Computer Science and Information Systems 18(1):67–91https://doi.org/10.2298/CSIS200330035A

DroidClone: Attack of the Android Malware Clones - A Step Towards Stopping Them ⋆

Shahid Alam[1] and Ibrahim Sogukpinar[2]

[1] Department of Computer Engineering, Adana Alparslan Turkes Science and Technology University, Adana, Turkey
salam@atu.edu.tr

[2] Department of Computer Engineering, Gebze Technical University, Gebze, Turkey
ispinar@gtu.edu.tr

Abstract. Code clones are frequent in use because they can be created fast with little effort and expense. Especially for malware writers, it is easier to create a clone of the original than writing a new malware. According to the recent Symantec threat reports, Android continues to be the most targeted mobile platform, and the number of new mobile malware clones grew by 54%. There is a need to develop techniques and tools to stop this attack of Android malware clones. To stop this attack, we propose *DroidClone* that exposes code clones (segments of code that are similar) in Android applications to help detect malware. *DroidClone* is the first such effort uses specific control flow patterns for reducing the effect of obfuscations and detect clones that are syntactically different but semantically similar up to a threshold. *DroidClone* is independent of the programming language of the code clones. When evaluated with real malware and benign Android applications, *DroidClone* obtained a detection rate of 94.2% and false positive rate of 5.6%. *DroidClone*, when tested against various obfuscations, was able to successfully provide resistance against all the trivial (Renaming methods, parameters, and nop insertion, etc) and some non-trivial (Call graph manipulation and function indirection, etc.) obfuscations.

Keywords: Android, Code Clones, MAIL, Malware Analysis and Detection, TF-IDF, Machine Learning.

1. Introduction

According to the McAfee threat report [34], number of malware (*clones*) found in the Google play increased by 30% in 2017. According to the Symantec [42, 43] threat reports, Android continues to be the most targeted mobile platform, and the number of new discovered mobile malware (*clones*) grew by 54% from 2016 to 2017. Further to this simple attack of *clones*, there are also Android malware *clones of clones*, i.e., *clones* of a malware family which itself is a *clone*. For example, *DroidKungFu1*, *DroidKungFu2*, *DroidKungFu3* and *DroidKungFu4* are 4 different families of the original Android *DroidKungFu* malware, and each of these 4 families have there own *clones* [47].

⋆ The work presented in this paper is an expansion of the authors' previously published work in conference paper [2].

Malware writers are using stealthy mutations (obfuscations) to continuously develop malware clones, thwarting detection by signature-based detectors. To protect Android applications against reverse engineering attacks, even legitimate applications are obfuscated [15]. Similar techniques are used by malware developers to prevent analysis and detection. Obfuscation can be used to make the code more difficult to analyze, or to create *clones* of the same malware in order to evade detection.

Code clones are frequent in use because they can be created fast with little effort and expense. Especially for malware writers, it is easier to create a clone of the original than writing a new malware. There are different ways to create code clones, some of them are: when a programmer copies and paste fragment of code after minor editing; part of a code is embedded (piggybacked) inside another code/program; when a code is obfuscated to create copies, which are syntactically different but semantically similar. Finding code clones can be useful for: detecting malicious software, plagiarism and copyright infringement; bug detection; code simplification; and code maintainability.

In general, clones are divided into four types [39]. *Type*-1 (Exact clones): Exact copies of each other except white spaces, blanks and comments, etc. *Type*-2 (Renamed clones): Similar copies except name of variables, literals, functions, etc. *Type*-3 (Near miss clones): Similar except all the above and some added/removed statements, etc. *Type*-4 (Semantic clones): Syntactically different but semantically similar.

There are several researches [9, 10, 16, 18, 22–25, 27–30, 32, 35, 45, 46] that have focused on code clone detection using different approaches, such as: (1) *textual*, based on token [9] and pattern [18] matching, and longest common subsequence [16]. (2) *lexical*, based on frequent subsequence mining [30], cosine similarity [46], and suffix array [35]. (3) *syntactical*, based on abstract syntax tree (AST) [10, 28, 45], hashed blocks [24], and AST to vectors [25]. (4) *semantic*, based on program dependence graph (PDG) [23, 27, 29, 32], and serialized AST [22].

Only two [27, 29] of the above approaches claim to detect *Type*-4 cloning, but the DR (detection rate) of [29] is low ranging from 17.3% – 45.8% and there is no DR to report for [27]. These two approaches can only find source code and not native code clones, and hence is also dependent on the programming language of the code. Both of them find isomorphic similarity of subgraph PDGs to detect clones, which is compute-intensive and is not scalable.

Clone detection technique can be improved by combining several different types of methods or reimplementing systems using a different programming language. It is hard to determine which is the best tool for detection because every tool has its strengths and weaknesses. Since text-based and token-based techniques have high recall and AST-based techniques have high precision, these techniques may be merged in a tool to get high recall and precision results. A PDG-based technique detects only Type-3 clones; this technique may be extended to detect Type-1 and Type-2 clones besides Type-3 clones.

Type-1 and Type-2 clones are easier to detect than Type-3 clones. Sequence alignment algorithms with gaps may potentially be used to detect Type-3 clones. To make clone detection independent of the programming language of the clone

and also combine different techniques in one we use a new intermediate language MAIL (Malware Analysis Intermediate Language) [3] to improve clone detection.

As mentioned earlier, Android mobile platform is facing an attack of clones. We need to find ways for detecting and stopping this attack. There are several researches [5,13,17,20,31,41] that have focused on Android malware clones detection. [17] uses API call graphs, [41] uses components based API calls to find code reuse, and [13] uses a text-based near-miss source code clone detector. [5] mines dominant API calls to find the reuse of malicious modules to detect malware. [31] captures system calls at thread level to detect malicious clones embedded inside an Android application and [20] uses SDHash [38] to detect application similarity for detecting malware. 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.

In this paper, we propose *DroidClone* as a step towards detecting and stopping these clones in Android malware. For this purpose, we utilize the new intermediate language MAIL [3] that helps us use specific control flow patterns to reduce the effect of obfuscations and unlike [5, 17, 41] can detect malware clones at a much-refined level that helps detect smaller size clones. Our technique, unlike [13, 20], can detect malware clones that are syntactically different but semantically similar up to a certain threshold. A malware writer has to employ an excessive (beyond a certain threshold, which is difficult to find) control flow obfuscation to create a clone to evade detection by such an anti-malware. [31] is based on dynamic analysis, may not cover all the program paths and hence can miss some malicious behaviors. *DroidClone* performs static analysis and covers all the program paths. *DroidClone* finds clones at a much refined level than [5]. Moreover, *DroidClone* can process and analyse Android native code clones.

DroidClone, when tested with 4180 real malware and benign Android applications using different validation methods, obtained detection rates (DRs) in the range of 90.3% − 94.2% and false positive rates (FPRs) in the range of 4% − 11%. *DroidClone*, when tested against various obfuscations (malware variants), was able to successfully provide resistance against all the trivial (Renaming methods, parameters, and nop insertion, etc) and some non-trivial (Call graph manipulation and function indirection, etc.) obfuscations.

