INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2015, VOL. 61, NO. 4, PP. 311–320
Manuscript received October 10, 2015; revised December, 2015. DOI: 10.1515/eletel-2015-0040

An Android Security Policy Enforcement Tool

Kathryn Cotterell, Ian Welch, and Aaron Chen

***Abstract*—The Android operating system (OS) has become the dominant smart phone OS in recent years due to its accessibility, usability and its open-source philosophy. Consequently, this has also made it a popular target for attackers who aim to install malware on Android devices and take advantage of Android's coarse-grained, non-revoking permission system. This project designs, implements and evaluates a security tool named COMBdroid, which addresses these security concerns in Android by enforcing fine-grained, user-defined policies. COMBdroid modifies an application before installation, allowing it to override points of security vulnerabilities at run-time. As a proof of concept we have implemented three policies in COMBdroid. This paper documents the development process of COMBdroid, deriving design decisions from the literature review, detailing the design and implementation, and proving the program's effectiveness through evaluation.**

***Keywords*—mobile computing, Java, Android, security**

I. Introduction

GOOGLE'S Android has dominated smart phone operating systems (OS) recent years, reaching 79.3% of the market share in the second quarter of 2013[1]. As Android is an open-source OS developed for smart phones with extensive documentation[2] it has been a popular choice among mobile vendors to customise and distribute on their own hardware.

The open-source philosophy of the Android OS means that attackers are able to exploit users into downloading malicious applications in various ways. This may be through social engineering, phishing emails, or drive-by downloads. The open-source approach is also reflected in the development and distribution model adopted by the Android ecosystem. While iOS requires developers to pay US$99 a year to distribute applications on the application store, Android Play charges a one-time fee of US$25 to register a developer account. Android developers are also not restricted to distributing their application on Google Play, as there are plenty of unofficial Android markets that will distribute applications. iOS users who wish to download applications from outside of the App store however, must first jailbreak their phone.

Every Android application package file (APK) downloaded requires user confirmation that the application may use a certain set of permissions to function correctly, however the permissions implementation is very coarse-grained [3]. Once installed on the users phone, the application has no obligation to inform the user of how, when or why it is accessing these permissions, and revoking permissions is impossible without uninstalling the application.

I. Welch and A.Chen are the School of Engineering and Computer Science, Victoria Univeristy of Wellington, New Zealand, (e-mail: ian.welch@ecs.vuw.ac.nz, aaron.chen@ecs.vuw.ac.nz).

K. Cotterell carried out this work as part of her Bachelor of Engineering final year project.

We have addressed the problem with too coarse grained control over Android applications by developing a new security tool providing greater transparency and control for users through enforcing fine-grained policies. Unlike previous approaches, our tool enforces these policies at the application-level without requiring modifications to the Android OS.

A. Contributions

The main contributions of this paper are:

- A review has been performed on the current state of Android threats. The review discusses Android malware and mitigation methods from researchers and commercial sectors. It was found the most effective detection mechanisms involved modification of the application to permit run-time monitoring.
- From the review a set of policies and requirements was designed. The requirements were drawn from the observations of the strengths and weaknesses of current solutions.
- A security system named COMBdroid was designed and integrated these requirements. The tool consisted of two main parts – classes to insert into the application that would intercept behaviour at run-time, and an interface to handle APK instrumentation.
- A security tool named COMBdroid was implemented and evaluated. Evaluation included verifying the enforcement mechanism for each policy and performance testing showed minimal impact on the application functionality. COMBdroid was also tested on closed-source applications presenting successful policy enforcement.

The name of the security tool that was created, COMBdroid, was decided as the tool functions as a **CO**vert **M**alware **B**locker, and the imagery of using a comb to closely inspect applications and enforce fine-grained policies is appropriate. The term *Droid* is also commonly associated with the Android devices, therefore *COMBdroid* accurately describes the overall purpose of the system.

B. Paper Organization

The rest of this paper is organised as follows, Section two provides a background review is performed which includes an analysis of previous contributions to this project, common malware threats and a literature review of previously proposed approaches. Section three covers the design and implementation of our tool. Section four presents the evaluation of the COMBdroid, showing the performance and testing enforcement mechanisms, and discusses the results. Finally, Section five gives the concluding remarks of the project and discusses opportunities of future work directions.

II. Background and Related Work

This Section reviews the state of Android technology and prior methods proposed by researchers. First, the Android OS is discussed explaining how permissions and activities play a role in security concerns. Android malware and examples of their applications are then reviewed. This is followed by a comprehensive study of recent research works that have proposed Android protection tools.

A. Android Architecture

The Android architecture is composed of several layers: application, Java class frameworks (Android middleware), native libraries, Dalvik virtual machine and core libraries, to the Linux kernel. Android applications are installed on the device and are given their own directory, with the intention that applications be unable to share data between one another. Applications can circumvent this restriction however, by using inter-process communication (IPC) where data will be serialised, sent through the kernel, then sent to the callee which will de-serialise the data. To invoke such behaviour the applications use intents, however intent filtering cannot be relied on as it will not prevent explicit intents[2].

1) Android Permissions: Android applications require the user to agree to a set of permissions on installation which cannot be revoked without removing the application. Permissions requested are declared in the APK's Manifest file such as `CALL_PHONE` for making phone calls, or `INTERNET` for establishing network connections. Usually, applications will have legitimate reasons for the permissions they are requesting. For example, a personalised keyboard application may require full internet access in order to download modules for other languages. The coarse-grained permissions however, do not require an application to declare when or for what purposes they are using the granted permissions. Usually it will be obvious to the informed user when not to trust an application, such as a puzzle game requesting permissions to write and send SMS messages. Other times, applications with malicious intent will not be so obvious. If a social network application requests access to the user's contacts, there are no restrictions to stop the application sending the contacts to an untrusted server without the users knowledge.

