TrUbi: A System for Dynamically Constraining Mobile Devices within Restrictive Usage Scenarios

Miguel B. Costa, Nuno O. Duarte, Nuno Santos, Paulo Ferreira
INESC-ID / Instituto Superior Técnico, University of Lisbon
miguel.costa@inesc-id.pt,nuno.duarte@inesc-id.pt,nuno.santos@inesc-id.pt,paulo.ferreira@inesc-id.pt

ABSTRACT

In certain restrictive usage scenarios, personal mobile devices are required to operate in some constrained manner. Security concerns tend to be the most typical motivation, for example, as in "Bring Your Own Device" use cases. However, because most device configurations are strictly controlled by their respective users, today it is practically infeasible to satisfy such requirements. In this paper, we present TrUbi, a system that allows for dynamic and temporary restriction of Android devices by disabling or locking specific functions for limited amounts of time, e.g. network blocked. TrUbi enforces global security policies by implementing an OS primitive named *trust lease*. Our TrUbi prototype can efficiently enforce security policies in unmodified real-world apps and paves the way for new apps that are currently unsupported by existing mobile platforms.

CCS CONCEPTS

•**Security and privacy → Mobile platform security;**

KEYWORDS

Trust lease, Android, Security, Runtime restrictions

1 INTRODUCTION

As mobile devices make their way to becoming truly ubiquitous personal assistants and permeate every aspect of human lives, usage scenarios begin to emerge that require them to operate under specific restrictions. One example includes the so called "Bring Your Own Device" (BYOD) scenarios in which personally owned devices are allowed to execute privileged processes or access sensitive data within a given organization. Regardless of how convenient this paradigm may be for professionals (allowing them to carry along a single device for both personal and professional uses), as of today, BYOD is far from gaining widespread acceptance, largely due to the lack of organizations' trust in personal mobile devices and fear of security breaches. In fact, because personal devices are entirely controlled by their owners, no assurances can be given with respect to compliance with companies' security policies.

The traditional way to ensure policy compliance is to use Mandatory Access Control (MAC) [1, 2, 18]. In this approach, each personal device runs under a predefined MAC policy centrally controlled by the IT department of the company. The MAC policy prevents the user from accessing potentially dangerous sites, running unauthorized apps, sharing classified documents, etc. Although this approach serves the company's interests well, for device owners it entails complete loss of control of the device. In particular, the device owner must surrender his administration privileges to the company which includes granting it access to all personal data and consequent loss of privacy. The question we address in this paper is how to enable personal devices to be allowed to (1) restrict their functionality according to specific usage scenarios, without the need to (2) yield control of the device to a third party.

To address this question, we present a system called TrUbi. TrUbi is an extension to Android OS that allows for dynamically constraining the functionality of a mobile device according to the security requirements of a given use case. The key feature of TrUbi that enables this behavior is a primitive called *trust lease*. A trust lease is a MAC security policy which allows for blocking access to certain resources (e.g., access to the Internet, changing the microphone settings) for a certain amount of time. What is specific to trust leases is that the restriction conditions to be applied are not unilaterally decided by a third party, but must be agreed upon between the user (i.e., the device owner) and the third party. Furthermore, the third party never gains administration control over the device. Instead, the third party is represented by a mobile application—named *strapp*—which must be freely installed by the user on the device and has the same privileges as any other standard mobile app. Then, when this application is executed, it issues a trust lease request to the OS in order to install a time-limited security policy specified by the third party. At this point the user is prompted whether he agrees with the policy or not, and is free to accept or decline it, respectively. Thus, in the BYOD scenario, for example, the company only needs to implement a strapp application containing the specification of the trust lease policy that needs to be enforced on the personal devices at working hours; the owner of each device must install this application and accept the terms of the lease.

The concept of trust lease first appeared in our recently published short workshop paper [19]. There, we not only introduce the basic idea behind this primitive, but also make the case of how it can be used to enable a broad range of new restricted mobile scenarios (e.g., applying restrictions to the camera and microphone for privacy purposes, place devices in silent mode in theaters). The current paper goes beyond this original proposal by presenting TrUbi. In particular, in this paper, we make the following contributions.

First, we present an OS-independent design of TrUbi which specifies the basic mechanisms that must be developed in order to provide trust lease support on a mobile platform (Section 2).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Mobihoc '17, Chennai, India
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
978-1-4503-4912-3/17/07...$15.00
DOI: http://dx.doi.org/10.1145/3084041.3084066

Second, we implemented a fully working version of TrUbi on Android OS. TrUbi allows for the specification of rich restriction policies, e.g., mute the sound, block network access by applications, or force all applications to terminate except a few trusted ones. In our current version, TrUbi can restrict up to 13 different functions on a device. It is possible to specify flexible trust lease termination conditions based on time or location (e.g., the vicinity to specific GPS coordinates). We leveraged Android Security Modules (ASM) [11] to build a fully workable TrUbi prototype (Section 3).

Lastly, we demonstrate the applicability and effectiveness of TrUbi's security policies. In particular, we built a BYOD use case app that enables exams to be answered on students' own devices without cheating (Section 4). We evaluated TrUbi using a testbed of 87 unmodified third-party apps on real devices, and found its performance overhead to be small (Section 5).

2 DESIGN

In this work, we pursue the following goals: (1) devise a general system architecture that supports the specification and enforcement of restriction policies according to the trust lease security model, (2) implement the system on commodity mobile devices, and (3) demonstrate the potential of our system using real applications.