Following are the major contributions of this paper:

- *DroidClone* is the first such effort that uses a new intermediate language MAIL for building the signatures to find Android clones for malware detection. We first build a MAIL CFG (control flow graph) and then extract specific control flow patterns to reduce the effect of obfuscations and detect code clones that are syntactically different but semantically similar (i.e., *Type*-3 and *Type*-4 clones) up to a threshold. Sometimes malicious code (bytecode or native code) clone consists of only a few statements, such as setting up a few registers and a jump to the actual malicious code location. To accommodate such clones, *DroidClone* uses MAIL blocks at a statement level, and can detect clones at a much-refined level (smaller size clones ≥ 3 statements) than other such techniques.

- We extend TF-IDF with a new weighting scheme for feature (control flow patterns) selection and improve the clone detection scheme. Moreover, we perform serialization of a MAIL CFG into strings of specific control flow patterns at block level, which also improves application matching.
- Most of the Android applications are written in C/C++ and Java. It is necessary to build a cross-platform clone detector for such applications. *DroidClone* designs cross-platform signatures for Android at the native code level, and is able to process, analyse and detect malware cloned as either bytecode or native code. This makes *DroidClone* independent of the programming language of the Android code clone.
- This paper significantly enhances the previous version of *DroidClone* [2] by:
  - updating and enhancing the clone detection scheme;
  - using only MAIL blocks for building the signatures, in turn improving accuracy and also runtime of the overall system;
  - improving the feature selection method;
  - lowering the false positive rate (8.5% $\Rightarrow$ 5.6%);
  - increasing the Accuracy (91% $\Rightarrow$ 94.3%);
- We provide cross-validation of *DroidClone*, using two methods holdout and n-fold, which is a more systematic way of determining the performance and accuracy of a system than is provided by most other similar works. We also test the resistance of *DroidClone* against various obfuscations.

The remainder of this paper is organized as follows. We discuss related works in Section 2. We present a detailed overview of our approach, its design and implementation in Section 3. Section 4 presents the evaluation and comparison of our approach with six other such works. Section 5 finally concludes the paper and presents some future works.

2. Related Works

A very detailed survey of the research done on code clones is presented in [39]. In this section, we briefly highlight seven recent research works of finding Android clones for detecting malware.

Lin et al. [31] propose SCSdroid, which captures thread-grained system call sequences during runtime to detect malicious clones embedded inside an Android application. A thread-grained system call sequence is the system calls recorded for a thread. The authors believe that the malicious behavior happens during a single thread, so mixing the system calls recorded for a process and for a thread can make it difficult to identify the malicious behavior. Android applications are usually multi-threaded, so it is possible to miss some malicious actions that encompass multiple threads.

Sun et al. [41] propose a technique using CBCFG (component-based control flow graph). CBCFG is a graph of Android APIs as nodes and their control flow precedence relationship as edges. These CBCFGs are then used to detect code reuse in Android repackaged applications and malware variants. The technique 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 app), 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).

Deshotels et al [17] use API call graph signatures and machine learning to identify piggybacked applications. Piggybacking can be used to inject malicious code into a benign app. First, a signature is generated of a benign application and used to identify whether another application is a piggyback of this app. This technique has the same limitations as discussed above regarding detection schemes based on API calls.

Chen et al. [13] present a technique that uses NiCad [16], 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 clones. 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.

Faruki et al. [20] propose a technique to use the similarity of applications to detect Android malware. They use SDHash [38], a statistical approach for selecting fingerprinting features. Therefore the results in the paper have a better false positive rate. Although the FPR reported in [20] is low (1.46%), because of the SDHash technique used, whose main goal is to detect very similar data objects, the ability to detect malware clones is much lower than the technique proposed in this paper.

Alam et al. [5] propose a technique that mines the dominant tree of API calls in an Android application to detect malware. Reused dominant API modules in an Android application are extracted to find clones. The technique works at the dominant API level and is more suitable for finding coarse (higher) level clones. Whereas, *DroidClone* works at MAIL CFG block level and is more refined.

Kalysch et al. [26] propose a technique based on the centroid of CFGs to measure the similarity between Android native codes. The centroid approach is faster than other approaches for matching CFGs. They achieve a DR of 89% and FPR of 10.8%. Disassembling native code to an intermediate code (e.g., a CFG or MAIL) is a non-trivial problem. The work presented in [26] processes only native code libraries and not native code applications. Moreover, they cannot process Intel x86 and 64 bit ARM native code, because of the limitations of the tools used for disassembling. Whereas, DroidClone processes both native code libraries and standalone applications, for both Intel x86 (32 and 64 bit) and ARM (32 and 64 bit) architectures.

Except [26], none of the other above works can find clones in Android native code. Some of them, such as [13] and [20] may not be able to detect syntactically different but semantically similar clones. The technique proposed in [31] is based on dynamic analysis and hence suffers from the known fact that it does not cover all the paths of a program, and may miss certain malicious behaviors.

The techniques proposed in [41], [17] and [5] are based on API calls. 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 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.

3. Overview of the System

Figure 1 provides an overview of the *DroidClone* architecture. Android applications are present either in bytecode or native code. First, the Android applications in bytecode are compiled to native code. Then, we use the tool DroidNative [4] to translate the native code to a MAIL program which is transformed into a CFG, called MAIL CFG for malware analysis and detection. In the next Sections, we explain these steps, and why and how MAIL is adapted for analysis and detection of Android malware code clones.

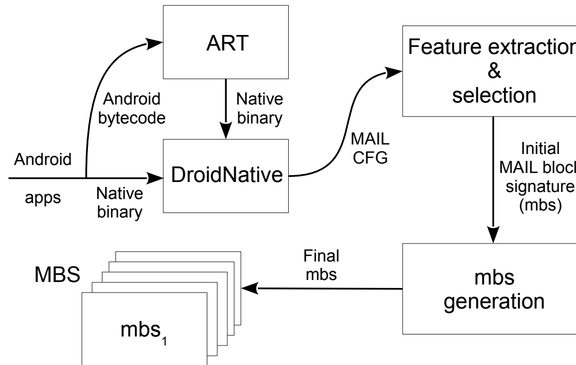

Fig. 1. Overview of *DroidClone*.

3.1. Android Runtime (ART) and DroidNative

Android applications are distributed in the form of APKs (Android application packages), and contains code as Android bytecode (in a *dex* file) and precompiled native binaries. The *dex* (Dalvik Executable Format) [44] files are specific to Android platform. These *dex* files were used to run under the Dalvik Virtual Machine [6] on older Android versions, and are similar to Java class files. Starting with Android 5.0, *dex* files are compiled to native binaries before they are installed. This task is performed by the ART (Android runtime) [7]. ART uses ahead-of-time (dex2oat) compiler for this purpose, which improves the overall execution efficiency and reduces the power consumption of an Android application.

In this paper we use DroidNative [4] to translate an Android native binary (x86 and ARM) to a MAIL program. This allows us to build cross-platform signatures, and process, analyse and detect malware cloned as either bytecode or native code. DroidNative first uses *dex2oat* to compile APKs into native code and then translate the native code to MAIL programs for clone detection.