Although an application is restricted to permissions outlined in its manifest file, they are not restricted to call another application to perform actions who have access to permissions the offending application does not. Applications taking advantage of this are performing privilege escalation attacks. When legitimate applications have been hijacked to carrying out tasks of malicious applications it is known as the confused deputy attack. While it is possible for an application to protect itself by including reference monitor checks in each component, there is no centralised enforcement of permission checking in the Android OS[4]. As demonstrated by Enck et al.[5] it is possible for an application to invoke a call to an unprotected component and request a phone number to dial, without the user's knowledge.

2) Dalvik Virtual Machine: APKs are built using Java and XML, and after being compiled to Java bytecode are converted into Dalvik[6] executable code (.dex) to run on the Dalvik virtual machine. Once packaged and signed, the source code of Android application files will not be accessible to those who download the applications, however there are tools that will disassemble the APKs to reveal their Dalvik bytecode. These tools, such as APKTool[7], are freely available online. Once the code has been disassembled developers with malicious intent are able to inject bytecode the reroute calls made to instrument their own malicious activities. Signing an application also does not require any special permissions, meaning applications originally distributed by a trusted vendor may be modified, re-signed and distributed as malware to unsuspecting users. This method that attackers use to modify code through disassembling tools will be the same approach this project takes, using APKTool to override targeted system calls.

B. Malware Detection

In recent years there have been several approaches to detecting and mitigating Android malware. These methods range from anti-virus software, Play-store scanners, and retrofitting the device to provide run-time monitoring[8][9][10].

1) Play Store Scanners: Google Bouncer[11], scans applications submitted to the Google Playstore to identify known malware through signature detection. Bouncer also executes submitted applications in an emulated environment to attempt to identify phone data being leaked without user input. Bouncer is not a complete solution however, as it is possible to buy verified Google play accounts online and update the application with malicious features. An application could also act benign until Bouncer has completed the visualisation scan – which is found to be around five minutes[9].

2) Anti-Virus Software: Anti-virus (AV) software has been found to be largely ineffective against the vast number of variants of Android malware. Zhou et al.[8] demonstrated that four popular AV's downloaded from the Google Play store were only able to detect between 20.2% and 79.6% malware samples out of 1260 given, showing it is not sufficient to rely solely on AV software.

3) Permission Analysis: Another approach is to analyse applications before being installed to determine if they might be malicious because of the types of permissions that they request (Kirin [12]) or use of external libraries (RiskRanker [13]). Although promising these approaches generate too many false positives because it the approach is quite coarse-grained because it may block applications that are in fact not using the permissions granted to it.

4) Run-Time Monitoring: Run-time monitoring is where an application is allowed to execute and is monitored at a fine-grained level. To perform run-time monitoring, either the application or firmware must be modified, allowing hooks to be inserted and altering application behaviour. The following discusses approaches that have inserted hooks either at the Android middleware or application level.

CRePe [14], SAINT [10], TaintDroid [15], AppFence [16] and Quire [17] all allow fine-grained monitoring and control

over the execution of Android applications. Both are implemented by modifying the Android middleware and inserting hooks to control IPC between applications. This allows access to services such as the phone or SMS features of the phone be controlled at a fine-grained level. However, all of these approaches require users to use modified version of Andoid operating sytem with hooks placed either with the Android middleware or operating system kernel and this not practical for widespread use.

Aurasium, developed by Xu et al.[18] presents a solution to monitoring application permission usage without a need for make modifications to the Android operating sytem. Aurasium modifies applications before they are installed on a device using APKTool to disassemble the APKs and insert their own java code and native libraries. The authors take advantage of the fact all functions requiring interaction with the OS must go through the *Bionic libc* library, creating a central point of restraint. This allows interception at the lowest possible boundary during run-time as applications make library calls, for example making a socket connection or initiating SMS requiring interaction with the `libc.so` library.

Aurasium meets our requirements of portability and our initial work looked at extending their approach. However, we decided to take a different approach which is avoid requiring changes to low level libraries. Instead our approach is a Java-only solution.

III. Design and Implementation

A. Design Requirements

We defined the following set of requirements based upon our original goals and the findings from our literature review.

1) Modified applications that are not violating enforced policies should function as expected. We do not want the tool to inconvenience users when the application is not violating policies, as this will result in user reluctance to use the tool, leading to compromised systems when malicious APKs are installed.
2) Instrumentation of the application should rely purely on having access to the APK and no assumptions should be in place of the source code. In commercial, closed-source projects there is currently no method of reverse engineering the class files to a higher level language. It is therefore not wise to presume the program's intentions.
3) An application violating one or more policies should be halted in execution until it has conferred with the user. Failure to consult the user will result in loss of user control, and reduce the user's confidence in the tool.
4) Users of the application should be conferred with on decisions of whether to permit or deny methods intercepted by policies (Section III-B), unless they have given a preference prior to the fact. The tool must also remember the policy preferences, so it will not intrude on trusted or untrusted connections during future interactions.
5) The design should make no assumptions about the user's technical knowledge other than the fact they are able to download and install Android applications. The requirement stems from the fact that users of all backgrounds own and operate Android devices. It therefore is potentially isolating to user groups if they are assumed to have high technical knowledge.

B. Policies

In order to evaluate our system a set of policies must be outlined, which will be used to ensure our system correctly intercepts the desired activities. The following policies have been chosen as they address operations that are commonly used without permission by malware[8][19][20][21].

1) Prevent applications sending SMS messages to unauthorised numbers. Applications attempting to send SMS messages should be intercepted, with the recipient's number displayed to the user. This allows the user to make informed decisions on whether they want the SMS to proceed.
2) Prevent phone calls to unauthorised numbers. Applications attempting to initiate phone calls should be intercepted, with the recipient's number displayed to the user.
3) Restricting the application from making connections via a web browser. A user should be made aware when an application is attempting connections to a particular domain, and should be given the option to permit or deny a URL.

C. Design and Implementation Decisions

The following key design and implementation decisions were made in order to satisfy the design requirements.

