Hunting ELFs: An investigation into Android malware detection

Thomas Atkinson

Technical Report

RHUL–ISG–2017–1

10 February 2017

Information Security Group
Royal Holloway University of London
Egham, Surrey, TW20 0EX
United Kingdom

Student Number: 100832601
Name: Thomas Sheridan, Atkinson

Hunting ELFs: An Investigation Into Android Malware Detection

Supervisor: Dr. Lorenzo Cavallaro

Submitted as part of the requirements for the award of the
MSc in Information Security at
Royal Holloway, University of London

I declare that this assignment is all my own work and that I have acknowledged all quotations from the published or unpublished works of other people. I declare that I have also read the statements on plagiarism in Section 1 of the Regulations Governing Examination and Assessment Offences and in accordance with it I submit this project report as my own work.

Signature:

Date:

Academic Year 2015/2016

Abstract

Hunting ELFs: An Investigation Into Android Malware Detection

by
Thomas Sheridan Atkinson

Master of Science
in
Information Security

Royal Holloway, University of London
2015/2016

Android has risen to become the de facto operating system for mobile technologies. Its huge success and ubiquity has made it a target for malware writers. It has been found that recently sampled Android malware apps make use of repackaged malicious ELF binaries. The research carried out by this project has found a previously unknown technique used by malicious ELF binaries to hide calls to external packed binaries. An original and robust method for detecting ELF binaries that make calls to a packed binary has been developed and implemented into a state of the art framework with an aim to improving how the framework detects malicious APKs. When trained and tested on 84,979 benign samples and 16,132 malware samples, an accuracy of 96.50% was maintained using XGBoost as the classifier algorithm. The accuracy achieved using the Extra Trees algorithm was improved by 0.01% to 95.79% and the accuracy achieved when using the Random Forest algorithm was increased by 0.05% to 95.34%.

Acknowledgement

This thesis would have not been possible without the work carried out by Guillermo Suarez-Tanigl, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto and Lorenzo Cavallaro on their Android malware detection and identification system DroidSieve upon which this project is based.

I would like to further extend my special thanks to Guillermo Suarez-Tanigl, Santanu Kumar Dash and Lorenzo Cavallaro whose continual help and support were essential to the success of this project. It has been a pleasure and a privilege to add to their work and I hope that I have added something of value.

I dedicate this work to strong cups of tea and
to everyone that helped me along the way!

Contents

List of Figures

List of Tables

Listings

1 Introduction

The number of devices running Android software has exponentially increased since the appearance of the HTC Dream in 2008 with google announcing in 2014 that it had over a billion active monthly users [22]. According to a survey by Gartner [13], as of November 2015 Android held 84.7% of the market share of worldwide smartphone sales [13] which equated to just under 0.3 billion Android smartphones [13] sold in the 3rd quarter of 2015 alone.

Android was conceived by Andy Rubin and his team and was quickly acquired a few years later by Google in 2005 [7]. Then in 2007 the Open Handset Alliance was formed by Google, HTC, Motorola, Qualcom and other industry leaders across the world for collaborative, open source development of the Android mobile operating system [2]. As the Open Handset Alliance put it themselves, before the advent of Android it had been difficult for mobile developers to respond to the ever changing and increasing demands of consumers [2]. Android went on to have a dramatic impact on the mobile market and became the de facto OS for mobile phones [13] [18].

The number of apps developed for Android has also seen incredible growth with over 1.6million apps available in the app store as of July 2015, which is 0.1million more than the iOS store [38]. The massive surge in market share and number of developed apps has brought with it the predictable security flaws and exploitation by attackers. Android smartphones are an ever increasing target for attackers. Indeed, the number of Android malware applications rose by 614% between March 2012 and March 2013 alone [21].

There have been many different approaches at detecting and categorising Android malware. Both static analysis and dynamic analysis have been widely used throughout the literature. A summary of the literature on Android malware detection and categorisation can be found in Chapters 3.2 and 3.3. The work carried out in this project focuses on static analysis based DroidSieve (see Chapter 4.1). This project has aimed to add to the DroidSieve framework and to improve it.

The objectives of this project are as follows:

- **Objective 1.** Improve how DroidSieve detects and analyses malicious ELF files.

  Rationale: DroidSieve currently only examines ELFs superficially and only makes use of a python implementation of readelf to accomplish this. There is likely potential for improvement in this area.

  Related Sections: Chapter 5.1.

- **Objective 2.** Evaluate existing literature and methods concerning detecting and examining ELF binaries and try to implement any parts that may be of use to the DroidSieve framework.

  Rationale: An overview of existing approaches will help guide and influence the efforts made by this project.

  Related Sections: Chapter 4.2 and Chapter 4.3.

- **Objective 3.** Create and test a new method of detecting potentially malicious ELFs in Android APKs. Implement this as a feature in the DroidSieve framework.

  Rationale: This will hopefully improve the framework's ability to detect malicious ELF files.

  Related Sections: Chapter 5.1 and Chapter 5.2.

By meeting with these objectives, this project has found a previously unknown method used by malware to make calls to external packed binaries and has produced a unique and highly reliable method for detecting these calls (see Chapters 4.3 and 4.4). With this new feature implemented into the DroidSieve framework, an accuracy of 96.50% was maintained during testing with the XGBoost algorithm. The accuracy achieved using the Extra Trees algorithm was improved by 0.01% to 95.79% and the accuracy achieved when using the Random Forest algorithm was increased by 0.05% to 95.34%. (see Chapter 5.3).

The rest of this report is organised as follows. Chapter 2.1 outlines the architecture and security models of Android and Android APKs. Chapters 2.2 and 3.1 set the scene and discuss the motivations for Android malware detection systems. Chapters 3.2 and 3.3 provide a summary of the background literature on Android malware detection and identification. Chapter 4.1 gives an overview of the DroidSieve framework. Chapter 4.2 describes previous approaches and tools for the detection and analysis of ELF binaries and packed binaries. Chapters 4.3 and 4.4 discuss how unknown methods used by ELFs in APKs to call external packed binaries were found as well providing relevant literature and background information. Chapter 5.1 shows how these findings were constructed into a reliable method of detection and implemented into the DroidSieve framework. Chapters 5.2 and 5.3 contain the tests and results. Finally, chapters 6.1 and 6.2 present the conclusions and suggestions for future work.

2 Background Information

2.1 Android Architecture and Security Model

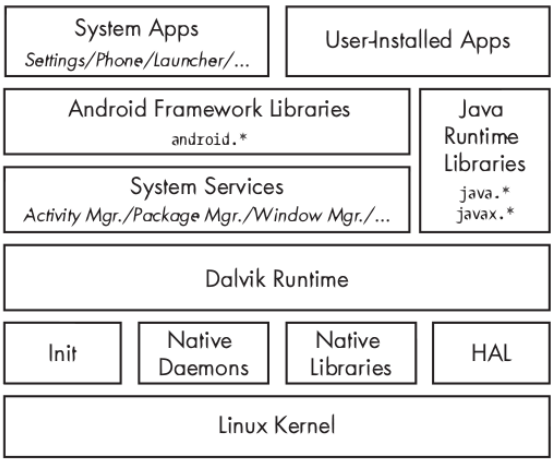

Figure 2.1: The Android OS architecture (source: Figure 1-1, page 2 [10]).

Figure 2.1 shows the layout of the Android architecture. At the bottom of which is an adapted version of the Linux kernel. The main differences between an ordinary Linux kernel and an Android Linux Kernel are that the Android version contains Binder, wake-locks, anonymous shared memory, alarms and networking. The networking feature uses permissions (explained further down) to restrict access to network sockets (page 2 [10]). Binder is a very important part of the Android kernel and deals with Inter Process Communication (IPC). The IPC mechanism restricts a process's address space to its own separate space and all calls to other process's address spaces must go through the IPC (Binder) (pages 4-5 [10]).

Figure 2.2 shows a simplified diagram of how Binder works. All communication between processes pass through the Binder Kernel driver which handles allocating memory for messages and transferring messages between processes.

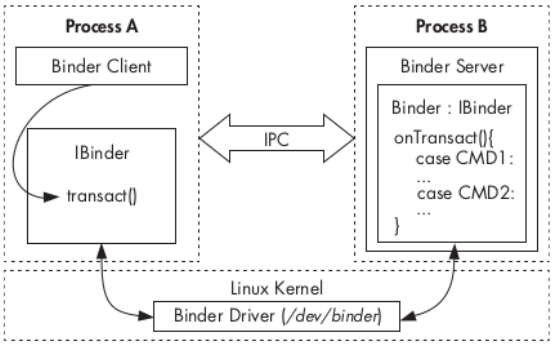

Figure 2.2: Simplified explanation of Binder IPC (source: fig1-5, page 6[10]).