3.2. Malware Analysis Intermediate Language (MAIL)

Intermediate languages have long been used in compilers to translate the source code into a form that is easy to optimize and provide portability. We apply the same concepts to malware analysis and detection. Several intermediate languages [3, 12, 14, 19, 40] have been developed for optimized analysis and detection of malware. The reason for using MAIL in *DroidClone* is that it has certain advantages over the other languages [12, 14, 19, 40], such as automating and minimizing the effect of obfuscations that makes it suitable for finding clones. Moreover, its publicly available formal model and open-source tools make it easy to use.

There are 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. Each statement in a MAIL program is assigned a type also called a *pattern*. This pattern can be used for matching to assist in clone detection.

Patterns for Annotation The MAIL language contains a total of 21 patterns as shown in Table 1. Each pattern represents the type of a MAIL statement and can be used for easy comparing and matching of MAIL programs.

To assist in matching, a MAIL program is annotated with these patterns. In this paper, an annotated MAIL program is used for matching clones to find malicious code in Android applications. For example, a MAIL jump statement with a constant value and one without a constant value are two different statements, and a MAIL jump statement with a reference to the stack and one with no reference to the stack are two different statements. The MAIL program annotations help make this distinction.

3.3. Control flow analysis

To evade detection, various obfuscations are implemented to create different types of clones. To build resistance against various obfuscations and successfully find

Table 1. Patterns used in MAIL. r_0, r_1, r_2 are the general purpose registers, zf and cf are the zero and carry flags respectively, and sp is the stack pointer.

Pattern	Description
ASSIGN	Assignment statement, *e.g.* $r_0 = r_0 + r_1$;
ASSIGN_CONSTANT	Assignment statement with a constant, *e.g.* $r_0 = r_0 +$ 0x1234;
CONTROL	Control statement with unknown jump, *e.g.* if ($zf = 1$) JMP [$r_0 + r_1 +$ 0x1234];
CONTROL_CONSTANT	Control statement with known jump, *e.g.* if ($zf = 1$) JMP 0x1234;
CALL	Call statement with unknown call, *e.g.* CALL r_2;
CALL_CONSTANT	Call statement with known call, *e.g.* CALL 0x1234;
FLAG	Statement with a flag, *e.g.* $cf = 1$;
FLAG_STACK	Statement that includes flag with stack, *e.g.* $eflags$ = [sp = sp – 0x1234];
HALT	Halt statement, *e.g.* halt;
JUMP	Jump statement with unknown jump, *e.g.* JMP [$r_0 + r_2 +$ 0x1234];
JUMP_CONSTANT	Jump statement with known jump, *e.g.* JMP 0x1234
JUMP_STACK	Return jump, *e.g.* JMP [sp = sp – 0x1234]
LIBCALL	Library call, *e.g.* compare(r_0, r_2);
LIBCALL_CONSTANT	Library call with a constant, *e.g.* compare(r_0, 0x1234);
LOCK	Lock statement, *e.g.* lock;
STACK	Stack statement, *e.g.* r_0 = [sp = sp – 0x1];
STACK_CONSTANT	Stack statement with a constant, *e.g.* [sp = sp + 0x2341] = 0x1234;
TEST	Test statement, *e.g.* r_0 and r_2;
TEST_CONSTANT	Test statement with a constant, *e.g.* r_0 and 0x1234;
UNKNOWN	Unknown assembly instruction that cannot be translated.
NOTDEFINED	The default pattern, and is assigned to every newly created statement.

clones, we extract control flow patterns in a MAIL program of an Android application. For extracting control flow patterns we perform control flow analysis and build a control flow graph (CFG) [1] of a MAIL program as follows.

Definition 1 *A **basic block** is a sequence of MAIL statements, and there are no branches except at the entry and exit points. MAIL statements starting a basic block can be: the first statement; a call to a function or a return statement; a statement following a branch; and target of a branch or a function call. MAIL statements ending a basic block can be: the last statement; call to a function; a return statement; and an unconditional or conditional branch.*

Definition 2 *Control flow edge is an edge between two basic blocks. A CFG is a directed graph $G = (V, E)$, where V is a set of basic blocks and E is a set of control flow edges. The CFG of a MAIL program represents all the paths that can be taken*

*during program execution. An annotated **MAIL CFG** is a CFG such that each statement of the CFG is assigned a MAIL Pattern.*

An annotated MAIL CFG is built for each function in a MAIL program. An example of an annotated MAIL CFG of a function of an Android malware program is shown in Table 2. For simplification, in the rest of the paper, an *annotated MAIL CFG* is just called a *MAIL CFG*. We describe in the following sections, how a MAIL CFG is used at a block level for matching clones to find malicious code in Android applications.

3.4. Preprocessing and Feature Extraction

This Section describes how a MAIL CFG on a block-level is serialized to a string for efficient matching. A MAIL CFG (program) consists of functions. The end of a function is tagged with the symbol `EOF`. In a MAIL CFG, a block starts with the tag `START` and ends with the tag `END`. All the MAIL statements inside these two tags become part of the block. Table 2 shows one of the CFG's for one portion of a MAIL program (a malware), and contains 1 function and 4 blocks. These blocks are parsed using MAIL patterns into block strings as follows:

Table 2. An annotated MAIL CFG (control flow graph) of a function of an Android malware program.

Num	Offset	MAIL Statement	Pattern	Block	Jump To
0	19018	r12 = sp - #8192; start_function_0	[ASSIGN_C]	START	
1	1901c	r12 = [r12, #0];	[ASSIGN_C]		
2	19020	[sp=sp+0x1] = r7;[sp=sp+0x1] = lr;	[STACK]		
3	19024	sp = sp - #16;	[STACK]		
4	19026	r7 = r0;	[ASSIGN]		
5	19028	[sp, #0] = r0;sp = sp - 0x1	[STACK]		
6	1902a	r5 = r1;	[ASSIGN]		
7	1902c	r6 = r2;	[ASSIGN]		
8	1902e	[r5, #8] = r6;	[ASSIGN_C]		
9	19030	if (r6 == 0 jmp 0x1903a);	[CONTROL_C]	END	1903a
10	19032	r2 = [r9, #120];	[ASSIGN_C]	START	
11	19036	r3 = r5 >> #7;	[ASSIGN_C]		
12	19038	[r2, r3] = r2;	[ASSIGN]	END	
13	1903a	lr = #12401;	[ASSIGN_C]	START	
14	1903e	lr = #29198;	[ASSIGN_C]		
15	19042	r0 = #13152;	[ASSIGN_C]		
16	19046	r0 = #28596;	[ASSIGN_C]		
17	1904a	r1 = r5;	[ASSIGN]		
18	1904c	jmp lr;	[JUMP]	END	
19	1904e	sp = sp + 20;	[ASSIGN_C]	START	
20	1904f	r6 = [sp=sp-0x1];pc = [sp=sp-0x1];	[JUMP_S]	END EOF	

```
block 1: ACACSSASAAACCC, block 2: ACACA,
block 3: ACACACACAJ and block 4: ACJS
where: ASSIGN ⇒ A, ASSIGN_C ⇒ AC, JUMP ⇒ J, JUMP_S⇒ JS,
CONTROL_C ⇒ CC, STACK ⇒ S
```

Each block in a MAIL CFG (program), in addition to the above string, is also assigned an initial weight, i.e., the number of times (frequency) it appears in the MAIL CFG. After these initial assignments, we select features and assign the final weight to each block and then build the database of MAIL block signatures for malware detection.

