# DeepDroid: Dynamically Enforcing Enterprise Policy on Android Devices

Xueqiang Wang*†, Kun Sun‡, Yuewu Wang* and Jiwu Jing*
* Data Assurance and Communication Security Research Center,
Institute of Information Engineering, Chinese Academy of Sciences
{wangxueqiang, ywwang, jing}@is.ac.cn
†University of Chinese Academy of Sciences
‡Department of Computer Science, College of William and Mary
ksun@wm.edu

***Abstract*—It is becoming a global trend for company employees equipped with mobile devices to access company's assets. Besides enterprise apps, lots of personal apps from various untrusted app stores may also be installed on those devices. To secure the business environment, policy enforcement on what, how, and when certain apps can access system resources is required by enterprise IT. However, Android, the largest mobile platform with a market share of 81.9%, provides very restricted interfaces for enterprise policy enforcement. In this paper, we present *DeepDroid*, a dynamic enterprise security policy enforcement scheme on Android devices. Different from existing approaches, DeepDroid is implemented by dynamic memory instrumentation of a small number of critical system processes without any firmware modification. DeepDroid can be easily deployed on various smartphone platforms with a wide range of Android versions. Moreover, based on the context information extracted from Binder interception, a fine-grained policy can be enforced. We develop a prototype of DeepDroid and test it on various smartphones and Android versions. The experimental results show that DeepDroid can effectively enforce enterprise resource access policies with negligible performance overhead.**

## I. Introduction

Nowadays, an increasing number of employees are allowed to use mobile devices in workplace and connect to enterprise assets. Cisco surveyed that 51% of end users rely on smartphones to perform their daily business activities in 2013 [1]. This trend will continue to influence the design and usage of mobile devices in the enterprise environments. While users are blurring the lines between company and personal usage, enterprises demand a secure and robust mobile device management to protect their business assets. For instance, in a building that forbids any audio recording, all mobile devices' microphones should be disabled when the users check in the building and be enabled when the users check out.

The permission model on Android, the largest mobile platform with a market share of 81% [2], only grants an "all-or-nothing" installation option for mobile users to either accept all the permissions an app asks for or simply decline to install the app. After installation, the app can keep accessing the approved system resources all the time. In Android 4.3, an experimental feature called "App Ops" [3] is added to permit mobile users to configure one app's runtime permissions. However, this feature has been removed from Android 4.4.2 due to "the increasing burden for user configuration and the impacts on advertisement market" [4]. SEAndroid has evolved from Permissive mode in Android 4.2 and 4.3 to Enforcing mode in Android 4.4 and later to provide flexible mandatory access control (MAC) mechanism in the Linux kernel. However, until now, even the newest Android 5.0 has not fully integrated MAC mechanism in Android middleware [5]. Moreover, SEAndroid is not available on legacy systems running old versions of Android.

Since Android 2.2, Google provides Device Administration APIs [6] to help enforce enterprise security policies; however, these APIs only provide a limited set of functionalities that vary among different Android releases. The Mobile Device Management concept (MDM) [7] has also been introduced to enterprise administrators for a long time; however, OEMs usually develop their


NDSS '15, 8-11 February 2015, San Diego, CA, USA
 ISBN 1-891562-38-X
http://dx.doi.org/10.14722/ndss.2015.23263

own proprietary MDM solutions [8], [9]. For instance, Samsung Knox provides a complete enterprise solution, including secure boot, kernel integrity checking, and SEAndroid [9]; however, it is only available on Samsung devices [10].

In this paper, we propose an enterprise-level security policy enforcement mechanism called DeepDroid that can be easily ported on various Android devices to dynamically enforce a fine-grained system service and resource access control policy by enterprise administrators. The basic idea is to apply dynamic memory instrumentation on the app runtime environment in Android. All current versions of Android share a common structure feature that system services and resource access are controlled by a small number of system processes. Thus, we only need to instrument these system processes whose structures are almost the same on all Android versions. DeepDroid dynamically hooks *system_server* process in Android and uses it as a centralized controller to enforce the enterprise-level permissions when an app requests to access a system service. It also tracks the *zygote* process to authorize native code's access requests. Moreover, DeepDroid can intercept the Binder interactions between apps and a few system processes to retrieve details of apps' requests for a fine-grained access control.

DeepDroid needs *root* privilege to instrument and track system processes. It can be satisfied in a corporate environment where companies usually either rent or purchase mobile devices from telecommunication companies and request vendors to customize the software image before distributing the devices to employees. Since DeepDroid does not need to statically change the Android middleware and the Linux kernel, it carries little burden on vendors for device customization.

Our enterprise-level policy enforcement solution has several good properties. First, it is portable. Our solution can be deployed on almost all Android mobile devices with very small customization efforts. Instead of tailoring various Android systems from different OEMs, we use dynamic instrumentation and process tracing techniques to hook the Android app runtime environment. The code base of our system remains largely unchanged when deployed on different Android versions. Our solution only requires minimal configuration changes in Android OS during the installation stage and does not need any changes on any app. We have evaluated our system on a variety of Android devices from different OEMs running Android 2.3 through Android 4.4.

Second, it is flexible to provide fine-grained enterprise-level control over each app. The enterprise administrators can dynamically update the policy rules for each mobile device's individual app. We cannot only enforce a general rule to constrain one specific service/resource access for an app, but also achieve a context-aware access control by inspecting the communications between apps and service processes through Android *Binder*. In this paper, we focus on providing security mechanisms instead of developing detailed policy rules given an enterprise's security requirements.

Finally, it has minimal impacts on user experience. DeepDroid is transparent to mobile users and supports location-aware automatic configuration. For instance, the enterprise policy enforcement mechanism only needs to be enabled when a user enters the workplace; when the user leaves the workplace, the policy enforcement mechanism can be automatically disabled. Moreover, since our mechanism only needs to instrument a very small number of critical Android processes and perform simple checkings, it has negligible performance overhead.

In summary, we make the following contributions.

- We solve the portability problems of enforcing enterprise's security policies on various Android devices. Our approach is based on dynamic code instrumentation and process tracing, which enforce access control policies in Android middleware and Linux kernel, respectively. Our system can be ported on different Android mobile devices with small changes.
- We can achieve a fine-grained control policy over each Android app. By hooking and tracing critical Android components, we can extract request details to better regulate one app's operations. Thus, it enables enterprise administrators to set fine-grained policy rules considering temporal and spatial constraints for each app.
- We minimize the impacts of our mechanism on Android system. Only a few critical processes (e.g., system_server) need to be dynamically instrumented in their memory spaces, and the performance overhead is minimal. Also, its impacts on Android system is temporary and can be easily removed when the policies are no longer desired. It is compatible with the existing permission mechanism.
- We develop a DeepDroid prototype and evaluate its effectiveness and efficiency on a number of mainstream Android devices with various Android versions. The evaluation results show that DeepDroid can work

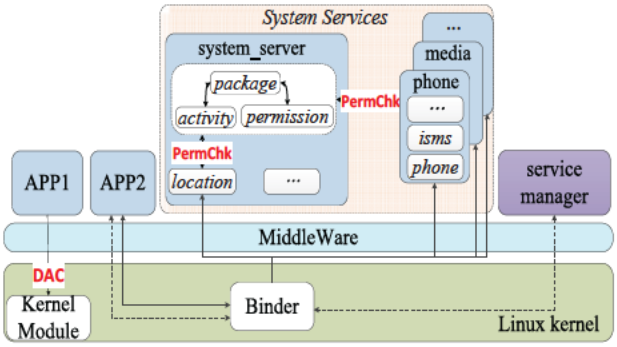


Fig. 1: Android Resource Accessing Framework

well on all tested devices and its impacts on Android system performance are negligible.