In Android, IPC is broken down into three main abstractions: *Intents*, *Messengers* and *Content Providers*. *Intents* are commands sent across processes with attached data. *Messengers* are objects that allow messages to be communicated between processes. *Content Providers* provide a cross-process data management interface. As well these Binder also handles developer defined service interfaces via the *Android Interface definition language* (page 6 [10]).

The next layer in the Android architecture shown in Figure 2.1 is the native user space. Here are the *init* binary that run initialisation processes, Native Demons and Libraries (page 2 [10]).

Next is the Dalvik Runtime environment. Dalvik is the Android version of the Java Virtual Machine (JVM). Most of the Android apps and code base are written in Java, hence the need for a JVM style environment. Unlike its JVM counterpart, Dalvik cannot run .class files and instead has its own native language called *Dalvik Executable*. Hence the Dalvik runtime environment runs .dex files that are stored in system Java libraries(JAR files) or inside Android application files(APK files). Dalvik bytecode differs somewhat from JVM bytecode but is still very much human readable (page 3 [10]).

The layer above and to the right of the Dalvik Runtime layer in Figure2.1 is the Java Runtime libraries. This section of the Android architecture contains the runtime libraries required by Java programs to run. Both the System Layer and the Application layer both directly access the Java Runtime libraries layer (page 4 [10]).

Down and to the left of the Java Runtime libraries layer in Figure 2.1 is the System Services layer. This layer hosts the core system services which are written mostly in Java and also in native code. Most system services have a remote interface that is used by other applications to call the system service. The system services come together with Binder to act as an object-oriented OS (page 4 [10]).

Above the System Services layer sits the Android Framework libraries. This layer contains all libraries that aren't contained within the standard Java runtime library. The Android Framework is used as the basis for creating Android applications (page 10 [10]).

Finally, at the top of Android architecture sit the system and user applications. The system applications are mounted under /system in read-only memory within the OS image and are unchangeable by users. Subsequently they are given higher levels privilege as they are thought of as being more secure due to their inability to be changed by the user. As of Android 4.4, apps installed in the /system/priv-app/ get treated as privileged applications. Applications that are not pre-installed can gain system permissions by being signed with the platform signing key (page 10 [10]).

User installed applications are sandboxed and usually mounted under /data in a dedicated read-write partition. User applications are granted access to resources based on the permissions they are granted by the user at install time (page 2 [10]).

All apps are a collection of components that are defined in the AndroidManifest.xml. The system parses this extensible mark-up language (XML) file at installation time and registers the package and components defined within. The main app components are: *Activities, Services, Content Providers and Broadcast Receivers*. *Activities* are used for the graphical user interface GUI. They occupy a single screen and an application may have multiple activities that can be called by either the application or other applications depending on how they are set up. *Services* do not have a GUI and are used for processes that need to run in the background. Just like activities, an application's services may have a defined interface via the Android Interface Definition Language (AIDL) so that other application can call them. Services at this layer in the architecture are run on demand as opposed to the OS services that are continually running in the background. *Content Providers* are accessed via IPC and provide an interface to an application's data. Which data they give access to can be fine tuned and restricted as required. *Broadcast Receivers* are used by an application to respond to *broadcasts* which are system wide events triggered by either the system or user applications (pages 11-12 [10]).

As well as the techniques already discussed, Android employs a few other techniques and methods as part of its security model. As part of the afore mentioned application sandboxing every single application is granted its own User ID (UID). This has the effect of ensuring that each app is restricted and isolated at both a process level and a file level. Android furthers this notion by restricting system processes and doing away with the usual Linux UID (0) for one super user. Instead each privileged system application has a UID starting at 1000 and is restricted in what it may do and access. Group IDs (GID) are generated based on the permissions granted to an application. UIDs can be shared by applications that need to do so (pages 12-13 [10]). Whenever a new user is added to the device that user is assigned a unique UID and for each app that is installed for a specific user Android will assign it a new effective UID (page 16 [10]).

Permissions are used to control an application's access rights to data, hardware and services. They are defined in the AndroidManifest.xml file. They are permanently granted at install time by the Android OS that decides whether to grant a permission or not. This can involve user interaction though some permissions can only be granted to Android system services and similarly third parties can also restrict access to their own permis-

sions. Permissions are enforced at different levels depending on what the permission is trying to access (pages 14-15 [10]).

All android APKs must be signed by the author. This ensures that updates can be validated to have come from the author and is also used to create a trust relationship between apps. Platform keys are used to sign system apps and similarly to UIDs, this allows system apps to share resources. These platform keys belong to the author of which ever version of Android is running on the device (pages 14-15 [10]). Android also employs a verified boot system that cryptographically verifies each stage of the boot process using the RSA public key stored in the device's boot partition (page 18 [10]).

The Android APK is compressed using the ZIP compression algorithm. The structure of an APK [6] is as follows: *META-INF* This folder contains the MANIFEST.MF manifest file and certificate files CERT.RSA, CERT.SF. *res* This is where the resources needed by an app are stored such as graphics, sounds etc. *AndroidManifest.XML* This contains information about an app's access rights, name, version and referenced library files. *assets* This folder contains assets used by an application such as ELF files. *classes.dex* This is the compiled Dalvik virtual machine code. *resources.arsc* This is a binary resource file.

2.2 Official and Unofficial App Stores

Malware	Official	M1	M2	M3	M4	Total	Distinct
Total Known	21	51	48	20	31	171	119
Total Zero Day	11	9	10	1	9	40	29
Total	32 (0.02%)	60 (0.35%)	58 (0.39%)	21 (0.20%)	40 (0.47%)	211	148

Table 2.1: The total number of malware samples detected by DroidRanger from the official Android App store (Official) and alternative markets (M1-4) (source: Table 9, page 10 [44]).

Users download applications through the official store on Google Play [14]. As well as the Google Play Store there are many alternative markets that make up a significant portion of the overall Android App market space.

In May and June of 2011 Yanjin et al [44] collected all free apps possible from the Google Play Store and four alternative markets. Of the apps they collected, the four alternative markets contributed to 25% of the unique applications they collected (page 5 [44]). Using permission based static analysis and dynamic analysis Yanjin et al. found results shown in Table 2.1. Table 2.1 shows that despite malware scanning by market places, malware still manages to slip through the net. Of the 204,040 apps they collected they found 211 malicious apps (page 10 [44]).

In 2012 Oberheide and Miller presented at SummerCon2012 their findings of how Google attempts to find malware in applications submitted to the Google Play Store with Google's malware discovery software "Bouncer" [19]. According to their findings, Google

Bouncer performs some kind of static analysis (though they did not manage to gain much insight into exactly what kind of static analysis Bouncer performs) (slides 50-53 [19]). They did however manager to ascertain that Bouncer runs dynamic analysis tests in a QEMU (a hypervisor environment used for virtualisation) emulated environment for 5 minutes (slides 27-28 [19]).

3 Related Literature

3.1 Early Malware Detection Approaches

A study [30] in 2007 showed that just before the mass adoption of smartphones the numbers of malware targeting mobile phones had drastically increased since 2004. Back then Bluetooth was the most prevalent way that malware spread. With phones infecting other phones via direct Bluetooth connections (slide 13 [30]).

At that time the iPhone had only just been released and had yet to make any headway into the mobile markets with an overall market share of 3.4% at the end of the 3rd quarter of 2007 (table 1 [3]). Windows mobile OS also had not made any significant impact with a market share of just 12.8%. Whilst Symbian OS dominated the markets with a total share of 63.1% (table 2 [3]). As already discussed, once the HTC Dream was released in 2008 Android grew to dominate today [13] and the number of malware for Android apps has also increased hugely since 2008.

In their pre-Android 2005 paper [29], Christodorescu et al. present a computer based malware detection system that incorporates semantic analysis of code using an algorithm the authors designed with resilience to obfuscation techniques. They used control flow graphs to model a malicious behaviour and to compare against (page 2 [29]). They achieved high success rates with good resilience to obfuscation techniques (pages 10-12 [29]).

Cavallaro et al. point out in their 2008 paper, the limitations of the afore mentioned approach with computer based malware [25]. As previously discussed, Android bytecode is very high level and the architectural arrangements of Android are such that a lot of information can be gained from such approaches applied to Android as will be shown shortly.

The first published attempt at finding Android malware was by Enk et al. in 2009 [43]. They sampled 311 apps from the Google Play store created from the 20 most popular apps from the 16 existing categories. Their system examined what permissions apps asked for at install time and was designed to be part of a phone's application installer.