1) Hooking Methods to Intercept APK at Run-time: To intercept an application during run-time, the most effective approach appears to be actual modification of the application before it is installed on the device. Existing approaches such as CRePe[14], TaintDroid[15], Apex[22] and Aurasium[18] either require modification of the Android firmware, or the Android application. As stated in the requirements we do not want to make assumptions of the user's technical knowledge, therefore deploying new firmware onto the Android device is not a desirable option. Our design is based upon programmatic modification of the application, in particular *overriding methods that could act as entry points for malicious behaviour*. This design came from the observation that many of the methods we were trying to intercept made calls to the same class, such as Activity, and it was possible to override this class and intercept the methods from a centralised manager. This approach involved disassembling the APK, inserting the custom interceptor class, and changing class paths where appropriate to direct the Activity class to the custom MyActivity class, before resuming to the normal task flow.

With this hooking approach we change the superclass *Activity*, which is used by all Android applications, to point to *MyActivity*. Our provided class *MyActivity* contains custom policy check methods that depend on the user policies.

This is a simpler approach than taken by exisiting approaches because our bytecode modification does not require modification of method bytecode.

2) User Interface: One of the design requirements outlines that there should be no assumptions made about the user's technical knowledge, and given that disassembling and reassembling applications recruits the help of `apktool`, `keytool` and `jarsigner` it seems sensible to implement a user interface to complement the hooking methods.

3) Policy Configuration File: One of the requirements of the security tool is that if it has conferred with a user about a connection, and the user indicates a preference on how to handle the connection, it must be able to store and recollect this information for future interactions. To allow fine-grained policies, it is ideal to give users a choice per number or URL whether they trust the recipient or not. For example, an application may be trying to send a message to a user contact. COMBdroid should be able to establish whether this is permitted or not based on past behaviour – if the number can be retrieved from a "Whitelist" then the connection should go through uninterrupted, however if the number has been previously flagged as a "Blacklist" number it should be automatically stopped. If COMBdroid cannot find any previous interaction with the number, it should first consult the user.

Storing persistent, dynamic information about past choices from the users cannot be implemented within the application. By the nature of Java-based applications, any parameters created at run-time and residing in the device's memory will be cleared once the application is killed, or the Android device is reset. For these reasons the policy configurations should be kept in a file that the COMBdroid application can refer back to every time an application is initiated.

We store the policy information within a file accessible only to the modified application. To achieve this, the `SecurityManager` will read from the PolicyConfig file when first started, or create a PolicyConfig file if one does not already exist.

D. System Overview

First, we present a complete overview of COMBdroid. Figure 1 shows a high level flow of events of instrumenting the application before installing it and injecting the classes that will provide run-time interception. The step-by-step process is as follows:

1) From the beginning, we assume a user has an APK file which they want to secure. Using a GUI, the user will select the APK and choose to disassemble it.
2) Disassembling will be performed with APKTool, revealing the `.smali` files which are then open to modification. Each `.smali` file will be searched line for line, looking for the specific pattern of `android/app/Activity`, which indicates the Activity class is being called. This string will be erased, and replaced with `combdroid/SecurityManager`, which means every time an Activity is called it will instead call the SecurityManager class.
3) The custom files which will provide interception at run-time – SecurityManager and Policy manager – will then be inserted into the `smali` directory, under their own folder named `combdroid`.
4) Finally, the application is reassembled, signed, and installed onto the device.

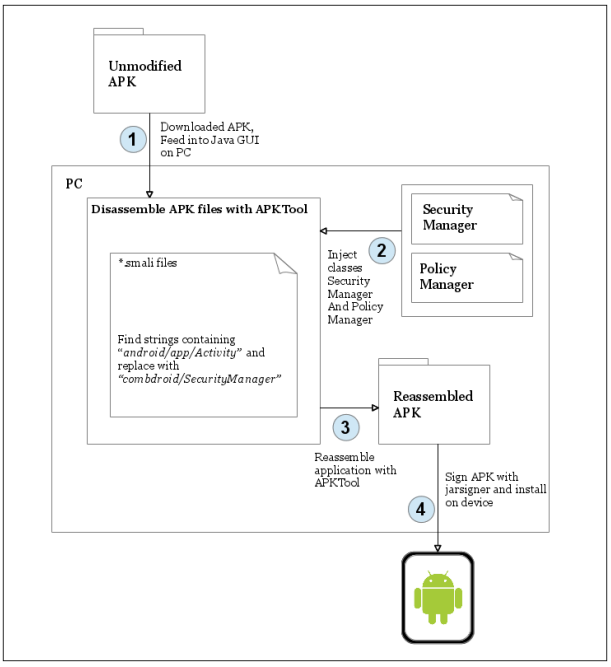

Fig. 1. Flow of instrumentation for COMBdroid

1) COMBdroid Instrumentation: The first part of the design requires creating an instrumentation tool which is able to manipulate the APK and insert the files that will intercept the application's behaviour at run-time. A step-by-step process of achieving this is outlined below. The disassembling and reassembling process has been designed from the documentation of the APKTool authors[7], and the modification steps derived from the hooking mechanism discussed in Section III-C1.

1) Ask the user to select an APK which they want to modify.
2) Disassemble the APK using apktool into assembly code `.smali` files.
3) Recursively replace the pattern `android/app/Activity` with `combdroid/SecurityManager`
4) Create a new directory named `combdroid` under the `smail` directory
5) Insert `SecurityManager.smali` and `PolicyManager.smali` into the *combdroid* directory
6) Reassemble the APK with `apktool`.
7) Sign APK with jarsigner using a key provided by ourselves so it will be accepted at loadtime.
8) Install signed APK onto device, using `adb` tool.

2) COMBdroid Run-time Flow: The files which will be inserted into the disassembled APK are the key factor behind intercepting and correctly handling undesired method calls at run-time. To handle interception, two files will be inserted – the SecurityManager class and the PolicyManager class. An

overview of the flow structure between the two classes can be seen in Figure 2.