With these initial assignments to each block in a MAIL CFG, in the next two Sections, we describe how features are selected and the final MAIL block signatures are build from a dataset of malware and benign samples, for malware/clone detection.

3.5. Feature Selection

Term Frequency and Inverse Document Frequency (TF-IDF) [33] is widely used, and often considered as an empirical method, in data mining to separate/select relevant features in a set of documents/samples. TF-IDF is used as the amount of information of a term weighted by its occurrence of probability. This Section describes how we adapt TF-IDF weighting method to assign the final weight to a MAIL block, and how this weight is used to select features (MAIL blocks) from a MAIL program.

Let $P = \{p_1, p_2, p_3, ..., p_N\}$ denote the N MAIL programs (preprocessed, as described in the above Section) in a dataset of either malware or binary samples, and $p = \{b_1, b_2, b_3, ..., b_n\}$, where n is the total number of MAIL blocks in program (sample) p. We define the TF and IDF of a MAIL block $b_i \in p$ as follows:

$$TF_i = \frac{f_i}{n} \qquad and \qquad IDF_i = log\left(\frac{N}{M_i}\right)$$

where, f_i is the number of times (frequency) b_i appears in a MAIL program p; and M_i is the number of all the MAIL programs with b_i in it.

Based on these definitions, we formulate our weight assigning approach to the MAIL block b_i as follows:

$$W_i = TF_i \times IDF_i \tag{1}$$

We only keep b_i, if $S_i \geq 3$ and $W_i \geq 0.5$, where S_i is the number of statements in b_i. These minimum values of S_i and W_i are computed empirically.

3.6. Signatures of MAIL blocks

We define signature of a MAIL block $b_i \in p$, in the vector space, as $s_i = \{Sig_i, W_i\}$, where Sig_i represents the MAIL statements in the block as a string of MAIL patterns, as described in Section 3.4.

As an example, for the MAIL CFG shown in Table 2, the following vector is generated: {ACACSSASAAACCC:1.58, ACACA:6.28, ACACACACAJ:4.04 and ACJS:15.35}. There are 4 blocks in this MAIL program and are also found in other portions of the same program (not shown here). 1.58, 6.28, 4.04 and 15.35 are the weights assigned, as defined in equation (1), to each block with respect to the block frequency in the whole MAIL program. Only blocks with 3 or more statements are used for generating the signature. The last block in this MAIL program is discarded because it contains only 2 statements. As we can see, this last block contains a typical epilogue of an assembly program, which is not essential for malware analysis and detection.

After building the signature of a MAIL block, the final signature of a MAIL program p is build as: $M_p = \{s_1, s_2, s_3, ..., s_n\}$. We take the common block signatures among MAIL programs out, and build our database of signatures as follows:

$$V = \bigcup_{p=0}^{N} M_p \tag{2}$$

We build signatures' database of both malware (V_m) and benign (V_b) MAIL programs separately using equation (2). To further improve feature selection, we take malware block signatures (V_m) that are common in benign (V_b) out, and build the final database of malware block signatures (MBS) as follows:

$$MBS = \{x \mid x \in V_m \wedge x \notin V_b\} \tag{3}$$

3.7. Malware/Clone Detection

Figure 2 gives an overview of how malware/clone detection is carried out in *DroidClone*. For malware/clone detection we process a new sample as described in Section 3.4. At this time signature of the new sample contains all its block strings and there respective initial frequencies. After this, a similarity score is computed for the new sample as follows:

$$SimScore = (\frac{\sum_{i=0}^{n} x_i}{N} \times 100) \times (\frac{\sum_{i=0}^{n} y_i}{n} \times 100) \tag{4}$$

where, n is the total number of blocks in the new sample; N is the total number of blocks in MBS; x_i is the similarity value of the ith block (b_i) in the new sample; and x_i and y_i are computed as follows:

$$x_i = \begin{cases} f_i \times W_i & b_i \in MBS \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad y_i = \begin{cases} 1 & b_i \in MBS \\ 0 & \text{otherwise} \end{cases}$$

where f_i is the initial frequency (Section 3.4) and W_i is the final weight (Equation (1)) of b_i.

A *SimScore* is assigned to each sample in the dataset using equation (4). The samples with there *SimScore* values are used for training a classifier for malware detection/classification. A new sample is tagged as malware if *SimScore* of the sample is $\geq$ a certain threshold.

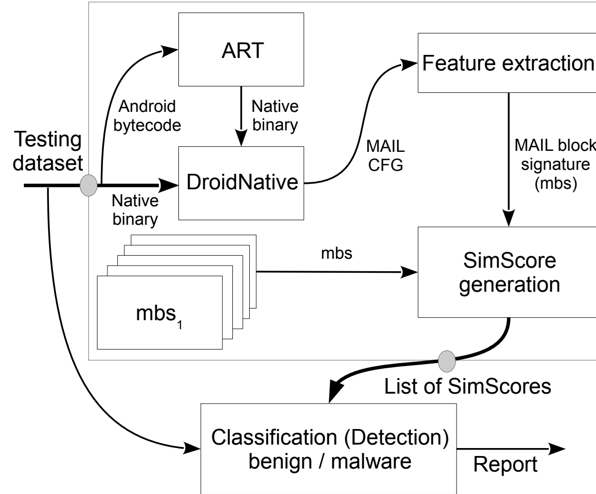

Fig. 2. Overview of malware/clone detection in *DroidClone*.

4. Experimental Evaluation

We carried out an empirical study to analyse the correctness and efficiency of our approach. We carried out two experiments, each using a different validation technique. In the first experiment, we used the holdout cross-validation and in the second experiment, we used n-fold cross-validation. We present in this section the dataset, evaluation metrics, the empirical study (the two experiments), obtained results and analysis. We also present the results of *DroidClone* resistance test against various obfuscations.

4.1. Dataset

Our dataset for the experiments consists of 4180 Android applications. Of these, 2050 are real Android malware programs collected from three different resources [8, 36, 47], and the other 2130 are benign programs containing applications downloaded from Google Play, Android 5.0 system programs, and shared libraries. Tables 3 and 4 shows distribution of the 2130 benign and 2050 malware samples respectively. The dataset also includes 284 Android malware variants. We picked 32 samples from the Miscellaneous class of malware families to generate 9 different classes of malware variants using the obfuscation techniques listed in Table 8. The purpose of generating these variants were to test *DroidClone* against various obfuscation techniques. These variants were also included in the other two validation tests (Sections 4.5 and 4.6).

Table 3. Distribution (native code or byte-code) of the 2130 Android benign samples

Code type	Number of samples
Byte code	1830
Native code	300

Partitioning of the dataset for different experiments, including threshold computation, holdout cross-validation, and n-fold cross-validation is shown in Table 5.

The benign dataset includes both native code (executables and libraries) and byte code. Some of the benign native code applications are: *atrace* – captures kernel events; *bugreport* – reports stack traces and diagnostic information etc; and *bmgr* – backup manager.