Their justification for their selection criteria was that most malware exploited permissions that the user willingly granted the app rather than trying to exploit a known or unknown vulnerability in the Android OS (page 2 [43]). They used security requirements to elicit and define what a secure Android phone is and create a model for identifying security requirements. They then used this model to define security rules at a mostly permissions level but also with some action strings (used for receiving intents) (pages 5-7 [43]). Of the 311 applications they examined, 12 failed to pass their rules (page 8 [43]). Their approach was basic but lead the way for others to follow and highlighted the importance of examining permissions that apps ask for as part of static analysis for malware.

3.2 Static Analysis of Android Malware

Many approaches have been carried out towards classifying and detecting Android malware with static analysis. What follows is a summary of some of the many tools created to solve these problems. The tools chosen have been chosen for their originality and or their relevance to this project.

In their 2011 paper [1] Felt et al. present a permissions based approach at examining app privileges. They created a map of which methods trigger which permissions checks (page 4 [1]). They point out the challenging problem of "reflection" in Java where Java's java.lang.reflect library can be used to change how a method acts when it is invoked (page 7 [1]). For testing, they used a small sample of 940 apps and found one third to be over privileged. The attribute much of this to developer confusion and find that most apps are only over privileged by a very small margin (page 11 [1]).

Rastogi et al. present in their 2013 paper, their work into overcoming transformation attacks by malware [40]. They looked at ten popular anti-malware products to see how resilient they were to common obfuscation techniques and none of them were at all resilient. They outlined transformations that are detectable by static analysis to be: changing package name in the android manifest, identifier renaming, i.e, classes, methods, field names; data encryption (the dex file contains all strings and arrays to be used and malware can be hidden here), call indirections to change call graphs, code reordering, for example, using goto instructions; junk code insertion, encrypting payloads and native exploits, function outlining and inlining (outlining - breaking down one function into many, inlining - replacing function call with entire function body), other simple transformations such as stripping off line numbers or changing names of local variables and finally combining any of the above (pages 2-3 [40]). They define behaviours not detectable by static analysis as: reflection (discussed earlier in this section) which is difficult to analyse statically and impossible if combined with encryption of function names and bytecode encryption - Cannot be statically analysed without decryption (page 3 [40]). The authors found that all tested products were vulnerable to common transformations. At least 43% of signatures used by the tested products were not based on code artefacts but instead used file names or hashes that are easily accessible. Only one out of the ten tested products performed static analysis of the byte code (page 5 [40]).

DroidSift [31] is an approach at Android malware classification that used weighted contextual API dependency graphs. Their approach was based on extracting program semantics rather than syntactic information. They point out previous approaches that used semantic based graphs are inherently unable to detect polymorphic variants of malware [31]. DroidSift's graph generation is built on top of Soot (page 4 [31]). Soot is a Java based platform for extracting information from Android apps in order to make analysis easier [37]. After generating graphs for malicious and benign behaviours they compared apps to both benign and malicious graphs and decide a best match. They have produced signatures based on feature vectors that they used to classify a malicious app's malware family. They have also built anomaly detection to detect zero-day malware (page 3 [31]). To the DroidSift authors' knowledge, their approach was the first to take the weighted distance of the API nodes in a graph into account and that previous approaches had treated nodes and edges the same (page 6 [31]). They collected 2,200 malware samples from the Android Malware Genome Project [45] and McAfee Inc [28]. They collected a total of 135,000 benign apps from McAfee and the Google Play Store [14] that they validated by scanning with VirusTotal [41]. They achieved a 93% detection success rate, 2% false negative rate and 5.15% false positive rate (pages 8-10 [31]).

Released in 2014, Apposcopy [47] was a static taint analysis, signature based approach at detecting Android applications that leaked private information. The authors point out that at the time of writing, there were two prevalent forms of malware detection: signature based and taint analysis. They remark that taint analysis approaches are flawed given that benign applications may need to leak user information. Therefore taint analysis cannot distinguish between benign and malicious applications and that great effort is required to differentiate between benign and malicious flow graphs. The authors refer to the work carried out by Rastogi et al. [40] when they say that signature based detection systems can be fooled with simple polymorphic techniques (page 1 [47]). Apposcopy uses a high level control flow graph called "inter-component callgraph (ICCG)" that is used to match control flow properties against those in stored in a signature. Taint analysis is used to compare data-flow analysis against a given data-flow property (page 2 [47]). Apposcopy uses Datalog with built in predicates to define signatures. They define Datalog as a set of rules that are based on a defined set of facts(page 3 [47]). They used Soot [37] to extract information from the .apk file to feed into Datalog (page 7 [47]). The ICCG is a directed graph where the nodes are components and edges are actions and data types (page 5 [47]).

Of the 1,027 samples taken from the Android Malware Genome Project [45], Apposcopy had a 90% success rate in identifying Malware (page 7 [47]). Of the 11,215 apps they sampled from the Google Play Store [14], 16 were found to be malicious (page 8 [47]). They used ProGuard [24] to obfuscate known malware samples and reported a dubious 100% success rate in resilience to malware obfuscation (page 9 [47]). The dubious nature of this success rate is discussed in Chapter 3.3. When compared to Kirin [43] on a sample of 1,027 apps, Kirin produced a 48% false positive rate and Apposcopy produced a 10% false positive rate (page 9 [47]).

Drebin [9] uses very broad static analysis and presents the user with a list of potentially malicious behaviours by means of explanation to the end user as to why the app has been flagged. Drebin aimed to be very lightweight taking 10 seconds to analyse malware on popular smartphone models of the time. The static analysis performed by Drebin is very broad indeed and collects as much information as possible (page 1-2 [9]). They used a data set of 131,611 applications collected from the Google Play Store [14], Russian and Chinese markets as well as many malware samples from all over the internet including the entirety of the Android Malware Genome Project [45]. To determine if the apps in their data set were benign or malicious they scanned them all using the top 10 anti-virus (AV) programs of the time and if one or more AV programs flagged an app as malicious then it was flagged as malicious in the data set. The work by Rastogi et al. the previous year [40] showed that this was probably not a good method to classify applications as malicious (page 6 [9]). The authors claim high success rates of detection and claim to have found sets of behaviours for popular malware families (pages 7-11 [9]).

RevealDroid, presented in 2015 [20] was designed to be an obfuscation resilient and efficient method of detecting malware and which family it came from. They argued that it is not good enough to just detect malware and remove it, one must also be able to identify it in order to reduce harm caused (page 1 [20]). RevealDroid examined sensitive API calls, categorised source-sink flows, intent actions and package API invocations. They reduced the size of their feature space by generalising sink-source relations and used similar methods for API calls. RevealDroid is Java based code using other pre-written libraries like Soot [37], Dexpler [11] - a translator from Dalvik bytecode to Soot intermediary formats and Weka [26] - a Java machine learning toolkit. FlowDroid [39] was used for flow analysis (pages 3-5 [20]). The authors found that RevealDroid obtained high accuracy for classification of both benign and malicious apps (pages 6-7 [20]). The overall accuracy of the family detection was between 87-95% (pages 7-8 [20]). RevealDroid experienced high resilience to obfuscation during testing (pages 8-9 [20]). They compared RevealDroid to MUDFLOW [42] and found RevealDroid to be more efficient (pages 9-10 [20]) and more accurate (pages 10-11 [20]). They compared RevealDroid's family identification to DenDroid's [15] family identification and found RevealDroid to be more accurate (pages 11-12 [20]).

3.3 Dynamic Analysis of Android Malware

This report has already mentioned how Google's Bouncer [30] uses dynamic analysis as part of its malware detection and there have been many approaches that used dynamic analysis to discover Android malware. For the sake of brevity and in keeping with the scope of this project, this report has avoided talking in depth about dynamic analysis. However no discussion of Android malware detection would be complete without at least mentioning dynamic analysis so this report will discuss briefly MARVIN [27], where dynamic analysis has been used as part of the approach.

In their 2015 paper [27] Lindorfer et al. make the point that static analysis alone may not be good enough (as previously alluded to in this report when discussing the

DroidChameleon [40] Project) to detect obfuscated malware. They also make the point that dynamic analysis approaches alone may suffer from analysis evasion. They present a system that combines the two to create a complete malware detection system. Their data set was of 135,000 apps including 15,000 malicious apps. They reported 98.24% correct classification of malicious apps with less than a 0.04% false positive rate and they discuss the re-training required to maintain performance (page 1 [27]). The authors show the need for re-training and show that retraining with new malware included included in the training data set is particularly important for maintaining accuracy (page 7 [27]).

3.4 History Aware Training

In their paper of March of 2015 [23], Allix et al. revealed that all the results presented in this report are likely to be considerably less accurate than reported and would not hold up to proper examination as will be shown below.

