Practical and Lightweight Domain Isolation on Android

Sven Bugiel[†], Lucas Davi[†], Alexandra Dmitrienko[‡], Stephan Heuser[‡],
Ahmad-Reza Sadeghi[†,‡], Bhargava Shastry[‡]

[†]Technische Universität Darmstadt
Darmstadt, Germany

[‡]Fraunhofer SIT
Darmstadt, Germany

ABSTRACT

In this paper, we introduce a security framework for *practical and lightweight domain isolation* on Android to mitigate unauthorized data access and communication among applications of different trust levels (e.g., private and corporate). We present the design and implementation of our framework, *TrustDroid*, which in contrast to existing solutions enables isolation at different layers of the Android software stack: (1) at the middleware layer to prevent inter-domain application communication and data access, (2) at the kernel layer to enforce mandatory access control on the file system and on Inter-Process Communication (IPC) channels, and (3) at the network layer to mediate network traffic. For instance, (3) allows network data to be only read by a particular domain, or enables basic context-based policies such as preventing Internet access by untrusted applications while an employee is connected to the company's network.

Our approach accurately addresses the demands of the business world, namely to isolate data and applications of different trust levels in a practical and lightweight way. Moreover, our solution is the first leveraging mandatory access control with TOMOYO Linux on a real Android device (Nexus One). Our evaluation demonstrates that *TrustDroid* only adds a negligible overhead, and in contrast to contemporary full virtualization, only minimally affects the battery's life-time.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security

1. INTRODUCTION

The market penetration of modern smartphones is high and sophisticated mobile devices are becoming an integral part of our daily life. Remarkably, smartphones are increasingly deployed in business transactions: They provide employees a means to remain connected to the company's network thereby enabling *on the road* access to company's data. In particular, they allow employees to read and send e-mails, synchronize calendars, organize meetings, attend telephone and video conferences, obtain news, and much more. On the other hand, mobile platforms have also become an appealing target for attacks threatening not only private/personal data but also corporate data.

Until today, the Blackberry OS is the most popular operating system used in the business world. However, recent statistics manifest that Google Android is rapidly expanding its market share[1], also in the business world, where it is currently the third-most used mobile operating system after Blackberry and iOS [6].

Security Deficiencies of Android. The core security mechanisms of the (open source) Google Android OS [21] are *application sandboxing* and a *permission framework*. However, recent attacks show that Android's security architecture is vulnerable to *malware* of many kinds. First, uploading malicious applications on the official Android market is straightforward, since anyone can become an Android developer by simply paying a fee of $25. Second, Google does not perform code inspection. Recent reports underline that these two design decisions have led to the spread of a number of malicious applications on the official Android Market in the past [29, 19, 25, 4]. Further, another attack technique against Android is privilege escalation. Basically, these attacks allow an adversary to perform unauthorized actions by breaking out of the application's sandbox. This can be achieved by exploiting a vulnerable deputy [15, 10, 17], or by malicious colluding applications [37, 5]. In particular, privilege escalation attacks have been utilized to send unauthorized text messages [10], to trigger malicious downloads [24, 29], to change the WiFi settings [17], or to perform context-aware voice recording [37]. Finally, approaches that rely on Android's permission framework to separate private applications and data from corporate ones (such as enterproid [1]) will likely fail due to the above-mentioned attacks. Moreover, any attack on the kernel-level will allow the adversary to circumvent such solutions.

Domain Isolation with Default Android. In the light of recent attacks, the Android OS cannot meet the security requirements of the business world. These requirements mainly comprise the security of a heterogeneous company network

[1]At the time of writing, Android has 36% market share and belongs to the most popular mobile operating systems worldwide [18].

SPSM'11, October 17, 2011, Chicago, Illinois, USA.

to which smartphones connect along with the protection of corporate data and applications (on the phone). In particular, Android lacks data isolation: For instance, standard Android only provides single database instances for SMS, Calendar, and Contacts. Hence, corporate and private data are stored in the same databases and any application allowed to read/write the database has direct access to any stored information. Apart from application sandboxing, Android provides no means to *isolate* corporate applications from private user applications in a system-centric way. Hence, an adversary could get unauthorized access to the company's network by utilizing privilege escalation attack techniques. Finally, Android fails to enforce isolation at the network-level which would enable the deployment of basic context-aware policy rules. For instance, there is no means to deny Internet access for untrusted applications while the employee is connected to the company's network.

To summarize, default Android has no means to group applications and data into *domains*, where in our context a domain compromises a set of applications and data belonging to one trust level (e.g., private, academic, enterprise, department, institution, etc.)

Existing Security Extensions to Android. Recently, a number of security extensions for Android have been proposed, the closest to our work being [32, 14, 31, 28, 5]. However, as we will elaborate in detail in related work (see Section 7), all of these solutions focus on a specific layer of the Android software stack (mainly Android's middleware) and fail if the attack occurs on a different layer, e.g., at the network layer by mounting a privilege escalation attack over socket connections [10]). Specifically, they do not address kernel-level attacks [30, 24] that allow an adversary to access the entire file system. Having said that, attacks on the kernel-level can be mitigated by enabling SELinux on Android [40]. However, SELinux only targets the kernel-level, and misses high-level semantics of Android's middleware.

In particular, we are not aware of any security extension providing efficient and scalable application and data isolation on different layers of the Android software stack, which is essential for deploying Android in the business world.

On the other hand, several virtualization-based approaches aim at providing isolation between private and corporate domains on Android [33, 3]. However, contemporary mobile virtualization solutions suffer from practical deficiencies (see Section 7): (1) they do not scale well on resource-constrained smartphone platforms which allow only a limited number of virtual machines to be executed simultaneously; (2) more importantly, virtualization highly reduces the battery lifetime, because it duplicates the whole Android operating system. This raises a severe usability problem.

Our Contribution. In this paper, we present a novel security architecture, called *TrustDroid*, that enables *practical and lightweight domain isolation on each layer* of the Android software stack. Specifically, *TrustDroid* provides application and data isolation by controlling the main communication channels in Android, namely IPC (Inter-Process Communication), files, databases, and socket connections. *TrustDroid* is lightweight, because it has a low computational overhead, and requires no duplication of Android's middleware and kernel, which is typically a must for virtualization-based approaches [33, 3]. As a benefit, *TrustDroid* offers a good scalability in terms of the number of parallel existing domains. In particular, *TrustDroid* exploits coloring of separate and distinguishable components (this approach has its origins in information-flow theory [36]). We color applications and user data (stored in shared databases) based on a (lightweight) certification scheme which can be easily integrated (as we shall show) into Android. Based on the applications colors, *TrustDroid* organizes applications along with their data into logical domains. At runtime, *TrustDroid* monitors all application communications, access to common shared databases, as well as file-system and network access, and denies any data exchange or application communication between different domains. In particular, our framework provides the following features:

- **Mediating IPC:** We extend the Android middleware and the underlying Linux kernel to deny IPC among applications belonging to different domains. Moreover, *TrustDroid* enforces data filtering on standard databases (e.g., Contacts, SMS, etc.) so that applications have access only to the data subset of the their respective domains.
- **Filtering Network Traffic:** We modified the standard Android kernel firewall to enable network filtering and socket control. This allows us to isolate network traffic among domains and enables the deployment of basic context-based policies for the network traffic.
- **File-System Control:** We extend the current Android Linux kernel with TOMOYO Linux based mandatory access control and corresponding TOMOYO policies to enforce domain isolation at the file-system level. This allows us to constrain the access to world-wide readable files to one specific domain. To the best of our knowledge, TOMOYO has never been applied on a real Android device (e.g., Nexus One) before.
- **Integration in Trusted Infrastructures:** Our design includes essential properties and building blocks for integrating Android OS based smartphones into sophisticated trusted infrastructures, such as Trusted Virtual Domains [12].