2.1 Threat model and assumptions

Our system must be robust against an adversarial user aiming to subvert the security properties enforced by trust leases. In particular, he may attempt to override the restrictive conditions of a trust lease by trying to execute applications, modify system settings, or rebooting the device. We assume that the operating system is correct, and that it is part of the Trusted Computing Base (TCB) of TrUbi. We are not going to address the problem of faulty OSes, or handle physical attacks to the hardware. Furthermore, we assume that the device has not been rooted, and that the integrity of the OS can be verified upon boot using integrity checking mechanisms provided by the firmware and hardware. The device can be equipped with trusted computing hardware technology, such as TPM or ARM TrustZone, and factory firmware to implement OS integrity measurement features at boot (aka trusted boot) and a platform ID consisting of a certified public key pair that can be used to implement remote attestation protocols. The public key is certified by the device manufacturer. We adopt Android, as our target platform.

2.2 Overview

We present TrUbi (Trusted Ubiquity), a system that provides a new OS primitive named *trust lease* to enforce application-specific restriction policies. To enforce trust lease restriction policies, TrUbi adopts a reference monitor architecture, as shown in Figure 1. Here, the light shaded boxes show a strapp (X) that owns a time-limited trust lease for restricting network access. A *trust lease reference monitor* maintains trust lease state information and coordinates their enforcement in the system by interacting with *restrictor* and *terminator* components. Restrictors implement specific restriction rules by setting the initial state of targeted objects and preventing unauthorized modifications to that state until the trust lease stops. Terminators are responsible for implementing policies' termination rules (Section 2.5). Each trust lease maintains a reference to the strapp that requested its

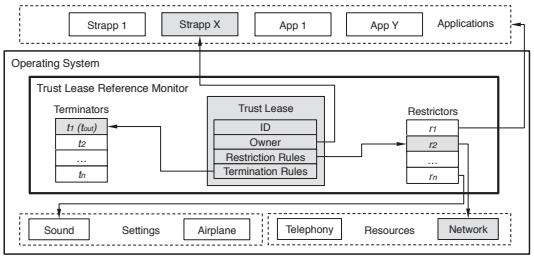

Figure 1: TrUbi architecture.

creation (*owner*). As explained in the following sections, only the owner can invoke certain trust lease operations, namely stoplease and quotelease. To illustrate how trust leases work in TrUbi, we use a simple example.

2.3 A motivating usage scenario

To illustrate the concept of trust lease, take the following example. Consider that we wanted to develop a mobile application for a "Bring your own device" (BYOD) usage scenario within the context of universities or other educational institutions. Essentially, the idea is to provide a digital alternative to printed exam copies, allowing students to do the exams by interacting with a mobile application running on their devices. This application would display the exam questions to the student, read his responses typed on the screen, and submit them to a centralized server whenever the exam duration had expired (or the student had voluntarily submitted his answers). In this scenario, the student would still be demanded to be present in an examination room to answer the exam in the presence of a professor to prevent students from cheating.

The trouble, however, is that if students are allowed to carry today's mobile devices, they can leverage numerous features of their devices to cheat. First, mobile devices give them access to the Internet, which means they can look for help in Web sites, join chat rooms, post questions in forums, etc. Even if the school were to temporarily block Internet connectivity through the local WiFi network, mobile devices allow for communication with remote or co-located peers through alternative interfaces, such as telephony, 3G/4G, Bluetooth, etc. Furthermore, students can retrieve helper material previously cached on the devices, or run certain applications to allow them to solve certain problems (e.g., a calculator). In fact, all these operations are possible, because personal mobile devices were designed to provide their owners a great degree of flexibility in terms of extensibility and customization. The downside, however, is that such devices cannot be constrained to operate under restrictive scenarios, such as the exam use case just described. A variety of other usage scenarios face similar challenges. For a broader discussion about this topic, we refer the reader to Santos et al. [19].

2.4 The concept of trust lease

The trust lease abstraction [19] aims to fill this gap. The key idea of this primitive is to allow applications to issue requests to the OS in order to restrict specific device functions for a limited amount of time. Such functions include, for example, access to the network interfaces, execution of certain applications, etc. The scope of these restrictions is such that, while a trust lease is active, the user will not be allowed to

override them until the trust lease expires. Thus, because trust lease requests are issued by applications, this simple primitive constitutes a building block that allows applications to restrict mobile devices according to the security requirements imposed by the specific usage scenario. To demonstrate this potential, in Section 4 we show how to leverage trust leases within the BYOD exam use case to prevent students from cheating.

An important aspect to highlight is that a trust lease cannot be issued without explicit approval by the user. This is because trust leases force the user to temporarily reduce some of his privileges on his personal device. For this reason, if a given application issues a trust lease request to block access to certain resources, the user is first notified about the resources to be constrained and duration of the trust lease, and must first approve these conditions before the trust lease can be activated by the operating system. Thus, a trust lease can be seen as a contract between an application and the user in which the user agrees to restrict certain functions of the device for a certain amount of time as requested by the application. We call such applications *strapp*s.

Note, however, that, as in any other contract, the user is responsible for his actions, and user mistakes may bring negative effects. In particular, it can happen that a user overlooks the restriction conditions of a trust lease and inadvertently accepts it without full awareness of its effects. In such cases, the consequence is that some functions of the device will be blocked temporarily, and the user will have to wait until the trust lease expires in order to regain complete control of his device. In the worst case, this feature can be exploited by malicious strapps aiming to cause harm by blocking some of the device's functions for a large amount of time, thereby causing a denial of service. To achieve this, the malware may try to induce the user to accept a trust lease with a large expiration date. To reduce the negative effects of such attacks, before granting a trust lease, the OS checks the requested trust lease duration against a maximum value defined as an OS configuration parameter.

