DroidNative: Automating and Optimizing Detection of Android Native Code Malware Variants

Shahid Alam[a], Zhengyang Qu[d], Ryan Riley[b], Yan Chen[d], Vaibhav Rastogi[c]

[a]*Department of Computer Engineering, Gebze Technical University, Gebze, Turkey*
[b]*Department of Computer Science and Engineering, Qatar University, Doha, Qatar*
[c]*Department of Computer Science, University of Wisconsin-Madison, Madison, WI, USA*
[d]*Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL, USA*

Abstract

According to the Symantec and F-Secure threat reports, mobile malware development in 2013 and 2014 has continued to focus almost exclusively (~99%) on the Android platform. Malware writers are applying stealthy mutations (obfuscations) to create malware variants, thwarting detection by signature-based detectors. In addition, the plethora of more sophisticated detectors making use of static analysis techniques to detect such variants operate only at the bytecode level, meaning that malware embedded in native code goes undetected. A recent study shows that 86% of the most popular Android applications contain native code, making native code malware a plausible threat vector. This paper proposes *DroidNative*, an Android malware detector that uses specific control flow patterns to reduce the effect of obfuscations and provides automation. As far as we know, DroidNative is the first system that builds cross-platform (x86 and ARM) semantic-based signatures at the Android *native code* level, allowing the system to detect malware embedded in either bytecode or native code. When tested with a dataset of 5490 samples, *DroidNative* achieves a detection rate (DR) of 93.57% and a false positive rate of 2.7%. When tested with traditional malware variants, it achieves a DR of 99.48%, compared to the DRs of academic and commercial tools that range from 8.33% to 93.22%.

Keywords: Android native code, Malware analysis, Malware variant detection, Control flow analysis, Data mining

Version 1.0
This is a preprint of the paper accepted in Elsevier Computers & Security
http://dx.doi.org/10.1016/j.cose.2016.11.011

1. Introduction

Worldwide losses from malware attacks and phishing between July 2011 and July 2012 were $110 billion (Symantec, 2012). In 2013, a 42% increase in malware attacks occurred compared to 2011. Web-based attacks increased by 30% in 2012 (Symantec, 2012). The total number of new malware variants added in 2013 and 2014 were 252 million and 317 million (a 26% increase from 2013), respectively (Symantec, 2015). Mobile malware development in 2013 (Symantec, 2014) and 2014 (F-Secure, Q1 2014) continues to focus almost exclusively (~99%) on the Android platform due to its popularity and open nature.

In recent years, a significant amount of research has focused on detection of Android malware through both static (Deshotels et al., 2014; Faruki et al., 2014b; Huang et al., 2014; Poeplau et al., 2014; Seo et al., 2014; Suarez-Tangil et al., 2014; Sun et al., 2014; Yang et al., 2014; Zhang et al., 2014) and dynamic (Burguera et al., 2011; Enck et al., 2014; Poeplau et al., 2014; Yan and Yin, 2012; Zheng et al., 2013; Zhou et al., 2012) analysis techniques. Among the static analysis techniques, all the systems operate at the Java bytecode level. This is logical, as most

Email addresses: salam@gtu.edu.tr (Shahid Alam), zhengyangqu2017@u.northwestern.edu (Zhengyang Qu), ryan.riley@qu.edu.qa (Ryan Riley), ychen@northwestern.edu (Yan Chen), vrastogi@wisc.edu (Vaibhav Rastogi)

Android applications are primarily written in Java. Many Android applications, however, contain functionality not implemented in Java but instead in *native code*.

In this paper, we refer to native code as machine code that is directly executed by a processor; this is as opposed to bytecode, which is executed in a virtual machine such as the Java virtual machine (JVM) or Dalvik-VM. While the standard programming model for Android applications is to write the application in Java and distribute the bytecode, developers may also include native code binaries as part of their applications. This is frequently done to make use of a legacy library or to write code that is hardware dependent, such as for a game graphics engine, video player, signal processing application, physics simulation, and so on.

Studies estimate that 37% of all Android applications (Afonso et al., 2016) and 86% of the most popular applications (Sun and Tan, 2014) contain native code. Given that existing, static analysis-based malware detection systems operate on a bytecode level, this means that these applications cannot be analyzed completely. If the malicious payload of the app is included in the native code, existing systems would be unable to detect it. Furthermore, native code has more capabilities than bytecode. This is because native code has direct access to the memory of the running process and hence can read and modify the behavior of the bytecode (Sun and Tan, 2014).

In addition to native code, another issue affecting malware detection on Android applications is obfuscation. To protect an Android application from reverse engineering attacks, even legitimate developers are encouraged to obfuscate (Collberg et al., 1997) their code. Similar techniques are used by malware authors to prevent analysis and detection. Obfuscation can be used to make the code more difficult to analyze, or to create *variants* of the same malware to evade detection. Previous studies (Faruki et al., 2014a; Rastogi et al., 2013; Zheng et al., 2012) have evaluated the resilience of commercial anti-malware products when tested against variants of known malware, and found that existing tools are unable to detect variants.

To try and detect malware variants, several works have focused on Android malware detection using static analysis. Some of these, such as (Zhang et al., 2014) and (Deshotels et al., 2014), use API call graphs (ACGs); (Sun et al., 2014) uses component-based ACGs; (Faruki et al., 2014b) uses byte sequence features. Other previous works have focused on graphs, such as control-flow graphs (Christodorescu et al., 2005) and data-dependency graphs (Fredrikson et al., 2010; Kolbitsch et al., 2009), to represent program semantics. These techniques make an exact match against manually crafted specifications for malware detection, and therefore they potentially can be evaded by malware variants. Other semantic-based techniques are computationally intensive, and attempts at reducing the processing time (Lee et al., 2010) have produced poor detection rates.

To alleviate these shortcomings, this paper introduces DroidNative, a malware detection system for Android systems that operates at the *native code level* and can detect malware in either bytecode or native code. DroidNative performs static analysis of the native code and focuses on patterns in the control flow that are not significantly impacted by obfuscations. This allows DroidNative to detect Android malware effectively, even with obfuscations applied and even if the malicious payload is embedded in native code. DroidNative is not limited to only analyzing native code. It is also able to analyze bytecode by making use of the Android runtime (ART) (Android-Development-Team, 2016a) to compile bytecode into native code for analysis. DroidNative optimizes the detection time without reducing the DR by making use of control flow with patterns that enable it to reduce the size of a signature. Our experimental analysis shows that DroidNative is able to detect 93.57% of malware properly, with a false positive rate of only 2.7%.

DroidNative achieves an average detection time of 62.68 sec/sample when tested with 5490 samples, and is ~1.2 and ~2.8 times faster than the other compared works (Kang et al., 2015) and (Zhang et al., 2014), respectively.

The contributions of this work are as follows:

- We propose DroidNative, which is the first system, as far as we know, that builds cross-platform (x86 and ARM) semantic-based signatures for Android and operates at the native code level, allowing it to detect malware embedded in either bytecode or native code. Here, we explain why this potential of DroidNative makes it superior to other Android anti-malware products. The design of the Android system allows applications to load additional code (malicious or benign) from external sources at runtime. A recent study (Poeplau et al., 2014) has shown that using this ability, malware can easily circumvent the Bouncer, a system that analyzes (by performing dynamic analysis) every Android application submitted to Google's store, and several mobile anti-malware products. In one of our experiments while analyzing various Android applications, we found that these applications were downloading some of the families of native code malware (discussed in Section 5.1.1) at runtime to perform malicious activities. Performing static analysis on these dynamically loaded native code

malware products, DroidNative was able to detect them successfully. This also demonstrates that DroidNative is capable of being used as part of a complete hybrid (performing both static and dynamic analysis) system installed for malware (Android native code and bytecode) protection.

- DroidNative adapts and makes significant improvements to the Malware Analysis Intermediate Language (MAIL) (Alam et al., 2013, 2015a,b), a malware analysis technique originally designed for Windows malware. These improvements include adding support for ARM instruction set (Android supports both x86 and ARM architectures); designing a hybrid approach (i.e., combining the two techniques available in MAIL) to increase the overall accuracy and analyzing speed of Android applications for malware detection; and updating the training phase of the classifier by only keeping the distinguished (unique) signatures of the malware samples, which increases the training time while reducing the testing time and memory usage.
- When applied to traditional, bytecode-based Android malware, the detection rate is superior to that of many research systems that only operate at the bytecode level, and when tested with previously unseen variants of existing Android malware, the detection rate is 99.48%, significantly surpassing that of commercial tools. Based on our analysis of related work, DroidNative is noticeably faster than existing systems, making it suitable for real-time analysis.

2. Background

This work has two important areas of background. The first area is the method by which Android applications are executed on Android. The second one involves intermediate languages designed for malware detection.

Android Application Execution. Android applications are available in the form of APKs (Android application packages). An APK file contains the application code as Android bytecode and precompiled native binaries (if present). Traditionally, these applications are executed on the phone using a customized Java virtual machine called Dalvik (Android-Development-Team, 2016b). Dalvik employs just-in-time compilation (JIT), similar to the standard JVM.

Beginning with Android 5.0, the Dalvik-VM was replaced by the Android Runtime, ART (Android-Development-Team, 2016a). ART uses ahead of time (AOT) compilation to transform the Android bytecode into native binaries when the application is first installed on the device. ART's AOT compiler can perform complex and advanced code optimizations that are not possible in a virtual machine using a JIT compiler. ART improves the overall execution efficiency and reduces the power consumption of an Android application. In this work, we use ART to translate bytecode into native code that can be analyzed using our techniques.

Intermediate Languages for Malware Detection. Traditionally, intermediate languages are used in compilers to translate the source code into a form that is easy to optimize and to provide portability. We can apply these same concepts to malware analysis and detection. Various intermediate languages (Alam et al., 2013; Anju et al., 2010; Cesare and Xiang, 2012; Christodorescu et al., 2005; Dullien and Porst, 2009; Song et al., 2008) have been developed for malware analysis and detection. They provide a high-level representation of the disassembled binary program, including specific information such as control flow information, function/API calls and patterns, etc. Having this information available in an intermediate language allows for easier and optimized analysis and detection of malware.

