FireDroid: Hardening Security in Almost-Stock Android

Giovanni Russello
University of Auckland
g.russello@auckland.ac.nz

Arturo Blas Jimenez
University of Auckland
a.blas@auckland.ac.nz

Habib Naderi
University of Auckland
h.naderi@auckland.ac.nz

Wannes van der Mark
University of Auckland
w.vandermark@auckland.ac.nz

ABSTRACT

Malware poses a serious threat to Android smartphones. Current security mechanisms offer poor protection and are often too inflexible to quickly mitigate new exploits. In this paper we present FireDroid, a policy-based framework for enforcing security policies by interleaving process system calls. The main advantage of FireDroid is that it is completely transparent to the applications as well as to the Android OS. FireDroid enforces security policies without modifying either the Android OS or its applications. FireDroid is able to perform security checks on third-party and pre-installed applications, as well as malicious native code. We have implemented a novel mechanism that is able to attach, identify, monitor and enforce polices for any process spawned by the Android's mother process Zygote. We have tested the effectiveness of FireDroid against real malware. Moreover, we show how FireDroid can be used as a swift solution for blocking OS and application vulnerabilities before patches are available. Finally, we provide an experimental evaluation of our approach showing that it has only a limited overhead. Given these facts, FireDroid represents a practical solution for strengthening security on Android smartphones.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*access controls, information flow controls*

General Terms

Security

Keywords

Android Security Enhancement, System Call Interposition, Policy-based Security

1. INTRODUCTION

Smartphones are the most successful consumer devices to date reaching 821 million units sold to end users in the last quarter of 2012 [1]. In this very competitive market, smartphones equipped with the Android OS reached 75% of the total sold devices [2]. According to Google chairman Erich Schmidt, 1.3 million Android devices are activated each day [3]. As these simple statistics testify, Android plays an important role in the mobile device market.

For the Android OS, developers have created more than 700,000 applications downloadable via well-known distribution points like the Google Play marketplace [4]. This openness is one of the key factors contributing to the success of Android. At the same time, however, it has raised new security concerns. Given the popularity of Android, it is not a surprise that malware is growing rapidly. In the last quarter of 2012, the number of unique instances of Android malware has increased from 5,000 to 51,500 compared to the previous quarter [5].

1.1 Motivation

Smartphones empower users with ubiquitous computing: users can perform the same tasks as with laptops and desktops but with more flexibility and mobility. In the private, public, and military sectors, organisations' personnel relies on smartphones for performing sensitive tasks by installing business applications. Android penetration in the business sector is on the rise and poses serious security challenges to IT departments and CIOs [1].

Because users can install third-party applications on their smartphones, several security concerns may arise. For instance, malicious applications can upload to remote servers private information stored on the device such as contact lists, SMS and MMS messages. By exploiting pre-installed application vulnerabilities [6], malware can reset the device to its factory settings deleting all the data that it contains. This poses serious security concerns to sensitive corporate data, especially when the standard security mechanisms offered by the platform are not sufficient to protect the users from such attacks. Solutions such as anti-virus software are not able to cope with the rapid evolution of malware. As reported by Zhou and Jiang in [51], the best anti-virus software is still failing to detected 21% of the malware currently in the wild. More worrisome is the fact that all the anti-virus solutions are vulenerable against simple malware transformation techniques [43].

The research community has addressed the security issues in Android proposing several approaches. One approach is to identify a malicious application by analysing the set of permissions it requests [32, 41]. Analysing permissions an application requires might not be very accurate at identifying real issues. For instance, most malicious applications request the SMS permission. Therefore, just requesting such a permission might flag an application as malicious although it might be a legitimate application. Moreover, these approaches do not take into account the problem of colluding applications performing privilege escalation attacks [25].

Another research trend has focused on extending the Android security mechanism by enforcing fine-grained policies [40, 20, 53,

ACSAC '13 Dec. 9-13, 2013, New Orleans, Louisiana USA

http://dx.doi.org/10.1145/2523649.2523678.

37], separating the runtime environment with light virtualisation [21, 45], and implementing full-virtualisation by hypervisor [39, 18]. The main issue of these approaches is that they require extensive modifications of the Android image installed on the device. As such, the widespread adoption of these approaches is hampered by the fragmentation problem of Android [7]. Each manufacturer introduces its own customised OS image for coping with the different hardware configurations. Therefore, these approaches would require maintaining a customised version of Android for each hardware configuration. Moreover, they are effective only within the Android middleware. However, malware in the wild can easily bypass these approaches given that it often relies on attacks executed by native code at the Linux level [51].

More recently, application-hardening approaches have been proposed where third-party applications are modified to inject security hooks [44, 49, 19, 26, 38]. When the application package is downloaded, extra-security checks are added to the application code generating a hardened version. The main advantage of these approaches is that (i) it does not require any changes of the Android code and (ii) the hardening tools can be easily deployed as standard applications on the device. However, there are several shortcomings in these approaches. First of all, they rely on the user to install the hardened version of an application. Nothing can prevent the user to install the original version of an application bypassing the security policies an organisation might want to enforce. Second, installing the hardened version of an application means that all the data associated with the original application will be lost. The reason is that Android application packages (APK) are digitally signed by their developers. After the tools repackage the application, the original signature is lost. Therefore for Android the hardened version of the application represents a new package with different access rights to the data storage. Third, because the hardening tools need to access the application bytecode, they can only be applied to third-party applications. Pre-installed applications and system services cannot be hardened by these tools since their code is part of the Android image on the device (unless they reflash the device image). Finally, these approaches can only control the bytecode of an application. However, applications can load at runtime malicious native code bypassing the injiected security monitor.

What is needed is a solution that (i) does not require to recompile any parts of the Android middleware and the underlying OS; and (ii) it provides support for enterprise security management where security administrators can define security policies to be enforced on the devices without relying on the device's user.

1.2 Contributions

In this paper we propose FireDroid, an effective security solution for enforcing fine-grained security policies without the need to recompile any internal modules of the Android OS. FireDroid monitors the execution of system calls and it is able to effectively confine the execution of processes within a secure sandbox. Although sandboxing based on system call interposition is not new, the main contribution of FireDroid lies in the exploit of the unique Android mechanism for spawning applications to automatically monitor applications executed in an Android device without the need of user intervention. This makes FireDroid an ideal tool for deploying and enforcing enterprise security policies. Previous interposition techniques such as [48, 42] have been proposed to allow users to contain untrusted applications. FireDroid differs from those tools as by default it automatically monitors any application a user launches without relying on the user's input.

As a main advantage compared to other Android security approaches, FireDroid is able to monitor any application and system code executed in a device. This makes FireDroid very effective also for controlling the execution of native code that is the main attacking vector used by malware. We have implemented FireDroid and evaluated its effectiveness against a large set of real malware samples. As another important contribution of this paper, we have created a set of policies that can protect a device from attacks performed by current malware families. Moreover, FireDroid can be used to stop attacks exploiting OS and applications vulnerabilities by simply enforcing dedicated policies.