The remaining of the paper is organized as follows. Section II introduces background knowledge. Section III presents our design goals and assumptions. We present the DeepDroid framework in Section IV. A prototype is detailed in Section V. Section VI discusses the evaluation results. The extensions of DeepDroid are discussed in VII. We describe related works in Section VIII. Finally, we conclude the paper in Section IX.

## II. Background

This section provides some background on Android system services and resources that can be accessed through both Android middleware and Linux kernel. Android java code uses Binder mechanism to control the communications between apps and system services; native code may use traditional kernel system calls to access system resources.

### A. Resource Access through Android Middleware

In Android, a number of system services provide interfaces for apps to access system resources (e.g. making phone calls, querying location information). In Google's factory image of Jelly Bean, 69 services provide almost 1300 callable interfaces for peripheral apps and they dwell only in a few processes (*system_server, phone, nfc, mediaserver, drmserver, etc.*). To handle communications between apps and system services processes, Android provides Binder IPC mechanism based on OpenBinder [11]. Binder is responsible for encapsulating IPC messages and interpreting them to corresponding procedure calls. Binder's implementation remains stable across different Android versions. A user-space process (such as an app process or an Android system server process) can call the functions in a shared library *libbinder.so* to dynamically create binder threads and invoke/handle remote requests between processes. Android adopts a permission mechanism to regulate behaviors of apps. To prevent unauthorized access to system resources, 145 built-in permissions are defined to constrain apps' capabilities. All packages' granted permissions are managed in *system_server* process, which works as a permission checking center.

Figure 1 shows the control flows for apps to access Android system resources. The *location, isms* and *phone* services are system services running in separate processes and handle app's requests forwarded by the Binder. These services are all registered into *servicemanager* process during system startup. The *servicemanager* is the context manager of the Binder mechanism and serves as a name server for system services. For example, when *APP2* makes a query to *location* service, it must first obtain a service proxy from the service manager by resolving the service's name. Through this proxy, binder transactions containing the query are sent to the *location* service. After receiving the query request, the *location* service will verify that if *APP2* has been granted necessary permissions such as *ACCESS_FINE_LOCATION* by consulting the service threads, namely, *package*, *activity* and *permission* of the *system_server* process. The *location* service serves the query only if the required permissions are granted to *APP2* or otherwise rejects the query and sends an exception signal to *APP2*.

### B. Resource Access through Linux Kernel

Android system service framework only provides permission checking mechanism in the middleware layer, while app's native code may bypass it by using low-level system calls directly. Android apps may read/write normal files and manage hardware modules, such as digital camera, and the access control of these operations is enforced by Linux Discretionary Access Control (DAC), as shown in Figure 1.

In Android, each app package is regarded as a user and assigned a unique *uid* [12]. All processes that belong to one package are granted a set of permissions defined in *manifest.xml* file. 17 permissions, such as *INTERNET* or *BLUETOOTH*, are mapped to unique groups as defined in *platform.xml*. When an app process is launched, *activity* service maps granted permissions to unique group IDs. Then these group IDs are passed in as parameters to the *zygote* process, which is responsible for creating a child process for the newly launched app. The *zygote*

process ensures that supplementary groups are properly set to the app process by initiating *setgroups*. Next, the app's process will have the privilege to access resources available to those groups.

## III. Goals, Challenges and Assumptions

DeepDroid targets at effectively enforcing enterprise security policies on Android devices, and it is designed to meet the following goals:

**Portability.** Our scheme should be easily deployed to various Android versions and different Android mobile devices. Current state-of-the-art solutions add proprietary enterprise mobile management interfaces into Android source code, so the interfaces are typically customized to specific Android versions and devices. When one company wants to adopt one mobile device that has not integrated the management interfaces, the development cost will increase and the delivery time will be extended. Our goal is to support various kinds of Android devices through making minimal configuration changes.

**Fine granularity.** Our scheme should support fine-grained service and resource access control on individual apps. It cannot only supervise the permission privileges of an app, but also regulate the service provision procedure. Thus, our solution can support various enterprise policies. For instance, during working hours, SMS is only allowed between employees. In this case, we need to limit the receiver of a SMS message rather than prohibiting all SMS messages blindly. Location Based Service (LBS) apps may lead to location privacy leakage; however, blindly forbidding location related operations may generate except signals, which leak the information that the user may be at workplace. This problem can be solved by regulating each location operation and replacing some sensitive locations with fake information.

**Trustworthy.** All access control policy rules should be completely enforced, so that malicious apps cannot violate any rules. We trust the Android middleware and the low-level Linux kernel. In Android, resource access operations through system services are controlled by Android permission mechanism; however, an app may access resources using native system calls that totally bypass the permission checking. Thus, to assure all resource accesses are being supervised, our scheme must enforce the policy rules in both Android middleware and Linux kernel layers.

**Ease of use.** Our mechanism could be promptly activated and deactivated according to the enterprise policy settings. For instance, it should be quickly activated when the user enters the workplace and removed immediately when the user leaves the workplace. Moreover, its impacts on Android system performance should be minimal.

DeepDroid requires root privilege for installation. This requirement is common among enterprise mobile management solutions, and typically the root privilege can be obtained from OEMs. With OEM support, since DeepDroid does not change the framework and the kernel of Android system, its installation is simple and straightforward. We assume the enterprise administrators can be trusted. Some keying material is shared between the smartphone and enterprise policy center to generate secret keys for communication protection. We assume the keying material is well protected and secure.

We assume the Android OS kernel can be trusted. Users have the freedom to install their favorite apps on their Android smartphones. Though some apps may be malicious, we assume they cannot compromise the OS kernel or get the root privilege. A malicious or uncooperative device user may attempt to disable our policy enforcement mechanism, so we should guarantee that DeepDroid is reliably active in the workplace. DeepDroid opens a management interface for policy enforcement, which may increase the attack surface for some new attacks misusing the interface.

## IV. DeepDroid System Architecture

DeepDroid consists of two parts: *Enterprise Policy Center* and *DeepDroid On-device*, as shown in Figure 2.

### A. Enterprise Policy Center

Enterprise Policy Center consists of three modules to authenticate the mobile devices, distribute enterprise policies, and monitor the mobile devices, respectively. The authentication module is responsible for authenticating the mobile devices. After being successfully authenticated, a device can share a temporary secret key with enterprise policy center. A policy repository provides policy rules according to enterprise's security requirements and the user's role. Since all communications between the policy center and the mobile device are protected by the secret key, the policy rules can be securely distributed to mobile devices. Moreover, the status of mobile devices should be continuously reported to the policy center for logging and auditing purpose. Particularly, the policy center can use encrypted heartbeat

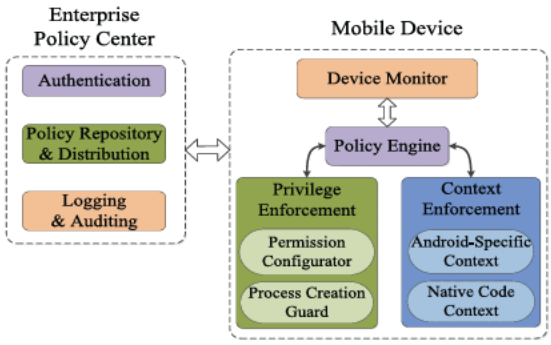


Fig. 2: DeepDroid Architecture

messages to verify that DeepDroid is correctly running on remote devices.