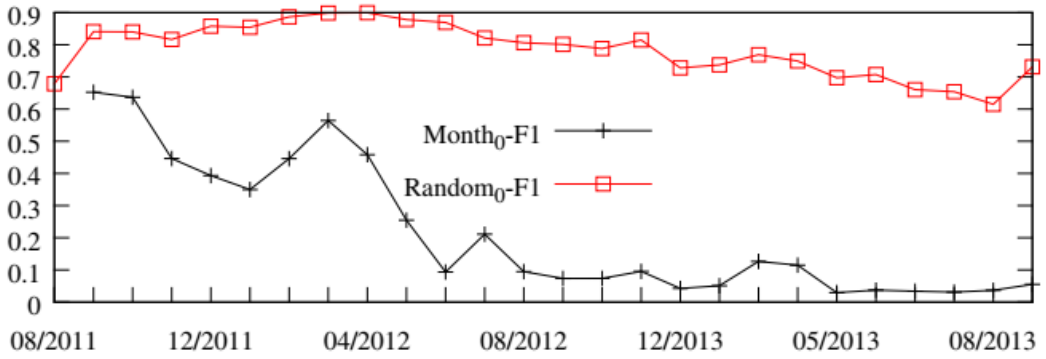

Figure 3.1: The effects of using history aware ($Month_0$) and history unaware ($Random_0$) on the performance of malware detection systems plotted as F Score versus time (source: Fig. 3, page 8[23]).

The seminal and most import claim that the paper makes is that by selecting a subset (randomly or otherwise) of data for testing from published or collected goodware and malware datasets, some of the apps in the training data set may have been created after the apps in the testing dataset thereby giving the system future knowledge that it would not have in a real world scenario (page 7 [23]). The implication that malware detection systems end up with knowledge of the future is likely why most previous approaches report detection success rates in the high 90%s as their systems have an unfair advantage over a real world system. The authors go on to show the effects of using a randomly selected data training set as shown in Figure 3.1 below.

Figure 3.1 shows the F score (see chapter 5.3 for a full explanation of F Score) which is a measurement of accuracy versus time for malware detection systems with and without history aware training. Figure 3.1 clearly shows that malware detection systems will falsely report better performance if it is trained on a random data set without history awareness. The authors go on to show that re-training is required to maintain performance and that that retraining may have to be done from scratch (pages 11-12 [23]).

4 Hunting ELFs

4.1 DroidSieve

This project builds upon the DroidSieve framework [16]. DroidSieve is a static analysis based approach at Android malware detection and classification with focus on feature selection specifically which features are important and most resilient. The authors reported 99.82% accuracy for malware detection and 99.62% accuracy for family identification of obfuscated malware. DroidSieve is implemented in python and is split into two halves. One for static feature analysis to extract static features from Android APKs in a given data set. Another for the static analysis AI learning used to create a classifier (of both malicious and family identification) of malware. This project seeks to add to the first half and improve upon the static feature extraction process.

Existing python libraries are used to extract and examine files from APKs. Each feature is then added to list of features for a given app and a tally of how many times a feature appears in the data set is also kept. Features are split into syntactic i.e. API calls, reflection, cryptography and resource centric i.e. certificate dates, incognito apps and embedded permissions.

In order to understand the work carried out in Chapter 4.3, it is necessary to briefly explain some of the details of how the DroidSieve framework operates. Within the feature extraction half of the framework, two pickle files are created. A pickle file is mechanism in python that allows an object to be serialised and written to a file [46]. This is useful as it allows the storage and transference of python objects from one program to another. The first pickle file contains a list of all APKs in a data set, their md5 hash and their location in memory. This first pickle file is used during the feature extraction phase to locate and identify APK files. The feature extraction phase then produces another pickle file that contains a list of all the features extracted from the data set. These pickle files must be created for all benign and malicious testing and training data sets. The pickle files containing the extracted features are then passed to AI learning portion of the framework that uses them to build an AI system and run tests to see how successful the feature selection is in malware identification and classification. The authors implemented the Extra Trees algorithm as their classifier algorithm (see 5.2

for more details on Extra Trees).

The authors rank the significance of a given feature for malware and family detection on obfuscated and non-obfuscated malware data sets. The interesting and most pertinent feature to this project, was the prominence of section flags 'AMS' (Allocatable, Mergable, Strings) which will be discussed in detail in Chapter 4.3.

4.2 Packed Binary Detection and Analysis

As will become clear later in Chapters 4.3 and 4.4, the detection of packed binary files is very pertinent to this project. So a brief look at previous attempts and tools will be useful to understand the context of this project.

In their 2008 paper [35], Perdisci et al. explore the detection of packed binaries within computer malware. Their rationale was that unpacking binaries is expensive so correctly identifying them as such before sending them to be unpacked greatly reduces computing time and resources required. Although the focus of this project is detection of calls to packed binaries as a feature of malicious Android APKs, the findings of this paper are still very much relevant.

Perdisci et al. state that sections where the Read, Write and Execute flags had been set were an important sign of a packed executable as the packed binary needs to execute to start the unpacking routine and needs to write to store the unpacked binary. Another key factor they looked for was the entropy of a section's header, content and the entropy of the overall file. A good encryption scheme should leave the encrypted data with high entropy. In the context of packed binaries, entropy refers to the randomness of the data. The authors make the observation that the closer the entropy of a section or file is to 8, i.e., the maximum byte entropy, the higher the probability that the section or file is packed. They found that an entropy threshold of 6.707 gave the best results for identifying packed binaries.

Before focussing on readelf, an initial investigation was made into other programs that had a python implementation or API that might be of potential use in examining ELFs. IDAPro [17] was discounted straight away due to licensing requirements that would not only be costly for this project but also anyone building upon this project. The angr [5] framework looked interesting but seemed to have a lot of emphasis on dynamic analysis which is beyond the scope of both this project and DroidSieve. Radare2 [32] looked very useful as it is command line interface (CLI) based and has a good python API. Radare2's capabilities are very advanced and come close to IDAPro's. For future investigation, Radare2 is highly recommended. Ultimately, it was decided that readelf still had a lot to offer as it has its own python implementation (as opposed to API) and was already being used in DroidSieve. So any extra functionality from readelf that was deemed useful would be very easy to incorporate into the existing framework and would also be the most efficient use of CPU time as DroidSieve uses a completely python based implementation of readelf.

4.3 Readelf and Malicious APK ELFs

In order to meet with Objective 1. (see 1) an investigation was carried out to determine what further use could be made of the python implementation of readelf within the DroidSieve framework. To begin examining what else readelf could potentially offer, it was first necessary to isolate some relevant samples of malware for examination. To accomplish this necessary step the *feature_search.sh* script was written.

As mentioned in Chapter 4.1, DroidSieve creates a meta_info file with a list of all samples and their corresponding md5 hashes and their location in the file system. It also creates a static pickle file containing the results of the feature extraction. *feature_search.sh* uses a lengthy series of regular expressions to grep these two files for a feature specified by the user. For example, file.ELF. *feature_search.sh* then collects the hashes of all the samples in the static pickle file containing this feature and then cross references these hashes with the meta info file to produce a list of file paths of samples with a requested feature. The user can then optionally tell the script to copy any APKs with the requested feature into a new subset.

Two subsets were created "testing_ELF" which contained all samples with the feature "file.ELF" and "testing_no_ELF" which contained all samples with the feature "NativeCodeWithoutElf". As it turned out, due to a misconfiguration in the apk_common.py file in DroidSieve, this later dataset actually contained all ELFs that were making calls to packed binaries and were not being scanned because of this misconfiguration. This misconfiguration was soon rectified before any further testing was done. However it is interesting to note that malware samples with the "NativeCodeWithoutElf" feature also had this packed binary feature. As will be discussed further on this is probably a coincidence resulting from the family or families of malware in the dataset. Once sufficient sub-samples were created, readelf was used to examine the ELFs. In these particular subsets, all the ELFs were located in the "assets" folder of the sample's APK.

Within the structure of an ELF, different sections may have different permissions flags set. For example the Execute - "X" flag allows a section to be executed and is usually set on the .text section of an ELF where the main executable code is set. As will explained further on in this paper, one might reasonably expect a malicious ELF to set the Execute flag on weird sections however this was not the case with the samples from the Malgenome data set. The main feature that came to light was the use the Allocate, Merge, Strings - "AMS" section flags on the .rodata (read-only data) section. Allocate is a standard flag used on most sections. Strings implies a section of null terminated "C" like strings and Merge allows other sections to access this section's data.

ELF files in the "testing_ELF" sub sample typically contained within the "AMS" flagged .rodata section, comments used during execution of the malware such "Loading please wait..." etc. and string paths to execve. ELF files in the "testing_no_ELF" sub sample set however contained within their "AMS" flagged .rodata section, paths to files in the "assets" folder (see Chapter 2.1). These files were data files which in several cases had names like "ratc" and random strings such as "gjwsvrc". This was a very interesting

feature that clearly pointed to some kind of packed binary file.