3. Static Analysis in DroidNative

DroidNative performs its analysis on an intermediate representation of an Android application's binary code. To accomplish this, DroidNative makes use of the Malware Analysis Intermediate Language (MAIL) (Alam et al., 2013). In this Section we discuss MAIL, two techniques that use MAIL to generate signatures for malware detection, as well as our extensions to MAIL that make it applicable to Android malware detection.

Table 1: Patterns used in MAIL. r_0, r_1, r_5 are the general purpose registers, zf and cf are the zero and carry flags respectively, and sp is the stack pointer.

Pattern	Description
ASSIGN	An assignment statement, *e.g.* $r_0 = r_0 + r_1$;
ASSIGN_CONSTANT	An assignment statement including a constant, *e.g.* $r_0 = r_0 + 0x01$;
CONTROL	A control statement where the target of the jump is unknown, *e.g.* if (zf == 1) JMP [$r_0 + r_1 + 0x10$];
CONTROL_CONSTANT	A control statement where the target of the jump is known. *e.g.* if (zf == 1) JMP 0x400567;
CALL	A call statement where the target of the call is unknown, *e.g.* CALL r_5;
CALL_CONSTANT	A call statement where the target of the call is known, *e.g.* CALL 0x603248;
FLAG	A statement including a flag, *e.g.* $cf = 1$;
FLAG_STACK	A statement including flag register with stack, *e.g.* $eflags = [sp = sp - 0x1]$;
HALT	A halt statement, *e.g.* HALT;
JUMP	A jump statement where the target of the jump is unknown, *e.g.* JMP [$r_0 + r_5 + 0x10$];
JUMP_CONSTANT	A jump statement where the target of the jump is known, *e.g.* JMP 0x680376
JUMP_STACK	A return jump, *e.g.* JMP [$sp = sp - 0x8$]
LIBCALL	A library call, *e.g.* compare(r_0, r_5);
LIBCALL_CONSTANT	A library call including a constant, *e.g.* compare(r_0, 0x10);
LOCK	A lock statement, *e.g.* lock;
STACK	A stack statement, *e.g.* $r_0 = [sp = sp - 0x1]$;
STACK_CONSTANT	A stack statement including a constant, *e.g.* $[sp = sp + 0x1] = 0x432516$;
TEST	A test statement, *e.g.* r_0 and r_5;
TEST_CONSTANT	A test statement including a constant, *e.g.* r_0 and 0x10;
UNKNOWN	Any unknown assembly instruction that cannot be translated.
NOTDEFINED	The default pattern, *e.g.* all the new statements when created are assigned this default value.

3.1. Malware Analysis Intermediate Language (MAIL)

Compared to other intermediate languages (Anju et al., 2010; Cesare and Xiang, 2012; Christodorescu et al., 2005; Dullien and Porst, 2009; Song et al., 2008) for malware analysis, MAIL (Alam et al., 2013) provides automation, reduces the effects of obfuscations, and provides a publicly available formal model and open-source tools that make it easy to use.

By translating native binaries compiled for different platforms (Android supports ARM, x86, and MIPS architectures) to MAIL, DroidNative achieves platform independence and uses MAIL *patterns* to optimize malware analysis and detection. Eight basic statements (e.g, assignment, control and conditional, etc.) in MAIL can be used to represent the structural and behavioral information of an assembly program.

A total of 21 patterns (see Table 1) are used in the MAIL language, and every MAIL statement is tagged by one of these patterns. A pattern represents the type of a MAIL statement and can be used for easy comparison and matching of MAIL programs. For example, an assignment statement with a constant value and an assignment statement without a constant value are two different patterns. These patterns are used to annotate a CFG (control flow graph) of a program to aid in pattern matching to detect malware.

3.2. Signature Generation Techniques

To counter simple signature-based malware detectors, malware will often use obfuscation techniques (Faruki et al., 2014a; Rastogi et al., 2013; Zheng et al., 2012) to obscure its host application and make it difficult to understand, analyze, and detect the malware embedded in the code. To detect malware protected by obfuscations, we apply two signature generation techniques designed to counter such approaches. MAIL patterns are used to build these signatures.

3.2.1. Annotated Control Flow Graph

The first technique, Annotated Control Flow Graph (ACFG) (Alam et al., 2015b), analyzes a control flow graph derived from an application and annotates it using the patterns listed in Table 1 to capture the control flow semantics of the program. Before describing the technique, we first formally define and describe how we build an ACFG:

Definition 1. *A basic block is a sequence of instructions (statements) with a single entry and single exit points; i.e., no branches (in or out) are used, except at the entry and the exit. Instructions starting a basic block can be: the first instruction; target of a branch or a function call; an instruction following a branch; and a function call or a return instruction. Instructions ending a basic block can be: the last instruction; a conditional or unconditional branch; a function call; and a return instruction.*

Definition 2. *Control flow edge is an edge between (joining) two basic blocks. A control flow edge e from block x to block y is denoted as e = (x, y).*

Definition 3. *A CFG is a directed graph G = (B, E), where B is a set of basic blocks and E is a set of control flow edges, that represents all the paths that can be taken during the program execution. An **ACFG** is a CFG such that each statement of the CFG is assigned a MAIL pattern.*

Annotating the CFG with MAIL patterns allows much smaller CFGs, which can be analyzed reliably and used for malware detection, than are traditionally used in CFG-based techniques. The advantage of this is that an application can be divided into a collection of smaller ACFGs instead of one, large CFG. This collection then can be analyzed and pattern matched with malware much more quickly than the larger CFG.

For example, an application being analyzed is broken up into a set of small ACFGs, which become that application's signature. At detection time, the ACFGs within that signature are compared to the ACFGs of known malware, and if a high percentage of ACFGs in the application match that of known malware, then the application can be flagged as malware. This matching is also very fast, because the individual ACFGs being compared against are typically small.

3.2.2. Sliding Window of Difference

The second technique for signature building, Sliding Window of Difference (SWOD) (Alam et al., 2015a), is an opcode-based technique for malware detection. One of the main advantages of using opcodes for detecting malware is that detection can be performed in real time. However, the current techniques (Austin et al., 2013; Rad et al., 2012; Runwal et al., 2012; Shanmugam et al., 2013) using opcodes for malware detection have several disadvantages: (1) The patterns of opcodes can be changed by using a different compiler, the same compiler with a different level of optimizations, or if the code is compiled for a different platform. (2) Obfuscations introduced by malware variants can change the opcode distributions. (3) The execution time depends on the number of features selected for mining in a program. Selecting too many features results in a high detection rate but also increases the execution time. Selecting too few features has the opposite effect.

A SWOD is a window that represents differences in the distributions of MAIL patterns (which provide a high-level representation of the opcodes of a program) and hence makes the analysis independent of different compilers, compiler optimizations, instruction set architectures, and operating systems. Therefore, the size of the SWOD can be changed to adapt to the distributions of the MAIL patterns of any specific dataset, and is used to compute the weight of a MAIL pattern for the dataset. The difference between the distribution of malware for a MAIL pattern and benign samples of a dataset is denoted by the number of SWODs that can completely slide through (or fit in) this window of difference. This number of SWODs become the weight of the MAIL pattern. For an example and a visual explanation of SWODs, the reader is referred to (Alam et al., 2015a).

To mitigate the effect of obfuscations introduced by malware variants, SWOD uses a control flow weight (CFWeight) scheme based on the following heuristics to extract and analyze the control flow semantics (amount of change in the control flow) of the program. (a) Each CONTROL statement is assigned a weight of 2. (b) The weight of a backwards jump (possibly a loop) is double the length of the jump. (c) The last statement of a block, and each JUMP and CALL statement, are assigned a weight of 1. (d) Each control flow change (JUMP, CONTROL, or CALL) is assigned a weight equal to the length of the jump. (e) A jump whose target is outside the function is assigned a weight equal to its distance (measured as the number of blocks) from the last block of the function + 1.

The final weight of a MAIL statement is the sum of its CFWeight and the weight of the pattern assigned to the statement. The final weights of the statements of a MAIL program are stored in a weight vector that represents the program signature. Then, this signature is sorted in ascending order for easy and efficient comparison using an index-based array.

3.3. Extensions

The existing MAIL system has limitations that affect its performance in detecting Android malware. We have extended it in the following ways:

1. MAIL, originally designed for Windows malware, only provides support for x86 binaries. We have extended MAIL to include support for ARM binaries, which was a significant engineering effort requiring translations to be built from each of the instructions in the ARM instruction set to MAIL.
2. While evaluating DroidNative (Section 5), we found that SWOD is fast but has a low detection rate when applied to Android malware. Meanwhile, ACFG was slow but had much better overall detection rates. As such, we designed a hybrid approach to combine the two techniques. First, DroidNative-SWOD is applied to all samples as a type of pre-filter; then, the samples that are detected as benign by DroidNative-SWOD are tested by DroidNative-ACFG. Next, the samples marked as malware from each phase are combined to produce the final results. This allows us to retain much of the accuracy of ACFG while taking advantage of the speed of SWOD.
3. We found the training phase of MAIL to be inefficient when applied to large numbers of Android apps. While training on known malware, MAIL retains the signatures of all malware in the training set, and during testing, each application is compared to every signature in the malware set. This means that as the training set grows, the size of the signature database becomes inefficiently large. We modified this approach so that during the training phase of DroidNative we only keep the distinguished (unique) signatures of the malware samples. If an existing sample is very similar to the new sample (as measured by a threshold of similarity), only one sample is kept. This process results in a much smaller signature database because similar signatures are pruned. Additionally, it increases the training time, but reduces the testing time and memory usage.

4. System Design and Implementation

In this section, we present the system design of DroidNative. DroidNative provides a fully automated approach to malware detection based on analysis of the native code of an Android application. It analyzes control-flow patterns found in the code to enable comparison to patterns seen in known malware.

Fig. 1 provides an overview of the DroidNative architecture. As mentioned previously, the system operates by analyzing native code. When an app is presented to DroidNative for analysis, the application is separated into bytecode and native code components. Any bytecode components are passed to the ART compiler, and a native binary is produced. Any native code components are passed directly to the next stage. At this point, all parts of the application exist as native code.