### B. DeepDroid On-device

On mobile device side, DeepDroid is composed of four major subcomponents: *Device Monitor*, *Privilege Enforcement*, *Context Enforcement*, and *Policy Engine*.

**Device Monitor.** The device monitor authenticates the device to the enterprise policy center and dynamically controls the activation and deactivation of the policy enforcement mechanism. It is responsible for accepting enterprise policies, reporting device status, and sending heartbeat messages to the policy center. All communications with the policy center should be protected by a shared secret key.

**Privilege Enforcement.** The privilege enforcement is in charge of authorizing access privilege to apps. It consists of two modules, *Permission Configurator* and *Process Creation Guard*, to control the app's access capabilities through Android middleware and Linux kernel, respectively. As an extension to the Android permission mechanism, the permission configurator enforces runtime permission policies by dynamically modifying the control flow of *system_server* process. Since an app may access system resources by calling Linux kernel system calls that are out of the control of Android middleware, the process creation guard is used to regulate app's native code by resetting its supplementary groups based on Linux access control mechanism.

**Context Enforcement.** Context contains detailed content (e.g. request parameters and return information) of resource accessing operations. With more detailed context information, we can provide a fine-grained resource control on smartphones. Since Android resource access operations are bounded to a series of system services in Android middleware, we monitor all running apps' operations by inspecting remote calls to these system services. Apart from monitoring resource access by Android middleware, we trace system calls to regulate apps' operations using native code.

**Policy Engine.** The policy engine maintains enterprise policy rules. Both privilege enforcement and context enforcement modules read the current policy rules from the policy engine. The policy rules can be dynamically updated after receiving new policy rules from the enterprise policy center.

In the following, we will present our design and implementation of DeepDroid architecture on real Android mobile devices.

## V. Implementation

We implement a DeepDroid prototype and describe the implementation details of device-side components in this section.

### A. Permission Configurator

The permissions configurator module is designed to enforce runtime permission policies for all installed apps. To accomplish this goal, we introduce an extra permission checking component in addition to original system permission mechanism by dynamically modifying the control flow of *system_server* process.

Table I lists a set of permission checking interfaces provided by *system_server*, which provides *package*, *activity* and *permission* services. The first three interfaces accept parameters in the form of permission strings like *android.permission.INTERNET*. The fourth *checkUriPermission* interface accepts parameters of ContentProvider-Related Uri like *content://contacts*. The fifth interface for permission service checks media-related permissions, such as *CAMERA* and *RECORD_AUDIO*. Since it has been covered by the third interface, we only need to intercept the first four interfaces in the *system_server* process in Table I.

Almost all Android processes, including *system_server*, have their own Dalvik instances. Loaded Java classes and methods are internally indexed as *ClassObject* and *Method* instances, respectively. These instances are allocated on *dalvik-LinearAlloc*, a memory region that linearly managed by Dalvik. One *Method* instance may point to a Java method or a native function. For *system_server*, the permission checking interfaces are related to their corresponding *Method* instances. Hence,

TABLE I: Permission Checking Interfaces

| Service | Service Descriptor | Interface |
|---|---|---|
| package | android.content.pm.IPackageManager | checkPermission |
| package | android.content.pm.IPackageManager | checkUidPermission |
| activity | android.app.IActivityManager | checkPermission |
| activity | android.app.IActivityManager | checkUriPermission |
| permission | android.os.IPermissionController | checkPermission |

we can intercept them by inserting our native functions to the original Java *Method* instances.

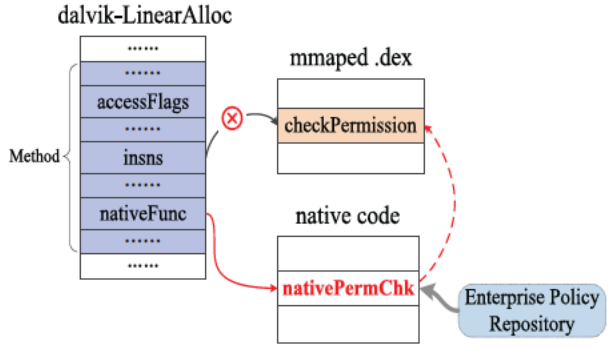


Fig. 3: Permission Checking Interception

Figure 3 shows the steps to intercept the permission checking. In *Method* structure, *accessFlags* indicates whether it is a Java or native method, where *insns* points to memory-mapped dex code and *nativeFunc* points to the actual native function or a JNI bridge. After changing *accessFlags* and remounting *nativeFunc* to the targeted *Method* instance described in Table I, all permission checks will be forwarded to our native method and then the policy rules can be reconfigured and enforced according to enterprise security requirements. For those permission items that are not defined in enterprise policy repository, the permission configurator forwards them to the original Java interfaces.

The permission configurator depends on dynamic runtime method interposition and it injects a piece of code into target process (e.g., the *system_server* process). Because Android is developed on Linux kernel, code injection techniques used in Linux also apply to Android. When our process is running with *root* privilege, we can call the *ptrace* system call to inject our code piece into a target process and get it executed.

Listing 1: Code Injection based on *ptrace*

```
1  void code_injection(pid_t pid,
                       const char* func,
3                      int length,
                       ...){
5      attach_to_target(pid);
       regs = get_regs(pid);
7
       free_mem = get_memory(pid);
9      put_code(pid, free_mem,
              func, length);
11
       manage_regs(&regs);
13     set_regs(pid, regs);
       detach_target(pid);
15     }
```

Listing 1 shows an overview of code injection procedure. First, we attach our tracer to the target process by calling *attach_to_target* method. Upon a successful attachment, registers of target process are reserved for site restoration. Then we can obtain free memory for the injected code by initiating a *mmap* function in target process. Because all Android processes share an identical mapping of system libraries including *libc.so* where *mmap* is defined, the *mmap* address in target process is the same as that in the tracer and can be easily obtained. After running *put_code* method, the free memory is filled with the injected code. By adjusting registers (*pc, lr, etc.*) of target process, the injected code gets executed before restoring normal code sequence. Now the environment of target process has been changed and we can detach our tracer. The injected code has the same privilege as the target process to perform operations such as modifying libraries and function definitions.

### B. Process Creation Guard

Android apps can also access system resources using native code, which can bypass the permission configurator. For example, an app that has been granted CAMERA permission will be added to the *camera* group and is authorized to perform native camera operations without using *camera* service. We propose to guard these uncontrolled operations by controlling the process creation procedure in *zygote*.