2.5 Trust lease lifecycle

The OS mechanisms provided by TrUbi (see Figure 1) aim to govern the lifecycle of trust leases. A trust lease comes into existence by explicit request of a given strapp application. Essentially, the strapp invokes a system call (startlease) which results in the activation of the trust lease and subsequent change in the device state. The device enters the *restricted mode* in which some functions are temporarily disabled. Eventually, the trust lease expires, and these restrictions are removed, causing the device to switch back to *unrestricted mode*. The signature of startlease is shown below:

$$\text{startlease}(p, f) \rightarrow l_{id} \mid FAIL$$

This system call takes a restriction policy p and a callback function f. The restriction policy specifies a set of *restriction rules* aimed to define which functions must be disabled, and a set of *termination rules* for specifying the termination conditions of the trust lease. The system call invocation can either succeed (returning a trust lease id l_{id}) or fail. In fact, before the trust lease can be issued, the OS requires the user to approve this request. If the request is denied the trust lease request is aborted. Otherwise, the OS instantiates a trust lease and applies the necessary restrictions. When the trust lease expires, before switching back states, the system invokes the callback function

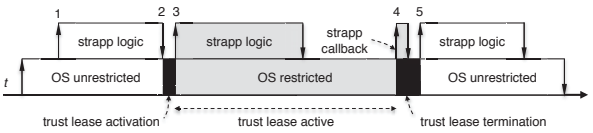

Figure 2: Timeline of a typical trust lease.

f, which enables the strapp to perform any cleaning operations (e.g., encrypt data). Figure 2 illustrates these operations: strapp starts (1), strapp invokes startlease (2), OS enters restricted mode upon user authorization (3), callback executed upon termination event (4), and OS leaves restricted mode (5).

2.6 Restriction rules and termination events

Restriction rules specify the system state that must remain unmodified while a trust lease is active. For example, a restriction rule "Network" may state that the network access must be disabled since the trust lease starts until it expires. In general, restriction rules may be designed to control access to peripherals (e.g., network, or camera), system services (e.g., screenshot service, etc.), system settings (e.g., muting the device, enabling bluetooth), or execution of applications. However, the specific restrictions supported in the system are implementation-dependent. In Section 3.2, we present the rules currently supported by our TrUbi implementation.

Termination events trigger the termination of an active lease, and can be delivered in one of two ways. One way is by defining *termination rules* in the restriction policy. Typical termination rules define *time* conditions, e.g., a timeout (T_{out}). Additionally, they can be based on *location* conditions, which confine trust leases to geographical areas defined, e.g., by GPS coordinates or vicinity to WiFi access points. An alternative method for terminating a lease is by explicit invocation of the system call stoplease. This system call aims to allow the strapp that owns the trust lease to terminate it before the termination rules occur. To invoke stoplease, the strapp only needs to provide the trust lease id (lid) as parameter:

$$\text{stoplease}(l_{id}) \rightarrow OK \mid FAIL$$

The reason for disabling a trust lease ahead of time is application specific. By judiciously defining termination rules and invoking stoplease, a strapp can generate a rich set of termination events. A strapp can also express *exceptional* termination conditions. Analogously, strapps can invoke startlease to start leases when only certain location, temporal, or exceptional conditions hold.

2.7 Trust lease attestation

For some usage scenarios, strapps may need to convince an external party that the device operates in restricted mode and possibly communicate securely with that party. In particular, if a trust lease expires while data is in transit, the external party may need to be immediately notified and stop further data transmission in order to prevent security breaches. Taking some ideas from the trusted computing world, we address this need by providing a *trust lease remote attestation* mechanism, which relies on two primitives:

$$\text{quotelease}(n) \rightarrow q \mid FAIL$$
$$\text{verifylease}(q,n) \rightarrow info \mid FAIL$$

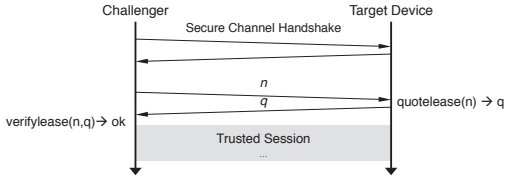

Figure 3: Trust lease attestation protocol.

Based on these two primitives, the strapp developer can implement a simple protocol between strapp and external party in order to communicate securely and with guarantees that the trust lease is active (see Figure 3). First, the strapp opens a secure channel with the server (e.g., using SSL). Over the secure channel, the server challenges the strapp by sending it a nonce, receiving a quote, and checking the quote. If the quote is valid, the strapp endpoint is trustworthy and the communication is safe. To remotely detect that the trust lease ended, the strapp developer can set a trust lease callback that closes the SSL connection, thus signaling the external party that the trust lease has expired and no more data should be sent. By combining trust lease attestation with SSL, it is therefore possible to create a *trusted session* between both parties.

Detailed trust lease attestation protocol: We now explain in more detail the cryptographic operations that enable an external party (challenger) to determine the trust lease state of a target device. We assume that each OS is provisioned with a unique keypair and that the private key is securely stored by the OS. The public key is certified by the device manufacturer. The key certificate indicates the version of the OS and whether the OS supports trust leases. We refer the reader to Figure 3.