The **SecurityManager** is the first point of contact when the `startActivity` method is called, and handles the extraction of intent information to pass to the PolicyManager. The SecurityManager should wait until the PolicyManager has assessed the information and returned a policy report. If the action is benign the SecurityManager should resume the application and if the action recipient has been blacklisted it should block the action. If the PolicyManager returns a report that indicates it has not seen this number before, the SecurityManager should raise an alert notifying the user of new behaviour from the application.

The **PolicyManager** is called from the SecurityManager for one of two reasons. Either the SecurityManager has been called for the first time and has PolicyConfig data to pass to the PolicyManager, or it requires the PolicyManager to evaluate potentially harmful intent data. If PolicyConfig data is passed in, the PolicyManager should store the information in a black or white list that it can use later for checking policies. If the SecurityManager is asking the PolicyManager to check a policy it should use the black and white listed data to determine the threat level, along with analysing whether the action is potentially an SMS, phone call or URL.

The **PolicyConfig** file is not inserted during APK instrumentation. Instead, it will be created by the SecurityManager during run-time. The SecurityManager will first check if the PolicyConfig file already exists, if it does not it will create one and if it does exist it will append new information to it. The PolicyConfig will be used to store blacklisted and whitelisted numbers that can be retrieved each time the application is used, as discussed in Section III-C3. The format of the stored file will be simple so that it may be read quickly from the SecurityManager on start-up. The file will use a new line for every whitelisted and blacklisted number, with each number preceded by the pattern `Whitelist:` or `Blacklist:`. Any comments may be inserted using the '#' character to indicate the text should not be parsed.

After both the SecurityManager and PolicyManager have been implemented each class will then be disassembled using `APKTool` so that the `.smali` version of each file may be inserted and recompiled with the modified APK. The following Section will explain the flow of events in greater detail with respect to Figure 2.

a) Run-time flow of Events: The following Section outlines the behaviour expected during run-time as a result of inserting the classes SecurityManager and PolicyManager. The steps followed correlate directly with Figure 2.

Step 1: The SecurityManager is called when an "original" class calls the `startActivity` method. If it is the first time the SecurityManager has been called it should first read in the PolicyConfig file, so that it may extract the blacklisted and whitelisted data defined by the user from previous uses of the application.

Step 2: The SecurityManager will then extract information about the activity intercepted and pass the activity data over to PolicyManager. The PolicyManager checks the activity data passed in, to determine whether the activity is likely a call, SMS or URL. If the activity is identified as a likely phone call, SMS or web browser activity, the activity will then be passed into a policy specific method, where the data is inspected at a more detailed level. Here, the PolicyManager will try match the recipient information to data stored in the black or whitelist it has, and create a notification accordingly to pass back to the SecurityManager.

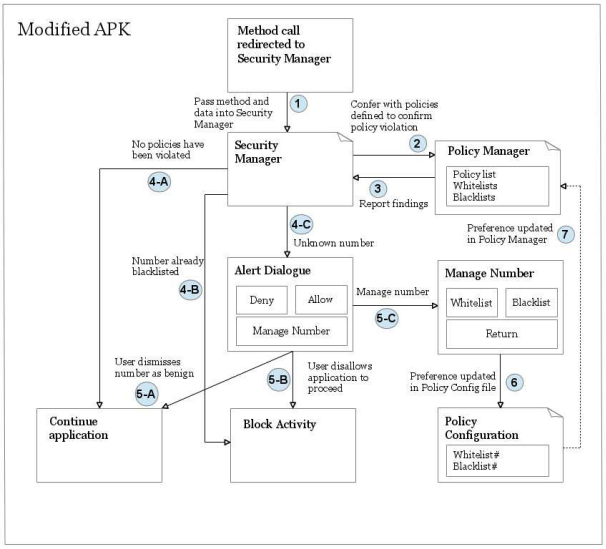

Fig. 2. Run-time flow SecurityManager, PolicyManager and PolicyConfig

Step 3: After the PolicyManager has performed its checks it will return a policy report to the SecurityManager.

Step 4-A: The SecurityManager will read in the information from the PolicyManager to determine whether the PolicyManager has identified a threat or not. If there are no threats found, or the recipient has been whitelisted, the SecurityManager will continue the application.

Step 4-B: If the PolicyManager has returned a report (from Step 3) indicating the number has already been blacklisted the activity is blocked and the application resumed.

Step 4-C: If the number returned from Step 3 is unknown it is assumed COMBdroid couldn't find any pre-existing numbers that matched the PolicyConfig file. This does not mean however that the number has not been alerted to the user before, it means that last time the number was alerted to the user they choose to simply "Deny" or "Allow" it. An alert is then raised presenting options to "Allow", "Deny" or "Manager Number".

Step 5-A: If the user "Allows" the activity the activity is permitted to proceed.

Step 5-B: If the user chooses to "Deny" the number the activity is blocked.

Step 5-C: If "Manage number" is selected another Dialogue Box presents itself, titled "Manage Number".

Step 6: Manage number presents options that will permanently store the user's preferences. "Whitelist" or "Blacklist" will update the number in the PolicyConfig file, and "Return" will continue the activity.

Step 7: If the user does choose to blacklist or whitelist the number from step 6, the PolicyManager is notified of the changes and the user preferences are persistently stored.

This flow of events means that the user does not have to commit to a permanent solution every time, especially if they are making one-off transactions to supervised numbers that they may not usually contact. It also allows for quick dismissal of a number if the user is unsure whether they should permit the call, and may choose at a later time to whitelist or blacklist the number.

E. COMBdroid Trust Assumptions

When designing a security-based tool there are certain assumptions that must be in place about the environment the tool is being deployed in, and how the user will interact with the environment after the tool deployment. These assumptions are the conditions COMBdroid holds while in operation, and will keep the system secure as long as the conditions are met before and after the application is installed. This Section discusses these conditions that are assumed of the environment COMBdroid will perform in, and presents reasoning of how COMBdroid will effectively secure a system if these conditions are kept.