The rest of this paper is organised as follows. Section 2 provides an overview of Android and its security model. Section 3 presents the system model of our approach. Section 4 focuses on the implementation details of FireDroid. The specification of the security policy language is given in Section 5. To demonstrate the effectiveness of FireDroid, we provide a security evaluation against real malware and attacks in Section 6. The evaluation of FireDroid performances is analysed in Section 7. In Section 8, we review existing approaches aiming at enhancing Android security. Section 9 highlights future research directions. Finally, Section 10 provides our concluding remarks.

2. ANDROID

Android is an open-source mobile platform developed by Google and the Open Handset Alliance (OHA) [17]. At the core of Android there is a Linux kernel optimised for mobile devices. In Android, applications are mostly coded in Java and compiled into Dalvik bytecode to be executed by the Dalvik Virtual Machine (DVM).

Each Android application is executed in a dedicated DVM. At the Linux level, each application is executed within a separate process. To control access to protected resources, Android combines the traditional Linux permissions with a Mandatory Access Control (MAC) mechanism [31, 46]. During install time, applications are assigned permission labels representing the resources they can access during runtime. The developer of an application must declare the permissions her application requires in its package manifest file. Once the permissions are granted at install time, the user cannot revoked them unless she uninstall the application.

Although each application executes within a dedicated sandbox, Android allows applications to communicate with each other through a well-defined Inter-Process Communication (IPC) mechanism. The IPC mechanism used in Android is called Binder [8] and it mediates tje communications between applications' components. The Binder takes care of migrating the execution of a request from the requester to the target process transparently to the applications.

In designing Android, Google engineers gave the highest priority to usability and integrability. However, in doing so they have provided a large attack surface easily exploited by malicious code. The main issue is that the Android security permissions are too coarse-grained to provide adequate protection to the user. In the rest of this paper, we present our approach to address this issue.

3. FIREDROID SYSTEM DESIGN

The main observation behind our approach is that although most of the Android applications do not interact directly with the Linux level, all their privileged operations rely on system calls executed by the Linux kernel. As such, by controlling the execution of system calls it is possible to control the behaviour of applications. However, defining security policies for such a low-level vetting mechanism is very tedious and error prone. That is why, we have developed a policy language (see Section 5) for specifying high-level policies that are then mapped to policies enforceable at the level of system calls.

There are several ways in which a system call interposition can be realised. Because our main goal is to extend the Android security without modifying applications, the Android middleware, and underlying Linux OS, we decided to use the `ptrace()` system call for tracing applications' executions. When FireDroid is deployed, each Linux process is monitored by a **FireDroid Application Monitor (FDAM)**. As shown in Figure 1, the FDAM attaches to the target process through the `ptrace()` system call meaning that each time the target process executes a system call, the kernel suspends the target process and notifies the FDAM. Within the FDAM, the **Policy Enforcement Point (PEP)** is responsible for gathering the required information from the system call executed by the target process including the parameters in the system call. The PEP forwards this information to the **Policy Decision Point (PDP)** that will retrieve the relevant policies from the **Policy Repository (PR)** within the FDAM. The PR contains policies specific to the process being monitored (more on this in Section 4). Depending on the policies, the PDP can decide to allow or deny the execution of the system call. Moreover, the policy evaluation might also return the decision to kill the target process. Another possible outcome of the policy evaluation is to inform the user and to ask her decision (either allow, deny or kill the process). To ask the user, the PDP contacts the **FireDroid Service (FDS)**. This service is implemented as an Android application and provides to the user an interface to interact with FireDroid.

The FDS provides a **Policy Administration Point (PAP)** to manage security policies defined at device level. In FireDroid, the user can dis/enable policies that can be applied to her device. However, in an enterprise scenario, enterprise-specific policies are not managed by the user. In this case, the security administrator can remotely manage the security policies by sending updated policies. The **Remote Policy Manager (RPM)** component handles the remote edit/update policy requests. New policies can be sent through SMS/MMS and/or Bluetooth. When a remote request for an update is made, first the RPM verifies the authenticity of the policies' owner. This can be done by including with the request a certificate with the identity and the owner's public key. If the authentication phase completes successfully, the RPM first stores the new set of policies in the **Global Policy Repository (GPR)** and then pushes the policies into the PRs of the respective FDAMs.

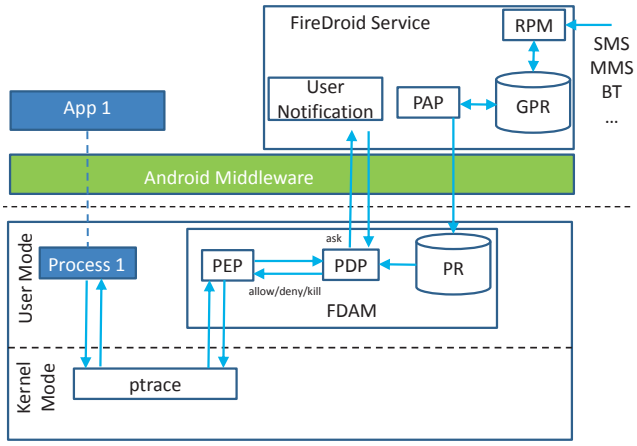

Figure 1: FireDroid Details.

4. IMPLEMENTATION DETAILS

This section is dedicated to the details of FireDroid implementation. FireDroid controls applications' behaviour by monitoring their interactions with the OS. Other approaches, such as Janus [48] and Systrace [42] have been proposed for system call interposition in the Linux OS on fully-fledged desktop machines.

There are two main differences between FireDroid and both Systrace and Janus. First of all, Systrace and Janus have been developed for Linux boxes where launching a process is commonly done with a shell under a user id. Android does not sport a shell (although emulators exist) and the concept of user id does not map with the one of a Linux box running on a PC. Android is a single-user OS. Although the underlying Linux kernel supports multiple user ids, these are used to differentiate applications developed from different authors. The second difference is that Systrace and Janus are implemented at kernel level which means that the Linux kernel needs to be modified to be able to intercept system calls. We are aware of the Systrace version based on `ptrace()` that does not require modifying the kernel [9]. However, even if a port of the `ptrace()`-based Systrace existed for Android, it would still require a Linux-based approach to be able to attach to a process. Although such a mechanism might be still feasible in Android, for the user is very awkward to use.

In more details, compared to a desktop PC, smartphones are still limited in computational power. To speed up the starting up of applications, the launching of a new application is managed by the **Zygote** process. Zygote is started when Android is booted and is the only process authorised to fork new processes. When an application is started, Zygote forks a new process copying its initialised structures. The sharing of this initialised components allows a faster start-up of the new application.