Next, the binary code is disassembled and translated into MAIL code. After MAIL is generated, DroidNative is divided into two phases, *training* and *testing*. In the training phase, a classifier is trained on known malware, and malware templates (*behavioral signatures*) are generated for both ACFG and SWOD separately (more details can be found below). In the testing phase, the *similarity detector* uses these malware templates and detects the presence of malware in the application. A simple binary classifier (decision tree) is used for this purpose. If the application being analyzed matches a malware template (within a given threshold computed empirically), then the app is tagged as malware.

4.1. Disassembler

When provided with the native code for an application, the first step in the DroidNative process is to disassemble it. A challenge related to this stage is ensuring that all code is found and disassembled. Two standard techniques (Schwarz et al., 2002) are used for disassembly. (1) The *Linear Sweep* technique starts from the first byte of the code and disassembles one instruction at a time until the end. This assumes that the instructions are stored adjacently, and

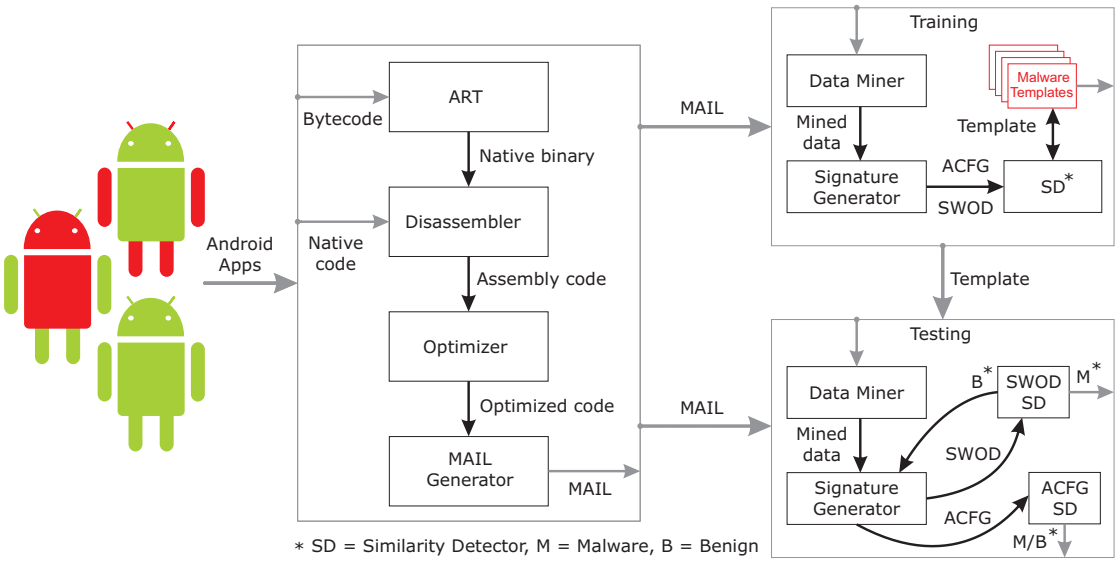


Figure 1: Overview of DroidNative.

hence does not distinguish between embedded data and instructions. When data is mixed with the code, either by the compiler or by malware writers, this technique may produce incorrect results. The advantage of this technique is that it provides complete coverage of the code. (2) The *Recursive Traversal* technique decodes instructions by following the control flow of the program. This technique only disassembles an instruction if it is referenced by another instruction. The advantage of this technique is that it distinguishes code from data. But in the case of a branch whose destination address cannot be determined statically, this technique may fail to find and disassemble valid instructions. To overcome the deficiencies of linear sweep and recursive traversal, we combine these two techniques while disassembling. For example, in the case of an unknown branch, we use MAIL to tag such an address, and use a linear sweep to move to the next instruction.

Another challenge related to this is that most binaries used in Android are stripped, meaning they do not include debugging or symbolic information. This makes it extremely difficult to statically build a call (API) graph for such binaries. It can also become difficult to find function boundaries. IDA Pro (IDA-Pro-Team, 2016), the most popular disassembler, uses pattern matching to find such functions, but these patterns change from one compiler to another and also from one version to another of the same compiler. Therefore, this technique is difficult to maintain and is not general . Our focus is mostly on the detection of malware variants, and we may only need to find where the control is flowing (i.e., just the behavior and not the function boundaries), and then compare this behavior with the previous samples of malware available to detect such malware. Therefore, we handle these problems by building control flow patterns and using them for malware detection.

4.2. Optimizer

A key challenge for static analysis of native code is the large number of instructions supported by the Intel x86 and ARM microprocessors (the two most popular architectures supported by Android). Hundreds of different instructions in instruction sets are employed for these architectures. To speed up static analysis, we need to reduce the number of these instructions that we fully process. Many instructions are not relevant to malware analysis. Examples include the instruction PREFETCH in Intel x86 and PRFM in ARM. These instructions move data from the memory to the cache. The optimizer removes these and other similar instructions. A complete list of these instructions is provided in Appendix A.

Other challenges for static analysis are building a correct control flow graph (CFG) (Aho et al., 2006) of a disassembled Android binary program and reducing the matching time (runtime) of these CFGs. An Android application is

divided into components; each component serves a different purpose and is called when an event occurs. In addition, exceptions permeate the control flow structure of an Android app. Exceptions are resolved by dynamically locating the code specified by the programmer for handling the exception. This produces several unknown branch addresses in a disassembled Android binary program. The existence of multiple entry points makes it difficult to build a correct CFG for the program using static analysis. To handle this problem, DroidNative builds multiple, smaller, interwoven CFGs for a program instead of a single, large CFG. Then, these smaller CFGs are used for matching and malware detection. Matching smaller CFGs, instead of a single large CFG, also helps in reducing the runtime.

4.3. MAIL Generation

The MAIL Generator translates an assembly program to a MAIL program. Some of the major tasks performed by this component are: (1) Translating each assembly instruction to the corresponding MAIL statement. Some of the instructions, such as PUSHA (x86) and STMDB (ARM), are translated to more than one MAIL statement. (2) Assigning a pattern to each MAIL statement. (3) Reducing the number of different instructions by grouping together functionally equivalent assembly instructions so that the MAIL Generator can combine them into one MAIL statement using the MAIL patterns listed in Table 1.

Further details regarding the transformation of Intel x86 and ARM assembly programs into MAIL can be found in our technical report (Alam, 2014).

4.4. Malware Detection

In this section, we explain how the components **Data Miner**, **Signature Generator**, and **Similarity Detector**, shown in Fig. 1, use the two techniques, ACFG and SWOD, for malware detection. The Data Miner searches for the control and structural information in a MAIL program to help the Signature Generator build a behavioral signature (currently two types of signatures, ACFG or SWOD) of the MAIL program. The Similarity Detector matches the signature of the program against the signatures of the malware templates extracted during the training phase, and determines whether the application is malware based on thresholds that are computed empirically, as explained in Section 4.5, for malware detection.

To further reduce the runtime, the current implementation of the Similarity Detector uses a simple binary decision tree classifier, which has proven exceptionally good for detecting malware variants. Currently, the classifier used in the Similarity Detector is only using commonalities among the malware samples. In the future, we will be conducting a study to also find commonalities among the benign samples, and between the malware and benign samples. These commonalities will be used to train the classifier that will help reduce the FPR of the DroidNative further.

4.4.1. DroidNative-ACFG

To detect Android malware variants, an ACFG (as defined in Section 3.2.1) is built for each function in the annotated MAIL program. These ACFGs (i.e., the signature of the Android program) are matched against the malware templates to determine whether the program contains malware. If a high number of ACFGs within a program are flagged as malware, then the application itself is flagged.

We formally define our malware detection approach in terms of subgraph isomorphism (Gross and Yellen, 2005).

Let $G = (V_G, E_G)$ and $H = (V_H, E_H)$ be any two graphs, where V_G, V_H and E_G, E_H are the sets of vertices and edges of the graphs, respectively.

Definition 4. *A vertex bijection (one-to-one mapping) denoted as $f_V = V_G \rightarrow V_H$ and an edge bijection denoted as $f_E = E_G \rightarrow E_H$ are **consistent** if for every edge $e \in E_G$ f_V maps the endpoints of e to the endpoints of edge $f_E(e)$.*

Definition 5. *G and H are **isomorphic graphs** if there exists a vertex bijection f_V and an edge bijection f_E that are consistent. This relationship is denoted as $G \cong H$.*

Based on these definitions, we formulate our ACFG matching approach as follows:

Let $P = (V_P, E_P)$ denote an ACFG of the program and $M = (V_M, E_M)$ denote an ACFG of the malware, where V_P, V_M and E_P, E_M are the sets of vertices and edges of the graphs, respectively. Let $P_{sg} = (V_{sg}, E_{sg})$ where $V_{sg} \subseteq V_P$ and $E_{sg} \subseteq E_P$ (i.e. P_{sg} is a subgraph of P). If $P_{sg} \cong M$ then P and M are considered as matching graphs.

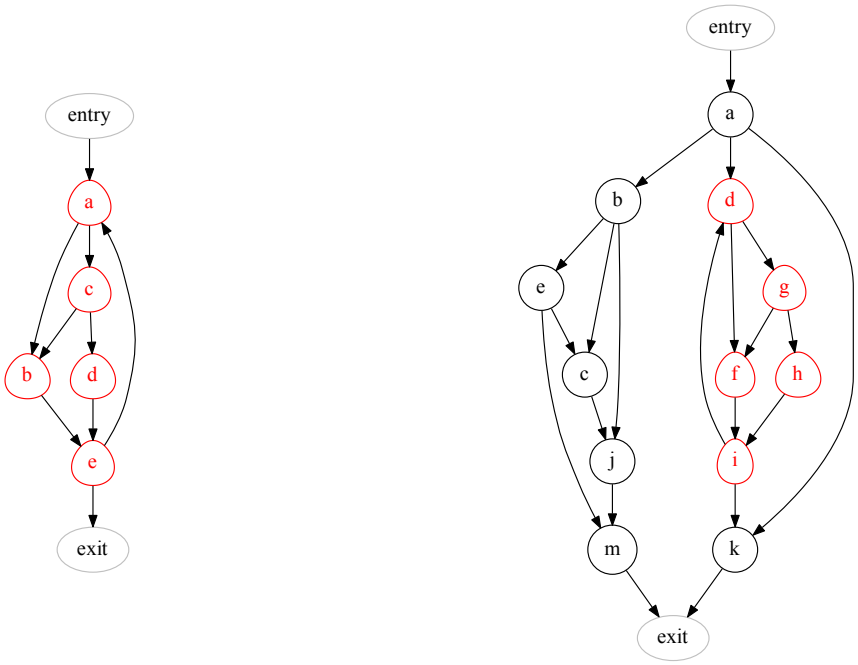


Figure 2: An example of ACFG matching. The graph on the left (M, a malware sample) is matched as a subgraph of the graph on the right (P_{sg}, the malware embedded inside a benign program); i.e., $M = (a, b, c, d, e) \cong P_{sg} = (d, f, g, h, i)$, where $P_{sg} \cong M$ denotes that P_{sg} is isomorphic to M.