The malware dataset shows a variety of samples from different families and includes both native and byte code. The 15 native code malware are standalone applications and the other 4 are libraries. Some of the malware native code applications are: *asroot* – A root exploit and an ELF32 ARM executable file detected by 31 anti-malware programs on virustotal.com, such as Kaspersky, McAfee, and Sophos, etc. *droidpak* – A PE32 Intel x86 executable file detected by 50 anti-malware programs on virustotal.com. It spreads to Windows PC from an infected Android phone. Some of its malicious features of the Android version include sending, uploading and deleting SMS messages, and uploading contacts and location, etc. Most of the Android byte code malware classes are piggybacked applications (ADRD, all of the DroidKungFu families, DroidDream, DroidDreamLight, Geinimi, JSMSHider, and Pjapps). GoldDream, YZHC and most of the malware in the Miscellaneous class are standalone malware applications. DREBIN also contains both piggybacked and standalone malware applications, such as GingerMaster and FakeInstaller families of malware.

4.2. Test Platform

All experiments were run on an Intel® Core(TM) i-7-4510U CPU @ 2.00 GHz with 8 GB of RAM, running Windows 8.1. The ART compiler [7], cross built on the above machine, was used to compile Android applications (malware and benign) to native code.

4.3. Metrics

Before performing the evaluation, we first define our metrics. **DR** (Detection Rate), also called the true positive rate, corresponds to the percentage of samples correctly recognized as malware out of the total malware dataset. **FPR** (False Positive Rate) corresponds to the percentage of samples incorrectly recognized as malware out of the total benign dataset. **Accuracy** is the fraction of samples, including malware and benign, that are correctly detected as either malware or

Table 4. Class distribution of the 2050 Android malware samples. The first 19 (upto DroidPak in the left column) are Android native code (ARM and Intel x86) malware. The rest of the 2031 are Android byte code malware.

Class/family	Number of samples	Class/family	Number of samples
Asroot	3	DroidDream	16
CarrierIQ	8	DroidDreamLight	46
ChathookPtrace	2	DroidKungFu1	35
Cuttherope	1	DroidKungFu2	30
DroidPak	5	DroidKungFu3	310
ADRD	21	DroidKungFu4	95
Airpush	11	Geinimi	68
Appinventor	17	GoldDream	45
AnserverBot	186	JSMSHider	15
BaseBridge	120	KMin	50
Coinkrypt	3	Pjapps	75
DREBIN	494	YZHC	22
DroidChameleonVariants	284	Miscellaneous[1]	88

[1] Includes Android malware samples, AntiObamaScan, DeathRing, FakeDefender, FakeJobOffer, FakeNotify, FakeTimer, FBIRansomLocker, RansomCollection, VoiceChange, and WindSeeker, etc.

benign. ROC (Receiver Operating Characteristic) curve is a graphical plot used to depict the performance of a binary classifier. **AUC** (Area Under the ROC Curve) [21] is equal to the probability that a detector/classifier will correctly classify a sample.

4.4. Threshold Computation

If the *SimScore* of a sample is $\geq$ a certain threshold the sample is detected as malware. That threshold must be determined experimentally. In order to compute the threshold, we separated out 3344 samples, including 1640 malware and 1704 benign. To optimize the results and pick the best threshold automatically we used a RandomTree classifier. During this testing, we perform 10 iterations, each time with a different 334 samples in the testing set and the remaining 3010 samples in the training set. RandomTree built a Decision Tree of size 825 nodes for this dataset, every time, i.e., at each decision node, picking a different *SimScore* ranging from ~0 to ~200, same as the range of the *SimScore* of the 1704 benign samples. Our priority is the DR, therefore at the end of this experiment, a threshold of 15 was picked based on the best DR results. This threshold was then used in the holdout cross-validation of *DroidClone* in Section 4.5.

Table 5. Partitioning of the dataset (total 4180 samples ⇒ 2130 benign and 2050 malware) for different experiments.

Experiment	Total number of samples (benign/malware)	Training samples	Testing samples
Threshold computation	3344[1] (1704/1640)	3010	334
Holdout cross validation	4180 (2130/2050)	3344[1]	836
N-fold cross validation	4050 (2025/2025)	3645[2]	405[2]

[1] Only the training data was used for computing the threshold.
[2] In this experiment we perform 10 (N = 10) iterations, each time with a different 405 samples in the testing set and the remaining 3645 samples in the training set.

Distribution of the 2025 benign and 2025 malware samples (used in n-fold cross-validation) based on their SimScore along with the computed threshold is shown in Figure 3. This distribution of 4050 samples contains 836 samples that were not used to compute the threshold.

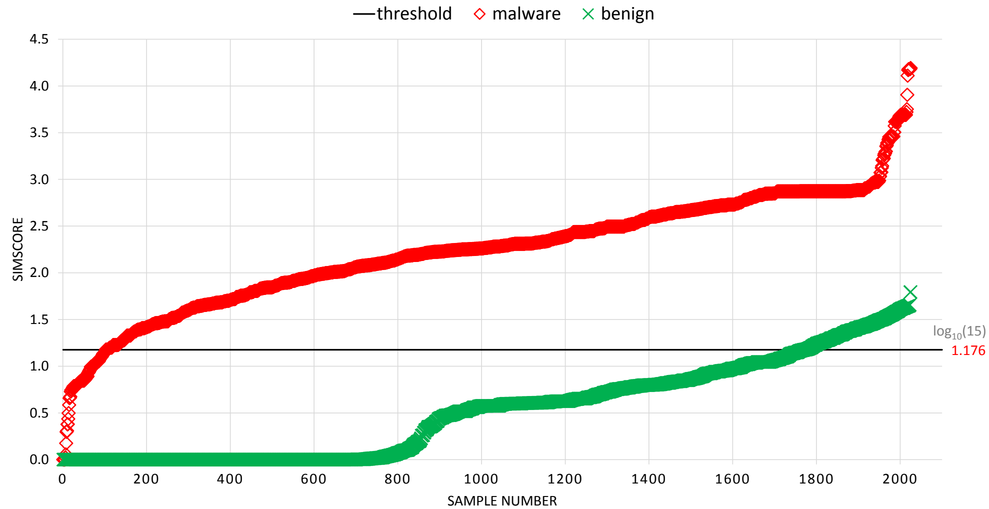

Fig. 3. Distribution of the 2025 malware and 2025 benign samples based on there *SimScore* as defined in equation (4). The threshold of 15 plotted here is computed by RandomTree algorithm (classifier) as explained in Section 4.4.

4.5. Holdout Cross Validation

After selecting the threshold, we carried out the holdout validation using our dataset of 4180 samples. In this method, we randomly divided the data into two parts. The larger part was used for training and the smaller part was used for testing. To keep the training set separate from the testing set, for the larger part we used the same dataset, i.e., the 3344 samples (1704 benign and 1640 malware samples), as used in Section 4.4 to compute the threshold. The smaller part consisted of a total of 836 samples, out of which 426 were benign and 410 malware.

Using the threshold of 15, we carried out the holdout validation experiment as follows.

First, we built the *MBS* database of the 3344 training samples (already labeled as malware or benign) using equation (3). Then we computed SimScore for each of the 836 testing samples (not yet labeled, i.e., unknown) using equation (4). Computing SimScore for each of the testing samples depends on the *MBS* database as explained in Section 3.7. If the SimScore of a testing sample was ≥ 15, it was tagged/labeled as malware otherwise benign.