The first assumption is that the environment COMBdroid is being deployed on **has not been compromised already by a malicious application**. If the Android device is already infected, this infected mechanism may have gained access to permissions that overrides COMBdroid's jurisdiction. For example, if there is a malicious rootkit already installed on the device the rootkit may override COMBdroid's alert which prevents applications from sending data[23].

As COMBdroid was designed to intercept malware by overriding potentially threatening methods in the Dalvik bytecode, it is assumed that the **attacker can only deliver their attacks through the APK byte code**, and we have correctly identified the entry points. If the user has permitted the APK to install extra information after COMBdroid has modified the application and been installed on the Android device, it is not possible for COMBdroid to intercept APK behaviour as a result. Furthermore, we assume the attacker cannot change or remove new subclasses at run-time. This situation should not occur however, as the Java language should prevent such actions occurring.

The PolicyConfig file created by the SecurityManager should be **protected by the Android OS if the file permissions state only the creator of the file has to access and modify it.** Therefore, we assume private files created will not be granted access from other applications by the OS, meaning malicious applications may not modify the PolicyConfig file to benefit malicious intentions. An example of how this may benefit attackers is if they are able to insert their own number as a whitelisted party, COMBdroid will then allow the activity to continue flagging the number as trusted by the user.

Finally, we assume that **COMBdroid will always run prior to execution of an activity**. If the user has modified the application after applying COMBdroid to the APK which overrides COMBdroid's authority to check the activity at runtime, there is no guarantee that COMBdroid will succeed in intercepting the application.

IV. Evaluation

This Section presents and discusses the results obtained from testing COMBdroid. First, the evaluation environment where COMBdroid was tested is outlined, followed by functional testing of COMBdroid. The functional testing is used to verify that COMBdroid is able to successfully intercept unknown recipients in activities and raise an alert to the user. The user preference should be stored, so that next time an activity is initiated with that recipient COMBdroid will remember that user's preference.

The enforcement testing showed it was successfully able to do this for each of it's call, SMS and URL policies. Next, the performance of COMBdroid is evaluated by testing the latency of the APKs from policy checking, the time taken to instrument the APKs using COMBdroid, and the size comparisons of an uninstrumented versus instrumented APK. Finally, COMBdroid is tested on Google Play applications to test its effectiveness on a closed-source application.

A. Evaluation Environment

The evaluation of COMBdroid was done on one PC using an Intel Core i7-3770 CPU processor running at 3.40GHz, with 8GB of RAM. The 3.7.5-1-ARCH kernel was used with the GNU desktop environment. The device used was a Samsung Galaxy Nexus running Android version 4.2.1, using a 1.2 GHz dual-core ARM Cortex-A9 CPU with 1GB of memory and 307 MHz PowerVR SGX540 GPU. The evaluation was performed in the School of Engineering and Computer Science at Victoria University of Wellington.

B. Functional Testing

We tested that policies were enforced correctly against misbehaving or malicious applications. This functional testing was implemented by creating our own test applications.

1) Phone Policy Enforcement: In order to test the phone policy enforcement mechanism an application has been manufactured that will make phone calls to predefined numbers without the user's intervention. Once the application initiates a call COMBdroid should raise an alert if the number that is being called is not found on the whitelist, or block the number if it is found to be blacklisted.

The testing will assume the PolicyConfig file to have no prior information on any numbers, meaning any call initiated should raise an alert. Two numbers will be dialled – the first call we will opt to blacklist the number (08001234567), and the second call we will opt to whitelist the number(08007654321). The two number will then be dialled again in the same order, the first time we expect to see a toast notification raised to indicate the number has been blacklisted and COMBdroid is blocking the call sequence. The second number dialled should be permitted to initiate a

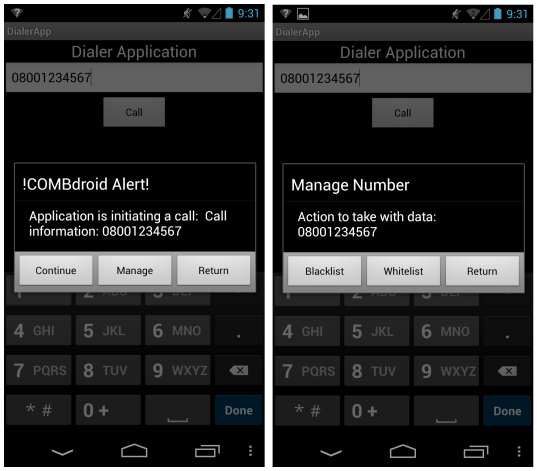

Fig. 3. COMBdroid intercepts an unfamilar number

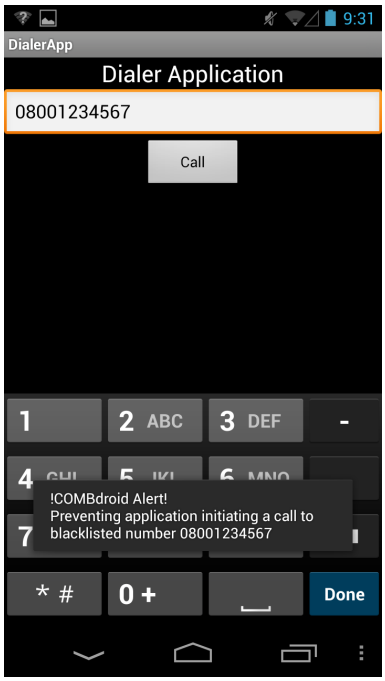

Fig. 4. COMBdroid prevents a blacklisted number initiating a call

call without interference from COMBdroid. After testing the `PolicyConfig` file will be inspected, where we will see the black and whitelisted numbers.