So any sections with the flags 'AMS' and that also contained path strings to other data files within the APK were a new feature that would hopefully help detect calls to packed binaries. Thus readelf would now be used to further probe into these 'AMS' flagged sections and search for mention of existing files within the APK.

This work was carried out independent of knowledge of the DroidSieve paper's findings. This was very interesting because once the DroidSieve paper became available it was apparent that the feature 'AMS' flags was one that ranked highly in their investigation. The initial findings of this project suggest why this feature was ranked so highly because not only for ELFs that contained paths to potential packed binaries but also to ELFs like "rageinthecage" that contained paths to execve but seemingly no packed binaries, both of these types of ELFS stored this information in a .rodata with flags 'AMS'. Hence why the 'AMS' flags feature ranked so highly.

This strongly suggested that this was an area worth pursuing.

4.4 Shannon Entropy and EntroPy

As discussed earlier, one of the most important aspects of the identification of packed binaries is the ability to calculate the entropy of a section or file. In his 1951 paper [36], Claude E Shannon presented a method for calculating the entropy of the English language. This method can be applied to any data set. Shannon's formula for calculating entropy is as follows.

$$H = \sum_{i=1}^{n} - P_i log_2 P_i \tag{4.1}$$

The entropy **H** is given by the sum of minus the probability of element **i** occurring in a data set multiplied the logarithm to base 2 of the probability of element **i** occurring where **i** takes the values of 1 to **n** where **n** is the number of elements in a data set. For values represented by a byte the maximum value of **H** is 8. This is because there are 8 bits in a byte and a value of 8 for **H** implies that all values from 0x00 to 0xff appear with equal frequency throughout the data set. This value is highly unlikely to be seen in a real data set and any values of **H** over 7 have very high entropy. Indeed, Perdisci et al found that an entropy value of 6.707 was optimal for their experiments [35].

In line with Objective 2. (see Chapter 1), the discoveries made in the evaluation process have lead to the development of a stand alone python module called EntroPy that can be used to calculate the Shannon entropy of a file or byte array.

EntroPy is designed to take either a file, path to file or byte array as input and return a numeric value for the Shannon entropy to a precision of 19 decimal places. This is level of precision is not necessary for this investigation but was included for the sake of modularity and future use. Likewise, although the current implementation passes a file the other input methods have been included for future work. For example, although the

initial investigation revealed that, of the malware sampled, none were using encrypted sections within the ELF, that does not mean that future malware will not.

EntroPy has been designed as a class so that multiple instance may exist. This is essential to the work with DroidSieve as it is a multi-threaded architecture. For the source code of EntroPy see Appendix A.

5 Experimental Evaluation

5.1 Implementation

The results of the work carried out to investigate how readelf could be of further use, suggested two important things to check for. The first was the 'AMS' flags and the second was the content of the section with 'AMS' (Allocate, Merge, Strings) flags to check for names of files that could potentially be packed binaries. The previous work by Perdisci et al outlined in Chapter 4.2 found that a threshold value of 6.707 (page 11 [35]) was optimal for their experiments. For this project however the preliminary investigations with the Malgenome data set revealed that the packed binaries generally had an entropy of 7.4 or higher. This being so, a threshold value of 7 was chosen for this project. With these three results in mind the following series of checks were devised to identify an ELF that contained a call to a packed binary and are represented in Algorithm 1 as pseudo code.

```
if section flags = 'AMS' : then
    for data in section : do
        if data = name of a file in APK : then
            E = Shannon Entropy of (file); if E>7 : then
                calls_packed_binary = true;
            end
        end
    end
end
```

Algorithm 1: Algorithm for the identification of ELFs that make calls to a packed binary

Within the feature extraction part of the DroidSieve framework, there are two main files relevant to this project. The first is apk_common.py. This file is responsible for the extraction of features from APKs. The first change that needed to be made here was to make sure the all ELF files were scanned. Upon first examination of the code, only incognito ELFs were being scanned. Next a section was added to implement the three

stage checks for an ELF calling a packed binary as outlined above.

Once an ELF has been established as making calls to a packed binary then the feature "calls_packed_binary" is added to the list of features for the APK.

The second file that required editing to accommodate the new feature was the feature_extraction.py python script that calls apk_common.py. This file builds the list of features and their prevalence for a given data set. This was a trivial edit to add in the new "calls_packed_binary" feature so that it would be added to the feature set.

Once these changes were made some preliminary tests were carried out on the "testing_no_ELF" data subset mentioned previously to ensure that the changes made to the framework were successful. The results of these tests identified all the ELFs with calls to packed binaries as expected and the new feature was then ready for testing on the AI half of the DroidSieve framework.

(See Appendix B for a list of changes made to the DroidSieve framework).

5.2 Testing

Before testing of the new feature was carried out, an investigation into the prevalence of benign ELFs that call packed binaries was carried out. For this initial experiment, 3,029 ELF binaries were scanned from an Arch Linux desktop computer. The ELF binaries in this sample originated from a wide variety of different sources such as Github, official and non official Arch Linux repositories. A separate script was written to scan all ELF binaries using the algorithm outlined in Chapter 5.1 as using the entire DroidSieve framework would have been unnecessarily slow and resource heavy.

Of the 3,029 ELF binaries scanned, none made calls to a packed binary. Only four ELFs had a section with the 'AMS' (Allocate, Merge Strings) flags set and of these four binaries only one made a call to an external binary and the binary it called was not packed. This was a very promising initial result as it suggested that benign ELFs making calls to packed binaries were either extremely rare or non existent on a Linux desktop environment. The Linux desktop environment is inherently more open source than Android App stores however so the result of this initial test was only indicative and not a direct comparison.

The next step in testing was to create feature extraction pickle files for all data sets used for both the original feature set used by DroidSieve and the new feature that included the feature produced by this project. Chapter 3.3 highlights the importance of history aware training and testing. In keeping with the original DroidSieve results, this project does not implement history aware training and testing. Not only is this out of scope, it would also hinder accurate comparison to the DroidSieve results. Due to non disclosure agreements, the data sets used for testing were a sub set of the DroidSieve data sets and included the following data sets. The malicious APKs from Drebin [9] and

the malicious and benign APKs from Marvin [27]. The Malgenome [45] data set was excluded from this testing phase as this data set had been used to develop the method of detecting APKs with ELFs that made calls to packed binaries and would therefore bias any tests. All the features previously used in the DroidSieve framework, as well as the newly created feature that identifies ELFs that make calls to a packed binary, were used in the feature extraction phase. Once feature extraction was complete, each static feature pickle file was then labelled as malicious or benign and testing or training. Note that here, the testing and training labels refer to the testing and training of the AI. DroidSieve's default value of 10-fold k-fold cross-validation was used. The Marvin dataset was split into testing and training data sets for both benign and malicious APKs and the Drebin data set was added to the malicious testing data set. In total the system was trained on 7,406 malware samples, 59,485 benign samples and tested on 5,551 malware samples, 25,494 benign samples.

Once the static feature extraction was complete, the pickle files were input into the AI section of the DroidSieve framework that trains and tests the pickle files created in the feature extraction phase using a specified AI algorithm. It quickly became apparent that the server deployed for this test (ten 3.3GHz Hasswell CPUs, 20GB DDR4 RAM) was no where near sufficient and instead a much greater server would be required. To this end an Amazon EC2 [4] instance was created with 40 CPUs and 160GB of RAM. Three AI algorithms were tested, XGBoost [8], Random forest and Extra Trees [33]. Random forest is a well known AI algorithm that averages over many decision trees. XGBoost was proposed by Tianqi Chen and Tong He in 2015. XGBoost stands for "extreme gradient boosting" and is a very fast implementation of the gradient boosting framework presented by Jerome H Friedman [12] [8]. Extra Trees is an extremely randomised decision tree based algorithm designed by Guerts et al [33].

5.3 Results

The results of testing conclusively prove that the new feature set improves the accuracy of the DroidSieve framework. Before the details of these results are discussed, it is necessary to define some terms first. Precision is defined as the proportion of retrieved information that is relevant and is given by:

$$precision = \frac{t_p}{t_p + f_p} \tag{5.1}$$

Where $t_p = true\,possitive$ and $f_p = false\,positive$ (chapter 7 [34]).

Recall is defined as the proportion of relevant information retrieved and is given by:

$$recall = \frac{t_p}{t_p + f_n} \tag{5.2}$$

Where $t_p = true\,possitive$ and $f_n = false\,negative$ (chapter 7 [34]).