We have tested *TrustDroid* with Android Market applications and show that it induces only a negligible runtime overhead and minimally impacts the battery life-time.

Outline. The remainder of this paper is organized as follows: In Section 2 we briefly recall the Android architecture and in Section 3 we provide a problem description, present our adversary model, and elaborate on our requirements and objectives. We present the architecture of *TrustDroid* in Section 4 and describe its implementation in Section 5. Our results are evaluated and discussed in Section 6. We summarize related work in Section 7 and conclude in Section 8.

2. ANDROID

In the following we briefly provide background information on Android. We explain the Android software stack, the types of communications present in the system and elaborate on the specifics of Android's security mechanisms.

2.1 Software Stack

Android is an open source software stack for mobile devices, such as smartphones or tablets. It comprises of a Linux kernel, the Android middleware, and an application layer (as depicted in Figure 1). The Linux kernel provides basic

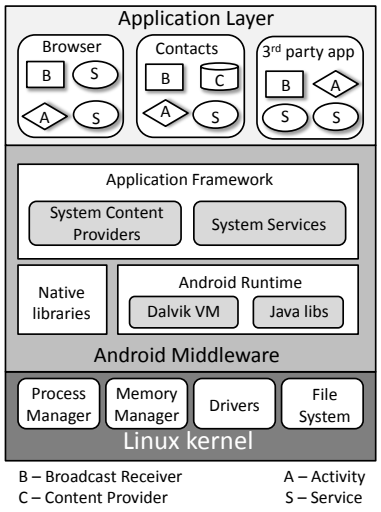

Figure 1: Android architecture

facilities such as memory management, process scheduling, device drivers, and a file system. On top of the Linux kernel is the middleware layer, which consists of native libraries, the Android runtime environment and the application framework. The native libraries provide certain core functionalities, e.g., graphics processing. The Android runtime environment is composed of core Java libraries and the *Dalvik* Virtual Machine, which is tailored for the specific requirements of resource constrained mobile devices.

The Android application framework consists of system applications written in C/C++ or Java, such as System Content Providers and System Services. These provide the basic functionalities and the essential services of the platform, for instance, the Contacts app, the Clipboard, the System Settings, the AudioManager, the WifiManager or the LocationManager. While System Content Providers are essential databases, System Services provide the necessary high-level functions to control the device's hardware and to get information about the platform state, e.g., location or network status.

At the top of the software stack is the application layer, which contains a set of built-in core applications (e.g., *Contacts* or *Web-browser*) and third party applications installed by the user (e.g., from the Android MarketStore[2]). Applications are written in Java, but for performance reasons may include native code (C/C++) which is called through the *Java Native Interface* (JNI). In general, Android applications consist of certain components: *Activities* (user interfaces), *Services* (background processes), *Content Providers* (SQL-like databases), and *Broadcast Receivers* (mailboxes for broadcast messages).

2.2 Communication

Android provides several means for application communication. First, it implements a Binder-based[3] lightweight Inter-Process Communication (IPC), which is based on shared memory. This is the primary IPC mechanism for the communication between the application components. This mechanism has been denoted as *Inter-Component Communication* (ICC) in [16] and since then this term has been well established. For ICC, a special definition language (*Android Interface Definition Language* – AIDL) is used to define the methods and fields available to a remote caller. An example of ICC calls is the binding to a remote service, thus calling remote procedures exposed by this service. Further, explicit actions on a different application can be triggered by means of an *Intent*, a message with an URL-like target address, holding an abstract description of the task to perform (e.g., starting an Activity). Second, the Linux kernel provides the standard IPC mechanisms, e.g., based on Unix domain sockets. Third, applications with the Internet permission are allowed to create Internet sockets. Thus, they are not only able to communicate with remote hosts but also connect to other local applications.

2.3 Security Mechanisms

Android implements a number of security mechanisms, most prominently application sandboxing and a permission framework that enforces mandatory access control (MAC) on ICC calls and on the access to core functionalities. In the following, we provide a brief summary of these mechanisms and refer to [16] for a more detailed discussion.

Sandboxing. In Android every installed application is sandboxed by assigning a unique user identifier (UID). Based on this UID the Linux kernel enforces discretionary access control (DAC) on low-level resources, such as files. For instance, each application has a private directory not accessible by other applications. Moreover, each application runs in its own instance of the Dalvik Virtual Machine under the assigned UID. This sandboxing mechanism also applies to native code contained in applications. However, applications from the same vendor (identified by the signature of the application package) can request a shared UID, thus basically sharing the sandbox.

Access Control. Figure 2 depicts the possible communication channels and their respective access control in Android. At runtime, Android enforces mandatory access control (MAC) on ICC calls between applications. The MAC mechanism is based on *Permissions* [20] which an application must request from the user and/or the system during installation. Android already contains a set of pre-defined permissions for the system services [20], but applications can also define new, custom permissions to protect their own interfaces. A reference monitor in the Android middleware checks if an application holds the necessary permissions to perform a certain protected action, for instance, to bind to a protected service or to start a protected activity of another application.

On the file system, apps can decide if their files are stored in their private directory, and are thus not accessible by any other application, or in a system-wide read-/writable location. Default Linux Inter-Process Communication, for instance, Unix domain sockets or pipes, can be created with certain modes that make them accessible by other processes. The creator of such sockets/pipes decides on the mode. Thus, both file system and default IPC are under discretionary access control.

Some permissions are mapped to Linux kernel group IDs, e.g., the Internet permission, thereby relying on Linux to prevent unprivileged applications from performing privileged

[2] `https://market.android.com/`

[3] Binder in Android is a reduced and custom implementation of OpenBinder[34]

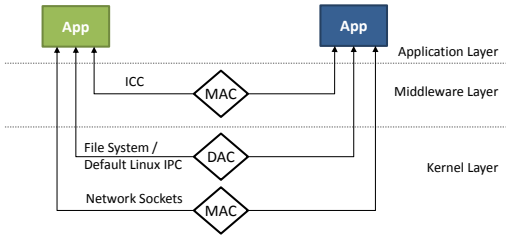

Figure 2: Communication channels and respective access control mechanisms in Android.

actions (e.g., creating Internet sockets, accessing sensitive information stored on external storage). However, since every application gets this permission granted (or not) at install time and the Android system henceforth enforces this decision, this falls under mandatory access control.

3. PROBLEM DESCRIPTION AND MODEL

We consider a corporate scenario which involves the following parties: (i) an enterprise (a company), (ii) a device (a smartphone), and (iii) an employee (the smartphone user). The enterprise issues mobile devices to its employees. The employees use their device for business related tasks, e.g., accessing the corporate network, loading and storing confidential documents, or organizing business contacts in an address book. To perform these tasks, the enterprise either deploys proprietary software, e.g., a custom VPN client including the necessary authentication credentials, on the device or provides a company-internal service, e.g., enterprise app market, from which employees can download and install those apps.

In this scenario, the enterprise is an additional stakeholder on the employees' devices and requires the protection of its delivered assets (software and data). Corporate assets may be compromised, e.g., when the user installs applications from untrusted public sources. Moreover, the employee accesses the enterprise internal network from his device and thus malware can potentially spread from the device into the corporate network.