Under these assumptions, the basic protocol is as follows. The challenger sends a nonce n to the target strapp, which invokes quotelease passing n as input. The OS simply returns the report message: quote= $\langle m||n\rangle_{K^-}^-, C(K^+)$. The quote contains a signature of its current restriction mode m (restricted or unrestricted) concatenated with n. The signature is produced with the target device's private key K^-. The certificate of this key is included in the quote. The quote is then returned to the challenger, which invokes verifylease to check the quote. The quote is valid if: (i) the nonce in the quote matches n, thereby detecting replay attacks, and (ii) the signature checks against K^+ enclosed in the certificate and the certificate indicates that the OS supports trust leases, which verifies that m returned in the quote is meaningful.

This basic protocol can be advanced in three ways. First, the challenger may need to validate the identity of the strapp, and the conditions of the trust lease. To enable this, the quote can include the strapp identity (name, version, and signature) and details about the trust lease (restriction policy and termination conditions): quote= $\langle m||id_A||p_{tl}||n\rangle_{K^-}^-, C(K^+)$. This extra information is returned by quotelease and can be validated by the challenger. Second, the challenger may need to determine if the endpoint strapp is indeed the owner of the lease. To ensure this, quotelease only returns a quote to the strapp that currently owns the trust lease, otherwise it sends an error message. Third, if there are concerns about the OS identity, the protocol can leverage trusted computing hardware. In this case, the quote must be extended with an additional signature issued with the platform's identity key. This new signature covers the original nonce n concatenated with a hash of the OS calculated upon boot. This hash enables the challenger to validate the OS identity. To provide this functionality, a unique identity key must be embedded into the device by the device manufacturer.

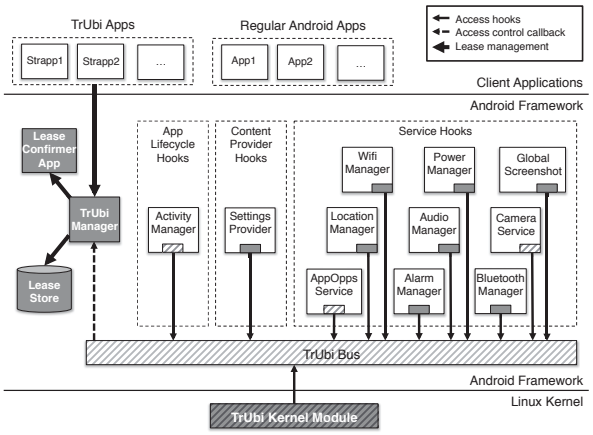

Figure 4: TrUbi implementation on Android.

2.8 Lease persistence and conflict resolution

Maintaining persistent trust lease state is a critical function of TrUbi. In fact, keeping track of active trust leases in volatile memory alone would constitute an important vulnerability. In such conditions, if the device were rebooted, information of active trust leases would be lost, resulting in a potential violation of terminator conditions. We address this problem by backing up all relevant trust lease state data persistently. When the phone bootstraps, TrUbi first checks for persistent trust lease data and, if necessary, resumes any previously interrupted trust leases.

Another important issue that TrUbi must handle is the case where different strapps submit overlapping trust lease requests with conflicting restriction rules. In general, restriction rules are conflicting whenever they act upon the same resource in order to impose irreconcilable states. For example, two trust leases p_1 and p_2 which require exclusive execution rights for strapp S_1 and S_2, respectively. In this case, a conflict clearly exists because only one of the strapps can be given exclusive right to execute on the device. To address this problem, we opted to reconcile conflicting requests by implementing a simple *first come, first served* conflict resolution policy. In other words, if a trust lease is active and a conflicting trust lease is issued, the latter will only be enforced after the former expires or is explicitly terminated.

3 IMPLEMENTATION

To validate our system design, we implemented TrUbi on Android-based mobile devices. To facilitate the implementation of system-wide hooks (explained in Section 3.3), we use an ASM implementation from November 2014, which requires Android KitKat. TrUbi source code was written in Java (10KLOC) and C (0.2KLOC).

```
// get reference to the TrUbi Manager
IASMAPP trubi = IASMAPP.Stub.asInterface(
                    ServiceManager.getService("TRUBI"));

// parse lease from XML file on "path"
LeaseHandler handler = new LeaseHandler(path);
Lease lease = handler.parseLease();
try {
  // instantiate the trust lease in the system
  int leaseId = trubi.startLease(lease, new TrustLeaseListener()
        {
    int callback(TerminationEvent e) {
        // invoked before the lease ends
    }
  });
} catch (TrUbiRemoteException e) {
    // lease creation failed
}
```

Listing 1: Java code fragment to start a lease.

T	Restrictors	Description	W
A	ExecApps	Prevents the execution of applications.	×
R	Network	Prevents access to the Internet.	×
	Camera	Denies access to the camera.	×
	Phone Call	Prevents making or receiving phone calls.	×
	SMS	Prevents sending or receiving SMS / MMS messages.	×
	Screenshot	Blocks screenshots.	
S	Microphone	Mutes the microphone.	
	Sound	Force the sound to be muted.	×
	Brightness	Dims device screen brightness and prevents changes.	×
	Airplane Mode	Disables all wireless transmission functions of the device.	
	LED	Forces the camera built-in LED to be disabled.	×
	Bluetooth	Disables Bluetooth communication.	
	Time	Prevents changes to system time.	

Legend: T: type of restrictor, A: application, R: resource, S: settings, W: app whitelist.

Table 1: Restrictors supported by TrUbi.

3.1 TrUbi internals

Figure 4 presents the blueprint of our implementation. TrUbi rests upon a standard Android stack, which comprises a modified Linux kernel (bottom layer), the Android framework, which includes the Android runtime, libraries, system services, and system apps (middle layer), and client applications (top layer). TrUbi adds a set of specific components (shown as dark shaded boxes) which implement the trust lease reference monitor.