If a part of the control flow of a program contains a malware, we classify that program as malware; i.e., if a percentage (compared to some predefined threshold computed empirically) of the number of ACFGs in the signature of a malware program match with the ACFGs in the signature of a program, then the program is classified as malware.

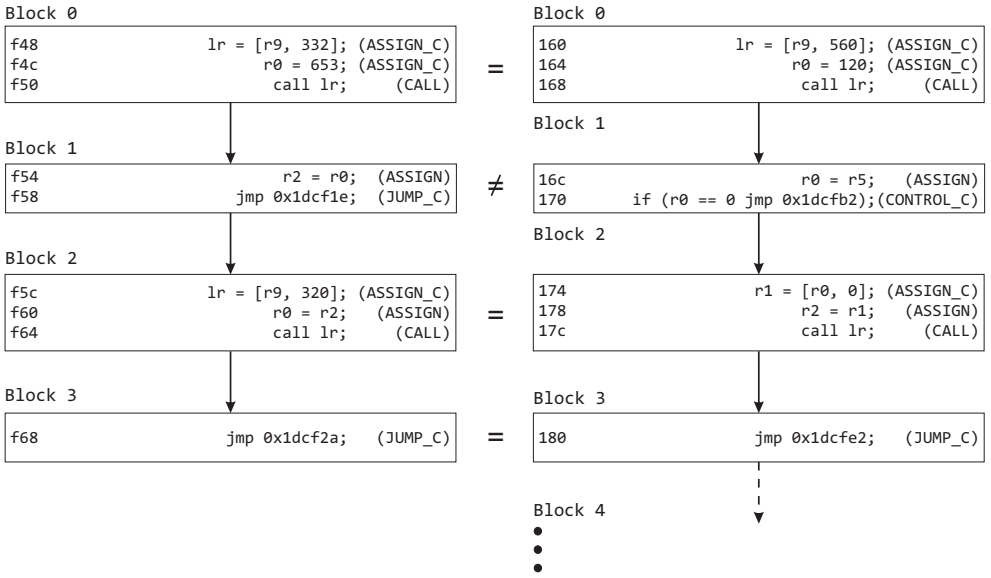
An example of ACFG matching can be found in Fig. 2. Each node $(a, b, c, d, e, f, g, h, i, j, k, m)$ is a block in the ACFGs and contains MAIL statements (one or more - not shown in the figure for simplicity), each with an assigned pattern. These patterns are used for pattern matching to optimize malware detection as described in the next paragraph. A complete (with MAIL statements) visual example of an ACFG is shown in Fig.5.

If an ACFG of a malware program matches with an ACFG of a program, we further use the patterns to match each statement in the matching nodes of the two ACFGs. A successful match requires all the statements in the matching nodes to have the same patterns, although differences in the corresponding statement blocks could exist. An example of a failed *pattern matching* of two *isomorphic* ACFGs is shown in Fig. 3. This process allows us to detect malware with smaller ACFGs, and also allows us to reduce the size (number of blocks) of an ACFG to optimize the detection time. We reduce the number of blocks in an ACFG by merging them together. Two blocks are combined only if their merger does not change the control flow of the program. Given two blocks a and b in an ACFG, if all the paths that reach node b pass through block a, and all the children of a are reachable through b, then a and b are merged. An example of ACFG reduction is shown in Fig.4.

4.4.2. DroidNative-SWOD

Every MAIL statement is assigned a pattern during translation from assembly language to MAIL. Each MAIL pattern is assigned a weight based on the SWOD that represents the differences between MAIL pattern distributions of malware and benign samples, as described in Section 3.2.2. The CFWeight (control flow weight) of a MAIL statement is computed by adding all the weights assigned to the elementary statements involved in it using the heuristics described in Section 3.2.2.

After the signature of a new program is built, as explained in the previous section, it is compared with the signatures of all the training malware samples. In case of a match with any of the signatures, we tag the new program as malware.



(a) One of the ACFGs of a malware sample

(b) One of the ACFGs of a benign program

Figure 3: Example of a failed *pattern matching* of two *isomorphic* ACFGs. The ACFG on the left (a malware sample) is *isomorphic* to the subgraph (blocks 0 – 3) of the ACFG on the right (a benign sample).

An example of a successful SWOD signature matching can be found in Fig. 5, where 11 out of 16 (a threshold computed empirically) index-based array values of the program match with the malware program M_5.

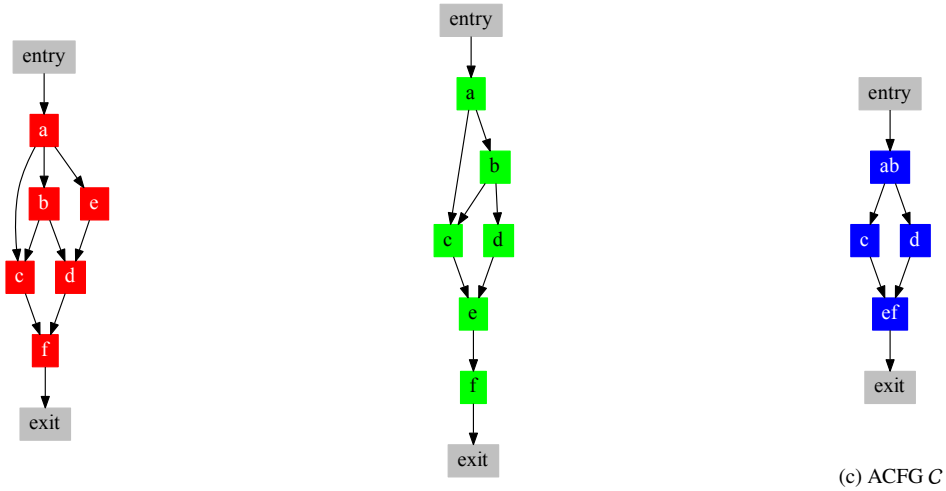
The example MAIL program shown in Fig. 5 (in the top box) contains 6 blocks and 15 MAIL statements, each with an assigned pattern. A weight is assigned to each MAIL pattern, based on the number of SWODs, that represents the differences between distribution of MAIL patterns for malware and benign samples, as described in Section 3.2.2. The MAIL pattern ASSIGN is assigned a weight of 15, ASSIGN_C a weight of 5, and so on. Then, these weights are added to the CFWeight of the statement to compute the total (final) weight of each statement. The first statement in the example is assigned a final weight of 5, the second statement a final weight of 10, and so on.

As an example, this is how the final weight of the statement at offset 0x3eb76 is computed. This statement is a backward jump (i.e., a loop), so its final weight = 2 (weight of pattern) + 2 x length of the jump (loop) + 1 (last statement of the block) + 1 (jump) = 22, where the length of the jump is the number of statements between the source and the target statements, and is 9.

These final weights become the original signature of the MAIL program, as shown in Fig.5. After sorting this array of weights, we store them in an index-based array. For example, if there are 3 statements with weight 5, we store 3 at index 5, and so on. This index-based array is compared with the similar index-based arrays containing the signatures of malware samples. On a successful match, the program under test is tagged as a malware. This increases the storage space of the signatures, but it significantly reduces the matching speed and hence the speed of the malware detection.

4.5. Training

The malware templates used for comparison are generated during a training phase that analyzes known malware and extracts their signatures as explained in Sections 3.2 and 4.4. In this phase, we find the threshold to be used for ACFG matching, and the differences between the MAIL patterns of the malware and benign samples to find the SWOD values, as described in Section 3.2.2. The threshold is the best similarity score of ACFGs matching that achieves the highest DR for the chosen dataset. The ACFG matching threshold computed (using 25% of the dataset) for the dataset used in this paper is 80%. The control flow weight of each MAIL pattern is computed, and then it is used with the SWOD to compute the total weight of each MAIL pattern. Then, the computed threshold and weight of each MAIL pattern are used to build the *Malware Templates* (malware signatures) shown in Fig. 1.



(a) ACFG $\mathcal{A}$

(b) ACFG $\mathcal{B}$

(c) ACFG C

Figure 4: Example of ACFG reduction. ACFG $\mathcal{A}$ is not reducible. ACFG $\mathcal{B}$ with 6 blocks is reduced to ACFG C with 4 blocks.

4.6. Implementation

DroidNative is implemented in C/C++ and contains 60,000+ lines of code, including 32,000+ lines of open-source third-party libraries. The AOT compiler used to compile Android applications to native code was executed on the testing server (as opposed to a phone) by cross compiling it for the server. Given that the AOT compiler to convert bytecode to machine code is provided by the Android open-source project, we have chosen not to include details regarding its algorithms or techniques in the paper. However, the script we used to configure and execute the compiler properly is available at (Riley, 2016). To facilitate the possibility of other authors comparing their systems to DroidNative, we have made the source code of DroidNative publicly available (Alam, 2016).

5. Evaluation

To analyze the correctness and efficiency of DroidNative, we performed a series of experiments of the system, comparing it to existing commercial systems as well as research systems.

5.1. Experimental Setup

5.1.1. Dataset

Our dataset for the experiments consists of 5490 Android applications. Of these, 1758 are Android malware programs collected from three different resources (Arp et al., 2014; Parkour, 2016; Zhou and Jiang, 2012), and the other 3732 are benign programs containing applications downloaded from Google Play, Android 5.0 system programs, and shared libraries.

Table 2 lists the class/family distribution of the 1758 malware samples. As can be seen, some of the malware in our set are native code based while others are bytecode based. *CarrierIQ*, an ARM native code malware, is an adware/spyware that performs keystroke logging and location tracking, and can intercept text messages. *DroidPak*, an x86 native code malware, is an APK loader. It is a malware specifically designed to infect Android devices by downloading a configuration file from a remote server, which is used to install a malicious APK on the device. *ChathookPtrace*, an ARM native code malware, performs malicious code injection through ptrace for the purpose of stealing private information.