Phone Policy Enforcement Results. The first number, 08001234567, is dialled. COMBdroid succeeds in recognising it is an unknown number, and alerts the user as shown in Figure 3. The blacklist option is chosen, and the activity returns to the prior screen. The same is done for number 08007654321, and again a COMBdroid alert appears – this number is whitelisted.

When the number 08001234567 is dialled again only a toast message appears this time, shown in Figure 4. This shows COMBdroid has been successful in storing 08001234567 as a blacklisted number after the first call sequence and is now preventing further action.

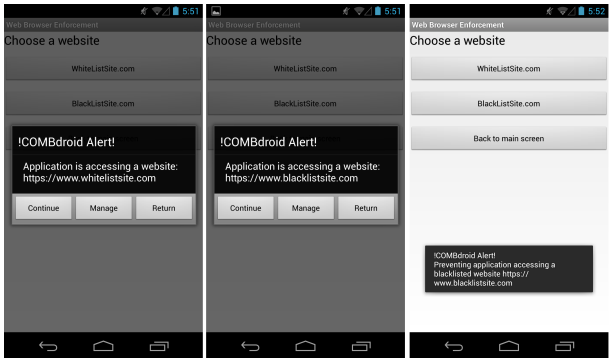

Fig. 5. COMBdroid raises alerts for unknown websites, and prevents a blacklisted website

The whitelisted number, 08007654321, is called once more. COMBdroid does not intervene this time, and instead the user is taken directly to the dialler screen.

To check COMBdroid has successfully stored both these numbers the `PolicyConfig` file is checked in the applications `/data/data/files/` folder, here the PolicyConfig file has been successfully created and has stored the black and whitelisted numbers as expected.

2) Web Browser Policy Enforcement: The second policy enforcement testing mechanism will test the web browser activity. The testing performed will be similar to testing the phone application, except COMBdroid will be testing intents to check whether a valid URL is included in the intent information. An application has been created of which the user will automatically be taken to a certain website when a button is clicked, which COMBdroid should intercept. The application will be installed fresh into the device with no `PolicyConfig` file, from there the first website chosen will be blacklisted and the second will be whitelisted. Connection to both websites will then be attempted again, where we should expect to see COMBdroid raise a toast and block the first one, and allow the second website to be displayed without interference.

Web Browser Policy Enforcement Results. The resulting actions from COMBdroid can be seen in Figure 5. The first run-through blacklisted *blacklistsite.com* and whitelisted *whitelistsite.com*. The second run-through attempted these websites again, where as expected COMBdroid raised a toast for *blacklistsite.com* (also shown in Figure 5) and performed no actions other than proceeding to the website when *whitelistsite.com* was called.

After testing the `PolicyConfig` file was checked, to see if the user choices had been updated correctly.

3) SMS Policy Enforcement: The SMS policy was the final policy that had been implemented. Similarly to the web browser and phone enforcement checks, two numbers were input to an SMS with a predefined number and text message body. The first number entered (0211234567) will be marked as a number to blacklist, and the second number (0217654321) will be whitelisted. Both numbers will be used again to create an SMS, where the first SMS should be blocked by COMBdroid and the second should be sent without COMBdroid

interfering. Again, the `PolicyConfig` file will not exist before testing, and will be checked to ensure COMBdroid has correctly stored the black and whitelisted numbers.

SMS Policy Enforcement Results. Similarly to sections IV-B1 and IV-B2 the enforcement mechanism for SMS worked as expected (raising similar alerts as shown in Figure 3). The PolicyConfig file also was updated correctly.

C. Performance Testing

The Section describes how we tested the performance impact of how applications behave prior and post modification from COMBdroid. Metrics that will be analysed are as follows:

- The latency of an application as it consults the policy manager during activity initiation, to determine whether an application's performance is noticeably impacted by COMBdroid's security checks.
- The time it takes to practically modify an application with the COMBdroid GUI. This testing was performed on both a single application and a group of applications.
- The size differences of an unmodified and modified application. Although the capacity of an Android device is growing with each product version released, many devices still have reduced capacity, forcing users to be resource conscious.

Performance metrics are critical to evaluation as our application is based on a mobile, resource conscience device. Users will be deterred from applying COMBdroid to applications if it noticeably affects their application's performance, or drains the device energy at an increased rate.

1) Application Latency: Whenever an application initiates an activity that the Security Manager flags as potentially unsafe it must pass this information to the Policy Manager to perform checks on the data. This injected behaviour takes time, and the objective of this test is to evaluate whether the checks are resulting in acceptable latency.

The applications used to test the enforcement mechanisms in Section IV-B will be modified so that 20 intents are called in a row, of either phone calls, SMS, or URL calls. The time will be measured from when `startActivity` is called (where Security Manager takes over) to when the activity is cleared. As the objective is to measure intrusion when the user is content with the data they have blocked and allowed, all data will assume to be whitelisted so time taken for user input does not interfere with testing results.

Table I shows 20 tests performed each on the phone, browser and SMS latency caused from consulting the policy manager, Figure 6 shows the data in a graphed format. For each, the initial overhead is comparatively higher – likely caused by COMBdroid having the read in the `PolicyConfig` file when it is first called. From there, the latency overhead subsides to 1-2 milliseconds per call. These results are satisfactory as a delay time of 1-2ms is unlikely to go noticed by the user, meaning the usability of the application during run-time is not degraded.