The core of the system is the *TrUbi Manager*, a system service that enforces trust lease restrictions, coordinates the trust lease lifecycle, and manages transitions between OS restriction modes. The TrUbi Manager serves trust lease calls issued by strapps, which run on the client application layer alongside regular Android applications. To interface with the user and obtain trust lease authorizations, the TrUbi Manager launches the *Lease Confirmer* system app, which displays lease conditions to the user and provides buttons for accepting or declining them. The TrUbi Manager keeps persistent state of ongoing trust leases on the *Lease Store*.

To enforce access control restrictions, the TrUbi Manager relies on a set of *hooks* spread across the system. Hooks monitor every access to restricted objects and trigger callbacks to the TrUbi Manager in order to perform access control decisions. Some of these hooks are placed in the Android framework (represented in Figure 4 as small shaded boxes inside pre-existing Android system components). Their goal is to control relevant events involving the lifecycle of applications (e.g., launching an app), access to content providers (e.g., system settings), or access to system services that control certain resources (e.g., sound, camera). Some other hooks are located in the Linux kernel (e.g., for restricting network access), more specifically in the TrUbi kernel module. To streamline the communication between hooks and TrUbi manager, access events are routed from the hooks to the TrUbi Manager via a dedicated system service named *TrUbi Bus*.

3.2 TrUbi API

To expose its services to strapps, TrUbi provides a simple API which implements the trust lease primitive introduced in Section 2. Listing 1 shows a sample Java code of the steps that a strapp developer would need to follow to start a trust lease, namely: obtain a reference to the TrUbi Manager, read the restriction policy from an XML file into a local object instance, and invoke method `startLease` with two parameters: the restriction policy object, and the trust lease termination callback. Alternatively to XML files, the restriction policy can be initialized programmatically. To terminate the trust lease explicitly (not shown in Listing 1), the developer must invoke `stopLease`.

To accommodate a large range of restriction policies, we implemented 13 different restrictors as listed in Table 1. For each restrictor, we indicate: the type, name, a brief description, and whitelist support. The type indicates the restrictor categories: application, resource, or settings. TrUbi provides one application restrictor, five resource restrictors (network, camera, phone call, SMS, and screenshot service), and seven setting restrictors (microphone, sound volume, screen brightness, airplane mode, LED, Bluetooth, and system time). By default the restrictions described in the table will be applied to all applications installed in the system, comprising the strapp that owns the lease. Some restrictors, however, can receive a white-list designating applications that will not be affected by the restrictor. The fourth column of Table 1 places mark "×" if a whitelist is supported.

The trust lease remote attestation primitives follow the cryptographic protocols specified in Section 2. For symmetric cryptography we use AES-256, and for digital signatures RSA-1024 and SHA-2. To secure the cryptographic keys, we use Android's Key Store. In our current implementation we do not verify the integrity of the kernel using trusted computing hardware. The reason is that, in order to ensure compatibility with ASM (see the next section), we used the same hardware testbed as ASM's authors did (Nexus 4 phones). However, this hardware offers no access to trusted computing primitives. Nevertheless, this limitation is not fundamental.

3.3 System hooks

To implement the restrictors indicated above it was necessary to modify the Android OS—both at framework and kernel levels—in order to override the access control decisions taken by Android itself. However, modifying the OS in such a range is not only complex and error-prone, but poses serious obstacles to the development and incorporation of new restrictors in the future. To tackle this challenge, the first approach we considered was simply to revoke Android permissions of the restricted resources. However, the spectrum of trust lease restrictions is wider than that covered by Android permissions, e.g., controlling app execution, system clock.

As a result, to reduce the complexity of this task and facilitate the extensibility of our system, we implemented a hooking mechanism based on Android Security Modules (ASM) [11]. ASM is a security framework that places hooks in the Android system and allows application developers to override the system's default access control mechanisms by implementing custom access decisions in their application code. ASM provides a backbone for placing and handling hooks in both the framework and kernel: the TrUbi Bus is implemented by the ASM Service, the TrUbi kernel module is based on the ASM LSM for intercepting kernel hooks, and some framework hooks are provided by ASM (in Activity Manager and Camera Service).

Note, however, that ASM alone is insufficient to implement TrUbi. Figure 4 illustrates the extent to which we used ASM: the grey background indicates that a TrUbi component was natively implemented by us, diagonal stripes under white background means that it was based on an unmodified ASM component, and diagonal stripes under grey background denotes that it was modified from a pre-existing ASM component. In particular, we had to implement various framework hooks unsupported by ASM, modified the ASM kernel module (to control suspension and termination of application processes), and built the remaining TrUbi components from scratch. Next, we provide more details of the most technically relevant features that had to be addressed specifically for TrUbi.

3.4 Blocking and resuming applications

TrUbi provides a restrictor named ExecApps, which prevents all but the user-level apps specified in a whitelist from executing. When a trust lease starts, TrUbi checks all running app processes that are not identified in the restrictor's whitelist, and suspends them until the lease terminates. While the lease is active, TrUbi forbids the execution of new processes for the apps not in the list. When the lease terminates, TrUbi removes the process execution restriction and resumes all suspended processes. Resuming application processes aims to reduce the impact on user experience, since the app's previous state will be recovered after the lease terminates.