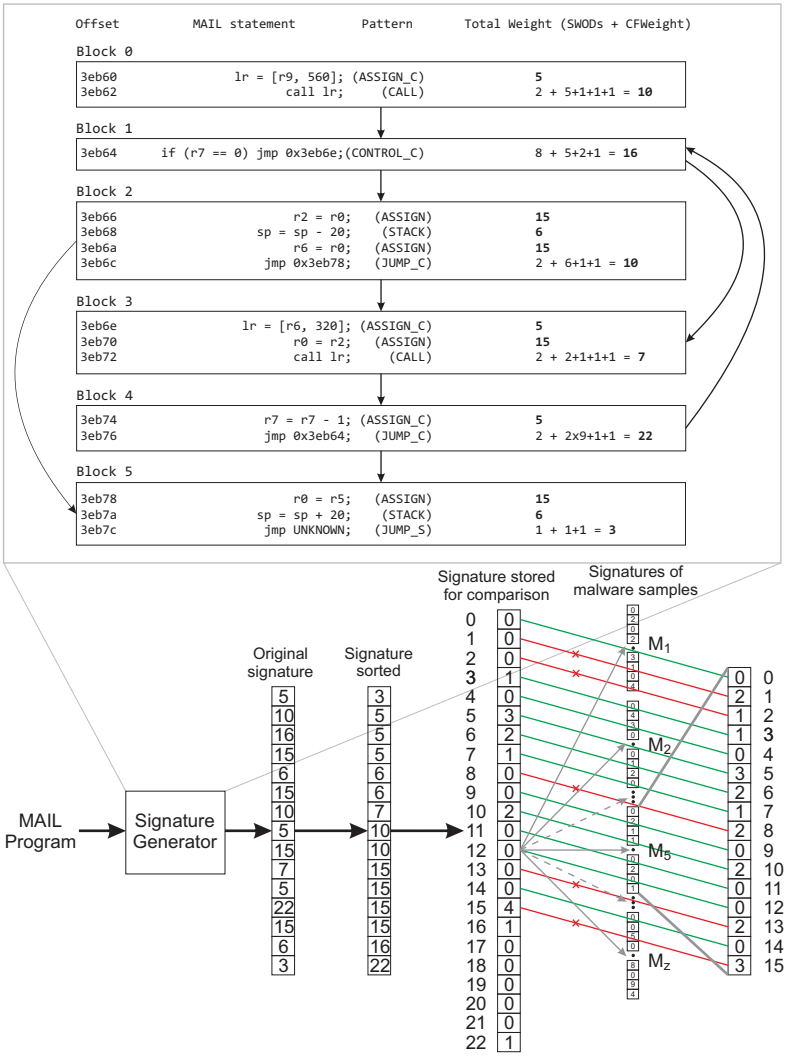


Figure 5: Example of a successful SWOD signature matching. Weights assigned to the MAIL patterns in the example: JUMP_S ⇒ 1, JUMP_C ⇒ 2, CALL ⇒ 2, ASSIGN_C ⇒ 5, STACK ⇒ 6, CONTROL_C ⇒ 8, and ASSIGN ⇒ 15.

The four *DroidKungFu* variants make use of obfuscations to evade detection, such as different ways of storing the command and control server addresses and change of class names, etc. *ADRD*, *DroidDream*, *DroidDreamLight*, and *BaseBridge* also contain a number of variants through obfuscations. *AnserverBot* employs anti-analysis techniques, such as tampering of the repackaged application; aggressively obfuscating its internal classes, methods, and fields to make them humanly unreadable; and detection of some anti-malware programs. The samples collected from Contagiominidump (Parkour, 2016) and Drebin (Arp et al., 2014) consist of a variety of malware applications, such as *RansomCollection*, *FakeMart*, *FakeJobOffer*, *FakeAntiVirus*, *Opfake*, *GinMaster*, *FakeInstaller*, *Plankton*, and *Iconosys* .

5.1.2. Test Platform

All experiments were run on an Intel® Xeon® CPU E5-2670 @ 2.60 GHz with 128 GB of RAM, running Ubuntu 14.04.4. Although such a powerful machine is not required for running DroidNative, it was used to speed the results of n-fold testing, which requires a significant number of repetitions of experiments. The ART compiler, cross built on the above machine, was used to compile Android applications (malware and benign) to native code.

Table 2: Class distribution of the 1758 malware samples, including whether they are native code or byte-code

Class/Family	Number of samples	Code Type
CarrierIQ	8	ARM
DroidPak	5	Intel x86
ChathookPtrace	2	ARM
DroidKungFu1	35	Bytecode
DroidKungFu2	30	Bytecode
DroidKungFu3	310	Bytecode
DroidKungFu4	95	Bytecode
JSMSHider	15	Bytecode
ADRD	21	Bytecode
YZHC	22	Bytecode
DroidDream	16	Bytecode
GoldDream	45	Bytecode
DroidDreamLight	46	Bytecode
KMin	50	Bytecode
Geinimi	68	Bytecode
Pjapps	75	Bytecode
BaseBridge	122	Bytecode
AnserverBot	186	Bytecode
Contagiominidump	89	Bytecode
Drebin	518	Bytecode

5.2. N-Fold Cross Validation

We use n-fold cross validation to estimate 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.

Before evaluating the proposed techniques, we first define our five evaluation 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 benign. **ROC** (Receiver Operating Characteristic) curve is a graphical plot used to depict the performance of a binary classifier. It is a two-dimensional plot, where DR is plotted on the Y-axis and FPR is plotted on the X-axis, and hence it depicts the trade-offs between benefits (DR) and costs (FPR). We want a higher DR and a lower FPR, so a point in ROC space at the top left corner is desirable. **AUC** (Area Under the ROC Curve) is equal to the probability that a detector/classifier will correctly classify a sample.

Through n-fold cross validation, four experiments were conducted using two different subsets of the dataset. The first set consisted of 40 benign and 40 malware samples, and the second set consisted of 350 benign and 330 malware samples. DroidNative-ACFG and DroidNative-SWOD were each tested with these two subsets of the dataset.

The ROC plot of both techniques using two different cross-validations can be seen in Fig. 6. DroidNative-ACFG produces better results than DroidNative-SWOD. This difference is highlighted by the AUC of the two techniques. The AUC of DroidNative-ACFG range from 97.86% – 99.56% while the AUC of DroidNative-SWOD range from 70.23% – 80.19%. Both the techniques show high DRs, but because of its low FPR, DroidNative-ACFG is more

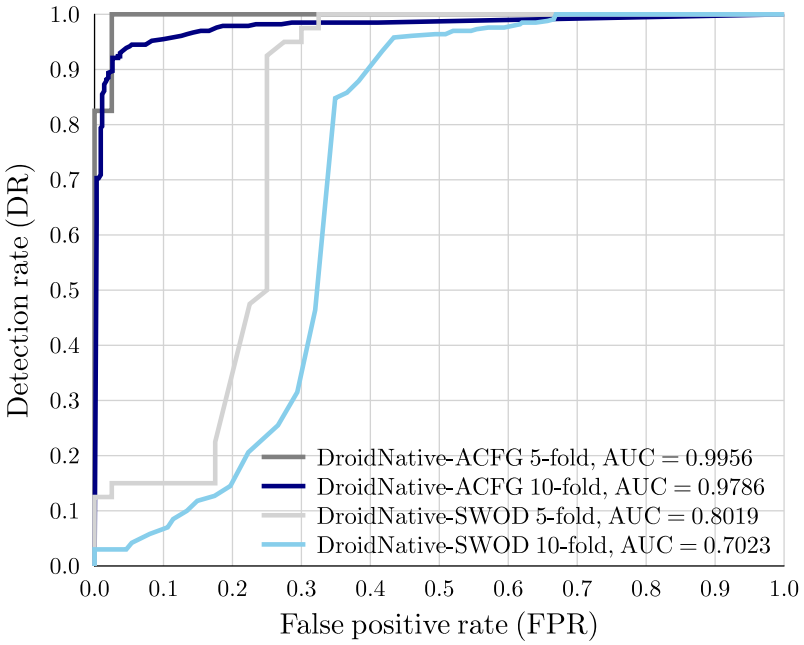


Figure 6: ROC graph plots of ACFG and SWOD of DroidNative

accurate (better AUC) than DroidNative-SWOD.

As mentioned in Section 3.3, we adapted an approach that combines the two techniques to improve the performance of DroidNative. The remainder of the experiments conducted in this paper using DroidNative are based on this hybrid approach.

5.3. Large-Scale Test

To demonstrate the scalability of our approach, we also performed an experiment with the larger dataset of 5490 samples. We trained DroidNative on 1400 malware samples and then performed detection on a set of 358 malware and 3732 benign samples. The results show that DroidNative detects 93.57% of malware samples with a false positive rate of 2.7% and overall accuracy of 97%. The average detection time is 62.68 sec/sample. These results show that DroidNative is applicable to larger datasets.

5.4. Comparison with Commercial Tools

To compare DroidNative to a variety of commercial tools, we generated variants of known malware to test their effectiveness. We selected 30 different families (class/type) of Android malware from (Parkour, 2016), and used DroidChameleon (Rastogi et al., 2013) to generate 210 malware variants based on them. DroidChameleon applies the following seven obfuscations: $\mathcal{NOP}$ = no-operation insertion; $\mathcal{CND}$ = call indirection; $\mathcal{FND}$ = function indirection; $\mathcal{RDI}$ = removing debug information; $\mathcal{RVD}$ = reversing order; $\mathcal{RNF}$ = renaming fields; $\mathcal{RNM}$ = renaming methods.

In total, we generated 210 variants, but 18 of them were unable to run on the new ART runtime. DroidChameleon was implemented and tested using the Dalvik-VM, and apparently some of the variants resulted in bytecode, which is considered invalid by ART. As such, we excluded these variants, leaving 192 samples.

We tested 16 commercial anti-malware programs and DroidNative with these 192 Android malware variants. For testing DroidNative, we included only the original 30 malware samples as the training dataset and the other 192 variants and 20 benign samples as the testing dataset. DroidNative was able to properly classify all benign samples and 191 out of 192 malware samples. This is an overall result of DR = 99.48% and FPR = 0%.