In FireDroid, we had to come up with a non-trivial technique to attach FDAMs to applications able to deal with the launching mechanism enabled by Zygote. When using `ptrace()` to monitor a process, the monitoring process needs to be its parent process. In a fully-fledged Linux OS, the user can launch the target process as a child of the monitoring process through the shell as prescribed for Systrace. As we have said above, such a mechanism is either not available or very awkward to use in Android.

In FireDroid, we solve this problem by monitoring the Zygote process itself. We implemented a monitor process called **FireDroid Main Monitor (FDMM)**. The FDMM is only responsible for monitoring when Zygote is executing the `fork()` system call. To be able to attach to Zygote, the FDMM has to be its parent process. Because Zygote is started when Android is booted, we have to attach the FDMM to Zygote at booting time. This can be achieved by modifying the `init.rc` file, that is the scripting sequence file in plain text used in Android for booting up. When the FDMM starts, it invokes the `ptrace()` system call to attach to Zygote and controls its execution since the first moment it starts. It should be noted that although the `init.rc` file is part of the Android image, it is only a text file. Any modification to this file does not require recompilation of the image. Using this mechanism, FireDroid is capable of automatically attaching to applications without relying on the user. This increases FireDroid's effectiveness as a security enforcement tool in an enterprise environment.

Modifying the `init.rc` requires access to the device image. We foresee the use of FireDroid in a corporate environment. Enterprises and organisations usually lease their devices from telcos. Telcos represent our entry point for FireDroid deployment since they can request that vendors customise the image installed on the devices before shipping them to corporate customers. Telcos could also sell devices with FireDroid installed to end users. In this way,

FireDroid could enable a corporation to support a BYOD policy through FireDroid devices owned by employees.

Alternatively, the `init.rc` file can be modified by rooting the device and extracting the booting partition. There are three main points to clarify. First of all, rooting the device does not void the warranty of the device. There are laws in the US (United States federal law, 15 U.S.C. ğ 2301 et seq.) [10] and EU (Directive 1999/44/CE) [11] that enforce a vendor does not void the entire warranty because a device has been modified by rooting/romming. Interestingly enough, majors vendors such as Samsung and HTC provide software tools and detailed descriptions on how to root their devices. Second, once the device has been rooted it might expose the user to some risks where applications might get to execute at root privileges. While this is certainly true, in FireDroid once the `init.rc` has been modified the device is unrooted. Finally, because the device needs to be rooted, installing FireDroid might not be a task for the average user. In this case, the security administrators of the enterprise can install FireDroid on the employees' devices. As a matter of fact, we have developed a set of scripts that automatically extracts the `init.rc` file from a device image, modifies it with our extension, and reflashes it in the device together with the FireDroid code. The scripts support several models of smartphones and tablets from different vendors. The security administrator would also be responsible for writing the security policies according to the organisation' security requirements.

Finally, OTA updates might represent a concern for devices with FireDroid installed. First of all, it must be necessary to perform some compatibility tests with the updated code. Another concern is that an OTA update might disable FireDroid. However, to disable FireDroid there are two main approaches: either modifying the `init.rc` file or disabling the `ptrace()` system call. The former cannot be performed as an OTA update because requires physical access to the device to extract the boot image. The latter, we believe it is very unlikely to happen.

FireDroid attaching mechanism enables us to monitor any process spawn from Zygote. This means we are able to control third-party applications as well as any pre-installed applications. As identified by Grace et al. in [36], pre-installed applications might suffer from capability leakages. To prevent those leakages, FireDroid can easily restrict their behaviour without the needs to wait for the vendors to provide patches to solve the issues. In principle, FireDroid is able to attach to all the processes and system services running on a device. However, it is possible to configure through the use of policies which processes FireDroid has to monitor limiting the overall impact on performance.

The enforcement of security policies through system call interposition needs to address several issues related to incorrectly replicating the OS and side effects introduced when a system call is denied [34]. In FireDroid, for most of these issues we have taken the same precautions as suggested in [42]. For instance, to avoid replicating the OS semantics, when a loopback network policy is enforced the FDAM retrieves the ports that processes are listening to using the information maintained in the `/proc` filestystem.

However, Android introduces some specific challenges that need to be addressed. For instance, colluding applications can use shared memory for changing the values of symbolic links used in system calls between the time of policy evaluation and the time when the parameters are actually used in the system call. In Android, the Linux native shared memory mechanism has been disabled. Instead, Android supports two services for sharing memory: `ashmem` and `ION` implemented as pseudo device drivers. In both cases, applications use file descriptors for allocated memories which can be shared through the Binder. FireDroid is able to control IPC through the Binder and it can enforce policies that do not allow processes to exchange file descriptors pointing to `ashmem`/`ION` shared memory regions. Moreover, when a system call is invoked, the values of the system call arguments are copied into a buffer in the FDAM address space. When the FDAM finishes the processing and before letting the system call to continue, it copies the arguments back to the original addresses. In this way, we reduce the time window a malicious process has to manipulate the system call arguments.

```
1 Requester Operation [param-list] on Target
2 [if condition] then outcome [do action]
```

Figure 2: The syntax of the FireDroid Policy Language.

5. FIREDROID POLICIES

FireDroid enforces security policies for controlling how processes execute system calls. However, specifying security policies at the level of system calls is quite a tedious task. In general, we can categorise system calls based on the resources that processes can access through their execution. For each of these categories, we can define security policies that are then translated in controls performed by the FDAM on the underlying system calls. For instance, a policy controlling how applications access the SMS service is translated in a set of low level policies that control the `ioctl` system calls to the Activity Manager and the iSMS service through the Binder.

We have defined the FireDroid policy language and its syntax is shown in Figure 2. The `Requester` represents the process requesting the `Operation` to be executed on the `Target`. Policies can have an optional `if condition` clause to evaluate boolean expressions defined over the parameters of the current operation being evaluated and/or over contextual data (such as time of the day, location, etc.). If a condition is specified the policy is enforced only if its condition evaluates to true. If no condition is specified then by default its value is always true.

The `then outcome` represents the result of the evaluation of the policy. Possible values for the `outcome` are: `allow`; `deny`; `kill`; and `ask`. When the `ask` outcome is selected, the policy will notify the user with a message providing information about the operation the application is about to perform and some information about the danger of letting the application execute the operation. The user can select one of the three options: `allow`, `deny`, or `kill`. Once the user selects an option, she can also request the system to remember her decision if the application executes the same operation under the same conditions in the future.[1] However, multiple policies might be evaluated for the same operation outputting different `outcome` values. To allow the PDP to select one `outcome`, the security administrator can set priorities for the different outcomes per policy type and application packages.