When launching an app, *system_server* sends a process creation request to *zygote* through socket. Listing 2

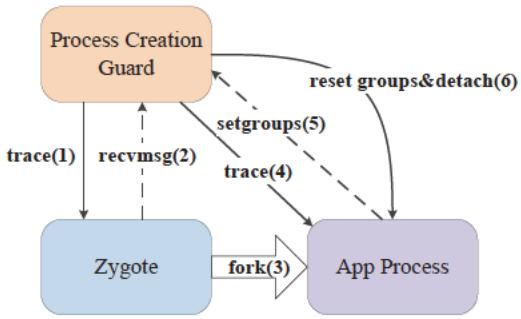


Fig. 4: Execution Flow of Process Creation Guard. (1) tracing *zygote* process; (2) extracting process creation request from *recvmsg*; (3) forking app process; (4) tracing new process upon forking; (5) extracting supplementary groups from *setgroups*; (6) resetting groups and detaching process.

shows a request message, which is used by *zygote* in forking an app process. 1015 (sdcard_rw), 3003 (inet), 1006 (camera), and 1007 (log) are supplementary group IDs and represent privileges of the new process. We narrow down this supplementary group set during process creation through tracing system calls of *zygote* process tree.

Listing 2: A Process Creation Request

```
1  --runtime-init
2  --setuid=10028
3  --setgid=10028
4  --setgroups=1015,3003,1006,1007
5  android.app.ActivityThread
```

Figure 4 shows an execution flow of the process creation guard. The guard continuously traces system calls of *zygote* process. Once a request is sent to *zygote* through *recvmsg*, our guard suspends *zygote* and extracts information from the request, such as *setuid* parameter that links the new process to a certain app. After that, our guard traces the new process upon its forking. Furthermore, it extracts the app process's supplementary groups from *setgroups*, resets groups according to enterprise policy, and finally detaches itself from the target app process.

We need to point out two things. First, when *setgroups* is called in the new process, there exists no straightforward way to associate the process to an app, since all app identifiers (*uid, gid*, etc.) have not been set then. We solve this problem by filtering app identifiers from arrived process creation requests through tracing socket calls of *zygote*, particularly, *recvmsg*. Second, when the child process returns from *fork*, the following code sequence will be executed: *setgroups-setgid-setuid*. After process's

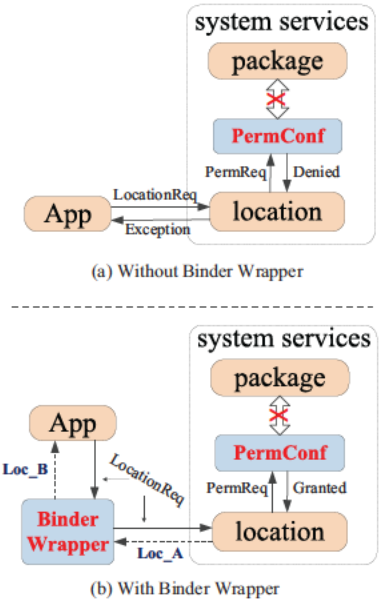


Fig. 5: Context Enforcement on Location

uid is reset by *setuid*, it runs as a normal process and its capabilities, include *setgroups*, will be revoked. Hence, we can detach the process from our tracer safely. Since this execution sequence requires less than 30 lines of code before any app code is loaded, it has no significant performance overhead to the app.

### C. Android-Specific Context Enforcement

We implement a fine-grained context enforcement by intercepting Binder. Figure 5 shows an example on regulating context-aware location information. Figure 5(a) shows the procedure of privilege enforcement where an app cannot obtain any location data since the location services may have been disabled by the permission configurator. Instead, we use a *Binder Wrapper* to intercept and regulate the service provision procedure, as shown in Figure 5(b). If the *Loc_A* provided by service is sensitive, *Binder Wrapper* replaces it with a fake *Loc_B* and sends it back to the app.

Most system services are running in a few service processes that provide a large number of interfaces for apps. Remote service calls from apps are delivered to these service processes by binder transaction. The binder wrapper manages all binder transactions in service processes in three steps: (1) intercepting binder transaction data; (2) parsing transaction data for high-level procedure calls; and (3) regulating the transaction.

**Binder Interception.** The *libbinder.so* provides communication interfaces for binder threads based on binder

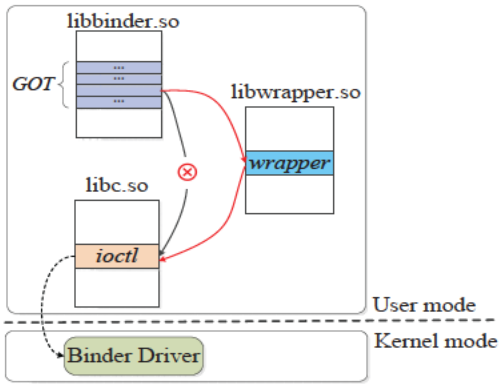


Fig. 6: Binder Interception

driver's file operations (e.g., open, ioctl). It is loaded into *zygote* and inherited by all app processes. Data transaction feature of *libbinder.so* depends on externally defined *ioctl*. The *ioctl* address is filled into Global Offset Table (GOT) of *libbinder.so* and all transaction data is intercepted by tampering this GOT item, as shown in Figure 6. Our *wrapper* function has an identical function signature with *ioctl* and transaction data is processed before being passed to *ioctl*.

In Android, the description of all shared objects is maintained as a *soinfo* struct in a linked list. A certain GOT item can be easily obtained and replaced with a different absolute function address after the memory segment is changed to "writable" by *mprotect*. Now all raw data of binder transaction can be inspected through this interception.

**Content Parsing.** The raw data from *ioctl* interception is packaged into a *binder_write_read* buffer. We manage to convert the raw data into comprehensive service procedure calls by a parser.

To better explain the role of parser, we use a remote *setWifiEnabled* call to *wifi* service as an example. If the raw transaction data (a *Parcel* object) matches interface name *android.net.wifi.IWifiManager* and procedure code 13, it is recognized as a procedure call. The number 13 works as a routing table item for the target procedure. After determining a remote procedure, we would map buffer data to procedure parameters or return values according to interface specification defined in intermediate files (e.g., *IWifiService.java*).

Table II shows a list of most accessed resources [13] and the information on how their corresponding interfaces are parsed. One *Process* may contain multiple binder or service threads and it is the target where binder data is collected. Descriptor of *Service* and *Code* are combined as an identifier to filter out a certain interface from a large amount of binder data. In particular, contacts are accessed from a content provider and an app must retrieve the provider from *activity* beforehand. Therefore, access to these resources through Android APIs can be intercepted and parsed in our *wrapper*.

**Transaction Regulating.** We can meet different requirements by processing on different points of binder transaction. A common pattern of binder transaction is shown in Figure 7. Binder thread #1 works as an initiator of transaction and thread #2 as its remote counterpart. Four transaction commands, BC_TRANSACTION, BR_TRANSACTION, BC_REPLY, and BR_REPLY, indicate state of the transaction, namely, a command to be sent, a command arrived, a reply to be sent and a reply arrived, respectively. In thread #2, a BR_TRANSACTION message containing RPC information (procedure code, interface name, calling parameters, etc.) arrives and is parsed into a target procedure call. After the procedure returns, a BC_REPLY message is sent to report return value to thread #1 if TF_ONE_WAY has not been set.

Two kinds of processing may be required. Preprocess is performed immediately after receiving a binder command and before target procedure is executed. By adapting procedure code or parameters, remote service can run a completely different operation or be forwarded to our control code. Thus, a SMS sent to a suspicious number can be controlled. Postprocess is performed before the return value is sent out. We can replace the return value with a fake value, which can help keep the app running without disclosing sensitive data, as shown in Figure 5.