To ensure that the entirety of an application's state is properly suspended and resumed, TrUbi must keep track of *all* the processes that belong to a given application. In fact, Android assigns a main process to each application that is launched, and from this process, the application code can spawn additional background processes using the system call interface. However, keeping track of all processes is relatively challenging. Although the ActivityManager controls the lifecycle of applications' main processes, it does not supervise their background processes. For this reason, we must suspend / resume applications' processes from the Linux kernel. When a trust lease starts, the TrUbi Manager sends a list of process UIDs that are allowed to execute to the TrUbi kernel module. From the kernel, TrUbi loops through the system's process list and sends a SIGSTOP signal to every process whose UID belongs to the userspace and is not present in the list. When the lease finishes, TrUbi sweeps the process list like before, but this time, sends a SIGCONT signal to the previously suspended processes, effectively unfreezing them. To implement these operations, we extended ASM's middleware-to-kernel communication mechanism.

To prevent apps from running while a lease is active, we simply used pre-existing ASM hooks placed in the ActivityManager. Some of these hooks supervise the lifecycle of applications' main processes and components. TrUbi leverages these hooks to deny operations that result in the execution of new application processes: start Activities and Services, bind to Services, and intent broadcasts to apps (which cause the execution of Broadcast Receivers). Since background processes can only be spawned from main processes, by blocking the latter TrUbi also blocks the former.

3.5 Keeping track of time and location

In order to satisfy the trust lease conditions imposed by strapps, TrUbi needs to securely keep track of time and location. To this end, given that the OS is part of TrUbi's trusted computing base, we rely on mechanisms provided by or deployed into the OS. In particular, with respect to time keeping, TrUbi uses a system background timer thread that terminates trust leases at the specified time. When a lease is about to be enforced, TrUbi calculates the absolute expiration time of the lease, and saves this information persistently. TrUbi keeps logging the time left for the trust lease expiration so as to prevent device reboots from interfering with the duration of the lease. Thus, even if a malicious user reboots his device, TrUbi ensures that the lease is resumed at system startup therefore ensuring full lease enforcement. Additionally, to prevent the user or applications from modifying the system time through the Settings app, TrUbi leverages hooks placed in the System Clock and Alarm Manager system services.

With respect to the location, TrUbi implements two location-based trust lease terminators: gps and wifi. For location tracking, TrUbi introduces changes to Android's WiFi and Location managers. For GPS, TrUbi waits for 30 location modification notifications indicating the user is outside the lease's perimeter before actually terminating the lease. For WiFi, a background thread scans surrounding networks every 10 seconds, which are identified by their MAC, and terminates the lease if the required WiFi networks are not within reach after 30 seconds. For GPS and WiFi, TrUbi again leverages its trust lease persistence mechanism in case of device rebooting. Upon system startup, TrUbi waits 30 seconds to assess the surrounding WiFi networks and 90 seconds for GPS signals, before deciding whether the lease should be revoked or not. Time and space values are provided to the terminators as parameters. By default, TrUbi defines a maximum timeout.

4 USE CASE

To demonstrate TrUbi in a concrete usage scenario, we prototyped a use case strapp named mExam, which was written in Java in about 2.7KLOC. This application provides support for a BYOD mobile scenario whose goal is to replace printed exam copies with digital exam submissions which can be performed from students' own mobile devices. In this context, exams must still be answered in examination rooms in the presence of a professor.

To this end, we want to ensure that from the moment the student enters the examination room, after being properly authenticated by the professor (before the exam starts), until the moment the student exits the room (upon completing the exam), students' mobile devices must be constrained to the execution of a *single* trusted application—the mExam strapp—responsible for loading the exam questions from a trusted server, displaying them on students' devices, reading students' local input, and submitting their responses back to the trusted server.

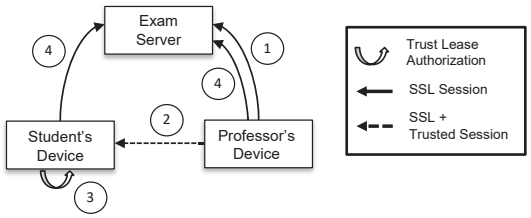

Figure 5: mExam workflow.

To prevent cheating (e.g., by accessing the Internet on a browser), TrUbi will be crucial by ensuring that these restrictions cannot be overridden by students.

Figure 5 illustrates the steps implemented by mExam in order to address these requirements. The first step represents a preparatory operation in which the professor retrieves the identities and credentials of enlisted students assigned to his examination room. As students enter an examination room, the professor authenticates each student and both their devices engage a protocol in order to enforce the necessary restrictions on the student's device. Communicating over NFC, the professor's device sends the required restriction policy to the student's device (step 2), and waits until the student accepts the trust lease (step 3). The restriction policy specifies an ExecApps restrictor granting exclusive execution rights to the mExam strapp package. After approving the policy, the device enters the restricted mode, and the student may take his place in the room. To start the exam, the professor signals the exam server to allow the devices of students in the room to retrieve the exam questions from the server (step 4). The trust lease expires when one of five termination conditions occur: the exam time limit is reached, the student submits the exam, the student aborts the exam, the professor terminates the student's exam for some reason, or the student exits the room and steps away from the local WiFi range.

Next, we briefly discuss relevant attacks that students can try to launch and respective defense measures implemented by mExam.

Device swapping: Students may leverage a second device to cheat. Although TrUbi does not offer a sound solution to this issue, applicational mechanisms coupled with human invigilation can allow for a professor to detect such frauds. In mExam, we implemented a brightness detection solution, that allows the professor to be notified when this level drops, possibly meaning the device in question is concealed. By doing so, we offer professors a justifiable reason to enquire a student about his device's state. At the same time we also avoid situations where a student swaps the device and looks for answers inadvertently, in plain sight of the professor. In this case, the professor would be notified the legitimate device was hidden, and the device the student was using was not the one authorized.