TABLE I
STATISTICS GAINED FROM LATENCY INDUCED THROUGH POLICY CHECKS

Attempt Number	Phone Latency (ms)	Browse (ms) Latency (ms)	SMS) Latency (ms)
1	13	9	11
2	1	3	4
3	2	3	2
4	1	2	1
5	2	2	1
6	1	1	1
7	1	7	2
8	1	1	3
9	2	1	3
10	1	1	2
11	1	3	1
12	2	2	1
13	1	2	1
14	1	2	1
15	1	1	2
16	2	1	2
17	2	2	1
18	1	1	1
19	1	1	1
20	1	2	1

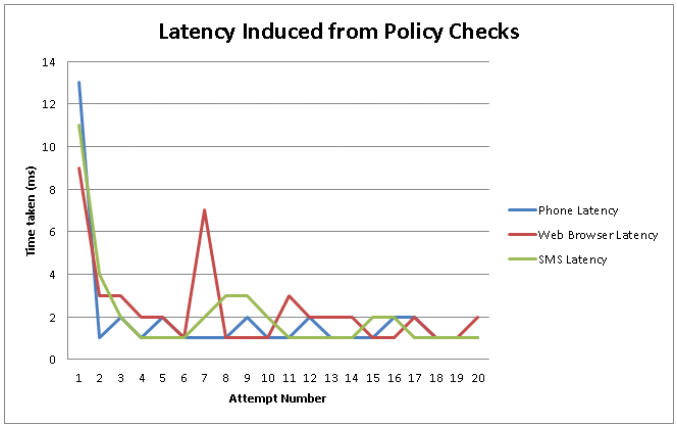

Fig. 6. Latency in application from policy check overhead

V. CONCLUSION AND FUTURE WORK

The Android operating system has quickly risen in popularity in recent years, consequently drawing the attention of attackers who wish to exploit users. Commonly this exploitation is through sending premium SMS or phone calls without the user's knowledge[24][25][19]. The permission system of Android makes these attacks easy to orchestrate, as once a user gives an application permissions such as `CALL_PHONE` or `INTERNET` the application may use these permissions however they please, regardless of whether user consent has been given. To address these issues, we looked to develop a security tool that would enforce fine-grained policies on applications, providing greater transparency between user and application. It was found through the related works that run-time instrumentation tools that were most effective at mitigating unwanted behaviour, and from there a system named COMBdroid was developed.

COMBdroid was able to instrument an application and inject its own custom classes, so that class methods that were

identified as points of vulnerabilities could be overridden and verified by our own policy checker. COMBdroid was designed to give users control, so that they may specify trusted and untrusted recipients, and have their preferences applied each time the instrumented application was run. Three policies were implemented for this project; SMS initiation, phone calls, and URL calls. The enforcement mechanism of each policy was verified, and performance evaluation showed COMBdroid to have minimal impact on the user's application functionality. COMBdroid was tested on Google Play store applications, where it shows promising results of intercepting unwanted behaviour.

Overall, COMBdroid has been a success, however this is only the beginning of what COMBdroid is capable of accomplishing. Potential improvements and alterations are discussed in the Section.

A. Future Work

Though the evaluation of COMBdroid has shown COMBdroid to be an effective tool for protecting users against unwanted activity calls initiated by Android applications. COMBdroid has potential to grow into an even more powerful policy checker however, and in this Section we outline areas of future work.

1) Greater Policy Control for Users: Currently, three policies that have been implemented for COMBdroid, however during the background review Section there were more threats identified which we could incorporate into the PolicyManager class. Once the appropriate entry points have been identified (such as an intent using `ACTION_CALL` with "tel;" in the data representing a phone call) they would be easy to integrate into the PolicyManager class. The PolicyManager class separates its methods for policy checking by defining a new method for each policy check, meaning new policies would be built into new methods. Such policies could include searching for alternate ways of initiating phone calls (for example, telephony manager), various SMS methods (such as SMSManager), and more browser policy checks (such as Webview). Completely new policies could involve subverting calls from foreign applications, attempting to use permissions from other applications for their own gain (resulting in privilege escalation) or restricting access to internal unique identifiers.

This project aims to give users greater control, so it is fitting that they have fine-grained control over the policies enforced. Instead of having one class which performs policy checking it would be possible to separate each policy into its own class, and have the PolicyManager class act as more of an abstract class, which each policy inherits. This approach would mean during instrumentation the COMBdroid interface could use a check-box method of getting users to decide which policies they would like to enforce, and only the classes of the policies chosen would be included when inserting COMBdroid's files.

2) Global, Centralised Management for COMBdroid: COMBdroid works on a per-application basis, meaning every time a user blacklists or whitelists a number their preference will only be recalled within the scope of that application. Implementing a global application that other instrumented applications could call out to would allow the user to allow or deny a number once, and have confidence that this preference will be applied across all applications.

The approach would be implemented by creating a custom application for COMBdroid users to install on their phone, containing the policy configuration settings for other applications to call out to. Each application participating would still require instrumentation for methods interception, however instead of using a local policy configuration file to determine policy violations the application would invoke the centralised COMBdroid application using `Intent.ACTION_MAIN`, retrieving black and whitelisted data. This approach would also involve notifying the centralised application when user preferences are defined, such as blacklisting or whitelisting a new recipient.

Having COMBdroid as its own application would also mean users would be able to edit the policy configuration file if they want to add or delete custom numbers. As COMBdroid currently resides in the background of applications, there has been no verifiable way to allow users to edit their policy configuration file. The COMBdroid interface as a standalone application would make it easy to permit text editing for the policy configuration file through application context control.

3) Flashback Capabilities in the COMBdroid Interface: COMBdroid alters an application by injecting custom classes and modifying the application `.smail` files. Once an application has been transformed there is no way for the user to revert back to the APK's original form, unless they uninstrument the APK manually or source the original APK. Having a "Flashback" mechanism for COMBdroid would give the user assurance that if for any reason they are not satisfied with COMBdroid's behaviour as a result of the transformation, they could easily revert back to the original state. This would not be too difficult to implement, as COMBdroid already has methods to disassemble and reassemble applications, and there would be no permission conflicts with removing the custom classes. There is also a find and replace method already implemented in the `ModifyApp` class, meaning it could be reversed to look for all instances of `combdroid/SecurityManager` and replace them with the original `android/app/Activity`.