Overall, only one malware variant sample was not detected by DroidNative, and it belonged to class $\mathcal{RVD}$. This class/obfuscation inserts *goto* instructions that change the CFG of a method/program. After further analysis of the

Table 3: Detection results of the 16 commercial anti-malware programs and DroidNative tested with 192 variants of 30 malware families.

Anti-malware	**Detection Rate (%)**							
	NOP	CND	FND	RDI	RVD	RNF	RNM	Overall
DroidNative	100	100	100	100	96.43	100	100	99.48
Kaspersky	92.85	92.85	92	92.59	96.43	93.1	92.59	93.22
Sophos	92.85	92.85	92	85.18	96.43	93.1	92.59	92.18
AVG	78.57	85.71	92	92.59	78.57	93.1	92.59	87.5
BitDefender	85.71	92.58	84	85.18	85.71	86.2	85.18	86.45
DrWeb	85.71	53.57	84	85.18	89.28	79.31	85.18	80.31
ESET	89.28	53.57	88	88.88	17.85	89.65	85.18	72.91
Microsoft	39.28	75	32	33.33	32.14	34.48	33.33	40.1
VIPRE	17.85	57.14	16	14.81	14.28	13.79	14.81	21.35
Symantec	14.28	53.57	8	3.7	7.14	10.34	7.4	15.1
Qihoo-360	10.71	53.57	8	7.4	7.14	6.89	7.4	14.58
Fortinet	10.71	53.57	8	7.4	7.14	6.89	7.4	14.58
TrendMicro	3.57	53.57	4	7.4	3.57	3.57	3.7	11.45
McAfee	7.14	53.57	4	3.7	3.57	3.57	3.7	11.45
TotalDefense	7.14	53.57	0	0	0	0	0	8.85
Malwarebytes	3.57	53.57	0	0	0	0	0	8.33
Panda	3.57	53.57	0	0	0	0	0	8.33

original and the variant malware samples, we noticed that the difference (the number of blocks and the control flow change per ACFG) in ACFGs of the two samples was greater than 50%. As we discuss later in Section 5.6, DroidNative may not be able to detect malware employing excessive control flow obfuscations. Therefore, DroidNative was unable to detect this variant as a copy of the original malware sample.

Next, we tested a variety of commercial anti-malware systems with the same 192 variants. The results are shown in Table 3. The majority (11/16) of the commercial anti-malware tools performed very poorly, with overall DRs ranging from 8.33% – 72.91%. Five of the tools performed significantly better, but still were not able to match the results of DroidNative. Overall, the commercial tools did better in our test than in previous reports (Faruki et al., 2014a; Rastogi et al., 2013; Zheng et al., 2012), which shows that their detection techniques are being improved continuously.

5.5. Comparison with Existing Research

In this section, we compare three (Deshotels et al., 2014; Faruki et al., 2014b; Zhang et al., 2014) state-of-the-art research efforts carried out in 2014 that use machine learning for Android malware variant detection. Very few works report the running time of their detector, and to compare the running time of our detector we selected DroidSift (Zhang et al., 2014). We selected DroidLegacy (Deshotels et al., 2014) because of their use of n-fold cross validation for evaluating the performance of their detector. Like DroidNative, DroidSift and AndroSimilar (Faruki et al., 2014b) also test their techniques on malware variants generated through obfuscations.

Table 4 compares DroidNative with these three approaches. For DroidSift, we have only reported the DR of their malware variant detector, because their anomaly detector does not provide automation of malware detection. In DroidLegacy, the same 48 applications were used in each of the 10-folds (confirmed through e-mail by the authors). During the n-fold testing performed in the paper, a total of 10 different benign applications were detected as malware. Therefore, according to our computation, the actual FPR = 10/48 = 0.2083.

The application of DroidSift and AndroSimilar to general malware and impartiality toward the testing dataset is not validated through n-fold cross validation or other such testing approaches. DroidLegacy requires a larger number

Table 4: Comparison with other malware detection techniques discussed in Section 6. The dataset size for each technique lists the number of benign samples used for testing and the number of malware samples used for training and testing. While testing DroidNative, out of the 1758 malware samples, 1400 were used for training and 358 for testing, which is almost the same ratio as used by the other techniques.

Technique	DR	FPR	Dataset size benign / malware	Runtime per sample	Android native binary
DroidNative	93.57%	2.7%	3732 / 1758	62.68	✓
DroidSift (Zhang et al., 2014)	93%	5.15%	2100 / 1243	175.8	✗
DroidLegacy (Deshotels et al., 2014)	92.73%	20.83%	48 / 1052	✗	✗
AndroSimilar (Faruki et al., 2014b)	76.48%	1.46%	21132 / 3309	✗	✗

of benign samples (more than 48) than was tested in this study for further validation. None of the other techniques can be used to analyze Android native binaries. Out of the three techniques, only AndroSimilar has the potential to be used in a real-time detector, but it has poor DR results. DroidNative, on the other hand, is automated, fast, platform independent, and can analyze and detect Android native code malware variants.

The techniques proposed in DroidSift and DroidLegacy are very similar to DroidNative-ACFG. The difference is that DroidNative-ACFG captures control flow with patterns of the native binary code of the Android application, whereas DroidSift and DroidLegacy capture the API call graph from the bytecode of the Android application, and may not be able to detect malware applications that obfuscate by using fake API calls, hiding API calls (e.g., using class loading to load an API at runtime), inlining APIs (e.g., instead of calling an external API, include the complete API code inside the application), and reflection (i.e., creation of programmatic class instances and/or method invocation using the literal strings, and a subsequent encryption of the class/method name can make it impossible for any static analysis to recover the call).

In DroidSift, during training the malware graphs that were common to the benign graphs are removed (i.e., there can be malware applications whose API calls are similar to the benign applications); hence, their technique may not detect such malware applications. The anomaly detector proposed in DroidSift is based on the behavior of known benign applications. Keeping track of benign applications is impractical in real life, and also makes the process more complex in terms of resources (time and memory) required than the technique proposed in this paper. Hence, DroidSift is much less efficient than DroidNative from a performance perspective. Unlike DroidNative, however, their anomaly detector may be able to detect zero-day malware.

DroidNative-SWOD is similar to AndroSimilar (Faruki et al., 2014b), which is based on SDHash (Roussev, 2010) and is most likely to detect very similar objects, and that is why it produces low DRs. DroidNative-SWOD is a trade-off between accuracy and efficiency, and it detects similar objects at a coarse level; therefore, it produces high DRs.

In terms of variant detection, DroidSift was tested with 23 variants of DroidDream and only one family (listed in Table 2) of malware, and it achieved an overall DR = 100%. One of the obfuscations described in (Rastogi et al., 2013) was used to generate these variants, but it is unclear what specific obfuscation was used. AndroSimilar was tested with 1013 variants of 33 families (most of them are listed in Table 2) of malware, and it achieved an overall DR = 80.65%. The following four obfuscations were used to generate these variants: *method renaming*, *junk method insertion*, *goto insertion*, and *string encryption*. DroidNative was tested with 192 variants of 30 families of malware that were generated using seven types of obfuscation, and achieved an overall DR = 99.84%.

5.6. Limitations

As with all static analysis techniques, DroidNative requires that the malicious code of the application be available for static analysis. Techniques such as packing (Guo et al., 2008) (compressing and encrypting the code) and applications that download their malicious code upon initial execution (Zhauniarovich et al., 2015) cannot be properly analyzed by the system. These issues can be mitigated by combining DroidNative with a dynamic analysis technique.

DroidNative may not be able to detect true zero-day malware. DroidNative excels at detecting variants of malware that have been previously seen, and will only detect a zero-day malware if its control structure is similar (up to a threshold) to an existing malware sample in the saved/training database.

DroidNative may not be able to detect a malware employing excessive control flow obfuscations. Excessive here means changing the control flow of a program beyond a certain percentage (threshold). These control flow obfuscations can be: *control flow flattening*, obscuring the control flow logic of a program by flattening the CFG; *irreducible flow graphs*, reducing these flow graphs loses the original CFG; *method inlining*, replacing a method call by the entire method body, which may increase the number of CFGs in the program; *branch function/code*, obscuring the CFG by replacing the branches with the address of a function/code; and *jump tables*, artificial jump table or jumps in the table are created to change the CFG.

The pattern matching of two ACFGs may fail if the malware variant obfuscates a statement in a basic block in a way that changes its MAIL pattern. DroidNative performs exact (100%) pattern matching for two ACFGs to match. To improve the resilience of DroidNative to such obfuscations, future work we will use a threshold for pattern matching.

6. Related Work

6.1. Native Code Analysis for Other Platforms

Several methods can be used for detecting malware native binaries for standard PCs. We briefly discuss some of these techniques. We also discuss the use of intermediate languages for detecting malware native binaries.

In (Song and Touili, 2012), a method is presented that uses model-checking to detect malware variants. They build a CFG of a binary program, which contains information about the register and the memory location values at each control point of the program. Model-checking is time consuming and can exhaust memory. Times reported in the paper range from a few seconds (10 instructions) to over 250 seconds (10000 instructions). (Eskandari and Hashemi, 2012) presents a method that uses CFGs for visualizing the control structure and representing the semantic aspects of a program. They extend the CFG with extracted API calls to provide more information about the executable program. (Lee et al., 2010) proposes a technique that checks similarities of call graphs (semantic-based signatures) to detect malware variants. Only system calls are extracted from the binary to build the call graph. The call graph is reduced to lower the processing time, but this also reduced the accuracy of the detector. (Leder et al., 2009) uses value set analysis (VSA) to detect malware variants. Value set analysis is a static analysis technique that keeps track of the propagation and changes of values throughout an executable. VSA is an expensive technique and cannot be used for real-time malware detection.

These techniques are compute intensive, and attempts to reduce the processing time tend to produce poor DRs, cannot handle smaller size malware, and are not suitable for real-time detection. Other techniques (Austin et al., 2013; Rad et al., 2012; Runwal et al., 2012; Shanmugam et al., 2013) that mostly use opcode-based analysis for detecting malware variants have the potential to be used for real-time malware detection, but they have several issues: The frequencies of opcodes can change by using different compilers, compiler optimizations, architectures, and operating systems; obfuscations introduced can change the opcode distributions; selecting too many features (patterns) for detection results in a high detection rate but also increases the runtime.