The results of this experiment, in the form of a confusion matrix, are shown in Table 6. Based on these results we compute DR, FPR and Accuracy of *DroidClone* as follows:

$$DR = \frac{386}{410} \times 100 = 94.2\%$$

$$FPR = \frac{24}{426} \times 100 = 5.6\%$$

$$Accuracy = \frac{386 + 402}{836} \times 100 = 94.3\%$$

Table 6. Results (Confusion Matrix) of *DroidClone* using the holdout validation method.

	Malware	Benign
Malware	386	24
Benign	24	402

From the confusion matrix shown in Table 6, 24 samples were falsely detected as benign and also 24 were falsely detected as malware, and hence *DroidClone* was able to achieve a DR of 94.2% and an FPR of 5.6% with an accuracy of 94.3%. Almost the same results are achieved by the majority of the classifiers during 10-fold cross-validation, as shown in Table 7.

4.6. N-fold Cross Validation

We also use n-fold cross-validation to evaluate the performance of our technique. In n-fold cross-validation, the dataset is divided randomly into n equal size subsets. $n-1$ sets are used for training, and the remaining set is used for testing. The cross-validation process is then repeated n times, with each of the n subsets used exactly once for validation. The purpose of this cross-validation is to produce very systematic and accurate testing results, to limit problems such as overfitting, and to give an insight on how the technique will generalize to an independent (unknown) dataset.

To evaluate the performance of our proposed technique, we trained multiple classifiers using the following machine learning algorithms: *BayesNetwork*: Based on the Bayesian theorem; *BFTree*: Best first decision tree; *NBTree*: Hybrid of decision tree and NaiveBayes classifiers; *RandomForest*: Forest of random trees; *RandomTree*: A decision tree built on a random subset of columns; and *REPTree*: Regression tree representative.

The results of 10-fold cross-validation with these classifiers are shown in Table 7.

Table 7. Results of *DroidClone* using 10-fold cross validation with six different classifiers.

Classifier	DR	FPR	Accuracy	AUC
BayesNetwork	94.2%	0.11	91.3%	0.975
RandomForest	93.1%	0.07	92.8%	0.969
RandomTree	93.1%	0.07	92.8%	0.917
NBTree	93.1%	0.10	91.2%	0.973
REPTree	90.6%	0.04	92.8%	0.969
BFTree	90.3%	0.04	92.9%	0.943

DroidClone successfully achieved DR $\geq$ 90.3% with all the classifiers. Highest DR reached by *DroidClone* is 94.2% with BayesNetwork. The highest AUC 97.5% reached is also with BayesNetwork. The range of FPR attained by *DroidClone* with the six classifiers is from 4% – 11%. The lowest FPR reached by *DroidClone* is with NBTree.

DroidClone reached similar results during n-fold cross-validation with four of the classifiers, as was attained with the holdout validation method in Section 4.5.

4.7. Resistance against Obfuscation

We also tested the resistance of *DroidClone* against various obfuscations. For this purpose, we used the 284 Android malware variants generated by Droid-Chameleon [37] as part of our dataset. The purpose of this experiment is to only

check the resistance of *DroidClone* against various obfuscations and to make this a fair experiment, we took out those malware variants whose original sample was not detected as malware by *DroidClone*. Therefore, out of 284, we selected 273 malware variants for this experiment.

We trained *DroidClone* with the original 28 Android malware and 28 benign samples, and tested with the 273 Android malware variants. Out of the 273 malware variants *DroidClone* was able to successfully classify 247. The description of different Android bytecode obfuscations implemented to test the *DroidClone* and the results obtained are shown in Table 8.

Table 8. Description of different Android bytecode obfuscations implemented to test the resistance of *DroidClone* against various obfuscations.

Obfuscation	Description	Type of clone	DR
ICI	Manipulating call graph of the application.	Type 4	31/31 = 100%
IFI	Hiding function calls through indirection.	Type 4	30/30 = 100%
JNK	Inserting non-trivial junk code, including sophisticated sequences and branches that change the control flow of a program.	Type 3 & 4	15/28 = 53.6%
NOP	Inserting No operation instruction.	Type 1 & 3	32/32 = 100%
RDI	Removing debug information, such as source file names, local and parameter variable names, etc.	Type 2	31/31 = 100%
REO	Reordering the instructions and inserts non-trivial goto statements to preserve the execution sequence of the program. Inserting goto statements changes the control flow of a program.	Type 3 & 4	17/29 = 58.6%
REV	Reverse ordering the instructions and inserting trivial *goto* statements to preserve the execution sequence of the program. Hence changing the control flow of a program.	Type 3 & 4	29/30 = 96.7%
RNF	Renaming fields, such as packages, variables and parameters, etc.	Type 2	31/31 = 100%
RNM	Renaming methods.	Type 2	31/31 = 100%

The results shown in Table 8 demonstrate that *DroidClone* successfully provides resistance to all the trivial (Type 1, 2 & 3 clones) and some non-trivial obfuscations (Type 3 & 4 clones). Type 3 clones can be created by using trivial and non-trivial obfuscations. For example, it depends on the complexity of the reordering of the statements carried out while creating the clone.

This experiment also highlights the limitations of *DroidClone*. Whenever there is a significant change in the control flow of a program it becomes difficult for

DroidClone to generate a matching signature, and hence it fails to detect the similarity. The obfuscation technique JNK, for example, not only adds trivial but also some non-trivial junk code, such as sophisticated sequences and jumps. This makes a significant change in the control flow of a program and making it difficult to detect the malware program based on control flow patterns. To improve this shortcoming, in the future we will add other patterns, such as call flow, etc., to *DroidClone*.

4.8. Comparison with Other Researches

Table 9 shows a comparison of *DroidClone* with other malware detection techniques discussed in Section 2. The reasons for including these works are: (1) all of them are using the similarity/cloning of Android applications to detect malware; (2) have used machine learning to improve the performance and reported at least the DR obtained; (3) have used almost similar kind of Android applications for training and testing as used in this paper.

Table 9. Comparison of *DroidClone* with other malware detection techniques discussed in Section 2

Technique	DR	FPR	Dataset size Benign / malware
DroidClone [1]	94.2%	5.6%	2130 / 2050
SCSdroid [31]	97.9%	2%	100 / 49
DroidSim [41] [2]	96.6%	NA [3]	0 / 706
DomTree [5]	94.3%	4%	150 / 200
NiCad [13] [2]	94.5%	81% [4]	473 / 1170
DroidLegacy [17]	92.7%	21%	48 / 1052
AndroSimilar [20]	76.5%	2%	21,132 / 3309

[1] The results of *DroidClone* reported here are obtained with Naive-Bayes classifier.
[2] No n-fold cross validation was used to evaluate the technique.
[3] The technique was only evaluated with malware samples (no benign samples). Therefore, there is no FPR to report.
[4] For an equitable comparison, FPR of the *Type*-3 clone detector is reported here.

Out of the six techniques compared, *DroidClone* obtained a DR $\sim\geq$ to four of them. Only SCSdroid and DroidSim have a better DR. SCSdroid is tested with only 49 malware and 100 benign samples, whereas *DroidClone* is tested with a much greater number of samples. DroidSim is not tested with benign samples. SCSdroid,

DomTree, and AndroSimilar achieved a lower FPR than *DroidClone*. Like SCS-droid, the number of samples used by DomTree is much lower than *DroidClone*. Although the FPR of AndroSimilar is low because of the SDHash technique [38] used, whose main criteria is to detect closely similar data objects, the ability to detect malware clones is much lower than *DroidClone*.