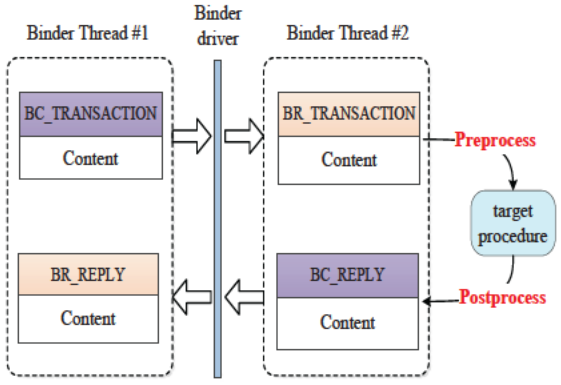


Fig. 7: A Typical Binder Transaction

We note that it is a challenge to apply this context-aware enforcement on all system resources. For instance,

TABLE II: Binder Resource Accessing Interfaces

| Resource | Process | Service | Function | Code |
|---|---|---|---|---|
| IMEI | phone | iphonesubinfo | getDeviceInfo | 1 |
| Phone # | phone | iphonesubinfo | getLine1Number | 5 |
| location | system_server | location | getLastKnownLocation | 17 |
| contacts* | system_server | activity | getContentProvider | 29 |
| camera | mediaserver | media.camera | connect | 3 |
| account | system_server | account | getAccounts | 4 |
| SMS/MMS | phone | isms | sendText | 5 |

it is difficult to define and enforce policy criterion on high volume media-related resources (such as camera and audio) without an efficient support on image or voice recognition. It is much easier for apps using Android APIs to perform operations like locating or photographing. However, an App may have multiple ways to bypass it using normal Android APIs. To solve this problem, we can extend DeepDroid with a behavior detection model similar to what has been proposed in FireDroid [14].

### D. Native Code Context Enforcement

The process creation guard in Section V-B constrains the access privileges of native operations with a course-grained Granted or Denied decision; however, enterprises may request a more fine-grained control on native operations. For instance, instead of disconnecting an app completely from the Internet, the app may be allowed to access some specific trusted web servers. Therefore, besides configuring *inet* group, we develop a native code context enforcement module to regulate network accessing operations by confining the context of socket calls, such as *connect*, *recvfrom* and *sendto*.

We trace system calls of the target process to constrain its native behaviors. Since it is important to guarantee that the tracing code is executed before any operation of target process, we must detect the launching of the target process and monitor the process through its entire life cycle. To achieve this goal, we trace *fork* calls of the processes (e.g., zygote, adbd) that are in charging of new app process creation and then map newly created processes to applications by matching their *uid*. Once the target app is identified, system calls of its process tree are traced recursively by setting *ptrace* options including *TRACEFORK*, *TRACEVFORK* and *TRACECLONE*. Since our module works as a tracer on a target process, when the target process is attached successfully, the tracer can receive signals at both the entry and exit of system calls, and meanwhile the target process is suspended after entering *syscall-enter-stop* and *syscall-exit-stop*. We maintain a flag for each target process to distinguish its system call entries from system call exits. In a typical work flow, this module first uses *ptrace* to extract CPU register information of a suspended target. According to routines of *ARM Procedure Call Standard* [15], the parameters of system calls are stored in registers *R0*$\sim$*R3* and *SP*, and the result of a system call is always stored in register *R0*. Therefore, our traces can easily parse the parameters and return value of the system calls. According to the policy rules, we can either simply decline a resource access request or send a fake return value.

Again, our work focuses on providing a mechanism to support native code context enforcement, instead of designing detailed policy rules. For various applications, different parsers may be required to inspect their unique context information. For instance, to restrict datagram communication with a certain remote server, we need to regulate system calls like *sendto, sendmsg, recvmsg* and *recvfrom*. At the entry of *sendto*, the destination address is extracted from *sockaddr* structure. Then, the system calls can go through if the destination address is valid or be stopped otherwise. Similarly, at the exit of *recvfrom*, we can recognize received buffer data and mask its content accordingly.

## VI. DeepDroid Evaluation

Our goal of DeepDroid evaluation is threefold: (1) to demonstrate that security policies on resource accessing can be effectively enforced; (2) to demonstrate that DeepDroid can be easily deployed to various Android platforms, and (3) to measure its performance overhead. We also analyze security and reliability of DeepDroid.

### A. Functional Evaluation

To test the effectiveness of DeepDroid, we choose the prevalent resources of Android identified in [13]

TABLE III: Tested Resources

| Resource | Permission | Group | PEP[1] | Result[2] |
|---|---|---|---|---|
| IMEI | READ_PHONE_STATE | | *package* | √ |
| Phone # | READ_PHONE_STATE | | *package* | √ |
| location | ACCESS_FINE_LOCATION | | *package* | √ |
| contacts | READ_CONTACTS | | *package* | √ |
| camera | CAMERA | camera | *package/PCG* | √ |
| account | GET_ACCOUNTS | | *package* | √ |
| logs | READ_LOGS | log | *PCG* | √ |
| SMS/MMS message | SEND_SMS | | *package* | √ |
| network | INTERNET | inet | *package/PCG* | √ |

[1] PEP is the policy enforcement point.
[2] The policy is enforced either in *package* service or by Process Creation Guard (PCG).

and run a number of popular apps that access these resources. For each resource, 5 most widely used apps that access this resource are chosen from *Google Play*. In our experiment, we manually instrument each resource related method call to check if the operations can be successfully regulated when we activate DeepDroid to prohibit accessing this resource. Table III shows the evaluation results, which verify that all resource accessing operations are controlled effectively.

### B. Portability Evaluation

We run DeepDroid on a series of smart phones with Android OS from version 2.3 to 4.x. As Table IV summarizes, DeepDroid can be successfully deployed on mainstream commercial Android devices with very small system modification.

TABLE IV: DeepDroid Portability

| Device | Android | Result |
|---|---|---|
| Nexus S(Samsung) | Android OS 2.3.6 | √ |
| Sony LT29i | Android OS 4.1.2 | √ |
| | Android OS 4.2.2 | √ |
| Galaxy Nexus(Samsung) | Android OS 4.0 | √ |
| Samsung Galaxy Note II | Android OS 4.1 | √ |
| Samsung Galaxy Note 3 | Android OS 4.3 | √ |
| Nexus 5(LG) | Android OS 4.4 | √ |
| Meizu MX II | Flyme 3.2[1] | √ |
| HUAWEI Honor 3c | Android OS 4.2 | √ |

[1] Flyme 3.2 is a customized version of Android OS 4.2.1

### C. Performance Evaluation

DeepDroid's system overhead is mainly introduced by the operations on permission enforcement, binder message interception, and system call tracing. Since a permission checking is always accompanied by a binder transaction, we combine the first two overhead factors.

*1) Permission Enforcement Overhead:* Permission enforcement overhead is mainly introduced by Permission Configurator in the *system_server* and behavior monitoring from binder interception. Permission Configurator overrules system original permission checking procedure with enterprise permission rules, and the Binder wrapper parses the binder messages to perform app's behavior monitoring.