A straightforward solution would be to prohibit any non-corporate app on the device (as proposed by, e.g., [11]). However, this is counter-intuitive to the idea of a *smartphone* and might even tempt employees to circumvent or disable this too restrictive security policy, e.g., by rooting the device. The default Android security mechanisms and recent extensions, on the other hand, are insufficient to provide enough isolation of untrusted applications and thus to protect the enterprise's assets. Virtualization can provide strong isolation between trusted and untrusted domains, but noticeably use up the battery life of the device, because major parts of the software stack are duplicated and executed in parallel in currently available virtualization solutions.

Consequently, an isolation solution is required, which preserves the battery life by minimizing the computational overhead and still provides isolation of corporate assets from untrusted applications.

3.1 Adversary and Trust Model

We consider *software* attacks launched by the adversary on the device at different layers of the Android software stack. The adversary's goal is to get access to corporate assets, e.g., to steal confidential data, to compromise corporate applications or to infiltrate the corporate network. The adversary can penetrate the system by injecting malware (e.g., by spreading it through the Android Market) or by exploiting vulnerabilities of benign applications. Malicious applications may either be granted by the user the privileges to access sensitive resources (see Gemini [25]) or try to extend their privileges by launching privilege escalation attacks [10, 37, 24, 19, 29, 30, 4].

We assume that the enterprise is trusted, and that the employee is not malicious, i.e., he does not intendedly leak the assets stored on his device. However, he is prone to security-critical errors, such as installing malware or disabling security features of his device.

The device is generally untrusted, but has a trusted computing base (TCB) which is responsible for security enforcement on the platform. The TCB is trusted by the enterprise.

3.2 Objectives and Requirements

We require the integrity and confidentiality of the corporate assets on the device, while preserving the usability. Furthermore, we require that the integrity of the corporate network will be preserved even if malware infiltrated employees' devices. With respect to these objectives, we define the following requirements:

- *Isolation.* Corporate assets must be isolated in a separate domain from untrusted data and software, and any communication between different domains must be prevented. In particular, the following communication channels must be considered: IPC channels, the file system and socket connections. In addition, potential malware on the device must be prevented from accessing the corporate network.

- *Access control.* Access of applications to assets stored on the device must be controlled by the enterprise by means of access control rules defined in a security policy, e.g., a new application can be installed in the corporate domain only if the policy states that it is trusted.

- *Legacy and Transparency.* To preserve the smartphone's functionality, we require our solution to be compatible to the default Android OS and to 3^{rd} party applications. Further, it should be transparent to the employee.

- *Low overhead.* With respect to the constrained resources of smartphones, in particular, the battery-life, our solution has to be lightweight.

3.3 Assumptions

We consider the underlying Linux kernel and the Android middleware as Trusted Computing Base (TCB), and assume that they have not been maliciously designed. Moreover, we assume the availability of mechanisms on the platform to guarantee integrity of the TCB (i.e., OS and firmware) on the device. For instance, this can be achieved with secure boot which is a feature of off-the-shelf hardware (e.g., M-Shield [41] and ARM TustZone [2]) or software security extensions for embedded devices (e.g., a Mobile Trusted Module (MTM) [42]).

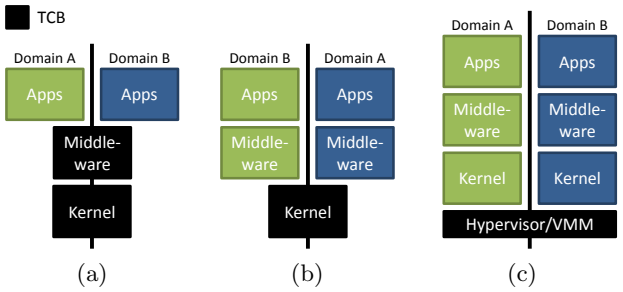

Figure 3: Approaches to isolation: (a) *TrustDroid*; (b) OS-level virtualization; (c) Hypervisor/VMM

4. DESIGN OF TRUSTDROID

In this section we describe the design and architecture of *TrustDroid*. The main idea is to group applications in isolated domains. With isolation we mean that applications in different domains are prevented from communicating with each other via ICC, Linux IPC, the file system, or a local network connection. Figure 3 illustrates different approaches to achieve isolation: (a) the approach taken by *TrustDroid*, which extends Android's middleware and kernel with mandatory access control; (b) OS-level virtualization, where each domain has its own middleware; (c) isolation enforced via a hypervisor and virtual machines, where each domain contains the full Android software stack. Comparing these approaches, *TrustDroid* has on the one hand the largest TCB, but on the other hand it is the most lightweight one, since it does not duplicate the Android software stack, and still provides good isolation, as we will argue in the remainder of this paper.

Our extensions to the Android OS are presented in Figure 4. The middleware extensions consist of several components: Policy Manager, Firewall Manager, Kernel MAC Manager, an additional MAC for Inter Component Communication (ICC), and finally a modified Package Installer. The Policy Manager is responsible for determining the color for each installed application, for issuing the corresponding policies to enforce the isolation between different colors, and to enforce these policies on any kind of ICC. The Firewall Manager and the kernel-level MAC Manager are instructed by the Policy Manager to apply the corresponding rules to enforce the isolation on the network layer and the kernel layer. To enforce the latter, *TrustDroid* relies on default features of the Linux kernel, which can also be activated in Android's Linux kernel: a firewall (FW) and a Kernel-level MAC mechanism. Since we modified the Android middleware a company which wants to make use of *TrustDroid* has to roll out a customized version of Android to their employees' smartphones.

In the subsequent sections, we elaborate in more detail on the components of *TrustDroid* that enforce domain isolation.

4.1 Policy Manager

In this section we explain the Policy Manager component of *TrustDroid* and elaborate in more detail on how it colors applications and enforces domain isolation in the middleware.

4.1.1 Application Coloring

The fundamental step in our architecture to isolate apps is to assign each app a trust level, i.e., to *color* them. In *TrustDroid*, we assume three trust levels for applications:

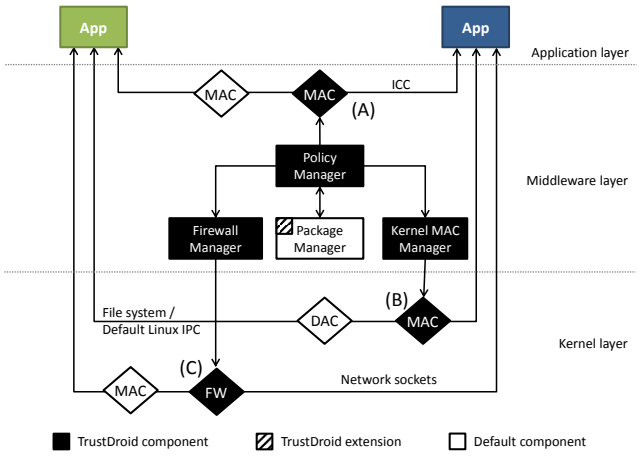

Figure 4: *TrustDroid* architecture with isolation of different colors (A) in the middleware, (B) at the file system/default Linux IPC level, and (C) at the network level.

1) pre-installed *system* apps, which include System Content Providers and Services (cf. Section 2); 2) *trusted* third party apps provided by the enterprise; 3) *untrusted* third party apps, which are retrieved from public sources such as the Android Market. While trusted and untrusted apps must be isolated from each other, system applications usually have to be accessible by all installed applications in order to preserve correct functionality of those applications and sustain both transparency and legacy compliance of our solution.