Like *DroidClone*, DomTree [5] also adapts TF-IDF [33] to improve feature selection. DomTree is the closest technique to *DroidClone*. Therefore, here we present some of the major differences between the two techniques: (1) DomTree does not provide native code malware analysis, and is not independent of the programming language of the code clone. (2) The similarity of two Android applications in *DroidClone* is based on the initial frequency and final weight of a MAIL block, whereas in DomTree it is only based on the presence of a dominant API module. (3) DomTree works at the dominant API level and is more suitable for finding coarse level clones. Whereas, *DroidClone* works at MAIL CFG block (statement) level and is more refined, and can detect clones of smaller size ≥ 3 statements.

Unlike the six works compared here *DroidClone*: uses an intermediate language MAIL to find Android malware clones; it is cross-platform, i.e., independent of the programming language of the Android code clone; can detect clones at a much-refined level; and achieves a DR better or comparable to others.

4.9. Malware Family Classification

The main purpose of the technique proposed in this paper is for binary classification, i.e., only two classes, *malware* and *benign*, and has been successfully used for this purpose. Because of the ability of DroidClone to detect clones we wanted to test if it can classify families of malware, i.e., grouping the samples into there respective families. DroidClone malware detection is based only on the *SimScore* of an Android application. It is difficult to find patterns specific to each family/class of malware just based on their *SimScore* values.

We carried out another experiment, with only selected malware families from the dataset, to test the potential of our proposed technique for predicting the family of a malware sample-based only on its *SimScore*. For this purpose, during training, we separated these classes into their respective *SimScore* groups. For example, $KMin$ was in the $288 - 298$ and $JSMSHider$ in the $41 - 47$ *SimScore* group. We have successfully used *DroidClone* for binary (two classes $\Rightarrow$ *benign* and *malware*) classification in Sections 4.5 and 4.6. Therefore, we used a variation of *one-versus-all* technique [11], which helps build a multiclass classifier from a binary classifier. A binary classifier was built for each class, that predicted the current class based on its *SimScore* group, i.e., if *SimScore* of a sample is in the current class group then it is predicted as positive and all the other samples are predicted as negative. This process was repeated for each class.

Our dataset for this experiment included a total of 197 malware samples from 11 different classes/families. All these samples have successfully been detected in Section 4.5 by *DroidClone* as malware. The results of these classifications into respective families are shown in Table 10. The results show that *DroidClone* was able to successfully separate (classify) most of the malware samples into their

families based only on their *SimScore*, however, the results are not as accurate as general malware classification.

To get good results, in multiclass (in our case 11 classes) classification the input, e.g., the *SimScore* of the input sample, should belong to exactly one class out of the 11 classes and not two (*benign* and *malware*). It is just like mapping the *SimScore* values from 2 dimensions to 11 dimensions. The accuracy is lower because it is difficult to find such mapping successfully just based on the *SimScore* values. *One-versus-all* technique may not always work, as some classes were not predicted accurately by the single binary classifier build for each class.

Table 10. Prediction of some of the selected families of the malware samples.

Malware Family	DR
Appinventor	6/6 = 100%
YZHC	6/6 = 100%
DroidDream	3/3 = 100%
AnserverBot	71/72 = 98.6%
KMin	12/13 = 92.3%
DroidKungFu4	19/21 = 90.5%
DroidKungFu3	37/46 = 80.4%
DroidKungFu2	4/5 = 80%
ADRD	4/5 = 80%
DroidKungFu1	9/12 = 75%
JSMSHider	6/8 = 75%

In the future, we would like to improve this classification by taking into account and combining other features (such as permissions, API and system calls, etc.) with a strong correlation for predicting the family (specific class) of a malware program. We would also like to work on selecting and using some of the parameters, that have been used in this paper to calculate the similarity score (*SimScore*) of a program sample, as a set of features to improve family classification.

4.10. Limitations

DroidClone is based on static analysis of a sample, therefore it requires that the malicious code be available for static analysis. If an Android application contains compressed or encrypted code or requires to download malicious code upon initial execution (dynamic code loading), then the sample will not be correctly analysed by the system. If an Android application dynamically (while executing) link a

third party library, which is not included in the application, then the library will not be processed by *DroidClone*.

DroidClone excels at detecting clone of a malware that has been previously known, but will only detect an unknown (zero-day) clone of a malware, if its control structure is similar, up to a threshold, to an existing malware sample in the training database.

If a clone of a malware is created, by obfuscating a statement in a basic block, in such a way that changes its MAIL pattern (e.g., control flow patterns) beyond a certain percentage (threshold), then *DroidClone* may not be able to detect such an obfuscated clone.

5. Conclusion

Android mobile platform is facing an attack of clones. In this paper, we propose *DroidClone* as a step towards detecting and stopping these clones in Android malware. *DroidClone* uses a new language MAIL to expose control flow patterns in a program, which helps in finding clones that are semantically similar up to a threshold. *DroidClone* is independent of the programming language of the code clones, as it builds cross-platform signatures. When evaluated with real malware and benign Android applications, *DroidClone* obtained a detection rate of 94.2% and false positive rate of 5.6%. *DroidClone*, when tested against various obfuscations, was able to successfully provide resistance against all the trivial and some non-trivial obfuscations.

The research carried out in this paper is just one step towards detecting and stopping Android malware clones. Some of the other works that need to be done in the future are: combining the static analysis performed in this paper with dynamic analysis to detect compressed, encrypted and dynamic loaded code clones; combining control flow patterns of MAIL with other structural features of Android applications to improve clone detection.

In the near future we will further improve the performance of *DroidClone* by combining the *SimScore* with other features, such as permissions, API and system calls, etc. We would also like to adapt the technique proposed in this paper for multiple (into families) classification by combining it with other such techniques. Different parameters were used to calculate the similarity score (*SimScore*) of a program sample. In the future, we would select some of these parameters as features to improve the family classification of *DroidClone*. When a significant change is made in the control flow of a program, it becomes difficult to detect the malware program based on control flow patterns. To improve this shortcoming in *DroidClone*, in the future we will add other patterns, such as call flow, etc.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Inc. (2006)
2. Alam, S., Riley, R., Sogukpinar, I., Carkaci, N.: DroidClone: Detecting android malware variants by exposing code clones. In: DICTAP. pp. 79–84. IEEE (July 2016)