The optional `do action` clause can be used to specify actions that are executed when the policy is evaluated. Actions use a set of functions provided by FireDroid to use the functionality of the Android layer. Examples of the functionality supported in the `do action` clause include but are not limited to: modifying the returned value of `Operation`; interacting with application components such as broadcast receivers and content providers; sending text messages and emails; and finally logging information. The `Requester` can be either an application or a system service (such as the Activity Manager). The `Target` can also be an application or a service, but it could also represent a resource, such as a file or

[1]This mechanism can be used for tailoring the decisions the system takes to the user's needs.

a content provider. For identifying applications, we use their full package name. It is possible to use the wildcard value `any` to define generic policies. For instance, using `any` as the `Requester` of a policy would allow a security administrator to specifying a system-wide policy that applies to any application. Similarly, the `any` value can be used in a policy as an operation and/or target.

One of the issues of a policy-based system is defining security policies that are robust enough to protect the system without compromising its functionality. Of course, there is no bullet-proof solution to this problem. As we have learned from years of using network firewalls, the set of policies used depends on the needs of the corporate network. The policy language to be effective has to provide the required constructs to satisfy those needs. The rest is up to the security administrator's knowledge and expertise.

Having said that, FireDroid can be used as a profiling tool to analyse the system calls and parameters applications execute during runtime. This information is then used for defining policies to either prevent or contain the undesired behaviour of an application. To this end, in the following section we present a compact set of few but effective policies created as a results of our analysis based on the 1260 malware samples provided by the Malgenome Project [12]. At the same time, these policies are generic enough to be applied to any application to protect the user's privacy and resources.

6. SECURITY VALIDATION

Given the recent increase in malware samples found in the wild aimed at the Android OS [5], it is essential to provide a robust solution against such threats. As discussed by Zhou and Jiang in [51], available anti-virus solutions for Android are only able to detect at most 79% of the samples collected untill October 2011. In more a recent study, it has been shown that today's anti-virus software is still vulnerable to simple evasion techniques [43]. The main reason anti-virus software fails in detecting even known malware samples once they have been slightly modified is related to the signature-based approach adopted.

FireDroid identifies at runtime if an application is executing illicit or potentially dangerous actions by intercepting the system calls the application executes. No matter if the malware is new or a repackaged version of an existing one: when the malware executes dangerous system calls, FireDroid can detect and enforce the appropriate security policies.

To prove the effectiveness of FireDroid, we stress-tested FireDroid against the malware samples in our collection. We have manually executed samples using a device where FireDroid was deployed. During this phase, an operator inspected the system calls the malware executed without enforcing any policies. Because FireDroid does not modify neither the application code nor the Android OS, its monitoring mechanism is an ideal inspection tool to fool even the more advanced malware samples (such as AnserverBot) that try to detect if they have been tampered with. When the attack was executed, the operator flagged the system call logs for a later inspection. In a second phase, we inspected the system call logs and created a set of policies to either stop or limit the offensive system calls. Finally, we deployed the policies in the device and executed the malware samples until the attack was performed and the respective policy enforced. However, before executing this last step the device was flashed and a new image installed to remove any side effect caused by the malware. In the following, we will provide details on the policies defined for each type of attacks.

6.1 Financial Charges

A large portion of the malware samples is responsible for causing financial charges to the user. The main method used is to subscribe the user to premium-rate SMS services by sending SMS messages with special codes. The malware requires the special permission `sendTextMessage` to be able to send SMS messages to any numbers without notifying the user. To prevent this type of attacks, we have used the policy shown in Figure 3. This policy performs several checks. For instance, it is possible to constrain the number of SMS messages applications can send. The policy checks whether the SMS message limit has been reached in which case it will ask the user. The policy language allows the declaration of variables that can also be associated with a specific identifier. For instance, the `numOfSentSMS` is an integer associated with the `App` identifier to limit its scope to this specific policy.

The policy also checks whether the destination number is not in the contact list: if this is the case, the user will be notified. The reference to the content provider containing the list of contacts is obtained through the `getContact()` API. Premium SMS services have special short codes (usually less than 7 digits). The policy checks whether the destination number matches this rule and in case notifies the user. Incidentally, Android JB also uses a similar approach to notify the user when an application is sending an SMS to a premium number. Finally, if the user allows the operation to be performed then the policy updates the number of sent SMS.

```
1 SMSLimit = 10
2 App->numOfSentSMS = 0
3 contact = getContact()
4 App sendText [destNum] on iSMS
5   if (App.numOfSentSMS > SMSLimit) then ask
6   if (!contact.contains(destNum)) then ask
7   if (destNum.length <=7) then ask
8   if (ask.outcome == allow) do App.numOfSentSMS++
```

Figure 3: Policy to control background SMS sending.

Some premium services require the user to send a second confirmation SMS to complete the subscription process. Some malware samples (such as `RougeLemon`) deal with this extra check by requesting the `receiveSMS` permission that allows them to get any received SMS messages. The malware registers to the `SMS_RECEIVED` event with the highest priority either programmatically through the `ActivityManager` or using the manifest file (in which case the `PackageManager` will be used). If the malware registers with the highest priority then it will be able to hijack the notification and the user will never know that an SMS has been received. To prevent this, we have defined the policy shown in Figure 4. The policy checks if the application is registering for a `SMS_RECEIVED` event with the highest priority. If this is the case, then it will reduce the priority to the lowest value and allow to complete the operation. In this way, the malware is not able to steal the SMS notification and at the least the user will be able to see notification of the SMS arrival.

```
1 App|PackageManager registerReceiver [intent,priority]  on
      ActivityManger
2   if (intent == SMS_RECEIVED) && (priority == HIGHEST)
3      then allow do set(priority, LOWEST)
```

Figure 4: Policy to control the registration of a broadcast receiver for the SMS_RECEIVED intent with highest priority.

6.2 Information Harvesting

Several malware families actively collect information from the infected devices and upload them on their C&C servers. `BeanBot`,

CoinPirate, DroidCoupon, and NickyBot are malware families that retrieve information such as IMEI (unique device id), IMSI (unique subscribed number), and the phone number. To access this information, the malware uses the IPhoneSubInfo service. To prevent the propagation of information without the user being aware, we have defined the policy shown in Figure 5. This policy denies access to the original information and provides the requesting application fake information. In any event, we also notify the user to make her aware of the application requests.

```
App get [code] on iPhoneSubInfo
  if (code == IMEI) then allow do replace(fakeIMEI) and
       notifyUser(imeiMessage)
  if (code == IMEI_SV) then allow do replace(fakeIMEI_SV)
        and notifyUser(imeisvMessage)
  if (code == IMSI) then allow do replace(fakeIMSI) and
       notifyUser(imsiMessage)
  if (code == ICC) then allow do replace(fakeICC) and
       notifyUser(iccMessage)
  if (code == PHONE_NUMBER) then ask
```

Figure 5: Policy to control access to the iPhoneSubInfo service.