In *TrustDroid*, system apps (i.e., pre-installed apps) are already colored during platform setup in accordance with the enterprise's security policies. Additionally installed third party apps are colored upon installation, before any code of the app is executed. In Android, the PackageManager is responsible for the installation of new applications and in *TrustDroid* we extended it to interface with the Policy Manager, such that the Policy Manager can determine the color of the new app, issue the necessary rules for its isolation in the middleware, and instruct the Firewall Manager and Kernel MAC Manager to enforce the corresponding policies on the lower levels.

Determining the color of an app can be based on various mechanisms. For instance, it can be based on a list of application hashes for each color or based on the information available about the new app, such as developer signature or requested permissions. For *TrustDroid* we opted for a certification based approach. The Policy Manager recognizes a special certificate (issued by the enterprise), which is optionally contained in the application package of apps. Based on this certificate, *TrustDroid*'s PolicyManager verifies the authenticity and integrity of the new app. Moreover, the certificate may define a platform state, (e.g., the already installed applications), in which the certificate is only valid. A trusted service on the device is responsible for verifying these certificates. This service also measures the platform state, provides secure storage for the certificate verification keys, and maintains the verification key hierarchy such that only the enterprise can issue valid certificates. We use a Mobile Trusted Module (MTM) and as certificate format Remote Integrity Metrics (RIM) certificates, both defined

by the Trusted Computing Group (TCG) [42]. We refer to Section 5 for more details on how we use and implement those.

If such a RIM certificate is present, it must be successfully verified to continue the installation, i.e., the certificate must have been issued by the enterprise, the application package's integrity must be verified, and the platform state defined in the certificate must be fulfilled. Otherwise, the installation is aborted. In case of a successful verification, the certificate determines the color of the new app. In our corporate scenario with only two domains, successfully verified apps are in the *trusted* corporate domain. If no certificate is found, the app is by default colored as *untrusted*. This applies, for example, to all Android Market apps.

Alternatively, the certificates can already be pre-installed on the phone and the Policy Manager checks for a pre-installed certificate corresponding to the new app.

Generating the RIM certificates for applications requires a corresponding PKI inside the company. However, almost all companies today have integrated a PKI into their IT infrastructure. For the initial setup of the mobile devices the certificates are generated and integrated once for every pre-installed trusted application. By integrating the deployment of RIM certificates into a mobile device management solution or a company internal app market the process of app-certification can be automated for updates or applications installed later.

4.1.2 Inter Component Communication

As described in Section 2.2, Android uses Inter Component Communication as the primary method of communication between apps. Although ICC is technically based on IPC at the kernel level, it can be seen as a logical connection in the middleware. Thus, enforcement of isolation in the middleware has to be implemented based on access control on ICC.

In general, one can distinguish different kinds of ICC which can be used by apps for communication.

Direct ICC. The most obvious way for apps to communicate via ICC is to establish direct communication links. For instance, an app could send an Intent to another app, connect to its service, or query the content provider of another app. The *TrustDroid* MAC on ICC detects this communication and prevents it in case the sender and receiver app of the ICC have different colors. It thereby acts as an additional MAC besides the default access control of Android. As mentioned in Section 4.1, system apps form an exception and direct ICC is not prohibited if either sender or receiver of the ICC is a system app.

If two applications depend on each other, it is the responsibility of the certificate issuer, i.e, the enterprise in our scenario, to take care that these applications are in the same domain and to resolve any conflict in case the applications should have different trust levels according to the issuer's security policy.

Broadcast Intents. Besides the obvious direct ICC, apps are also able to send broadcast Intents, which are delivered to all registered receivers. Similar to the approaches taken in [32] and [5], *TrustDroid* filters out all receivers of a broadcast which have a different color than the sender before the broadcast is delivered. Again, system apps are an exception and are not filtered from the receivers list.

System Content Providers. A mechanism for apps

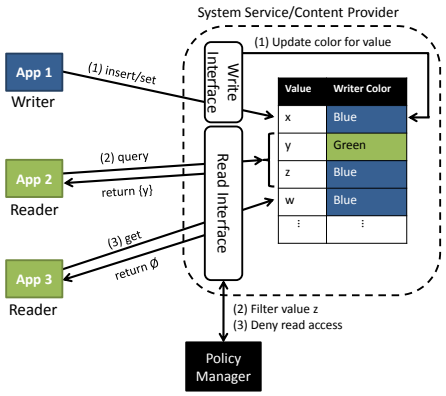

Figure 5: Coloring of data (1) and isolation of data from different colors in the (2) System Content Providers and (3) System Service.

from different domains to communicate indirectly is to share data in System Content Providers, such as the Contacts database, the Clipboard, or the Calendar. The ICC call to read data from such a provider does not give any information on the origin of the data, i.e., who wrote the data to the provider. We achieve domain isolation for System Providers as depicted in Figure 5. *TrustDroid* extends the System Content Providers such that all data is *colored* with the color of its originator app (Step (1) in Figure 5). Upon read access to a provider, all data colored differently than the reader app is filtered from the response (Step (2) in Figure 5).

System Service Providers. A covert method for apps to communicate are System Service Providers, such as the Audio Manager [37]. However, in our adversary model (cf. Section 3.1), we assume that corporate apps are trusted and not malicious and thus no sender for such a covert channel exists in the trusted corporate domain. Nevertheless, data might leak via System Services from the trusted to the untrusted domain and thus isolation should be enforced here as well. Thus, as for the System Content Providers, *TrustDroid* tags the read-/writable data values of the System Service Providers with the color of the last app updating them, e.g., when setting the volume level (Step (1) in Figure 5). Read access to these values is denied in case the colors of the reader and the data differ (Step (3) in Figure 5). Although this approach does not prevent this kind of covert channel per se, it drastically reduces its bandwidth to 1-bit, because the reader only gains information if his corresponding writer changed the value or not.

Alternatively, *TrustDroid* could return a pseudo or null value instead of denying the read access. However, in contrast to System Content Providers, on which a read operation by design might return an empty response, System Services are expected to return the requested value. Thus, returning a pseudo or null value may crash the calling app, or even cause more severe harm to the hardware or user, for instance, if the app reads a very low volume level when instead the real volume level is very high.

4.2 Kernel MAC Manager

The Kernel MAC Manager is responsible for communication with and management of the MAC mechanism provided by the underlying Linux kernel. Such mechanisms,

like SELinux [26] or TOMOYO Linux [22], are already by default features of the Linux kernel and provide mandatory access control on various aspects of the OS, including the file system and the Inter-Process Communication. Thus, by employing such a MAC mechanism, *TrustDroid* achieves the isolation of domains on file system and IPC level. More explicitly, we create a MAC domain for each color and each app is added to the domain of its color upon installation. The Policy Manager instructs the Kernel MAC Manager to which domain a new application has to be added and the Kernel MAC Manager translates this instruction into low-level rules, which are inserted into the MAC mechanism and which define the isolation of domains at file system and IPC level.

File System. The file system is a further communication channel for applications. Apps are able to share files system-wide, by writing them to a system-wide readable location. Thus, a sending application can write such a file and a receiving application would simply read the same file. The mandatory access control mechanism enforces isolation on the file system in addition to the discretionary access control applied by default. *TrustDroid* applies rules, which enforce that a system-wide readable file can be only read by a another app of the same color as the writer. Thus, if an app declares a file system-wide readable, it is shared only within the domain of the writer.