REFERENCES

[1] I. C. USA, Apple cedes market share in smartphone operating system market as android surges and windows phone gains, according to idc, http://www.idc.com/getdoc.jsp?containerId=prUS24257413. [Online]. Available: http://www.idc.com/getdoc.jsp?containerId=prUS24257413

[2] Google, Android developers, http://developer.android.com/index.html. [Online]. Available: http://developer. android. com/index. html

[3] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, Android permissions demystified, in Proceedings of the 18th ACM conference on Computer and communications security, ser. CCS 11. New York, NY, USA: ACM, 2011, pp. 627638. [Online]. Available: http://doi. acm. org/10. 1145/2046707. 2046779

[4] L. Davi, A. Dmitrienko, A. -R. Sadeghi, and M. Winandy, Privilege escalation attacks on android, in Proceedings of the 13th international conference on Information security, ser. ISC10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 346360. [Online]. Available: http://dl. acm. org/citation. cfm?id=1949317. 1949356

[5] W. Enck, M. Ongtang, and P. Mcdaniel, Mitigating android software misuse before it happens, Tech. Rep. , 2008.

[6] Google, Dalvik technical information, Android open source project, http://source. android. com/tech/dalvik/. [Online]. Available: http://source. android. com/tech/dalvik/

[7] B. Alll and C. Tumbleson, A tool for reverse-engineering android apk files, apktool, https://code. google. com/p/android-apktool/. [Online]. Available: https://code. google. com/p/android-apktool/

[8] Y. Zhou and X. Jiang, Dissecting android malware: Characterization and evolution, in 2012 IEEE Symposium on Security and Privacy (SP). IEEE, 2012, pp. 95109.

[9] O. Hou, A look at google bouncer j malware blog j trend micro, http://blog. trendmicro. com/trendlabs-security-intelligence/a-lookat- google-bouncer/. [Online]. Available: http://blog. trendmicro. com/ trendlabs-security-intelligence/a-look-at-google-bouncer/

[10] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, Semantically rich application-centric security in android, in Proceedings of the 2009 Annual Computer Security Applications Conference, ser. ACSAC 09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 340349. [Online]. Available: http://dx. doi. org/10. 1109/ACSAC. 2009. 39

[11] H. Lockheimer, Android and security - official google mobile blog, http://googlemobile. blogspot. co. nz/2012/02/android-andsecurity. html. [Online]. Available: http://googlemobile. blogspot. co. nz/ 2012/02/android-and-security. html

[12] W. Enck, M. Ongtang, and P. McDaniel, On lightweight mobile phone application certification, in Proceedings of the 16th ACM conference on Computer and communications security, ser. CCS 09. New York, NY, USA: ACM, 2009, pp. 235245. [Online]. Available: http://doi. acm. org/10. 1145/1653662. 1653691

[13] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, Riskranker: scalable and accurate zero-day android malware detection, in Proceedings of the 10th international conference on Mobile systems, applications, and services, ser. MobiSys 12. New York, NY, USA: ACM, 2012, pp. 281294. [Online]. Available: http://doi. acm. org/10. 1145/2307636. 2307663

[14] M. Conti, V. T. N. Nguyen, and B. Crispo, Crepe: contextrelated policy enforcement for android, in Proceedings of the 13th international conference on Information security, ser. ISC10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 331345. [Online]. Available: http://dl. acm. org/citation. cfm?id=1949317. 1949355

[15] W. Enck, P. Gilbert, B. -G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones, in Proceedings of the 9th USENIX conference on Operating systems design and implementation, ser. OSDI10. Berkeley, CA, USA: USENIX Association, 2010, pp. 16. [Online]. Available: http://dl. acm. org/citation. cfm?id=1924943. 1924971

[16] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, These arent the droids youre looking for: retrofitting android to protect data from imperious applications, in Proceedings of the 18th ACM conference on Computer and communications security, ser. CCS 11. New York, NY, USA: ACM, 2011, pp. 639652. [Online]. Available: http://doi. acm. org/10. 1145/2046707. 2046780

[17] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, Quire: lightweight provenance for smart phone operating systems, in Proceedings of the 20th USENIX conference on Security, ser. SEC11. Berkeley, CA, USA: USENIX Association, 2011, pp. 2323. [Online]. Available: http://dl. acm. org/citation. cfm?id=2028067. 2028090

[18] R. Xu, H. Saidi, and R. Anderson, Aurasium: practical policy enforcement for android applications, in Proceedings of the 21st USENIX conference on Security symposium, ser. Security12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2727. [Online]. Available: http://dl. acm. org/citation. cfm?id=2362793. 2362820

[19] H. Pieterse and M. Olivier, Android botnets on the rise: Trends and characteristics, in Information Security for South Africa (ISSA), 2012, 2012, pp. 15.

[20] T. Wyatt, Security alert: Geinimi, sophisticated new android trojan found in wild j the official lookout blog, https://blog. lookout. com/blog/2010/12/29/geinimi trojan/. [Online]. Available: https://blog. lookout. com/blog/2010/12/29/geinimi trojan/

[21] E. Erturk, A case study in open source software security and privacy: Android adware, in Internet Security (WorldCIS), 2012 World Congress on, 2012, pp. 189191.

[22] M. Nauman, S. Khan, and X. Zhang, Apex: extending android permission model and enforcement with user-defined runtime constraints, in Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ser. ASIACCS 10. New York, NY, USA: ACM, 2010, pp. 328332. [Online]. Available: http://doi. acm. org/10. 1145/1755688. 1755732

[23] J. Bickford, R. OHare, A. Baliga, V. Ganapathy, and L. Iftode, Rootkits on smart phones: attacks, implications and opportunities, in Proceedings of the Eleventh Workshop on Mobile Computing Systems #38; Applications, ser. HotMobile 10. New York, NY, USA: ACM, 2010, pp. 4954. [Online]. Available: http://doi. acm. org/10. 1145/1734583. 1734596

[24] S. S. of Trustwave, Focus stealing vulnerability in android, https://www. trustwave. com/spiderlabs/. [Online]. Available: https://www. trustwave. com/spiderlabs/

[25] C. A. Castillo, Android malware past, present, and future, McAfee,[online], 2010.