Malware also tries to collect other information stored in the device such as call logs and stored SMS messages. For instance, Zitmo and Spitmo (the Android versions of Zeus and SpyEye) attempt to retrieve the SMS verification messages to initiate fraudulent transactions on the user's account. The policy in Figure 6 requests the user's permission to allow an application to access the call logs and deny access to the stored SMS message notifying the user of the application's attempt.

```
App query on ContentProvider
  if (call_log/calls) then ask
  if (sms/inbox || sms/sent) then deny notifyUser(
       storedsmsMessage)
```

Figure 6: Policy to control access to content providers.

6.3 Vulnerabilities

A great proportion of the malware samples embed one or more rootkits to exploit vulnerabilities present in the Linux kernel. For instance, the DroidKungFu family uses RATC and exploid root exploits for getting root privileges on the device. RATC works by forking new processes to reach the limit of allowed user processes. Then it kills the adbd daemon. When the adbd daemon is restarted as a root process it will fail to downgrade to a user level process and it can be used to execute processes with root privileges. exploid uses a NETLINK message to create a user-controlled copy of the init process, thus gaining root access. In both cases, FireDroid is able to identify the sequence of system calls that are performed by the root exploits. For instance, by counting the number of fork a process executes we can detect when the limit of user processes is reached. This would give us an indication that the attack is being performed. To contrast the attack, we could limit the number of fork a process can execute to avoid reaching the limit of user processes imposed by the OS. For exploid, we can specify a policy that does not allow a user process to open a socket to NETLINK where the protocol option is set to NETLINK_KOBJECT_UEVENT. This option should only be used when the kernel sends a message to a user process. Both of these policies are shown in Figure 7.

A very recent Linux kernel vulnerability [13] (affecting version 2.6.37 to 3.8.9) has been discovered that allows a process to gain

```
numOfForked = 0
delta = 10
App fork on System
  if (numOfForked < userProcLimit() - delta) then deny
App socket [domain] on System
  if (domain == PF_NETLINK) then deny
```

Figure 7: Policy to control the number of processes forked by a process and the the opening of a socket to NETLINK.

root privileges in Linux kernels with the PERF_EVENTS option enabled. This vulnerability affects even kernels with SELinux enabled. The exploit uses the perf_event_open system call to force the kernel to transfer execution to a known user-process address where malicious code will elevate the process credentials to execute a shell as root. The main problem is that the perf_event_open uses as an argument a struct where a 64-bit unsigned integer is eventually casted into a 32-bit signed integer. Any negative value will be used by the kernel as an offset for a memory address that a user program can control.

Although as of June 2013, there is no porting of the this exploit on ARM architecture, the most recent Android versions (JB and ICS) might be affected. For instance, the PERF_EVENTS compilation option is enabled in kernels included in several devices such as the Nexus 7, Nexus 4 and Samsung Galaxy S4. Because this vulnerability is able to bypass SELinux, this means that also SE-Android will be affected [47]. Interestingly enough, there is no function wrapper for this function in libc. Therefore the function has to be called through the syscall system call or inlined assembly. This means that an approach such as Aurasium that uses GOT injection is not able to intercept the execution of this exploit.

With FireDroid, we can block the attack with the policy shown in Figure 8 that checks the value of the config field in the struct passed as an argument of the system call.

```
App perf_even_open [attr] on System
  if (attr.config < 0) then deny
```

Figure 8: Policy to stop an application to gain root privileges using the `perf_even_open` vulnerability.

Vendor	Model	Android Version
HTC	One X	4.0.3
Samsung	Galaxy Note 10.1	4.0.4
	Galaxy Note 2	4.1.1
	Galaxy S4	4.2.2
	Galaxy S3	4.0.4
	Galaxy Nexus	4.1.2
	Galaxy S2	2.3.3
Google	Nexus 4	4.2.1
	Nexus 7	4.2.1
Sony-Ericson	Xperia Arc S	4.0.4

Table 1: Devices from different vendors and Android versions where FireDroid has been tested.

7. PERFORMANCE EVALUATION

In this section, we discuss our experimental findings on FireDroid overhead.

To check whether the deployment of FireDroid on a device is compatible with the Android standards set by Google, we have used

the Compatibility Test Suite (CTS) [14]. Google has designed these tests to provide vendors with a benchmark to test the compatibility of their devices. The CTS checks whether a given hardware/software configuration is compatible with Google's standard policies [15]. In particular, the performance tests check the performance of the system against defined benchmarks, such as rendering frames per second or the time required for a browser to start up. To run the CTS tests, we have used several Android devices with FireDroid deployed listed in Table 1.

To measure the overhead introduced by FireDroid, we used two popular benchmark applications from Google Play to represent the execution of general-purpose operations. Following, we synthesised an application performing invocations of the Android API to assess FireDroid's impact on Android-specific activities. The experiments were executed using a HTC One X [2] running Android 4.0.3 (Ice Cream Sandwich) on a Linux 2.6.39.4 kernel. The results of the experiments presented in the following are averaged over five different executions.

Benchmark	Test	Reference	FireDroid	Overhead
Quadrant	Total	4617	4104.6	12.5%
	CPU	12638	12013	5.2%
	Memory	3078	2960.4	3.9%
	I/O	4331	2193.2	97.5%
	2D	984.4	982	0.2%
	3D	2422.6	2344.8	3.3%
BenchmarkPi	CPU	349	351.1	0.59%

Table 2: Measurements of FireDroid overhead using two benchmark applications.

The first set of experiments is aimed at assessing the overhead FireDroid introduces when computationally-intensive applications are executed (i.e., gaming applications). We have performed several measurements using two benchmark applications: the Quadrant benchmark [3] and the BenchmarkPi application [4]. The former benchmark is a general-purpose benchmark for CPU, memory, I/O, 2D and 3D graphics testing. The latter is mainly a CPU-intensive application that approximates the numerical value of π. The results are provided in Table 2. The Quadrant benchmark generates points (pts) for each of the tests (higher values indicate better performance) while the BenchmarkPi measures the time in milliseconds (ms) to complete the computation (the lower the value the better the performance).

Firstly, we measured the scores of the benchmarks when FireDroid is not deployed on the device. The reference values for both benchmarks are provided in the Reference column in Table 2. Following, we deployed FireDroid on the device and executed again the same experiments. This time the benchmark applications have each a FDAM attached to them monitoring all the performed system calls. From the results, we can see the overhead introduced by FireDroid is very low for the CPU, memory, 2D and 3D tests. Only in the I/O test, FireDroid introduces a very high overhead of 97.5%. This is explained by the fact that when files are opened, FireDroid normalises the file path which requires extra time. Also, the use of `ptrace` introduces a high overhead given the context switches necessary for passing the control to the FDAM. We are aware of this issue and we are working on a new version of FireDroid that does not use `ptrace` (see Section 9 for more details). However, looking at the Quadrant total score, the overhead introduced by FireDroid is 12.5% which we belive is acceptable. The results for the BenchmarkPi test shows that the CPU overhead is negligible (only 0.59%).