The F_1 Score is the harmonic mean of precision and recall and is a measure of accuracy. It is given by:

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{5.3}$$

(section 4.1 [16], chapter 7 [34])

The results of the testing showed that XGBoost performed the best, achieving an accuracy score of 96.50% for both the new feature set (see Table 5.3) and the old feature set (see Table 5.2). When using the Random Forest algorithm for the classifier, the new feature set improved the accuracy of the DroidSieve framework by 0.05% (see Table 5.4 and Table 5.5). The new feature set also improved the accuracy of the DroidSieve framework by 0.01% when used with the Extra Trees algorithm (see Table 5.7 and Table 5.6).

The improvements achieved with the new feature set are not inconsequential when the following are taken into account. Firstly, that the original feature set already achieves percentile accuracies in the high nineties. Secondly, within the feature set of the higher scoring XGBoost algorithm, the feature 'File_ELF' ranked very highly and 'e_sh_flags.AMS' (this was the feature indicating that the 'AMS' flags were set within a section of an ELF) ranked highly also (see Table 5.8). However with none of the algorithms did the new feature of 'calls_packed_binary' appear in the list of top features. This is not a surprising result when the occurrence of this feature within the data sets used is taken into account. Table 5.1 shows that number of APKs in all data sets with the feature 'calls_packed_binary' is substantially lower than APKs with the 'File_ELF' feature and are a very small fraction of the total number of APKs.

Data Set	Type	Samples	File_ELF	calls_packed_binary
Drebin	Malware	5,551	1,073	69
Marvin Testing Benign	Benign	25,494	948	1
Marvin Testing Malicious	Malware	3,175	196	9
Marvin Training Malicious	Malware	7,406	453	16
Marvin Training Benign	Benign	59,485	2,210	5

Table 5.1: Results from feature extraction phase showing data set, type of samples, number of samples within the data set, number of APKs with feature 'File_ELF', number of APKs with feature 'calls_packed_binary'

Old Feature Set XGBoost	precision	recall	f1-score	support
Goodware	0.955938	0.999059	0.977023	25494
Malware	0.996832	0.865460	0.926512	8726
avg / total	0.966366	0.964991	0.964143	34220

Accuracy 0.9650

Table 5.2: Results from testing the old feature set using the XGBoost Algorithm.

New Feature Set XGBoost	precision	recall	f1-score	support
Goodware	0.955938	0.999059	0.977023	25494
Malware	0.996832	0.865460	0.926512	8726
avg / total	0.966366	0.964991	0.964143	34220

Accuracy 0.9650

Table 5.3: Results from the new feature set using the XGBoost Algorithm.

Old Feature Set Random Forest	precision	recall	f1-score	support
Goodware	0.940593	0.999882	0.969332	25494
Malware	0.999579	0.815494	0.898201	8726
avg / total	0.955634	0.952864	0.951194	34220

Accuracy 0.9529

Table 5.4: Results from testing the old feature using the Random Forest Algorithm.

New Feature Set Random Forest	precision	recall	f1-score	support
Goodware	0.941252	0.999882	0.969682	25494
Malware	0.999580	0.817671	0.899521	8726
avg / total	0.956126	0.953419	0.951791	34220

Accuracy 0.9534

Table 5.5: Results from testing the new feature set using the Random Forest Algorithm.

Old Feature Set Extra Trees	precision	recall	f1-score	support
Goodware	0.946594	0.999765	0.972453	25494
Malware	0.999177	0.835205	0.909863	8726
avg / total	0.960003	0.957802	0.956493	34220

Accuracy 0.9578

Table 5.6: Results from testing the old feature set using the Extra Trees Algorithm.

New Feature Set Extra Trees	precision	recall	f1-score	support
Goodware	0.946698	0.999725	0.972489	25494
Malware	0.999041	0.835549	0.910010	8726
avg / total	0.960045	0.957861	0.956557	34220

Accuracy 0.9579

Table 5.7: Results from testing the new feature set using the Extra Trees Algorithm.

XGBoost Top Features	
Score	Feature
0.0494	u'package.package_entropy'
0.047	u'num_permissions'
0.0435	u'num_files'
0.0435	u'file.JPEG'
0.0411	u'file.data'
0.0376	u'package.package_length'
0.0317	u'file.empty'
0.0317	u'file.PNG'
0.0282	u'PackageMissmatchIntentConsts'
0.0247	u'num_activities'
0.0188	u'num_services'
0.0188	u'file.ASCII'
0.0188	u'entry_points_under'
0.0176	u'android.permission.SEND_SMS'
0.0165	u'num_intent_consts'
0.0165	u'num_intent_const_android_intent'
0.0153	u'android.permission.ACCESS_WIFI_STATE'
0.0141	u'num_intent_actions'
0.0141	u'file.ELF Executable'
0.0129	u'intent_actions.android.intent.action.BOOT_COMPLETED'
0.0118	u'android.permission.WRITE_EXTERNAL_STORAGE'
0.0118	u'android.permission.VIBRATE'
0.0118	u'android.permission.READ_PHONE_STATE'
0.0118	u'PackageMissmatchService'
0.0118	u'PackageMissmatchMainActivity'
0.0106	u'intent_actions.android.intent.action.SIG_STR'
0.0106	u'file.UTF-8'
0.0106	u'android.permission.READ_CONTACTS'
0.0106	u'android.permission.ACCESS_FINE_LOCATION'
0.0094	u'android.permission.INTERNET'
0.0094	u'android.permission.ACCESS_NETWORK_STATE'
0.0082	u'package.package_subpckg'
0.0082	u'num_intent_action_android_intent'
0.0082	u'file.Audio'
0.0082	u'CoIs.ImageFileExtensionMismatch'
0.0071	u'num_libraries'
0.0071	u'file.HTML'
0.0071	u'e_sh_flags.AMS'
0.0071	u'com.android.launcher.permission.UNINSTALL_SHORTCUT'
0.0071	u'PackageMissmatchReceiver'
Total number of features	23,134

Table 5.8: List of top features from the new feature set used by the XGBoost algorithm sorted by their score

Random Forest Top Features	
Score	Feature
0.0902	u'android.permission.SEND_SMS'
0.0554	u'android.permission.RECEIVE_SMS'
0.0375	u'file.UTF-8'
0.0369	u'num_permissions'
0.0297	u'android.permission.READ_PHONE_STATE'
0.0287	u'file.empty'
0.022	u'file.ASCII'
0.0211	u'num_intent_consts'
0.021	u'num_intent_const_android_intent'
0.0191	u'package.package_length'
0.019	u'file.data'
0.0189	u'package.package_entropy'
0.0181	u'num_files'
0.0174	u'file.PNG'
0.0173	u'intent_actions.android.intent.action.BOOT_COMPLETED'
0.017	u'intent_actions.android.intent.action.DATA_SMS_RECEIVED'
0.016	u'num_activities'
0.0155	u'entry_points_under'
0.0142	u'target_sdk.5'
0.0133	u'android.permission.install_packages'
0.0126	u'android.permission.read_sms'
0.0124	u'com.android.launcher.permission.install_shortcut'
0.0124	u'android.permission.receive_boot_completed'
0.012	u'android.permission.access_network_state'
0.011	u'packagemissmatchintentconsts'
0.0105	u'num_intent_const_other'
0.0103	u'num_intent_action_android_intent'
0.01	u'package.package_subpckg'
0.0095	u'num_receivers'
0.0092	u'num_intent_actions'
0.0088	u'packagemissmatchservice'
0.0085	u'num_services'
0.0085	u'android.permission.write_sms'
0.0084	u'android.permission.change_configuration'
0.008	u'file.jpeg'
0.0076	u'intent_actions.ru.alpha.alphaservicestart76'
0.0071	u'android.permission.receive_wap_push'
0.007	u'android.permission.access_coarse_updates'
0.0068	u'android.permission.write_external_storage'
0.0059	u'com.android.launcher.permission.uninstall_shortcut'
total number of features	11,567

Table 5.9: list of top features from the new feature set used by the Random Forest algorithm sorted by their score