Malicious strapp: Students could be induced to install a malicious mExam application which could compromise the normal course of the exam, e.g., by terminating the trust lease ahead of time without students' awareness, which would prevent them from submitting their responses. To prevent this problem, mExam relies on the app's signature verification feature implemented by TrUbi's trust lease negotiation mechanism. This means that during lease negotiation, the mExam app installed on each student's device has its signature checked against the signature of a mExam application endorsed by the school. This process allows the professor to detect the installation of illegitimate mExam strapps.

WiFi spoofing: By spoofing a WiFi signal a student could lead mExam to assume he was still taking the exam in case he left the examination room without terminating it explicitly. Regarding this issue, we assume it is the responsibility of the professor to ensure students submit / abort the exam through mExam before leaving the room, thus preventing further submission of exam answers. In this case, professors benefit from mExam's termination notification, i.e., a callback function similar to the one in Section 2.7, that notifies the professor every time a student finishes his exam.

User impersonation. Another potential attack consists of allowing a student physically located outside the premises of the examination room, yet covered by the local WiFi network, to enrol and submit an exam on behalf of another student. However, given that students must be authenticated through NFC, i.e., a close proximity technology, we force the presence of the student before the professor, thus eliminating the possibility to perform this attack.

5 EVALUATION

Our evaluation of TrUbi aims to determine its impact to the functionality of existing applications (Section 5.1), to the performance of applications (Section 5.2), and to the user experience (Section 5.3).

5.1 Impact to the functionality of applications

To assess the impact of TrUbi to the functionality of applications we need 1) to determine whether TrUbi's trust leases are effective when applied to real applications, and 2) to study potential side-effects to applications when subjected to trust lease restrictions.

Methodology: To achieve our goals, we collected a test suite of unmodified real-world applications and applied trust leases to them under different patterns. For these patterns, we changed both the type of restrictor applied by the trust lease and also the relative point in time where the trust lease was applied: before, during, or after the application has accessed a restricted resource.

We selected applications that require access to resources that TrUbi can restrict: process, network, camera, phone call, SMS, microphone, sound, brightness, LED, and Bluetooth. To test each restrictor, we use a mix of: system apps from the vanilla Android distribution (e.g., System Camera), and apps from the Google Play app market [9]. We chose apps that were popular and highly rated by March 2015. Each restrictor was tested with 10 apps, except for the restrictors: airplane mode, time, and screenshot. The reason for not testing these restrictors with multiple apps is that they can be triggered only through a limited set of entrypoints, such as the system settings, the power button, or the Android Debug Bridge. In total, we collected 87 different apps. Most of these apps were used to test a single restrictor. To test the process restrictor, we reused 9 of these apps.

The tests were performed on two Nexus 4 devices, featuring a quad-core 1.5 GHz CPU, 2 GB of RAM, 16 GB of memory, 802.11 WiFi interface, 768 x 1280 display, 8MP camera, 3264 x 2448 pix, and a LED flash. Both devices were flashed with Android 4.4.1 AOSP patched with TrUbi code. For communications, we used WiFi.

Restrictors	Size S / 3 / T	ABL B / K / F	ADL C / I / D	AAL L / F / R
CPU	1 / 9 / 10	– / – / 10	– / 10 / –	10 / – / –
Network	– / 10 / 10	– / 10 / –	9 / 1 / –	4 / 6 / –
Camera	1 / 9 / 10	– / 10 / –	10 / – / –	– / – / 10
Phone Call	1 / 9 / 10	10 / – / –	10 / – / –	10 / – / –
SMS & MMS	1 / 9 / 10	10 / – / –	10 / – / –	9 / – / 1
Microphone	– / 10 / 10	10 / – / –	10 / – / –	9 / – / 1
Sound	– / 10 / 10	10 / – / –	5 / 4 / 1	10 / – / –
Brightness	– / 10 / 10	10 / – / –	2 / 8 / –	9 / – / 1
LED	– / 10 / 10	10 / – / –	– / 10 / –	– / 10 / –
Bluetooth	– / 10 / 10	10 / – / –	5 / 5 / –	5 / 5 / –
Total:	**4 / 96 / 100**	**70 / 20 / 10**	**61 / 38 / 1**	**66 / 21 / 13**

Legend: Size = test suite size (S = number of system apps, 3 = number of apps, T = total number of apps); ABL = access before lease (B = access to resource blocked, K = app killed, F = app frozen); ADL = access during lease (C = app state and GUI consistent, I = app state and GUI inconsistent, D = app dies); AAL = access after lease (L = seamless access, F = requires refresh operation, R = requires app restart).

Table 2: Trust leases on real mobile applications.

Trust lease effectiveness and side effects: For all tested applications, TrUbi was effective at enforcing trust lease restrictions, i.e., whenever a trust lease is active, applications have no access to the resources that are constrained by the trust lease. Table 2 summarizes our findings for three complementary cases.

Under "Access Before Lease" (ABL), the trust lease is activated after the tested app has started to access a resource. As expected, we observed that the access to the resource by the application was blocked immediately since the activation of the trust lease. The way how this access was blocked by TrUbi was manifested in three different forms: some applications stop having access to the resource (1), others are killed (2), and others are frozen (3). Form 1 applies when TrUbi can intercept all operations issued by the application to a given resource. Examples include the microphone or the LED flash. In such cases every access is preceded by an access control decision that TrUbi can control. In contrast, for resources where an access decision is made only when the application uses the resource for the first time, TrUbi must kill the app in order to ensure that no further accesses can be carried out by said application (form 2). This policy is implemented for 10 camera apps and 10 networking apps. For trust leases that restrict app execution, freezing unauthorized apps is expected (form 3).