The next set of experiments focuses on evaluating the performance penalty introduced by FireDroid when applications interact with each other and access the device's resources through the Android API. In particular, we have created a test application that performs the following operations: `HttpGet` to simulate web browsing, `BroadcastIntent` to send an implicit intent, `QueryContact` to access the phone contact list, and `GetLastLocation` to access the location information provided by the device's GPS. The FDAM attached to the test application has a set of FireDroid policies enabled that allow the execution of the operations. However, for each operation the FDAM has to evaluate the policies to decide whether to authorise the operation execution.

Each test run consists in measuring the execution time of 200 operations first with FireDroid disabled and then with FireDroid enabled. Table 3 provides the results of the execution of each operation in milliseconds (ms). In more details, for the `HttpGet` operation the overhead is very negligible. In this case, the device was connected to the Internet using a WiFi connection and performing an `HttpGet` to `http://www.google.com` . We measured that 95% of the total execution time is due to network latency to retrieve the data from the remote server.

The rest of operations executed by the application require IPC with other processes through the Binder. This represents the worst case, since the FDAM has to extract information from the `ioctl` system call with the Binder to be able to evaluate the policies.

The `BroadcastIntent` sends an implicit intent that is captured by a service component through the `sendBroadcast(intent)` method. When this operation is executed under FireDroid the running time increases by 5%. The `QueryContact` is performed using the `ContentResolver` interface to query the id, displayed name, and phone number of each contact. Each time this operation is performed, the FDAM extracts from the `ioctl` system call the authority of content provider. In this case, the overhead introduced by FireDroid is 3.9%. Finally, the `GetLastLocation` is executed by getting a handle to the system service responsible for the location by invoking the `getSystemService` method. Afterwards, the `getLastKnownLocation` method is invoked using the service's handle. When this operation is executed under FireDroid, the FDAM intercepts first the `ioctl` to request the service's handle and then the system call executed for requesting the remote execution of the service's method. Intercepting the second system call allows us to enforce filtering operations such as changing the returned value of the location with a less precise or a fake one. For performing this fine-grained level of control the overhead introduced by FireDroid is 29.9%.

Operation	Reference	FireDroid	Overhead
`HttpGet`	553	558.77	1.03%
`BroadcastIntent`	0.443	0.445	5%
`QueryContact`	10.4	10.8	3.9%
`GetLastLocation`	0.769	0.999	29.9%

Table 3: Overhead measurements when executing the API under FireDroid. Values represent execution time in milliseconds per each operation.

To measure the overhead introduced by FireDroid on battery utilisation, we have executed the following test ten times and averaged the results. With a fully charged battery, we measured how long the battery lasts when we let our test application perform every 10 seconds all the operations until the battery is completely depleted and

[2]Quad-core ARMv7 Cortex-A9 (1.5 GHz) CPU, Nvidia ULP GeForce GPU and 1GB RAM

[3]version 2.1.1

[4]version 1.11

the device turns off. During the test, the screen and the WiFi were always on. When FireDroid was not enabled the battery lasted for 496 minutes in average. When FireDroid was enabled, the battery lasted for 480 minutes in average (3.3% shorter than previous case), with a 95% confidence interval of (479,481).

Concluding, when applications performs API invocations FireDroid is able to enforce fine-grained security policies while introducing an acceptable performance overhead. This is further confirmed by the results obtained when executing the Google CTS tests. The worst case scenario is for I/O intensive applications as shown in the Quadrant benchmark. However, we can argue that I/O intensive behaviour is not typical for mobile applications. FireDroid's impact on battery utilisation is negligible. Overall, our findings show that FireDroid's overhead is not prohibitive and in most typical scenarios, such as web browsing and CPU intensive applications, not noticeable to the user.

8. RELATED WORK

In this section, we provide an overview of the related work aimed at enhancing the security mechanism of the Android framework. The research activities have focused on several aspects of Android.

Two approaches have studied malware found in the wild. Felt et al. [33] analysed a set of 46 samples for several smartphone platforms. A much larger set of Android malware samples have been extensively studied in the MalGenome project [51]. These research efforts are very useful in providing information for contrasting malware. For detecting malware other approaches have been proposed. DroidMOSS [50] is a tool for detecting malware in application markets using the signature of known malware samples. DroidRanger [52] is designed for identifying repackaged applications. Finally, RiskRanker [35] aims at detecting unknown malware in application markets without relying on known samples. These approaches have their merits in keeping the markets clean from malicious applications. However, they are incapable of restraining malware on devices that have been already exposed. Another line of research has been concerned with ranking the risk associated with applications that require more permissions than necessary to perform their functionality. In this respect, Enck et al. [29] have studied the permissions of the top 1100 free applications in Google Play to understand their security characteristics. Systems like Kirin [30] can be used to stop the installation of applications requesting a dangerous combination of permissions. Stowaway [32] and the approach proposed in [41] can be used to detect if applications are overprivileged. However, the smishing attack shows that even an application without any permission can perform malicious operations. Compared to FireDroid, these approaches have a different goal: to prevent that malicious applications get on the device. FireDroid can use the information these approaches provide for extending its capabilities to block malware at run-time.

Another security problem in Android is that of the confused deputy attack where benign applications unrestrictedly expose dangerous functionalities. Tools such as ComDroid [23] and Woodpecker [36] have been proposed for detecting such capability leaks only in third-party and pre-installed applications. These approaches are complementary to FireDroid. A security administrator can use these approaches for detecting vulnerabilities and then specify FireDroid policies to prevent their exploitation.

Given that Android source code is available, several approaches have been proposed to modify Android code for extending and/or refining its reference monitor. An extension to the Android IPC stack has been proposed in [27] to address the confused-deputy attack. TaintDroid [28] employs dynamic taint analysis to control data flows. TaintDroid tracks the source and destination of specific tainted data to avoid that sensitive information is leaked from the device. MockDroid [20] and TISSA [53] control how applications retrieve user's private information and applying filtering mechanisms. AppFence [37] uses TaintDroid's tainting mechanism to filter sensitive information and to block unauthorised leakage of data via network access. Apex [40] and Crepe [24] use contextual information for triggering security policies. To detect escalation attacks performed by colluding applications, Bugiel et al. [22] have proposed a run-time monitor for detecting such a malicious behaviour that could be not noticed by the user. For supporting the BYOD policy in an enterprise environment, virtualisation approaches have been proposed. In particular, TrustDroid [21] and MOSES [45] support lightweight virtualisation where corporate applications and data are confined in a security domain not accessible from any other applications the user has installed in her device. A very important limitation of these approaches is that native code can easily circumvent them. In full virtualisation such as proposed in [39, 18] several instances of the same OS are run on the same device. Similarly to FireDroid, all these approaches require rooting of the devices to have their image deployed. In contrast with FireDroid, they require extensive changes of the Android code making them impractical for a wide-scale deployment. Another difference with FireDroid is that these approaches are not capable to deal with threats posed by malicious native code.