6.1.1. Intermediate Languages

REIL (Dullien and Porst, 2009) is being used for manual malware analysis and detection. Unhandled native instructions are replaced with NOP instructions, which may introduce inaccuracies in disassembling. Furthermore, REIL does not translate FPU, MMX, and SSE instructions, nor any privileged instructions, because these instructions, according to the authors, are not yet being used to exploit security vulnerabilities. SAIL (Christodorescu et al., 2005) represents a CFG of the program under analysis. A node in the CFG contains only a single SAIL instruction, which can make the number of nodes in the CFG extremely large and therefore can make analysis excessively slow for larger binary programs. In VINE (Song et al., 2008), the final translated instructions have all the side effects explicitly exposed as VINE instructions, which makes this approach general but also difficult to maintain platform independence and less efficient for specific security applications such as malware detection. CFGO-IL (Anju et al., 2010) simplifies transformation of a program in the x86 assembly language to a CFG. By exposing all the side effects in an instruction, CFGO-IL faces the same problems as VINE-IL. Furthermore, the size of a CFGO-IL program tends to be much larger than the original assembly program. WIRE (Cesare and Xiang, 2012) is a new intermediate language, and like MAIL, it is specifically designed for malware analysis. WIRE does not explicitly specify indirect jumps, making malware

detection more complicated. Furthermore, the authors do not discuss any side effects of the WIRE instructions, and is unclear how the language is used to automate the malware analysis and detection process.

In contrast to other languages, side-effects are avoided in MAIL, making the language much simpler and providing the basis for efficient malware detection; control flow with patterns in MAIL provide further optimization and automation of malware detection; and publicly available formal models and tools makes MAIL easier to use.

6.2. Detection of Android Malware Variants

Because of the popularity of Android devices, several research efforts on malware detection for Android are underway. For a comprehensive survey of Android malware detection techniques, the reader is referred to (Faruki et al., 2015). Here, we only discuss the three (Deshotels et al., 2014; Faruki et al., 2014b; Zhang et al., 2014) research efforts that are also used for comparison in Section 5.5, and one other research effort (Kang et al., 2015) for comparing its runtime with DroidNative.

DroidSift (Zhang et al., 2014) detects unknown malware using anomaly detection. They generate an API call graph and assign weights based on the data dependency and frequency of security-related API calls. A graph of an application is matched against a graph database of benign applications, and if a matching graph is not found, then an anomaly is reported. These reports are sent to a human expert for verification. For known malware, a signature is generated for each malware sample using the similarity score of the graphs. A classifier, which is used for signature detection, is known as the malware variant detector.

In general, changing (obfuscating) control flow patterns is more difficult (i.e., a comprehensive change in the program is required) than simply changing API call patterns of a program to evade detection. ACG-based techniques look for specific API call patterns (including call sequences) in Android malware programs for their detection, which may also be present in Android benign applications that are protected against reverse engineering attacks. These API call patterns can include packing/unpacking, calling a remote server for encryption/decryption, dynamic loading, and system calls. Moreover, sometimes (e.g., for stripped native binaries of Android) it becomes impossible to build a correct ACG of a program.

DroidLegacy (Deshotels et al., 2014) uses API call graph signatures and machine learning to identify piggybacked applications. Piggybacking can be used to inject malicious code into a benign application. First, a signature is generated of a benign application and used to identify whether another application is a piggyback of this application. This technique has the same limitations as discussed above regarding detection schemes based on API calls.

AndroSimilar (Faruki et al., 2014b) is based on SDHash (Roussev, 2010), a statistical approach for selecting fingerprinting features. SDHash relies on entropy estimates and an empirical study, and is most likely to pick features unique to a data object. Therefore, the results in the paper have a better false positive rate. Although the FPR reported in (Faruki et al., 2014b) is low (1.46%), because of the SDHash technique used (whose main goal is to detect very similar data objects), the ability of detecting malware variants is much lower than the technique proposed in this paper.

The technique proposed in AndroTracker (Kang et al., 2015) profiles Android applications using the creator information and similarity scoring metrics based on API sequence, permissions, and system commands. AndroTracker is not specifically designed for detecting malware variants, and in the paper the authors do not test their detector on malware variants generated through obfuscations.

The runtime of AndroTracker is 70 sec/MB, which is near to the runtime, 59.81 sec/MB, of DroidNative. The improved runtime performance of DroidNative is because of signature size reduction, and pre-filtering of the samples with DroidNative-SWOD, which on average takes 2–3 sec/sample. Extracting the API sequence of an application and system calls is time consuming. Neither is the source code available nor do the authors mention details about the implementation (such as how the API and system calls are extracted) of AndroTracker in the paper (Kang et al., 2015). Therefore, we cannot compare this aspect of AndroTracker.

In terms of performance, only AndroTracker and DroidSift reported runtime results. Although we acknowledge that it is less than ideal to directly compare performance numbers of two different systems tested on completely different hardware, the fact that DroidNative is ~1.2 and ~2.8 times faster than AndroTracker and DroidSift, respectively, is a stronger indicator of the significant performance advantage of the system. More thorough performance testing would require access to the source code of the other systems, which was not available to us.

6.3. App Analysis on iOS

One other effort (Egele et al., 2011) targets Apple's iOS, and performs static binary analysis by building a CFG from the Objective-C binaries. The main goal of the work is to detect privacy leaks and not malware analysis and detection. The authors perform complex data flow analysis, such as backward slicing and others that are time consuming and are not suitable for real-time malware detection. Some differences exist between Android applications and Apple iOS applications, including the use of C, C++, and Java in Android and Objective-C in iOS, and the use of components in Android and use of message passing in iOS. Because of these differences, binary analysis of these applications have significant differences and challenges.

7. Conclusion and Future Work

In this paper, we have proposed *DroidNative* for the detection of both bytecode and native code Android malware. DroidNative uses control flow with patterns, and implements and adapts the two techniques ACFG and SWOD to reduce the effect of obfuscations; uses the language MAIL to provide automation and platform independence; and has the potential to be used for real-time malware detection. To the best of our knowledge, none of the existing static analysis techniques deal with the detection of Android native code malware, and this is the first research effort to detect such malware. Through n-fold cross validation and other test evaluations, DroidNative shows superior results for the detection of Android native code and malware variants compared to both other research efforts and commercial tools. We believe our work provides a promising basis for future researchers interested in the area of Android native code malware variant analysis and detection.

In future work, we will make DroidNative more resilient to other control flow obfuscations through normalization and improve its runtime through parallelization. We will also explore the possibility of using DroidNative as part of a complete hybrid (performing static and dynamic analysis) system, such as a cloud-based malware analysis and detection system. Because of the ability of DroidNative to detect Android malware variants, we would also like to test if it can be used for detecting code clones (Baxter et al., 1998) among benign Android applications. This will help us to detect plagiarism and investigate copyright infringements (Roy and Cordy, 2007) among Android applications.

Acknowledgments

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

References

Afonso, V., Bianchiy, A., Fratantonioy, Y., Doupez, A., Polinox, M., de Geus, P., Kruegely, C., Vigna, G., 2016. Going Native: Using a Large-Scale Analysis of Android Apps to Create a Practical Native-Code Sandboxing Policy. In: NDSS.

Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D., 2006. Compilers: Principles, Techniques, and Tools. Addison-Wesley, Inc.

Alam, S., 2014. MAIL: Malware Analysis Intermediate Language. Technical Report, College of Engineering, University of Victoria. `http://web.uvic.ca/~salam/mail/TR-MAIL.pdf`.

Alam, S., 2016. DroidNative Source Code. `https://bitbucket.org/shahid_alam/droidnative`.

Alam, S., Horspool, R. N., Traore, I., November 2013. MAIL: Malware Analysis Intermediate Language - A Step Towards Automating and Optimizing Malware Detection. In: Security of Information and Networks. ACM SIGSAC, pp. 233–240.

Alam, S., Sogukpinar, I., Traore, I., Nigel Horspool, R., 2015a. Sliding Window and Control Flow Weight for Metamorphic Malware Detection. Journal of Computer Virology and Hacking Techniques.

Alam, S., Traore, I., Sogukpinar, I., 2015b. Annotated Control Flow Graph for Metamorphic Malware Detection. The Computer Journal.

Android-Development-Team, 2016a. Android Runtime (ART). `http://en.wikipedia.org/wiki/Android_Runtime`.

Android-Development-Team, 2016b. Dalvik Virtual Machine. `https://en.wikipedia.org/wiki/Dalvik_(software)`.

Anju, S., Harmya, P., Jagadeesh, N., Darsana, R., 2010. Malware detection using assembly code and control flow graph optimization. In: ACM-W. ACM, p. 65.

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., 2014. Drebin: Effective and explainable detection of android malware in your pocket. In: NDSS.

Austin, T. H., Filiol, E., Josse, S., Stamp, M., 2013. Exploring Hidden Markov Models for Virus Analysis: A Semantic Approach. In: HICSS. pp. 5039–5048.

Baxter, I. D., Yahin, A., Moura, L., Anna, M. S., Bier, L., 1998. Clone detection using abstract syntax trees. In: Software Maintenance, 1998. Proceedings., International Conference on. IEEE, pp. 368–377.

Burguera, I., Zurutuza, U., Nadjm-Tehrani, S., 2011. Crowdroid: Behavior-based Malware Detection System for Android. In: SPSM. SPSM '11. ACM, pp. 15–26.

Cesare, S., Xiang, Y., 2012. Wire–a formal intermediate language for binary analysis. In: TrustCom, 2012. IEEE, pp. 515–524.

Christodorescu, M., Jha, S., Seshia, S. A., Song, D., Bryant, R. E., 2005. Semantics-Aware Malware Detection. In: Security and Privacy. SP '05. IEEE Computer Society, pp. 32–46.

Collberg, C., Thomborson, C., Low, D., 1997. A Taxonomy of Obfuscating Transformations. Tech. rep., University of Auckland.

Deshotels, L., Notani, V., Lakhotia, A., 2014. DroidLegacy: Automated Familial Classification of Android Malware. In: SIGPLAN. ACM, p. 3.