Since most benchmarks focus on overhead of the entire process or the whole system, we need to design a mechanism to benchmark the overhead of permission enforcement operations. To minimize measurement errors, we choose sensitive operations that do not depend on a certain hardware module or volatile surroundings (e.g., stability of network signal). As shown in Table V, we construct a test case from a subset of prevalent resources described in [13].

TABLE V: Tested Permission Checking Transactions

| Class | Permission | Trigger Point |
|---|---|---|
| phone_state | android.permission. READ_PHONE_STATE | *TelephonyManager. getDeviceId* |
| contacts | android.permission. READ_CONTACTS | *ContentResolver. query(Phone.CONTENT_URI...)* |
| SMS messages | android.permission. SEND_SMS | *SmsManager. sendTextMessage* |

We evaluate the performance by initiating resource access operations that require a permission checking procedure in the following scenarios.

- **Normal mode (S)**. The tested apps are granted required permissions by original permission mechanism and the trigger points are executed successfully.
- **Intercepted mode (S)**. The tested apps are granted required permissions by Permission Configurator and the trigger points are intercepted and monitored by the binder.
- **Normal mode (F)**. The tested apps are denied required

permissions by original permission mechanism and an exception is thrown to run apps without executing trigger points.

- **Intercepted mode (F)**. The tested apps are denied required permissions by Permission Configurator and an exception is generated by the binder and sent back to the apps.

For each of the above scenarios, we run each trigger point in Table V 100 times. The results are shown in Figure 8. Contacts access in Intercepted mode (F) consumes 9.75% more time (about 744 *us*) than Normal mode (F), which is the highest overhead observed. It's worth mentioning that SMS messages and phone_state access use less time in Intercepted mode (S) than in Normal mode (S), since permission checking path is shortened in Permission Configurator and less time is spent on permission decisions.

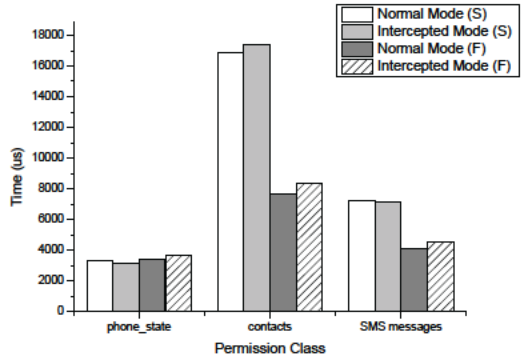


Fig. 8: Permission Enforcement Overhead

*2) Sys-call Tracing Overhead on Apps:* System call tracing is used to monitor app's native behaviors, as described in Section V-D. We evaluate performance impact of tracing running apps by two different benchmarks. CaffeineMark [16] measures running speed of Java programs. Quadrant is a CPU, I/O and 3D graphics benchmark [17]. They both support Android platforms.

We run the two benchmarks on three smart phone devices: *Meizu MX II*, *Sony LT29i* and *Nexus S(Samsung)*. We run each experiment 30 times and calculate the average number. Experimental results are shown in Figure 9 and Figure 10. CaffeineMark results show the highest overhead of 2.51% on MX II and Quadrant results show the highest overhead of 2.31% on Nexus S.

*3) Sys-call Tracing Overhead on Zygote:* An important tracing target in our work is *zygote* process, and we need to know the tracing impacts on the efficiency of *zygote* when creating new app processes. We use the time

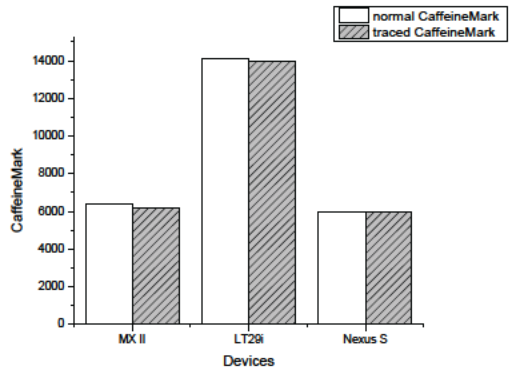


Fig. 9: Impact of Syscall Tracing by CaffeineMark

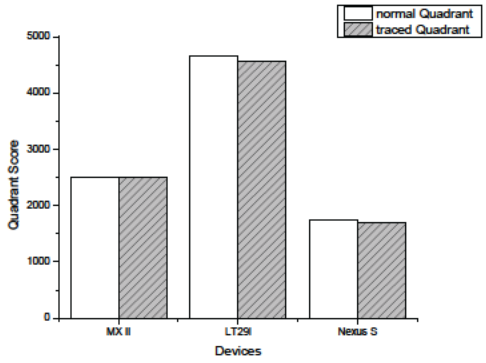


Fig. 10: Impact of Syscall Tracing by Quadrant

difference between initiating a *startService* request and that the service succeeds in running *onCreate* to measure efficiency of *zygote*. We compare a vanilla *zygote* and a traced one on different devices and show the results in Figure 11. We see the maximal time difference is within 20 $ms$, which can be ignored especially considering the low frequency of app creation requests.

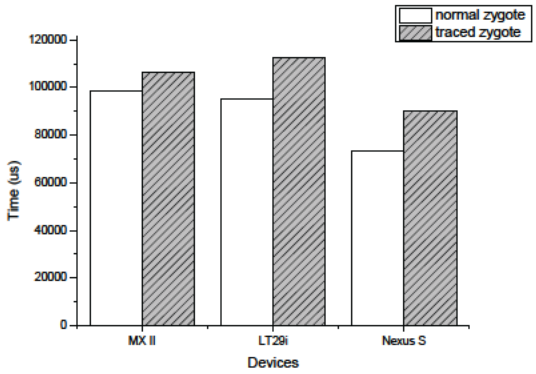


Fig. 11: Impact of Syscall Tracing on Zygote

### D. Security Analysis

Our system can ensure an enterprise security policy enforcement on Android device through dynamic memory instrumentation of several critical system processes.

**Malicious apps.** DeepDroid may allow device users to install their favorite apps on their Android smartphones. Some apps may be malicious and target at compromising our policy enforcement mechanism. However, since the user-level malicious processes are securely isolated into separate containers, they cannot manipulate the code or the control flow of DeepDroid unless they have the root privilege, which is strictly protected and monitored by enterprise administrators. We assume the Android OS can be trusted. Therefore, without the root privilege, malicious apps cannot compromise our mechanism.

**Permission escalation attacks.** An Android system may suffer from permission escalation attacks, such as confused deputy attack and collusion attack [18], [19], [20]. In confused deputy attack, a malicious application exploits the vulnerable interfaces of another privileged (but confused) application to perform unauthorized operations. This kind of attack usually happens when a privileged app unintentionally exposes interfaces of sensitive operation to an app without required permissions. In collusion attack, malicious apps collude and combine their permissions in order to perform actions beyond their individual privileges. DeepDroid can be used to regulate two apps' communications that go through the binder and system calls; however, if two apps may communicate through some covert channels that are out of the control of DeepDroid, we need to deploy other mechanisms to help remove the covert channels.

**Uncooperative user.** Some employees may be reluctant to conform to enterprise's security policies due to various reasons, and they may simply deactivate our system. Therefore, it is critical to guarantee that DeepDroid is correctly running on employees' mobile devices, and we use heartbeat messages to prove it. Therefore, a mobile device that stops sending encrypted heartbeat messages will trigger further investigation. Moreover, we can use software based attestation approach [21], [22], [23] to make sure the integrity of DeepDroid. On the mobile platforms with TrustZone hardware support [24], we can also use TrustZone to keep monitoring the integrity of DeepDroid, similar to what has been done in Knox [9].