There are extensions of the Linux kernel to make Android more secure. These include TOMOYO [16] (used in TrustDroid) and more recently the port of SE Linux to Android [47]. Similarly to the above approaches, they require extensive modifications of the Android OS. Another limitation of SEAndroid is that it supports only MAC policies at install time. Any changes to the MAC model would require extensive changes on the device installation. Finally, the MAC model of SEAndroid although very powerful is not able to prevent the `perf_even_open` attack unless a dedicated fix is released. With FireDroid, just one simple policy can prevent the attack from being performed.

Recently, new approaches have appeared where no modifications of the Android OS are required. These approaches have in common that they modify the bytecode of the installed applications to inject code for monitoring the application behaviour. Thus, these approaches can operate on unrooted devices. To inject the monitoring code, the Android bytecode disassemblers are used to manipulate the dex files of the target applications. This injected code redirects the execution flow of the application to monitoring code that enforces security policies. AppGuard [19], I-ARM-Doid [26], and DrDroid [44, 38] inject control code around dangerous Java method invocations. However, malware found in the wild may use native code to perform its malicious actions [51]. Therefore, these approaches are not capable of stopping native code to perform such attacks. The only application hardening approach that is able to partially cope with native code is Aurasium [49]. Aurasium performs library function call interposition by dynamically changing the entries of the GOT of a process with pointers to Aurasium-monitored functions. In this way, Aurasium is able to detect the `dlopen()` system call used for replacing the current process with a new image loaded from native code. However, when the application starts executing native code it can break Aurasium's sandbox quite easily. For instance, the native code can have system call libraries (such as `libc`) statically linked. Another approach to escape Aurasium's sandbox is to use in the native code inlined assembly code to call directly system calls. On the other hand, FireDroid is more robust and does not suffer from these issues. In both the above cases, when the system calls are executed the FDAM will be able to detect them and enforce the required policies. Although

these approaches are ideally suited for the security-minded users that want to protect their devices (provided that the users know how to specify the right security policies), they are less suited for an enterprise deployment. In fact, these approaches rely on the user to perform the application hardening process and to execute exclusively the hardened version of the application. Instead, FireDroid approach gives an enterprise complete control over a device since it does not rely on the user and is able to control third-party and pre-installed applications.

9. FUTURE WORK

Our future work is mainly focused in reducing the overhead introduced by FireDroid. Currently, the main penalty is generated by the use of `ptrace()`. Each time a system call is intercepted, a context switch is required for passing the control to the monitoring process to perform its security checks. Moreover, the monitoring process needs to extract the arguments of the system calls to evaluate security policies. We are currently working on an extension of FireDroid where `ptrace()` is used to attach to Zygote and to modify its Global Offset Table (GOT) to point to the FireDroid enforcement point. When a process is spawned from Zygote it will get the modified GOT. This means that for every function call invoked by the process, FireDroid is able to capture the function arguments and to enforce security policies without doing any context switch as required for `ptrace()`. Our preliminary results show that we can achieve a speed-up of 80x.

GOT indirection can be bypassed when the process executes a `dlopen()` to load native code. In FireDroid, we are taking care of this by allowing the FDAM to switch to `ptrace` when a `dlopen()` is executed. In this way, even if there is a loss in performance we can still control the execution of the process. We believe that this is more advantageous compared to Aurasium where the user has to choose between continuing the execution of the native code without any security guarantees or killing the application.

Given the low-level at which FireDroid operates, it is possible to extend the monitoring facilities to support intrusion and anomaly detection techniques. Finally, the policies presented in this paper focus on specific threats coming from the analysed malware samples. We are planning to introduce high-level policies to support dynamic analysis such as information flow and taint analysis.

We have deployed FireDroid on devices we use in our daily activities. In our experience, FireDroid does not introduce a noticeable performance loss. However, we are planning to perform more extensive usability tests with a larger set of users.

10. CONCLUSIONS

In this paper, we have presented FireDroid a system call interposition mechanism to enforce fine-grained security policies for Android. The main advantage of FireDroid is that it does not require recompilation of any internal modules of the Android OS. FireDroid provides a robust enforcement mechanism to protect an Android device from malware. We have tested its effectiveness against a large sample of real malware found in the wild and have defined a set of policies to protect the devices from the malware attacks. FireDroid can be used as a quick-fix against vulnerabilities without the need to wait for patches. We have also provided an example of a how to protect a device from a very serious vulnerability of the Linux kernel. We have deployed FireDroid on several devices and performed a quantitative analysis of its performance overhead. Our results show that FireDroid overhead is measurable but to an acceptable level with respect to Google compatibility tests.

11. ACKNOWLEDGEMENTS

We would like to thank Daniel Bertinshaw, Xiao Bao Clark, Daniel Lewis, and Kris Pritchard for their contributions to the FireDroid project. Also, we are grateful for the financial support for realising FireDroid provided by Auckland UniServices Limited. Finally, we want to express our appreciation to Dr. William Enck and the anonymous reviewers for their invaluable suggestions.

[1] `http://www.gartner.com/it/page.jsp?id=2227215`.
[2] `http://www.gartner.com/newsroom/id/2482816`.
[3] `http://news.cnet.com/8301-1035_3-57545513-94/five-years-of-android-by-the-numbers/`.
[4] `http://www.appbrain.com/stats/number-of-android-apps`.
[5] `http://www.ibtimes.co.uk/articles/401395/20121105/android-malware-increae-ten-fold.htm`.
[6] `http://www.computerworld.com/s/article/9231758/USSD_attack_hit_SIM_cards_and_Samsung_Android_devices`.
[7] `http://theunderstatement.com/post/11982112928/android-orphans-visualizing-a-sad_history-of-support`.
[8] `http://www.angryredplanet.com/~hackbod/openbinder/docs/html/BinderIPCMechanism.html`.
[9] `http://www.citi.umich.edu/u/provos/systrace/index.html`.
[10] `http://en.wikipedia.org/wiki/Magnuson%E2%80%93Moss_Warranty_Act`.
[11] `http://eur-lex.europa.eu/Notice.do?val=330258:cs&lang=en&list=340508:cs,330258:cs,&pos=2&page=1&nbl=2&pgs=10&hwords=&checktexte=checkbox&visu=#texte`.
[12] `http://www.malgenomeproject.org/`.
[13] `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-2094`.
[14] `http://source.android.com/compatibility/overview.html`.
[15] `http://static.googleusercontent.com/external_content/untrusted_dlcp/source.android.com/en//compatibility/4.2/android-4.2-cdd.pdf`.
[16] `http://tomoyo.sourceforge.jp/`.
[17] Android Project. `http://www.android.com`.
[18] ANDRUS, J., DALL, C., AND ET AL. Cells: a virtual mobile smartphone architecture. In *Proc. of the 23th ACM Symposium on OS Principles* (New York, NY, USA, 2011), ACM, pp. 173–187.
[19] BACKES, M., GERLING, S., AND ET AL. Appguard âĂŤ real-time policy enforcement for third-party applications. Tech. Rep. A/02/2012, Saarland Uni., Germany, 2012.
[20] BERESFORD, A. R., RICE, A., AND SKEHIN, N. MockDroid: trading privacy for application functionality on smartphones. In *Proc. HotMobile '11* (2011).
[21] BUGIEL, S., DAVI, L., AND ET AL. Practical and lightweight domain isolation on android. In *Proceedings of the 1st ACM*

workshop on Security and privacy in smartphones and mobile devices (2011), pp. 51–62.