Moreover, mandatory access control can, with corresponding policies, even be used to constrain the superuser account. Hence, even if a malicious application gains superuser privileges, it's file system scope could be limited to it's domain.

Inter-Process Communication. To prevent any communication of apps through Linux IPC (e.g., pipes, sockets, messages, or shared memory), *TrustDroid* leverages the same domains already established for the file system access control. Thus, apps are not able to establish IPC with differently colored apps. However, system applications form an exception, since denial of communication to system apps renders any application dysfunctional.

Potentially, ICC, which is based on Binder (and hence on shared memory based IPC), can be essentially addressed with kernel level MAC. However, in this case the policy enforcement would be limited to direct ICC between apps and would miss indirect communications, e.g., via Content Providers or Broadcast Intents. In this sense, MAC on shared memory based IPC is supplementary to the ICC MAC, because it enforces policies even in case (malicious) applications manage to disable the ICC MAC.

4.3 Firewall Manager

A further channel that has to be considered is Internet networking, i.e., network sockets used for communication via Internet protocols (such as TCP/IP). Based on these sockets applications are able to communicate with remote hosts, but also with other applications on the same platform. Thus, isolation with respect to the corporate smartphone scenario has to take both local and remote communication into consideration. To enforce isolation, *TrustDroid* employs a firewall to modify or block Internet socket based communication. Managing the firewall rules based on the policies from the Policy Manager is the responsibility of the Firewall Manager component.

Local Isolation. To locally enforce isolation between domains on the platform, *TrustDroid* prohibits any communication from a local network socket of an untrusted application to another local network socket. Although, on first glance, this might appear over-restrictive, it is a reasonable enforcement, because applications residing on the same platform usually employ lightweight ICC to communicate instead of network channels.

Alternatively, network communication within each domain could be allowed and only cross-domain traffic be prevented. However, this would require that the Firewall Manager knows which Internet socket belongs to which application and which address has been assigned to each socket.

Remote Isolation and Context-Awareness. Enforcing isolation between domains on the network traffic between the platform and remote hosts, e.g., web-servers, is a harder problem than local enforcement. All data that leaves the phone is beyond the policy enforcement capabilities of *TrustDroid*. For instance, applications in different domains could exchange data via a remote web-service. Moreover, with respect to the corporate scenario, one must consider that malware on the phone might spread into the corporate network once the phone connects to it.

To address the former problem, *TrustDroid* uses a firewall that is able to tag (*color*) the network traffic, e.g., VLAN. If the network infrastructure supports the isolation of traffic, for instance in Trusted Virtual Domains (TVDs) [12], the policy enforcement is extended beyond the mobile platform.

To address the latter problem, *TrustDroid* employs context-aware policy enforcement on outgoing traffic. The context can be composed of various factors, for instance, the absence/presence of a user, the temperature of the device, or the network state. In *TrustDroid*, the context means the physical location of the device and the network the device is connected to. Each context definition is associated with a policy that defines how to proceed with the network traffic of untrusted applications, e.g., blocking all traffic or manipulating it in a particular way. Thus, if the platform is physically on corporate premises or connected to the corporate network, all untrusted, non-corporate apps could be denied network access or their traffic can be manipulated, for instance, to reroute it to a security proxy or an isolated guest network.

5. IMPLEMENTATION AND EVALUATION

5.1 Implementation

We implemented *TrustDroid* based on the Android 2.2.1 sources and the Android Linux kernel version 2.6.32.

We extended the default Android ActivityManager with a new component for the *TrustDroid* Policy Manager and the additional policy enforcement on ICC. We implemented the Firewall Manager and Kernel MAC Manager as new packages in the system services in the middleware.

The Policy Manager contains a minimal native MTM implementation, which is loaded as a shared library and called via the Java Native Interface (JNI). Alternatively, *TrustDroid* could use more sophisticated and secure MTM implementations as proposed in [13, 44, 45]. The MTM provides the means to verify Remote Integrity Metrics (RIM) certificates, to measure the software state of the platform, and to securely maintain monotonic counters.

Figure 6 illustrates the control flow for coloring a new application during installation and mapping the policies from the Policy Manager to the kernel and network level. Solid lines illustrate the control flow in case the application

package contains a RIM certificate. Dashed lines show the deviation from this flow in case no RIM certificate is included in the package.

Application Coloring. To color new apps during installation, we extended the Android PackageManager to call the *TrustDroid* Policy Manager during the early installation procedure (step 1 in Figure 6) in order to verify the certificate potentially included in the application package (denoted APK) and determine the color of the new app. Therefore, the certificate is first extracted from the APK (steps 2 and 3a) and the resulting APK is verified with this certificate (steps 4a and 5a). In case the verification fails, the installation is aborted by throwing a Security Exception back to the PackageManager (step 6a). In case no RIM certificate is contained in the APK, the installation proceeds normally (step 3b). If the installation is continued and succeeds (step 7), a second remote call from the PackageManager informs the Policy Manager of this success (step 8) and thus triggers the issuing of corresponding policies to isolate the new app from other apps with a different color at ICC level (steps 9a and 9b), at file system and IPC level (steps 10a and 10b), and the network level (step 11).

RIM Certificates and life-cycle management. As certificate format, we chose the RIM certificates as defined in the TCG Mobile Trusted Module (MTM) specifications [42]. In addition to the authenticity and integrity verification provided by other certificate standards such as X.509, RIM certificates additionally provide valuable features for a trusted life-cycle management. RIM certificates define a platform state in which the certificate is valid. This state is composed of monotonic counter values of the MTM and the measured software state. If either the counter value or software state defined in a RIM certificate mismatches the corresponding value of the MTM, the certificate verification fails. RIM certificates are signed with so-called *verification keys*. These verification keys form a key hierarchy, whose root key can be exclusively controlled by a particular entity, the enterprise in our scenario. Thus, only the enterprise is able to create valid RIM certificates for it's employees' devices and thus only successfully certified apps are considered as trusted. Examples for MTM-based enhanced life-cycle management of apps are the prevention of version rollback attacks based on monotonic MTM counters, the binding of the installation to a certain platform state, or the trustworthy reporting of the software state, i.e., installed applications.

To certify APKs, we developed a small tool written in Java and that makes use of the jTSS[4].

Network, Default IPC, and File System Isolation. To implement isolation at network, default Linux IPC, and file system level, our implementation employs *netfilter*[5] present in the kernel and *TOMOYO Linux*[6] v1.8 available as a kernel patch. To maintain them from the Firewall Manager and Kernel MAC Manager, respectively, we cross-compiled and adapted the user-space tools *iptables* and *ccs-tools*. The former is used to administrate netfilter and the latter for TOMOYO Linux policy management.

In *TrustDroid*, we created two TOMOYO Linux domains for third party apps, *trusted* and *untrusted*, and policies that isolate these domains on file system and default Linux IPC level. Upon installation of a new application, the UID of the new application is inserted into either one of those domains (steps 10a and 10b in Figure 6). A third domain for system apps is accessible by both the trusted and untrusted domains.

By default, *TrustDroid* denies all untrusted applications network communication to local addresses on the phone and the Policy Manager instructs the FW Manager to enforce this isolation also for newly installed untrusted applications (step 11 in Figure 6). Thus, any local network communication between trusted apps is isolated from untrusted apps.