**DeepDroid misuse.** An attacker may impersonate the enterprise administrators to send false policy rules to the mobile devices; however, since the attacker cannot obtain the keying material shared between the device and the enterprise server, it cannot perform this type of attacks. DeepDroid exposes one control interface of Android system to third parties; however, since the code base of DeepDroid is small and may be formally verified, the attacker can hardly misuse our mechanism to attack the system. Moreover, since all the communications between the trusted enterprise server and the mobile device are protected by a shared secret key, an external attacker cannot steal the policy rules through eavesdropping.

## VII. Discussion

At Google I/O 2014 conference, Android L was unveiled and the previously experimental Android Runtime (ART) [25] has replaced Dalvik as a default environment. ART compiles byte code into executable ELF only once during app installation. In spite of the runtime transformation, foundations of DeepDroid, including permission mechanism, system service architecture, and binder IPC, are barely changed. In other words, DeepDroid can be easily ported to ART. Only the implementation of runtime method interception in section V-A needs to be changed accordingly. In ART, the *.oat* executable file compiled from Java byte code is mapped into process by calling *dlopen* function, and all *Methods* refer to their native code in *oatexec* segment of *.oat*. ART runtime method interception can be achieved by native code inline-hooking. Hence, DeepDroid can also work on ART-enabled devices with little modification.

DeepDroid requires root privilege for installation. This requirement is common among enterprise mobile management solutions, and usually the root privilege can be obtained from OEMs. With OEM support, DeepDroid installation is simple and straightforward. OEMs only need to modify the "init.rc" file and import DeepDroid as a service. Then DeepDroid can run with a root privilege and all other Android security features remain intact. Since DeepDroid does not change the framework and the kernel of Android system, the above configuration incurs very little impact on OEMs. Some other work such as FireDroid [14] adopts a similar idea to obtain root privilege from OEMs. Alternatively, with the support of ARM TrustZone mechanism [24], DeepDroid may use the secure domain for the installation in the normal domain.

There exists a number of system access control solutions for enterprise management, but our solution has

some advantages when comparing to those solutions, particularly, SELinux [26] and Knox [9]. SEAndroid enforces mandatory access control (MAC) in Android kernel. Android's support for SELinux has evolved from permissive in 4.3 to full enforcement in 5.0 (L). SELinux enhances system security by confining privileged processes and enforce policies on various domains. However, it is unavailable or disabled by default on Android versions older than 4.4. Thus, old devices cannot be well protected by SEAndroid. Moreover, it requires manufacturers to have a better understanding of SELinux implementations. Until now, SEAndroid has not been fully supported since Android 5.0 does not include middleware MAC mechanism [5]. On the contrary, DeepDroid does not rely on any unique kernel features and thus works well on almost all Android versions and platforms. We may enhance the security of DeepDroid with SEAndroid mechanism.

Samsung Knox has risen up to provide a complete enterprise solution. It focuses on providing capabilities including Trusted Boot, TrustZone-based Integrity Measurement Architecture (TIMA), SE for Android, and Knox container, to protect Android system from adversaries and isolate different working scenarios [9]. Through secure boot and kernel integrity checking, Knox can ensure a trusted OS in the normal world based on ARM TrustZone hardware. As for policy enforcement, Knox integrates SEAndroid and provides management APIs to customize security policies. Despite that Knox APIs are integrated into Android 5.0 [27], its adoption is limited to Samsung devices [10]. Moreover, Knox requires ARM TrustZone hardware support, which limits its deployment to only certain Android platforms. Our DeepDroid system is a software-based solution that can be deployed on almost all Android platforms. Meanwhile, on TrustZone-enabled platforms, DeepDroid may utilize TrustZone to obtain the root privilege of the normal domain and protect the integrity of the rich OS in the normal domain.

## VIII. Related Work

Enterprise demands system resource access control interfaces in Android when employees are equipped with Android smartphone in workspace. Android access control mechanisms can be generally categorized into four classes by their technical approaches.

**Modifying Android source code.** Android source code can be directly modified to support new access control mechanisms when we can access the source code for the enterprise-customized Android system. Because Android permission framework does not provide flexible runtime configuration interface [28] and permission privilege leaks happen to commercial images [20], several security extensions have been proposed to the permission framework [29], [30], [31], [32], [33], [26], [34], [13], [35], [36]. Apex [29] enables users to grant a selected set of permissions and supports user-defined restrictions on apps. CRePE [30] can enforce fine-grained permission policies by using context information of the mobile devices. By introducing a privacy mode, TISSA [31] empowers users to define what kinds of personal information are accessible to apps. To mitigate security problems aroused by a certain third-party component, Compac [32] manages to distribute a narrowed set of permissions to one component. While the above approaches focus on protection of system resources, Saint [33] provides an infrastructure that protects apps' interfaces and resources.

Another way to enhance access control is achieved by introducing Security Enhanced Linux (SELinux). A flexible mandatory access control (MAC) can be supported on both Android's middleware and kernel layers [26], [34]. Besides directly hardening access control system, privacy data can be further protected by being replaced with some dummy data before providing it to apps [13]. TaintDroid [35] monitors usage of sensitive data by dynamic taint tracking and analysis. Based on Taintdroid, TreeDroid [36] presents a novel scheme to monitor security policies on data processing.

Since all these approaches require Android source code modification, they have portability problem due to the high cost incurred when customizing a specific Android branch from different OEMs. Our system performs dynamic memory instrumentation on the stable Android structures, so it can be deployed similarly on various Android versions.

**Rewriting apps.** Compared with modifying Android system code, app code can be rewritten to ensure a resource access policy. Due to good portability, enforcing security on apps themselves becomes a competitive approach. It is mainly implemented by integrating security measures into Android app with app rewriting. [37] enables identifying and interposing of Security Sensitive APIs by dalvik bytecode rewriting. [38] supports retrofit of app's behaviors by static and dynamic method interception. [39] is an on-the-phone instrumentation scheme, which enables flexible policies on apps by intercepting

high-level java calls. Security policies of [40] are enforced by low-level *libc.so* rewriting.

Interactions between an app and Android system can be recovered from system calls. [41] helps to perform fine-grained permissions on resource accessing by introducing a new module that supports parameterized permissions. Any access to sensitive resources from apps is forwarded to this module. To support behavior studying, [42] allows user to insert instrumentation code into an app from a high-level of abstraction. App rewriting is an effective way that requires no modification to Android ROM. However, incomplete implementations of bytecode rewriting may result in several potential attacks [43]. It is difficult to assure that all apps are rewritten which is critical to enterprise security management. In addition, due to signature difference of repackaging process, all history information of the original app cannot be shared by the rewritten app.

**Isolating business apps.** Besides adding extra control measures on Android system or apps, creating an isolated secure domain is another promising solution for running business related apps. It divides all user apps into two categories: personal apps and business apps. Business apps are running in a secure isolated environment, and thus attacks originated from personal domain can be blocked out. Trustdroid [44] is an isolation framework based on modifying Android system source code. KNOX [45] provides a complete enterprise management solution including an integrity checking component in the secure domain. All these solutions do not directly aim at enforcing enterprise security policies on employee's mobile device. It also has the portability problem due to the changing of Android system or hardware supports.