[22] BUGIEL, S., DAVI, L., AND ET AL. Towards taming privilege-escalation attacks on Android. In *Proc. of the 19th Annual Network & Distributed System Security Symposium* (Feb. 2012).

[23] CHIN, E., FELT, A. P., AND ET AL. Analyzing inter-application communication in android. In *Proc. of the 9th international conf. on Mobile systems, applications, and services* (New York, NY, USA, 2011), ACM, pp. 239–252.

[24] CONTI, M., NGUYEN, V. T. N., AND CRISPO, B. Crepe: context-related policy enforcement for android. In *Proc. of the 13th international conf. on Information security* (Berlin, Heidelberg, 2011), Springer-Verlag, pp. 331–345.

[25] DAVI, L., DMITRIENKO, A., AND ET AL. Privilege escalation attacks on android. In *Proc. of the 13th international conf. on Information security* (2011), pp. 346–360.

[26] DAVIS, B., SANDERS, B., AND ET AL. I-arm-droid: A rewriting framework for in-app reference monitors for android applications. *IEEE Mobile Security Technologies, San Francisco, CA* (2012).

[27] DIETZ, M., SHEKHAR, S., AND ET AL. Quire: Lightweight provenance for smart phone operating systems. In *20th USENIX Security Symposium* (2011).

[28] ENCK, W., GILBERT, P., AND ET AL. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of OSDI 2010* (Oct. 2010).

[29] ENCK, W., OCTEAU, D., AND ET AL. A study of android application security. In *Proc. of the 20th USENIX conf. on Security* (San Francisco, CA, 2011), pp. 21–21.

[30] ENCK, W., ONGTANG, M., AND MCDANIEL, P. On lightweight mobile phone application certification. In *Proc. CCS '09* (2009), pp. 235–245.

[31] ENCK, W., ONGTANG, M., AND MCDANIEL, P. Understanding android security. *IEEE Security and Privacy 7*, 1 (Jan. 2009), 50–57.

[32] FELT, A. P., CHIN, E., AND ET AL. Android permissions demystified. In *Proc. of the 18th ACM conf. on Computer and communications security* (New York, NY, USA, 2011), ACM, pp. 627–638.

[33] FELT, A. P., FINIFTER, M., AND ET AL. A survey of mobile malware in the wild. In *Proc. of the 1st ACM workshop on Security and privacy in smartphones and mobile devices* (New York, NY, USA, 2011), ACM, pp. 3–14.

[34] GARFINKEL, T. Traps and pitfalls: Practical problems in system call interposition based security tools. In *In Proc. Network and Distributed Systems Security Symposium* (2003), pp. 163–176.

[35] GRACE, M., ZHOU, Y., AND ET AL. Riskranker: scalable and accurate zero-day android malware detection. In *Proc. of the 10th international conf. on Mobile systems, applications, and services* (New York, NY, USA, 2012), ACM, pp. 281–294.

[36] GRACE, M., ZHOU, Y., AND ET AL. Systematic detection of capability leaks in stock Android smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)* (Feb. 2012).

[37] HORNYACK, P., HAN, S., AND ET AL. These aren't the droids you're looking for": Retroffiting android to protect data from imperious applications. In *18th ACM Conf. on Computer and Communications Security (CCS'11)* (2011).

[38] JEON, J., MICINSKI, K. K., AND ET AL. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proc. of the 2nd ACM workshop on Security and privacy in smartphones and mobile devices* (New York, NY, USA, 2012), ACM, pp. 3–14.

[39] LANGE, M., LIEBERGELD, S., AND ET AL. L4android: a generic operating system framework for secure smartphones. In *Proc. of the 1st ACM workshop on Security and privacy in smartphones and mobile devices* (Chicago, USA, 2011), ACM, pp. 39–50.

[40] NAUMAN, M., KHAN, S., AND ZHANG, X. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proc. ASIACCS '10* (2010), pp. 328–332.

[41] PENG, H., GATES, C. S., AND ET AL. Using probabilistic generative models for ranking risks of android apps. In *the ACM Conf. on Computer and Communications Security, CCS'12, Raleigh, NC, USA, Oct. 16-18, 2012* (2012), pp. 241–252.

[42] PROVOS, N. Improving host security with system call policies. In *Proc. of the 12th conf. on USENIX Security Symposium* (Washington, DC, 2003), vol. 12, pp. 18–18.

[43] RASTOGI, V., CHEN, Y., AND JIANG, X. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security* (New York, NY, USA, 2013), ASIA CCS '13, ACM, pp. 329–334.

[44] REDDY, N., JEON, J., AND ET AL. Application-centric security policies on unmodified Android. Tech. Rep. UCLA TR 110017, University of California, LA, Computer Science Department, July 2011.

[45] RUSSELLO, G., CONTI, M., AND ET AL. Moses: supporting operation modes on smartphones. In *SACMAT* (2012), V. Atluri, J. Vaidya, and et al., Eds., ACM, pp. 3–12.

[46] SHABTAI, A., FLEDEL, Y., AND ET AL. Google android: A comprehensive security assessment. *IEEE Security and Privacy 8* (2010), 35–44.

[47] SMALLEY, S., AND CRAIG, R. Security enhanced (se) android: Bringing flexible mac to android. In *Proc. of the 20th Network and Distributed System Security Symposium (NDSS 2013)* (2013).

[48] WAGNER, D. A. Janus: an approach for confinement of untrusted applications. Tech. Rep. UCB/CSD-99-1056, EECS Department, University of California, 1999.

[49] XU, R., AND ANDERSON, H. S. R. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st conf. on USENIX Security Symposium* (2012), vol. 21.

[50] ZHOU, W., ZHOU, Y., AND ET AL. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proc. of the 2nd ACM conf. on Data and Application Security and Privacy* (New York, NY, USA, 2012), ACM, pp. 317–326.

[51] ZHOU, Y., AND JIANG, X. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy* (2012), pp. 95–109.

[52] ZHOU, Y., WANG, Z., AND ET AL. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proc. of the 19th Annual Network & Distributed System Security Symposium* (Feb. 2012).

[53] ZHOU, Y., ZHANG, X., AND ET AL. Taming Information-Stealing Smartphone Applications (on Android). In *Proc. TRUST 2011* (2011).