A particular technical challenge was the adaption of the TOMOYO Linux user-space programs, in order to be able to maintain the TOMOYO Linux policies locally on the device. Recent documentation for TOMOYO Linux on the Android emulator describes the policy administration from a remote host instead of locally on the device and thus required certain adaption for *TrustDroid*.

Although TOMOYO Linux in version 1.8 provides MAC for Internet sockets as well, thus the means to exclude applications with fine-grained policies (e.g., UID, port or IP address) from Internet access, we opted for netfilter for two major reasons: 1) Unexpectedly denying access to sockets is much more likely to crash affected applications, in contrast to simply blocking the outgoing traffic and thus faking a disabled network connection; 2) netfilter provides much more flexibility than simply access control, e.g., manipulating or tagging network traffic for advanced security infrastructures such as TVDs or security proxies.

An alternative building block to TOMOYO Linux would be SELinux, which is based on extended file attributes and thus provides a more intuitive solution for domain isolation at the file system level. On the other hand, it is more complex to administer than TOMOYO Linux and requires modifications to the default Android file system, because the default file system does not support extended file attributes.

Context-Awareness. A context in our current implementation is simply the definition of a WiFi state and/or location. For instance, it could be the SSID of the wireless network, the MAC address of the access point, a certain latitude/longitude range, or proximity to a certain location.

To implement the context-aware management of the netfilter rules (cf. Section 4.3), the current Firewall Manager uses two state listeners – one for changes of the WiFi state and one for updates on the location. The former is simply a receiver for notification broadcasts about the changed Wifi state. The latter is an LocationListener thread registered at the LocationManager. In case one of the two listeners is triggered, the new state is compared with the installed contexts and the policies of any matched context are activated. The active policies of contexts that are not fulfilled anymore are revoked.

Middleware Isolation. The implementation of the additional policy enforcement on ICC is based on the *XManDroid* framework presented in [5], which provides the necessary hooks in the Android middleware to easily implement policy enforcement on direct ICC, broadcast Intents, and channels via System Content Providers and Services.

To prevent direct ICC between applications with different colors, we wrapped the *checkPermission* function of the ActivityManager, which is called every time a new ICC channel shall be established. If the default MAC of Android permits the new ICC, *TrustDroid* performs an additional check to compare the colors of the caller and callee. On

[4] `http://trustedjava.sourceforge.net/`
[5] `http://www.netfilter.org/`
[6] `http://tomoyo.sourceforge.jp/`

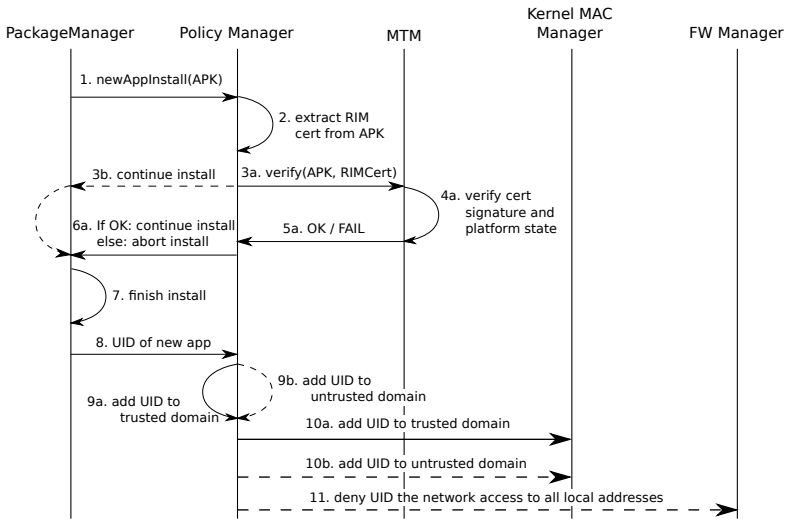

Figure 6: Control flow for the installation of a new application in case the installation package contains a RIM certificate (solid lines). If no RIM certificate is included in the package, this flow deviates (dashed lines).

mismatch, the previous decision is overruled and the ICC denied. To prevent data flow between different domains via Broadcast Intents, our implementation is similar to [32] and implements hooks in the broadcast management in the ActivityManager to filter out all receivers of a broadcast that do not have the same color as the sender. As described in Section 4.1.2, we extended the interfaces of System Content Providers, such as the Settings or Contacts, and of the System Services, such as the Audio Manager, to color data upon write access and filter data/deny access upon read access.

Moreover, the PackageManagerService allows applications to iterate over the information of installed packages, e.g., to find a specific application that might provide supplementary services. In *TrustDroid* we extended this functionality with additional filters, such that applications can only receive a list of and information about applications of the same color or about system applications.

5.2 Evaluation

We evaluated the performance overhead and memory footprint of our extensions to the middleware with 50 apps from the Android Market, categorized in two domains (plus one domain for system apps). On average our additional policy enforcement on ICC added $0.170ms$ to the decision process on whether or not an ICC is allowed (default Android requires on average $0.184ms$). The standard deviation in this case was $1.910ms$, caused by high system-load due to heavy multi-threading during some measurements. The verification of the RIM certificate during the installation of new packages required on average $869.750ms$ with a standard deviation of $645.313ms$. The average memory footprint of our extensions to the Android system server was 348.2 KB with a standard deviation of 200.8 KB, which is comparatively small to the default footprint of approximately 2 MB.

In our prototype implementation, the policy file of TOMOYO consumes on average a little more than 200 KB of memory. The policy file includes access control rules for file system and standard IPC mechanisms e.g. communication based on Unix domain sockets.

6. DISCUSSION

In this section we discuss the security of *TrustDroid* and highlight possible extensions.

6.1 Security Considerations

The main goal of *TrustDroid* is to provide an efficient and practical means to enforce domain isolation on Android. In particular, *TrustDroid* isolates applications by their respective trust levels, meaning that applications have no means to communicate with each other if their trust levels mismatch. Our requirement of access control is achieved by including certificates into an application package. Further, to control as many communication channels as possible, *TrustDroid* targets different layers of the Android software stack. First, IPC traffic (in the middleware and the kernel) is completely mediated by *TrustDroid* and is target to domain policies. Hence, malicious applications cannot use interfaces of applications belonging to other domains, even if the interfaces are exposed as public. Thereby *TrustDroid* prevents privilege escalation attacks from affecting other domains. Second, *TrustDroid* prevents unauthorized data access, by performing fine-grained data filtering on application data and data stored in common databases (SMS, Contacts, etc.). In particular, this prevents malicious applications from reading data of the corporate domain, as long as the malicious application has not been issued by the enterprise itself, which is excluded in our Adversary Model (cf. Section 3.1). Third, *TrustDroid* successfully mitigates the impact of kernel-exploits, because our TOMOYO policies prevent an adversary from accessing files of another domain. Finally, communication over socket connections are constrained to the domain boundary.

Although our approach is lightweight and practical, it does not provide the same degree of isolation as full-virtualization would do. In particular, *TrustDroid* only mitigates kernel-level attacks by restricting access to the file-system, but in general, it cannot prevent an adversary from compromising the Trusted Computing Base (TCB), which for *TrustDroid* includes the underlying Linux kernel and the Android middleware (see Section 3.3). In practice, static integrity of

the TCB can be insured by means of secure boot. However, the TCB is still vulnerable to runtime attacks subsequent to a secure boot. Solving this problem is orthogonal to the solution presented in this paper.

The primary cause for runtime attacks on Android is the deployment of native code (shared C/C++ libraries) [30]. Although Android applications are written in Java, a type-safe language, the application developers may also include (custom) native libraries via the Java Native Interface (JNI). Moreover, many native system libraries are mapped by default to the program memory space.