Extra trees top features	
score	feature
0.1216	u'android.permission.send_sms'
0.1154	u'android.permission.RECEIVE_SMS'
0.0338	u'target_sdk.5'
0.0283	u'android.permission.READ_PHONE_STATE'
0.0221	u'android.permission.CHANGE_CONFIGURATION'
0.0217	u'intent_actions.android.intent.action.DATA_SMS_RECEIVED'
0.0191	u'android.permission.RECEIVE_BOOT_COMPLETED'
0.0187	u'android.permission.INSTALL_PACKAGES'
0.0174	u'intent_actions.ru.alpha.AlphaServiceStart76'
0.0166	u'android.permission.ACCESS_COARSE_UPDATES'
0.0154	u'com.android.launcher.permission.INSTALL_SHORTCUT'
0.0148	u'android.permission.READ_SMS'
0.0143	u'intent_actions.ru.mskdev.andrinst.NewsReaderService'
0.0137	u'num_permissions'
0.0132	u'android.permission.ACCESS_NETWORK_STATE'
0.013	u'file.empty'
0.0127	u'android.permission.RECEIVE_WAP_PUSH'
0.011	u'intent_actions.android.intent.action.BOOT_COMPLETED'
0.011	u'com.android.launcher.permission.UNINSTALL_SHORTCUT'
0.0096	u'package.package_entropy'
0.0087	u'android.permission.GLOBAL_SEARCH_CONTROL'
0.0087	u'android.permission.EXPAND_STATUS_BAR'
0.0084	u'package.package_length'
0.0082	u'intent_actions.com.android.launcher.action.INSTALL_SHORTCUT'
0.008	u'android.permission.SYSTEM_ALERT_WINDOW'
0.0079	u'android.permission.WRITE_SMS'
0.0077	u'package.package_subpckg'
0.0076	u'com.android.launcher.permission.WRITE_SETTINGS'
0.0075	u'com.android.launcher.permission.READ_SETTINGS'
0.0073	u'intent_actions.android.intent.action.SIG_STR'
0.0072	u'intent_actions.com.software.CHECKER'
0.0068	u'intent_actions.android.intent.action.BATTERY_CHANGED_ACTION'
0.0068	u'android.permission.WRITE_EXTERNAL_STORAGE'
0.0067	u'android.permission.READ_CONTACTS'
0.0062	u'android.permission.SET_WALLPAPER_HINTS'
0.006	u'intent_actions.com.android.settings.APPLICATION_DEVELOPMENT_SETTINGS'
0.0059	u'num_intent_const_android_intent'
0.0059	u'com.software.application.permission.C2D_MESSAGE'
0.0058	u'num_intent_consts'
0.0058	u'file.UTF-8'
Total number of features	11,567

Table 5.10: List of top features from the new feature set used by the Extra Trees algorithm sorted by their score

6 Conclusions and Future Work

6.1 Conclusions

This project has investigated how DroidSieve analyses ELF files, what other approaches and tools exist and how they might apply to the DroidSieve framework. The results of this first stage of investigation lead to the discovery of a completely unknown technique used by malicious ELFs to hide calls to packed binaries. A highly reliable method for the detection of these hidden calls was developed, implemented into the DroidSieve framework and tested. The inclusion of this new feature within the DroidSieve framework has been proven to increase DroidSieve's malware detection accuracy.

Objective 1. Improve how DroidSieve detects and analyses malicious ELF files.

Based on the research carried out as part of this project, a highly reliable method for identifying ELF's that contain calls to external packed binaries has been developed into an algorithm, implemented into the DroidSieve framework and tested. The method identifies with near absolute certainty ELFs that make calls to a packed binary. The implementation of this new method builds upon and extends DroidSieve's use of the python implementation of readelf and adds to this with some original python code for calculating the Shannon entropy of a file. No other works exist that use the method developed in this project.

The new feature set improved the performance of the DroidSieve framework by 0.01% when used with the Extra Trees algorithm as the classifier and by 0.05% when used with the Random Forest algorithm as the classifier. Both the new feature set and old feature set achieved an accuracy of 96.50% This is a very interesting result as Extra Trees was the best performing algorithm in the original DroidSieve study. This high performance of the XGBoost algorithm compared to the Extra Trees algorithm (which was found to be the best by the original DroidSieve paper) could be due to the significantly smaller data set used in this project. The low occurrence of APKs with the new feature 'calls_packed_binary' in the data sets used for this experiment may also explain why the improvements in accuracy seen were only incremental. It is likely that with a larger data set these improvements will be higher.

Objective 2. Evaluate previous attempts at detecting and examining ELF binaries and try to implement any parts that may be of use to the DroidSieve framework.

This project examined a wide range of tools for ELF analysis as well as how DroidSieve used readelf and what more readelf could offer. An examination of the literature on previous attempts at classification of packed binaries proved to be very useful and formed the basis of further investigation into how malicious Android APKs make use of packed binaries. This further research was unique and its findings were previously unknown and unreported in the literature. These findings were that the allocate, merge and string (AMS) flags were very important as they allowed executable sections of an ELF access to path names to packed binaries that were stored outside of the ELF. This explained why the feature 'e_sh'AMS' (section contains AMS flags) had ranked so highly in the original DroidSieve results.

Objective 3. Create and test a new method of detecting potentially malicious ELFs in Android APKs. Implement this as a feature in the DroidSieve framework.

As previously discussed for *Objective 1.*, this project focused on identifying ELFs that made calls to external packed binaries and created a robust method to reliably do so. This new method was successfully integrated into the DroidSieve framework and tests have proved its success.

This project has met all its objectives very well especially considering the resource issues and time constraints. It has produced original findings that were previously unreported in the literature. It has successfully developed and tested new methods based on these original findings that identify with high certainty ELFs that call an external packed binary. Testing has shown that this new method is very capable at feature extraction and that the accuracy DroidSieve framework is improved with this new feature when used with Extra Trees and Random Forest algorithms. The accuracy remains the same with the XGBoost algorithm tested on the data sets available to this project.

6.2 Future Directions

The following future directions are suggested for improvement of the current findings and the continuation of this research.

Firstly, it is suggested that all of the data sets used in the original DroidSieve paper be used to test the DrodSieve framework with new feature. Due to non disclosure agreements only a subset of the original data sets were available to this project for training and testing. The occurrence of the newly added feature was not very high within the data sets used by this project and it would be very interesting to see how the new feature performs on larger data sets.

Secondly, it would be very interesting to examine the six benign APKs that contained

calls to packed binaries and see whether they are in fact malicious. This might require some dynamic analysis to be certain. Given that the benign data set used by this project is an order of magnitude greater than the malicious data set, these six APKs stand out as being very suspicious. The assertion that these six APKs are benign was based on the state of the art in malware detection and this project has gone beyond the state of the art. The method developed by this project is entirely unique, therefore it is possible that these six APKs represent a new family or families of malware that were previously undetectable with static analysis methods.

Lastly, the research carried out by this project found that Radare2 had a lot of potential for use with the DroidSieve framework for analysis of ELFs. It is recommended that this be further investigated. Due to time constraints, this project did not specifically examine the .text section (which is the main section of executable code). Which would likely contain more features such as an embedded unpacker or call to a generic unpacker. This is one example where Radare2 could be very useful.

Bibliography

[1] Steve Hanna Dawn Song David Wagner Adrienne Porter Felt, Erika Chin. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[2] Open Handset Alliance. Industry leaders announce open platform for mobile devices. http://www.openhandsetalliance.com/press_110507.html, November 2007.

[3] Amazon. Gartner says worldwide smartphone sales reached its lowest growth rate with 11.5 per cent increase in third quarter of 2008. http://www.gartner.com/newsroom/id/827912, December 2008.

[4] Amazon. Amazon web services. https://aws.amazon.com/ec2/, 2016.

[5] Computer Security Lab at UC Santa Barbara. What is angr? https://angr.io, 2016.

[6] Michal Blazek. Google android apk application package file format description. http://www.file-extensions.org/article/android-apk-file-format-description, August.

[7] Bloomberg. Google buys android for its mobile arsenal. http://www.bloomberg.com/bw/stories/2005-08-16/google-buys-android-for-its-mobile-arsenal, August 2005.

[8] Tianqi Chen and Tong He. xgboost: extreme gradient boosting. *R package version 0.4-2*, 2015.

[9] Malte Hbner Hugo Gascon Konrad Rieck Daniel Arp, Michael Spreitzenbarth. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[10] Nikolay Elenkov. *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. No Starch Press, No Starch Press Inc. 245 8th Street, Sanfrancisco, CA94103 USA, first edition, 2015.

[11] Alexandre Bartel et al. Dexpler. http://www.abartel.net/dexpler/, 2016.

[12] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[13] Gartner. Gartner says emerging markets drove worldwide smartphone sales to 15.5 percent growth in third quarter of 2015. http://www.gartner.com/newsroom/id/3169417, November 2015. Table 2.

[14] Google. Google play. https://play.google.com/store, February 2016.

[15] Pedro Peris-Lopez Jorge Blasco Guillermo Suarez-Tangil, Juan E Tapiador. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Systems with Applications*, 41(4):1104–1117, 2014.

[16] Mansour Ahmadi-Johannes Kinder Giorgio Giacinto Guillermo Suarez-Tanigl, Santanu Kumar Dash and Lorenzo Cavallaro. Droidsieve: Fast and accurate classification of obfuscated android malware. http://s2lab.isg.rhul.ac.uk/, 2016.

[17] Hex-Rays. Ida: About. https://www.hex-rays.com/products/ida/, 2016.