3. Alam, S., Horspool, R.N., Traore, I.: MAIL: Malware Analysis Intermediate Language - A Step Towards Automating and Optimizing Malware Detection. In: Security of Information and Networks. pp. 233–240. ACM SIGSAC (November 2013)
4. Alam, S., Qu, Z., Riley, R., Chen, Y., Rastogi, V.: DroidNative: Automating and optimizing detection of Android native code malware variants. Comput. Secur. 65(C), 230–246 (Mar 2017)
5. Alam, S., Yildirim, S., Hassan, M., Sogukpinar, I.: Mininng Dominance Tree of API Calls for Detecting Android Malware. In: 2018 2nd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT). pp. 1–4. IEEE (2018)
6. Android-Development-Team: Dalvik Virtual Machine. `https://en.wikipedia.org/wiki/Dalvik_(software)` (2016)
7. Android-Development-Team: Android Runtime (ART). `http://en.wikipedia.org/wiki/Android_Runtime` (2021)
8. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K.: Drebin: Effective and explainable detection of android malware in your pocket. In: NDSS (2014)
9. Baker, B.S.: On finding duplication and near-duplication in large software systems. In: Proceedings of 2nd Working Conference on Reverse Engineering. pp. 86–95. IEEE (1995)
10. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272). pp. 368–377. IEEE (1998)
11. Bishop, C.M.: Pattern recognition and machine learning. springer (2006)
12. Cesare, S., Xiang, Y.: Wire–a formal intermediate language for binary analysis. In: TrustCom. pp. 515–524. IEEE (2012)
13. Chen, J., Alalfi, M.H., Dean, T.R., Zou, Y.: Detecting android malware using clone detection. Journal of Computer Science and Technology 30(5), 942–956 (2015)
14. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-Aware Malware Detection. In: Security and Privacy. pp. 32–46. SP '05, IEEE Computer Society (2005)
15. Collberg, C., Thomborson, C., Low, D.: A Taxonomy of Obfuscating Transformations. Tech. rep., University of Auckland (1997)
16. Cordy, J.R., Roy, C.K.: The nicad clone detector. In: Program Comprehension (ICPC), 2011 IEEE 19th International Conference on. pp. 219–220. IEEE (2011)
17. Deshotels, L., Notani, V., Lakhotia, A.: DroidLegacy: Automated Familial Classification of Android Malware. In: SIGPLAN. p. 3. ACM (2014)
18. Ducasse, S., Rieger, M., Demeyer, S.: A language independent approach for detecting duplicated code. In: Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360). pp. 109–118. IEEE (1999)
19. Dullien, T., Porst, S.: Reil: A platform-independent intermediate representation of disassembled code for static code analysis. Proceeding of CanSecWest (2009)
20. Faruki, P., Laxmi, V., Bharmal, A., Gaur, M., Ganmoor, V.: Androsimilar: Robust signature for detecting variants of android malware. Journal of Information Security and Applications 22, 66–80 (2014)
21. Fawcett, T.: An Introduction to ROC Analysis. Pattern Recogn. Lett. 27, 861–874 (2006)
22. Funaro, M., Braga, D., Campi, A., Ghezzi, C.: A hybrid approach (syntactic and textual) to clone detection. In: Proceedings of the 4th International Workshop on Software Clones. pp. 79–80. ACM (2010)
23. Higo, Y., Yasushi, U., Nishino, M., Kusumoto, S.: Incremental code clone detection: A pdg-based approach. In: 2011 18th Working Conference on Reverse Engineering. pp. 3–12. IEEE (2011)

24. Hotta, K., Yang, J., Higo, Y., Kusumoto, S.: How accurate is coarse-grained clone detection?: Comparision with fine-grained detectors. Electronic Communications of the EASST 63 (2014)
25. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: Deckard: Scalable and accurate tree-based detection of code clones. In: Proceedings of the 29th international conference on Software Engineering. pp. 96–105. IEEE Computer Society (2007)
26. Kalysch, A., Milisterfer, O., Protsenko, M., Müller, T.: Tackling androids native library malware with robust, efficient and accurate similarity measures. In: Proceedings of the 13th International Conference on Availability, Reliability and Security. p. 58. ACM (2018)
27. Komondoor, R., Horwitz, S.: Using slicing to identify duplication in source code. In: International static analysis symposium. pp. 40–56. Springer (2001)
28. Koschke, R., Falke, R., Frenzel, P.: Clone detection using abstract syntax suffix trees. In: 2006 13th Working Conference on Reverse Engineering. pp. 253–262. IEEE (2006)
29. Krinke, J.: Identifying similar code with program dependence graphs. In: Proceedings Eighth Working Conference on Reverse Engineering. pp. 301–309. IEEE (2001)
30. Li, Z., Lu, S., Myagmar, S., Zhou, Y.: Cp-miner: Finding copy-paste and related bugs in large-scale software code. IEEE Transactions on software Engineering 32(3), 176–192 (2006)
31. Lin, Y.D., Lai, Y.C., Chen, C.H., Tsai, H.C.: Identifying android malicious repackaged applications by thread-grained system call sequences. Computers & Security 39, 340–350 (2013)
32. Liu, C., Chen, C., Han, J., Yu, P.S.: Gplag: detection of software plagiarism by program dependence graph analysis. In: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 872–881. ACM (2006)
33. Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge University Press, New York, NY, USA (2008)
34. McAfee, C.: McAfee mobile threat report Q1 (© McAfee Corporation 2017)
35. Murakami, H., Hotta, K., Higo, Y., Igaki, H., Kusumoto, S.: Folding repeated instructions for improving token-based code clone detection. In: 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation. pp. 64–73. IEEE (2012)
36. Parkour, M.: Mobile Malware Dump. http://contagiominidump.blogspot.com (2021)
37. Rastogi, V., Chen, Y., Jiang, X.: DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks. In: ASIA CCS. pp. 329–334. ASIA CCS '13, ACM (2013)
38. Roussev, V.: Data Fingerprinting with Similarity Digests. In: Chow, K.P., Shenoi, S. (eds.) Advances in Digital Forensics VI, pp. 207–226. Springer (2010)
39. Roy, C.K., Cordy, J.R.: A survey on software clone detection research. Tech. rep., 541, Queen's University at Kingston, ON, Canada (2007)
40. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: Bitblaze: A new approach to computer security via binary analysis. In: Information systems security, pp. 1–25. Springer (2008)
41. Sun, X., Zhongyang, Y., Xin, Z., Mao, B., Xie, L.: Detecting Code Reuse in Android Applications Using Component-Based Control Flow Graph. In: ICT, pp. 142–155. Springer (2014)
42. Symantec, C.: Symantec security threat report (© Symantec Corporation 2017)
43. Symantec, C.: Symantec security threat report (© Symantec Corporation 2018)

44. Team, A.D.: Android Dalvik Virtual Machine Opcodes. http://developer.android.com/reference/dalvik/bytecode/Opcodes.html (2021)
45. Wahler, V., Seipel, D., Wolff, J., Fischer, G.: Clone detection in source code by frequent itemset techniques. In: Source Code Analysis and Manipulation, Fourth IEEE International Workshop on. pp. 128–135. IEEE (2004)
46. Yuan, Y., Guo, Y.: Boreas: an accurate and scalable token-based approach to code clone detection. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. pp. 286–289. ACM (2012)
47. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: Security and Privacy. pp. 95–109. IEEE (2012)

Shahid Alam is currently working as an assistant professor in the department of Computer Engineering at Adana Alparsalan Turkes Science and Technology University, Adana, Turkey. He recevied his PhD in Computer Science from University of Victoria, Canada in 2014. His research interests include software engineering, programming languages, and cyber security.

Ibrahim Sogukpinar received his PhD degree in Computer and Control Engineering from Technical University of Istanbul in 1995. Currently he is a Professor in the Department of Computer Engineering at Gebze Technical University, Gebze, Turkey. His main research areas are information security, computer networks, applications of information systems, and computer vision.

Received: March 30, 2020; Accepted: August 10, 2020.