A straightforward countermeasure against native code based attacks would be to prohibit the installation of applications that include native code. However, this is rather over-restrictive and, similar to prohibiting any non-corporate app (cf. Section 3), contradicts the actual purpose of smartphones or might even tempt the phone user to break the security mechanisms in place.

Another approach to address native code attacks is Google's Native Client [38], which provides an isolated sandbox for native code. However, this solution requires the recompilation of all available applications that contain native code.

Moreover, as argued and shown in [46], mandatory access control can also be efficiently deployed on mobile platforms to enforce isolation for the complete Linux kernel. We consider this as a valuable extension to *TrustDroid* to mitigate kernel attacks, which could easily be integrated in *TrustDroid*, since a kernel-level MAC mechanism is already a building block of our design (see Section 4).

Finally, *TrustDroid* uses a separate, accessible domain for system applications and services, which is due to the fact that all applications require these system apps to work correctly. If an adversary identifies a vulnerability in one of these applications, he may potentially circumvent domain isolation and access data not belonging to his domain. However, until today, vulnerabilities of system applications were constrained to confused deputy attacks and did not allow an adversary to access sensitive data [17]. Protecting system applications and services from being exploited is orthogonal to harden the kernel, and we aim to consider this in our future work. Alternatively, one could deploy apps in the trusted domain which offer the functionality of certain system apps (e.g., business contacts app or enterprise browser; cf. [1]) and isolate the now redundant system apps by classifying them as untrusted.

6.2 Trusted Computing

Our *TrustDroid* design leans towards possible extensions with Trusted Computing functionality.

Currently, we leverage a Mobile Trusted Module (MTM) to validate application installation packages and to determine their color. The features of the employed RIM certificates in contrast to established certification standards such as X.509 provide the means for an enhanced life-cycle management based on monotonic counters and the platform state, e.g., version rollback prevention. The current implementation of our MTM is simple, but more sophisticated approaches may be integrated into our current design [13, 44, 45].

Moreover, our design includes the fundamentals for the integration of Trusted Computing Group (TCG) mechanisms such as the attestation of the domains [28], e.g., in the context of Trusted Network Connect [43], or the isolation of network traffic for infrastructures like Trusted Virtual Domains [12].

7. RELATED WORK

In this section we provide an overview of related work with respect to the establishment of domains and policy enforcement on Android.

7.1 Virtualization

A "classical" approach from the desktop/server area to establish isolated domains on the same platform is based on virtualization technologies. This approach has been ported to the mobile area [33, 3, 39]. Although virtualization provides strong isolation, it duplicates the entire Android software stack, which renders those approaches quite heavy-weight in consideration of the scarce battery life of smartphones. Possible approaches to mitigate this problem could be the automatic hibernation of VMs currently not displayed to the user or the application of a *just-enough-OS/Middleware* to minimize the resident memory footprint of domains. However, currently available mobile virtualization technology does not provide these features. In contrast, our solution is more lightweight, since the creation of a new domain simply requires the definition of a new string value and deployment of a new MTM verification key. Moreover, from our past experience with mobile virtualization technology [9], we conclude that our solution is more practical in the sense that it is more portable to new hardware, because we can re-use the provided proprietary hardware drivers, while virtualization requires new (re-implemented) drivers or an additional *driver-domain* that multiplexes the hardware between the VMs (e.g., *dom0* in Xen [23]).

7.2 Kernel-level Mandatory Access Control

Another well established mechanism, that is now being ported to the Android platform, is kernel-level mandatory access control like SELinux or TOMOYO Linux [40, 8]. These mechanisms allow, e.g., policy enforcement on processes, the file system, sockets, or IPC. In *SEIP* [46], SELinux was used to establish trusted and untrusted domains on the LiMo platform in order to protect the platform integrity against malicious third party software. The work further shows how unique features of mobile devices can be leveraged to identify the borderline between trusted/untrusted domains and to simplify the policy specification, while maintaining a high level of platform integrity. The authors of [35] show how policies in the context of multiple mobile platform stakeholders can be created dynamically and present a prototype based on SELinux. Low-level mandatory access control is an essential building block in our design (see Section 4). However, it is insufficient for isolating domains because it does not consider the Android middleware system components, such as System Content Providers/Services or Broadcast Intents, as communication channels between domains (see Section 3). Without high-level policy enforcement in the middleware, low-level MAC mechanisms can only grant/deny applications the access to System Content Providers and Services as a whole. However, generally denying an app access to system components most likely crashes this app or at least renders it dysfunctional. Moreover, although these mechanisms allow to some extend fine-grained access control policies on the network, they do not support the manipulation of network packets like netfilter does (cf. Section 4.3). Nevertheless, the approach of [46] could enhance the integrity protection of our TCB (see Section 6).

7.3 Android Security Extensions

In the last few years, a number of security extensions to the Android security mechanisms have been introduced [7, 27, 31, 15, 14, 5]. Based on very similar incentives to *TrustDroid*, *Porscha* proposes a DRM mechanism to enforce access control on specifically tagged data, such as SMS, on the phone. However, this approach is limited to isolate data assets, but is not suitable to isolate particular (sets of) apps.

Similarly, the sophisticated framework *TaintDroid* [14] tracks the propagation of tainted data from sensible sources (in program variables, files, and IPC) on the phone and detects unauthorized leakage of this data. However, it is limited to tracking data flows and does not consider control flows. Moreover, it does not enforce policies to prevent illegal data flows, but notifies the user in case an illegal flow was discovered. Nevertheless, TaintDroid could form a very valuable building block in our *TrustDroid* design to isolate data assets, if it would be extended with policy enforcement.

Both APEX [27] and CRePE [7] focus on enabling/disabling functionalities and enforcing runtime constraints. While APEX provides the user with the means to selectively choose the permissions and runtime constraints (e.g., limited number of text messages per day) each application has, CRePE enables the enforcement of context-related policies of the user or a third party (e.g., disabling bluetooth discovery). In this sense, both are related to our design goal to isolate untrusted applications based on the context (cf. Section 4.3) or protect data assets in shared resources like System Content Providers. However, the enforcement described in [27] and [7] is too coarse-grained. For instance, networking would be disabled for all applications, not just particular ones, or not only the access to certain data but to the entire Content Provider would be denied to selected applications.

Saint [32] introduces a fine-grained, context-aware access control model to enable developers to install policies to protect the interfaces of their apps. Although Saint could, with a corresponding system centric policy, provide the isolation of apps on direct and broadcast ICC, it can not prevent indirect communication via System Components (see Section 4.1.2).

XManDroid [5] addresses the problem of ICC-based privilege escalation by colluding apps and is also able to enforce policies on ICC channels via System Components. The XManDroid framework formed the basis for our *TrustDroid* implementation, but had to be extended to enable application coloring and mapping of policies for domain isolation from the middleware onto the network and kernel level.

In general, none of these extensions provides any policy enforcement on the file system, IPC, or local Internet socket connections in order to enforce isolation of domains. However, TaintDroid with its data flow tracking mechanism has the potential to implement fine-grained policy enforcement.

8. CONCLUSION

Existing smartphone operating systems, such as Google Android, provide *no data and application isolation* between domains of different trust levels. In particular, there exists no efficient solution to isolate corporate and private applications and data on Android: the existing security extensions for Android only focus on one specific layer of the Android software stack, and hence, do not provide a general and system-wide solution for isolation.