[18] IDC. Smartphone os market share, 2015 q2. http://www.idc.com/prodserv/smartphone-os-market-share.jsp, August 2015. Chart: "Worldwide Smartphone OS Market Share".

[19] Charlie Miller Jon Oberheide. Dissecting the android bouncer. *SummerCon2012, New York*, 2012.

[20] Bahman Perood Ali Bagheri-Khaligh Sam Malek Joshu Garcia, Mahmoud Hammad. Obfuscation-resilient, efficient, and accurate detection and family identification of android malware. Technical report, George Mason University, 2015.

[21] Juniper. Juniper networks mobile threat center third annual mobile threats report: March 2012 through march 2013. http://www.juniper.net/us/en/local/pdf/additional-resources/3rd-jnpr-mobile-threats-report-exec-summary.pdf, March 2013. Page 2.

[22] Justin Kahn. Google shows off new version of android, announces 1 billion active monthly users. http://www.techspot.com/news/57228-google-shows-off-new-version-of-android-announces-1-billion-active-monthly-users.html, June 2014.

[23] Jaques Klein Yves Le Traon Kevin Allix, Tegawendé F. Bissyandé. *Engineering Secure Software and Systems: 7th International Symposium, ESSoS 2015, Milan, Italy, March 4-6, 2015. Proceedings*, chapter Are Your Training Datasets Yet Relevant?, pages 51–67. Springer International Publishing, Cham, 2015.

[24] Eric Lafortune. Proguard. http://proguard.sourceforge.net/, 2016.

[25] R Sekar Lorenzo Cavallaro, Prateek Saxena. On the limits of information flow techniques for malware analysis and containment. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 143–163. Springer, 2008.

[26] R J Marsan. Weka. https://github.com/rjmarsan/Weka-for-Android, 2016.

[27] Christian Platzer Martina Lindorfer, Matthias Neugschwandtner. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, volume 2, pages 422–433. IEEE, 2015.

[28] McAfee. Mcafee. www.mcafee.com, 2016.

[29] Sanjit A. Seshia Dawn Song Randal E. Bryant Mihai Christodorescu, Somesh Jha. Semantics-aware malware detection. In *Security and Privacy, 2005 IEEE Symposium on*, pages 32–46. IEEE, 2005.

[30] Mikko Miiko Hyppnen. Mobile malware. In *16th USENIX Security Symposium*, 2007.

[31] Heng Yin Zhiruo Zhao Mu Zhang, Yue Duan. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1105–1116. ACM, 2014.

[32] pancake. Radare2. https://radare2.org, 2016.

[33] Loius Wehenkel Pierre Geurts, Damien Ernst. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[34] CJ Van Rijsbergen. Information retrieval. dept. of computer science, university of glasgow. *URL: citeseer. ist. psu. edu/vanrijsbergen79information. html*, 1979.

[35] Wenke Lee Roberto Perdisci, Andrea Lanzi. Classification of packed executables for accurate computer virus detection. *Pattern Recognition Letters*, 29(14):1941–1946, 2008.

[36] Claude E Shannon. Prediction and entropy of printed english. *Bell system technical journal*, 30(1):50–64, 1951.

[37] Soot. Soot. https://sable.github.io/soot/, 2016.

[38] Statista. Number of apps available in leading app stores as of july 2015. http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, July 2015.

[39] Christian Fritz Eric Bodden Alexandre Bartel Jaques Klein Yves Le Traon Damien Octeau Patrick McDaniel Steven Arzt, Siegfried Rasthofer. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Notices*, volume 49, pages 259–269. ACM, 2014.

[40] Xuxian Jiang Vibhav Rastogi, Yan Chen. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 329–334. ACM, 2013.

[41] Virustotal. Virustotal. https://www.virustotal.com, 2016.

[42] Alessandra Gorla Andreas Zeller Steven Arzt Siefried Rasthofer Eric Bodden Vitalii Avdiienko, Konstantin Kuznetsov. Mining apps for abnormal usage of sensitive data. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 426–436. IEEE, 2015.

[43] Patrick McDaniel William Enck, Machigar Ongtang. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245. ACM, 2009.

[44] Zhou Wu Xuxian Jiang Yajin Zhou, Zhi Wang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.

[45] Xuxian Jiang Yanjin Zhou. Android malware genome project. http://www.malgenomeproject.org/, 2016.

[46] YASOOB. What is pickle in python? https://pythontips.com/2013/08/02/what-is-pickle-in-python/, 2013.

[47] Isil Dillig Alex Aiken Yu Feng, Saswat Anand. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 576–587. ACM, 2014.

A

EntroPy Source Code

Listing A.1: EntroPy source code

```
#!/usr/bin/python

from collections import Counter
import math
import sys

class entroPy:


    def __init__(self):

        self.entropy=0
        self.filePathOfInterest=""
        self.fileOfInterest = None
        self.byteArray=[]


    def __shannonEntropy(self):

        numTotalChars=len(self.byteArray)
        charCount=Counter(self.byteArray)
        self.entropy=0

        for key, value in sorted(charCount.iteritems()):
            prbChar=('%.19f' % (value/float(numTotalChars)))
            self.entropy+= -float(prbChar)*math.log(float(prbChar), 2)

        return;
```

```
def fromFilePath( self, inputFile ):

    self.filePat:OfInterest = inputFile

    with open(self.filePathOfInterest, "rb") as opnFile:
        for line in opnFile:
            for byte in line:
                self.byteArray+=[byte]

    self.__shannonEntropy()

    return self.entropy;

def fromFile( self, inputFile ):

    self.fileOfInterest = inputFile
    for line in self.fileOfInterest:
        for byte in line:
            self.byteArray+=[byte]

    self.__shannonEntropy()

    return self.entropy;

def fromArray(self, arrayOfInterest):

    self.byteArray = arrayOfInterest[:]
    self.__shannonEntropy()
    return self.entropy;
```

B

Changes Made to DroidSieve

Listing B.1: Diff between original and updateda feature_extraction.py source code used for building extracted features into pickle files

```
--- feature_extraction.py       2016-07-30 16:13:09.992000000 +0100
+++ feature_extraction.py.new   2016-07-30 16:32:57.920000000 +0100
@@ -588,6 +588,10 @@
         for f in hls[feature]:
             sample.add_feature_freq(feature + '.' + f, hls[feature][f])

+    feature = 'calls_packed_binary'
+    if feature in hls:
+        sample.add_feature(feature)
+
     return sample

 def read_black_list(meta_file_path):
@@ -938,6 +942,14 @@
             if sample.features.freq_fname(feature) > 0:
                 try:
                     count = apps_with_a_feature[feature]
+                except KeyError:
+                    count = 0
+                apps_with_a_feature[feature] = count + 1
+
+            feature = 'calls_packed_binary'
+            if sample.features.freq_fname(feature) > 0:
+                try:
+                    count = apps_with_a_feature[feature]
                 except KeyError:
                     count = 0
                 apps_with_a_feature[feature] = count + 1
```

Listing B.2: Diff between original and updated apk_common.py source code used for feature extraction

```
--- apk_common.py         2016-08-15 11:31:36.288000000 +0100
+++ apk_common.py.new     2016-07-30 16:12:42.900000000 +0100
@@ -10,7 +10,7 @@
 from time import sleep
 import logging, logging.config
 import datetime
-
+from entroPy import *
 import Cols
 from Cols import *

@@ -227,7 +227,7 @@
     extensions={}
     components_of_interest=[]
     num_componenets_of_interest = {}
-
+
     #---------- Strings
     su = 0
     emulator = 0
@@ -505,13 +505,14 @@

         if file_type == 'DEX' and 'classes' != fileName:
             incognito_dex.append(f)
-        #ELF files are scanned!
+
         if file_type == 'ELF' or file_type == "ELF Executable":
             incognito_elf.append(f)

         if file_type == "Text Executable":
             incognito_sh.append(f)
+
+        #print f, file_type


     text_executable_commands = {}
@@ -536,6 +537,7 @@
     if num_componenets_of_interest:
         data['Cols'] = num_componenets_of_interest
     '''
+
@@ -615,9 +617,21 @@
                 except KeyError:
                     count = 0
                 sections_flags[flag] = count + 1
```

```
+
+                if flag=='AMS':
+                    hxdmp = section.data()
+                    for itm in files:
+                        fle = itm.rsplit('/', 1)[-1]
+                        if fle in hxdmp:
+                            shnnonEntrpy = entroPy()
+                            fileToScan = app.get_file(itm)
+                            if shnnonEntrpy.fromFile(fileToScan) > 7 :
+                                data['calls_packed_binary'] = True

            if sections_flags:
                data['e_sh_flags'] = sections_flags
+               #print sections_flags

            if symbols_shared_libraries:
                data['symbols_shared_libraries'] = symbols_shared_libra
```