Dullien, T., Porst, S., 2009. Reil: A platform-independent intermediate representation of disassembled code for static code analysis. Proceeding of CanSecWest.

Egele, M., Kruegel, C., Kirda, E., Vigna, G., 2011. PiOS: Detecting Privacy Leaks in iOS Applications. In: NDSS.

Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., Sheth, A. N., 2014. TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones. Comm of the ACM, 99–106.

Eskandari, M., Hashemi, S., 2012. A Graph Mining Approach for Detecting Unknown Malwares. Journal of Visual Languages and Computing 23, 154 – 162.

F-Secure, C., © F-Secure Corporation Q1 2014. F-Secure mobile threat report.

Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M., Conti, M., Muttukrishnan, R., 2015. Android Security: A Survey of Issues, Malware Penetration and Defenses. Comm Surveys Tutorials, IEEE 17 (2), 998–1022.

Faruki, P., Bharmal, A., Laxmi, V., Gaur, M., Conti, M., Rajarajan, M., 2014a. Evaluation of Android Anti-malware Techniques against Dalvik Bytecode Obfuscation. In: TrustCom. pp. 414–421.

Faruki, P., Laxmi, V., Bharmal, A., Gaur, M., Ganmoor, V., 2014b. Androsimilar: Robust signature for detecting variants of android malware. Journal of Information Security and Applications 22, 66–80.

Fredrikson, M., Jha, S., Christodorescu, M., Sailer, R., Yan, X., 2010. Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors. In: Security and Privacy. SP '10. IEEE Computer Society, pp. 45–60.

Gross, J. L., Yellen, J., 2005. Graph Theory and Its Applications, Second Edition. Chapman & Hall, London, UK.

Guo, F., Ferrie, P., Chiueh, T.-C., 2008. A study of the packer problem and its solutions. In: RAID. Springer-Verlag, pp. 98–115.

Huang, J., Zhang, X., Tan, L., Wang, P., Liang, B., 2014. AsDroid: detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In: ICSE. pp. 1036–1046.

IDA-Pro-Team, 2016. IDA Pro. `https://www.hex-rays.com/products/ida/index.shtml`.

Kang, H., Jang, J.-w., Mohaisen, A., Kim, H. K., 2015. Detecting and classifying android malware using static analysis along with creator information. Int. J. Distrib. Sen. Netw. 2015, 7:7–7:7.

Kolbitsch, C., Comparetti, P. M., Kruegel, C., Kirda, E., Zhou, X., Wang, X., 2009. Effective and Efficient Malware Detection at the End Host. In: USENIX. SSYM'09. USENIX Association, pp. 351–366.

Leder, F., Steinbock, B., Martini, P., 2009. Classification and Detection of Metamorphic Malware Using Value Set Analysis. In: MALWARE. pp. 39 – 46.

Lee, J., Jeong, K., Lee, H., 2010. Detecting Metamorphic Malwares Using Code Graphs. In: SAC. ACM, pp. 1970 – 1977.

Parkour, M., 2016. Mobile Malware Dump. `http://contagiominidump.blogspot.com`.

Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., Vigna, G., 2014. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In: NDSS. pp. 23–26.

Rad, B., Masrom, M., Ibrahim, S., 2012. Opcodes Histogram for Classifying Metamorphic Portable Executables Malware. In: ICEEE. pp. 209 – 213.

Rastogi, V., Chen, Y., Jiang, X., 2013. DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks. In: ASIA CCS. ASIA CCS '13. ACM, pp. 329–334.

Riley, R., 2016. AOT Compiler Script. `https://gist.github.com/rriley/ce38ab20532c1d7a2667`.

Roussev, V., 2010. Data Fingerprinting with Similarity Digests. In: Chow, K.-P., Shenoi, S. (Eds.), Advances in Digital Forensics VI. Springer, pp. 207–226.

Roy, C. K., Cordy, J. R., 2007. A survey on software clone detection research. Tech. rep., 541, Queens University at Kingston, ON, Canada.

Runwal, N., Low, R. M., Stamp, M., 2012. Opcode Graph Similarity and Metamorphic Detection. Journal in Computer Virology 8, 37 – 52.

Schwarz, B., Debray, S., Andrews, G., 2002. Disassembly of Executable Code Revisited. In: WCRE. IEEE Computer Society, Washington, DC, USA, p. 45.

Seo, S.-H., Gupta, A., Mohamed Sallam, A., Bertino, E., Yim, K., 2014. Detecting mobile malware threats to homeland security through static analysis. Journal of Network and Computer Applications, 43–53.

Shanmugam, G., Low, R. M., Stamp, M., 2013. Simple substitution distance and metamorphic detection. Journal of Computer Virology and Hacking Techniques 9, 159–170.

Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P., 2008. Bitblaze: A new approach to computer security via binary analysis. In: Information systems security. Springer, pp. 1–25.

Song, F., Touili, T., 2012. Efficient Malware Detection Using Model-Checking. In: Formal Methods. Vol. 7436 of Lecture Notes in Computer Science. Springer, pp. 418–433.

Suarez-Tangil, G., Tapiador, J. E., Peris-Lopez, P., Blasco, J., 2014. Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. Expert Systems with Applications, 1104–1117.

Sun, M., Tan, G., 2014. NativeGuard: Protecting Android Applications from Third-party Native Libraries. In: WiSec. WiSec '14. ACM, pp. 165–176.

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

Symantec, C., © Symantec Corporation 2012. Symantec cybercrime and Internet security threat report.

Symantec, C., © Symantec Corporation 2014. Symantec cybercrime and Internet security threat report.

Symantec, C., © Symantec Corporation 2015. Symantec cybercrime and Internet security threat report.
Yan, L. K., Yin, H., 2012. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In: USENIX. Security'12. pp. 29–29.
Yang, C., Xu, Z., Gu, G., Yegneswaran, V., Porras, P., 2014. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In: Computer Security-ESORICS. Springer, pp. 163–182.
Zhang, M., Duan, Y., Yin, H., Zhao, Z., 2014. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In: CCS. ACM, Scottsdale, AZ, pp. 1105–1116.
Zhauniarovich, Y., Ahmad, M., Gadyatskaya, O., Crispo, B., Massacci, F., 2015. Stadyna: Addressing the problem of dynamic code updates in the security analysis of android applications. In: CODASPY. ACM, pp. 37–48.
Zheng, M., Lee, P. P., Lui, J. C., 2012. ADAM: an automatic and extensible platform to stress test android anti-virus systems. In: DIMVA. Springer, pp. 82–101.
Zheng, M., Sun, M., Lui, J., 2013. Droid Analytics: A Signature Based Analytic System to Collect, Extract, Analyze and Associate Android Malware. In: TrustCom. IEEE, pp. 163–171.
Zhou, Y., Jiang, X., 2012. Dissecting android malware: Characterization and evolution. In: Security and Privacy. IEEE, pp. 95–109.
Zhou, Y., Wang, Z., Zhou, W., Jiang, X., 2012. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In: NDSS.

Appendix A. Ignored Instructions

List of x86 and ARM instructions, in alphabetical order, that are not translated to MAIL statements:

x86

```
3DNOW, AAA, AAD, AAM, AAS, AESDEC, AESDECLAST, AESENC, AESENCLAST, AESIMC, AESKEYGENASSIST, ARPL
BOUND, DAA, DAS, EMMS, ENTER, GETSEC, CLFLUSH, CLTS, CMC, CPUID, CRC32
FCLEX, FDECSTP, FEDISI, FEEMS, FENI, FFREE, FINCSTP, FINIT, FLDCW, FLDENV, FNCLEX, FNINIT
FNSAVE, FNSTCW, FNSTENV, FNSTSW, FRSTOR, FSAVE, FSETPM, FSTCW, FSTENV, FSTSW, FXRSTOR
FXRSTOR64, FXSAVE, FXSAVE64, FXTRACT
INT 3, INVD, INVEPT, INVLPG, INVLPGA, INVPCID, INVVPID, LEAVE, LFENCE, LZCNT
MFENCE, MONITOR, MPSADBW, MWAIT, PAUSE, PREFETCH, PSAD, PSHUF, PSIGN
RCL, RCR, RDRAND, RDTSC, RDTSCP, ROL, ROR, RSM
SFENCE, SHUFPD, SHUFPS, SKINIT, SMSW, SYSCALL, SYSENTER, SYSEXIT, SYSRET
VAESDEC, VAESDECLAST, VAESENC, VAESENCLAST, VAESIMC, VAESKEYGENASSIST, VERR, VERW, VMCALL
VMCLEAR, VMFUNC, VMLAUNCH, VMLOAD, VMMCALL, VMPSADBW, VMREAD, VMRESUME, VMRUN, VMSAVE
VMWRITE, VMXOFF, VSHUFPD, VSHUFPS, VZEROALL, VZEROUPPER
WAIT, WBINVD, XRSTOR, XSAVE, XSAVEOPT
```

ARM

```
BKPT, CDP, CDP2, CLREX, CLZ, CPS, CPSID, CPSIE, CRC32, CRC32C, DBG, DCPS1, DCPS2, DCPS3, DMB, DSB
HVC, ISB, LDC, LDC2, MCR, MCR2, MCRR, MCRR2, MRC, MRC2, MRRC, MRRC2, PLD, PLDW, PLI, PRFM
SETEND, SEV, SHA1C, SHA1H, SHA1M, SHA1P, SHA1SU0, SHA1SU1, SHA256H, SHA256H2, SHA256SU0
SHA256SU1, SMC, SSAT, SSAT16, STC, STC2, SVC
USAT, USAT16, VCVT, VCVTA, VCVTB, VCVTM, VCVTN, VCVTP, VCVTR, VCVTT, VEXT, VLD1, VLD2, VLD3, VLD4
VLD1, VLD2, VLD3, VLD4, VQRDMULH, VQRSHL, VQRSHRN, VQRSHRN, VQRSHRUN, VQRSHRUN, VQSHL, VQSHLU
VQSHL, VQSHRN, VQSHRN, VQSHRUN, VQSHRUN, VRINTA, VRINTM, VRINTN, VRINTP, VRINTR, VRINTX, VRINTZ
VRSHL, VRSHR, VRSHRN, VRSRA, VRSUBHN, VST1, VST2, VST3, VST4, VTBL, VTBX, VTRN, VUZP, VZIP
WFE, WFI, YIELD
```