2015) 86–98.

[59] V. Balachandran, D. J. Tan, V. L. Thing, et al., Control flow obfuscation for android applications, Computers & Security 61 (2016) 72–93.

[60] V. Haupert, D. Maier, N. Schneider, J. Kirsch, T. Müller, Honey, i shrunk your app security: The state of android app hardening, in: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, 2018, pp. 69–91.

[61] P. Faruki, A. Bharmal, V. Laxmi, M. S. Gaur, M. Conti, M. Rajarajan, Evaluation of android anti-malware techniques against dalvik bytecode obfuscation, in: 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications, IEEE, 2014, pp. 414–421.

[62] Z. Kan, H. Wang, L. Wu, Y. Guo, D. X. Luo, Automated deobfuscation of android native binary code, arXiv preprint arXiv:1907.06828.

[63] Y. Moses, Y. Mordekhay, Android app deobfuscation using static-dynamic cooperation, VB2018.

[64] B. Bichsel, V. Raychev, P. Tsankov, M. Vechev, Statistical deobfuscation of android applications, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 343–355.

[65] D. Hang, N.-q. HE, H. Ge, L. Qi, M. ZHANG, Malware detection method of android application based on simplification instructions, The Journal of China Universities of Posts and Telecommunications 21 (2014) 94–100.

[66] T. Chen, Q. Mao, Y. Yang, M. Lv, J. Zhu, Tinydroid: a lightweight and efficient model for android malware detection and classification, Mobile information systems 2018.

[67] G. Canfora, A. De Lorenzo, E. Medvet, F. Mercaldo, C. A. Visaggio, Effectiveness of opcode ngrams for detection of multi family android malware, in: 2015 10th International Conference on Availability, Reliability and Security, IEEE, 2015, pp. 333–340.

[68] S. Hahn, M. Protsenko, T. Müller, Comparative evaluation of machine learning-based malware detection on android., Sicherheit 2016-Sicherheit, Schutz und Zuverlässigkeit.

[69] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, et al., Deep android malware detection, in: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, 2017, pp. 301–308.

[70] V. Sihag, A. Mitharwal, M. Vardhan, P. Singh, Opcode n-gram based malware classification in android, in: 2020 Fourth World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4), IEEE, 2020, pp. 645–650.

[71] A. Ali-Gombe, I. Ahmed, G. G. Richard III, V. Roussev, Opseq: Android malware fingerprinting, in: Proceedings of the 5th Program Protection and Reverse Engineering Workshop, 2015, pp. 1–12.

View publication stats