**Modifying Android runtime.** Our system relies on dynamically modifying the Android runtime environment. Patchdroid [46] uses a similar technique to apply security patches for the entire Android system, while our work focuses on enforcing enterprise policies on all installed apps.

## IX. Conclusions

In this paper, we present a mechanism named DeepDroid to enforce enterprise security policy on Android devices. DeepDroid provides good properties including portability, fine-grained control, and minimal performance overhead through dynamically injecting control code into Android framework, which has a stable set of process structures in various versions. Thus, DeepDroid may be deployed on various Android versions with a similar installation procedure. Since DeepDroid can regulate each app's service access operations by intercepting Binder transactions and tracing system calls, it can achieve a fine-grained context-aware policy enforcement. The evaluation results of a prototype show that DeepDroid can work effectively on various Android platforms with negligible performance overhead.

## X. Acknowledgment

The authors would like to thank the shepherd, Xiaofeng Wang from Indiana University, and the anonymous reviewers for their valuable comments and suggestions. This work is partially supported by National 973 Program of China under award No. 2013CB338001 and No. 2014CB340603. Dr. Kun Sun's work is supported by U.S. Office of Naval Research under award number N00014-11-1-0471 and U.S. Army Research Office under Grant W911NF-12-1-0448. This paper's corresponding author is Yuewu Wang.

## References

[1] "Cisco global work your way study," http://www.cisco.com/c/dam/en/us/solutions/collateral/borderless-networks/unified-access/ua_survey_infographic.pdf.

[2] "Android pushes past 80156.0http://www.idc.com/getdoc.jsp?containerId=prUS24442013.

[3] "App ops: Android 4.3's hidden app permission manager, control permissions for individual apps!" http://www.androidpolice.com/2013/07/25/app-ops-android-4-3s/hidden-app-permission-manager/-control-permissions-for/-individual-apps/.

[4] "App ops removed by google in android 4.4.2 update," http://www.phonearena.com/news/App-Ops-removed-by-Google-in-Android-4.4.2-update_id50340/.

[5] "Merge status," http://seandroid.bitbucket.org/MergeStatus.html.

[6] "Device administration," http://developer.android.com/guide/topics/admin/device-admin.html.

[7] "Mobile device management," http://en.wikipedia.org/wiki/Mobile_device_management.

[8] Samsung, "Samsung For Enterprise," http://www.samsung.com/us/business/samsung-for-enterprise/index.html.

[9] S. Electronics, "Samsung KNOX," http://www.samsung.com/global/business/mobile/solution/security/samsung-knox.

[10] "Knox workspace supported devices," https://www.samsungknox.cn/en/solutions/knox/technical/knox-supported-devices.

[11] "Openbinder," http://www.angryredplanet.com/ hackbod/openbinder/docs/html/.

[12] "System permissions," http://developer.android.com/guide/topics/security/permissions.html.

[13] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're 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, 2011.

[14] G. Russello, A. B. Jimenez, H. Naderi, and W. van der Mark, "FireDroid: Hardening Security in Almost-stock Android," in *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013.

[15] http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf.

[16] "Caffeinemark 3.0," http://www.benchmarkhq.ru/cm30/.

[17] "Quadrant standard edition," https://play.google.com/store/apps/details?id=com.aurorasoftworks.quadrant.ui.standard.

[18] A. D. T. F. A.-R. S. B. S. Sven Bugiel, Lucas Davi, "Towards taming privilege-escalation attacks on android," in *NDSS*, 2012.

[19] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission Re-Delegation: Attacks and Defenses," in *USENIX Security Symposium*, 2011.

[20] Z. W. X. J. Michael Grace, Yajin Zhou, "Systematic detection of capability leaks in stock android," in *NDSS*, 2012.

[21] V. K. Y. K. Mark Shaneck, Karthikeyan Mahadevan, "Remote software-based attestation for wireless sensors," *Security and Privacy in Ad-hoc and Sensor Networks*, vol. 3813, pp. 27–41, 2005.

[22] v. D. L. K. P. Seshadri. A, Perrig A, "Swatt: software-based attestation for embedded devices," in *IEEE Symposium on Security and Privacy*, 2004.

[23] 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*, 2011, pp. 23–23.

[24] T. Alves and D. Felton, "TrustZone: Integrated hardware and software security," *ARM white paper*, vol. 3, no. 4, 2004.

[25] http://source.android.com/devices/tech/dalvik/art.html.

[26] S. Smalley and R. Craig, "Security Enhanced (SE) Android: Bringing Flexible MAC to Android," in *NDSS*, 2013.

[27] "A closer look at knox contribution in android," https://www.samsungknox.cn/en/androidworkwithknox.

[28] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Permission Evolution in the Android Ecosystem," in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012.

[29] 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*, 2010.

[30] M. Conti, V. T. N. Nguyen, and B. Crispo, "CRePE: Context-related Policy Enforcement for Android," in *Proceedings of the 13th International Conference on Information Security*, 2011.

[31] X. J. Yajin Zhou, Xinwen Zhang and V. W. Freeh, "Taming Information-Stealing Smartphone Applications (on Android)," in *In: Trust and Trustworthy Computing*, 2011.

[32] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du, "Compac: Enforce Component-level Access Control in Android," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, 2014.

[33] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in android," *Computer Security Applications Conference, Annual*, 2009.

[34] S. Bugiel, S. Heuser, and A.-R. Sadegh, "Flexible and Fine-Grained Mandatory Access Control on Android for Diverse Security and Privacy Policies," in *22nd USENIX Security Symposium (USENIX Security '13)*, 2013.

[35] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *OSDI*, 2010, pp. 393–407.

[36] A. L. Mads Dam, Gurvan Le Guernic, "Treedroid: A tree automaton based approach to enforcing data processing policies," in *CCS*, 2012.

[37] B. Davis, B. S, A. Khodaverdian, and H. Chen, "I-arm-droid: A rewriting framework for in-app reference monitors for android applications," in *In Proceedings of the Mobile Security Technologies 2012, MOST 12. IEEE*, 2012.

[38] B. Davis and H. Chen, "RetroSkeleton: Retrofitting Android Apps," in *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, 2013.

[39] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "AppGuard: Enforcing User Requirements on Android Apps," in *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2013.

[40] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical Policy Enforcement for Android Applications," in *Proceedings of the 21st USENIX Conference on Security Symposium*, 2012.

[41] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein, "Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2012.

[42] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "SIF: A Selective Instrumentation Framework for Mobile Applications," in *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, 2013.

[43] H. Hao, V. Singh, and W. Du, "On the Effectiveness of API-level Access Control Using Bytecode Rewriting in Android," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, 2013.

[44] A. D. S. H. A.-R. S. B. S. Sven Bugiel, Lucas Davi, "Practical and lightweight domain isolation on android," in *SPSM*, 2011.

[45] Samsung Electronics, "White Paper: An Overview of Samsung KNOX," http://www.samsung.com/global/business/business-images/resource/white-paper/2013/06/Samsung_KNOX_whitepaper_June-0.pdf.

[46] C. Mulliner, J. Oberheide, W. Robertson, and E. Kirda, "PatchDroid: Scalable Third-party Security Patches for Android Devices," in *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013.