In this paper we present *TrustDroid*, a framework which provides *practical and lightweight domain isolation* on Android, i.e., it does not affect the battery life-time significantly, requires no duplication of Android's software stack, and supports a large number of domains. In contrast to existing security extensions, *TrustDroid* enforces isolation on many abstraction layers: (1) in the middleware and kernel layer to constrain IPC traffic to a single domain, and to enforce data filtering for common databases such as Contacts, (2) at the kernel layer by enforcing mandatory access control on the file system, and (3) at the network layer to regulate network traffic, e.g., denying Internet access by untrusted applications while the employee is connected to the corporate network. Our evaluation results demonstrate that our solution adds a negligible runtime overhead, and in contrast to contemporary virtualization-based approaches [33, 3], only minimally affects the battery's life-time.

We also provide a detailed discussion on the design of *TrustDroid* and argue that *TrustDroid* can be used as a foundation for Trusted Computing enhanced concepts such as Trusted Virtual Domains (TVD), a distributed isolation concept known from the desktop world. In our future work, we aim to adopt domain isolation on the underlying Linux kernel so that an adversary can no longer exploit kernel vulnerabilities to circumvent domain isolation.

9. REFERENCES

[1] enterproid. `http://www.enterproid.com/`.

[2] ARM. TrustZone technology overview. `http://www.arm.com/products/security/trustzone/index.html`.

[3] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis. The VMware mobile virtualization platform: is that a hypervisor in your pocket? *SIGOPS Operating Systems Review*, 2010.

[4] T. Bradley. DroidDream becomes Android market nightmare. `http://www.pcworld.com/businesscenter/article/221247/droiddream_becomes_android_market_nightmare.html`, 2011.

[5] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. XManDroid: A new Android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Universität Darmstadt, 2011.

[6] P. Carton. New burst of momentum for Google Android OS. `http://www.investorplace.com/18151/google-android-os-major-corporate-smart-phone-winner/`, 2010.

[7] M. Conti, V. T. N. Nguyen, and B. Crispo. CRePE: Context-related policy enforcement for Android. In *13th Information Security Conference (ISC)*, 2010.

[8] N. Daisuke and G. L. Tona. Tomoyo-android: TOMOYO Linux on Android. `http://code.google.com/p/tomoyo-android/`.

[9] L. Davi, A. Dmitrienko, C. Kowalski, and M. Winandy. Trusted virtual domains on OKL4: Secure information sharing on smartphones. In *6th ACM Workshop on Scalable Trusted Computing (STC)*, 2011.

[10] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *13th Information Security Conference (ISC)*, 2010.

[11] A. Distefano, A. Grillo, A. Lentini, and G. F. Italiano. SecureMyDroid: enforcing security in the mobile devices lifecycle. In *ACM CSIIRW*, 2010.

[12] A. Dmitrienko, K. Eriksson, D. Kuhlmann,

G. Ramunno, A.-R. Sadeghi, S. Schulz, M. Schunter, M. Winandy, L. Catuogno, and J. Zhan. Trusted Virtual Domains – design, implementation and lessons learned. In *INTRUST*, 2009.

[13] J.-E. Ekberg and S. Bugiel. Trust in a small package: Minimized MRTM software implementation for mobile secure environments. In *4th ACM Workshop on Scalable Trusted Computing (STC)*, 2009.

[14] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[15] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *16th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[16] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android security. *IEEE Security and Privacy Magazine*, 2009.

[17] A. P. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.

[18] Gartner Inc. `http://www.gartner.com/it/page.jsp?id=1689814`, 2011.

[19] D. Goodin. Android bugs let attackers install malware without warning. `http://www.theregister.co.uk/2010/11/10/android_malware_attacks/`, 2010.

[20] Google. The Android developer's guide - Android Manifest permissions. `http://developer.android.com/reference/android/Manifest.permission.html`, 2010.

[21] Google Inc. Google Android. `http://www.android.com/`.

[22] T. Harada, T. Horie, and K. Tanaka. Task Oriented Management Obviates Your Onus on Linux. In *Linux Conference*, 2004.

[23] J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim. Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones. In *IEEE CCNC*, Jan. 2008.

[24] A. Lineberry, D. L. Richardson, and T. Wyatt. These aren't the permissions you're looking for. BlackHat USA 2010. `http://dtors.files.wordpress.com/2010/08/blackhat-2010-slides.pdf`, 2010.

[25] Lookout Mobile Security. Security alert: Geinimi, sophisticated new Android Trojan found in wild. `http://blog.mylookout.com/2010/12/geinimi_trojan/`, 2010.

[26] National Security Agency. Security-Enhanced Linux. `http://www.nsa.gov/research/selinux`.

[27] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In *ACM ASIACCS*, 2010.

[28] M. Nauman, S. Khan, X. Zhang, and J.-P. Seifert. Beyond kernel-level integrity measurement: Enabling remote attestation for the Android platform. In *TRUST*, 2010.

[29] Nils. Building Android sandcastles in Android's sandbox. BlackHat Abu Dhabi 2010. `https://media.blackhat.com/bh-ad-10/Nils/Black-Hat-AD-2010-android-sandcastle-wp.pdf`, 2010.

[30] J. Oberheide. Android Hax. SummerCon 2010. `http://jon.oberheide.org/files/summercon10-androidhax-jonoberheide.pdf`, 2010.

[31] M. Ongtang, K. Butler, and P. McDaniel. Porscha: Policy oriented secure content handling in Android. In *ACSAC*, 2010.

[32] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *ACSAC*, 2009.

[33] Open Kernel Labs. Ok:android. `http://www.ok-labs.com/products/ok-android`.

[34] Palm Source, Inc. Open Binder. Version 1. `http://www.angryredplanet.com/~hackbod/openbinder/docs/html/index.html`, 2005.

[35] V. Rao and T. Jaeger. Dynamic mandatory access control for multiple stakeholders. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2009.

[36] J. M. Rushby. Design and verification of secure systems. In *8th ACM Symposium on Operating System Principles (SOSP)*, 1981.

[37] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *18th Annual Network and Distributed System Security Conference (NDSS)*, 2011.

[38] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary CPU architectures. In *USENIX Security Symposium*, 2010.

[39] M. Selhorst, C. Stüble, F. Feldmann, and U. Gnaida. Towards a trusted mobile desktop. In *TRUST*, 2010.

[40] A. Shabtai, Y. Fledel, and Y. Elovici. Securing Android-powered mobile devices using SELinux. *IEEE Security and Privacy Magazine*, 2010.

[41] J. Srage and J. Azema. M-Shield mobile security technology, 2005. TI White paper. `http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf`.

[42] Trusted Computing Group. Mobile Trusted Module Specification. Version 1.0 Revision 6, 26 June 2008.

[43] Trusted Computing Group (TCG). *TNC Architecture for Interoperability, Version 1.4, Revision 4*, 2009.

[44] J. Winter. Trusted computing building blocks for embedded linux-based ARM trustzone platforms. In *3rd ACM Workshop on Scalable Trusted Computing (STC)*, 2008.

[45] X. Zhang, O. Acıiçmez, and J.-P. Seifert. A trusted mobile phone reference architecture via secure kernel. In *2nd ACM Workshop on Scalable Trusted Computing (STC)*, 2007.

[46] X. Zhang, J.-P. Seifert, and O. Acıiçmez. SEIP: simple and efficient integrity protection for open mobile platforms. In *12th International Conference on Information and Communications Security (ICICS)*, 2010.