For the case "Access During Lease" (ADL), we activated the trust lease first, and only then launched the application, which would then attempt to access the restricted resource. In this case, despite the fact that applications had been given legitimate access to the resource (explicitly declared in the manifest and authorized by the user), TrUbi was effective at suspending applications' access rights to that resource while the trust lease was active. However, different applications reacted differently to this operation. For example, for the camera, 9 out of 10 apps terminate; a single app remains executing, but shows a black screen to the user and takes no photos. In other cases, applications provide different outputs to the user. For example, regarding the microphone, 9 apps tape silence, while a single app notifies the user about the lack of input. As a result of lease restrictions, we detected in some cases side effects to the consistency between the internal state of the app and its GUI. In particular, we experienced inconsistencies in 38 apps. For example, if the user turns on the Bluetooth and a lease is active, an application will misleadingly show the Bluetooth status as enabled. In contrast, 61 apps update their visual interfaces consistently. For instance, all phone dialer apps refresh their GUI properly by disabling calls. Under ADL, one app crashed due to a null pointer exception.

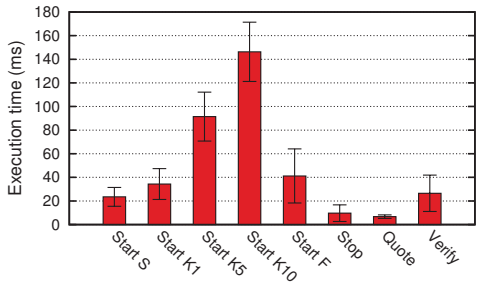

Figure 6: Execution time of API calls.

Lastly, "Access After Lease" (AAL) covers the case where an app attempts to regain access to a resource that had been blocked by a trust lease. If the trust lease has expired, the app must be able to regain access. Under AAL, we detected that, in most cases, applications managed to regain access to the resource seamlessly: in 66 out of 100 tests. In other cases, however, the user must perform some manual operation to "refresh" the GUI, e.g., by pressing the back button and restarting an application activity. A refresh operation is required by 21 apps. In more extreme cases, apps must be entirely restarted. This occurs in 13 cases: 10 camera apps, and three apps handling microphone, brightness, and SMS. Note that the reason for both this abnormal behavior and the inconsistencies detected in ADL cases is that trust leases modify the functionality that applications are expecting from the Android API. As a result, in order to work properly with TrUbi, applications must be adapted to handle events of revocation and reacquisition of resource access.

5.2 Impact to the performance of applications

To study the performance overheads of TrUbi, we measure the execution time of the API calls, and the time to setup trust sessions. All measurements were taken from Nexus 4 devices using microbenchmarks. Unless stated otherwise, for each experiment, we report mean and standard deviation of 50 runs. To avoid performance reading noise from battery consumption, devices remain connected to the power plug during our experiments.

Performance of TrUbi API: Overall, we find that the performance overhead is small. Figure 6 presents our evaluation results of the TrUbi API primitives. The relatively large standard deviation values are mainly caused by the execution of Android's garbage collector. As for the start lease primitive, it was tested under different conditions. Figure 6 shows that the minimum execution time of start lease is 23.5 ms and takes place when we restrict one resource without needing to kill or to suspend applications (Start S). If we grow the number of applications that need to be killed when starting the trust lease (Start Kn), the execution times increase proportionally to the number (n) of killed applications: between 34.32 and 146.3 ms, for 1 and 10

Trust Lease Installer Times				
		SSL phase	TRB phase	Total
WiFi	Time	0.32s ± 0.04	0.78s ± 0.07	1.10s ± 0.09
	Percentage	29.09%	70.91%	100%
	Round Trips	2	4	6
NFC	Time	0.70s ± 0.05	0.89s ± 0.05	1.59s ± 0.06
	Percentage	44.03%	55.97%	100%
	Round Trips	8	11	19

Table 3: Performance of attestation and lease deployment.

apps, respectively. If the trust lease freezes all services and apps, i.e., applies the process restrictor, the start lease takes 41.18 ms (Start F).

As opposed to start lease, all other TrUbi API primitives execute in constant time since they are independent of the restrictors being enforced. As shown in Figure 6, the primitives to stop, quote, and verify a trust lease take 9.66, 6.7, and 26.54 ms, respectively. Comparing the remote attestation primitives, we see that quotelease is more efficient than verifylease. The reason is that the former performs a single digital signature while the latter verifies two signatures: from a quote, and from a certificate (see Section 2.7).

Performance of attestation and lease deployment: According to our experience, a common operation performed by strapps is the negotiation of trust leases. This operation leads one party to first attest a second party, and then, upon successful attestation, instruct that second party to deploy a specific trust lease. We call this operation trust lease installer. Table 3 shows the total execution time of this combined operation when the communication takes place over WiFi (1.1 seconds) or NFC (1.59 seconds).

To better understand these numbers, we break them down into its two constituent parcels: SSL handshake (SSL) and TrUbi handshake (TRB). The SSL handshake is the first action to be performed by a trust lease installer and takes two round trips to execute. The TrUbi handshake ensues the SSL handshake and comprises two round trips for (1) sending the restriction policy and starting the trust lease, and (2) issuing and verifying quote signatures. With WiFi, the SSL handshake takes about 29% of the total time, whereas with NFC, this contribution increases to 44%. This is explained by NFC's slower connection setup time but also because of the difference in the number of round trips. Because NFC messages have a smaller payload, we had to break down messages, resulting in a performance penalty, where NFC takes over two times longer to complete the SSL phase, when compared with WiFi. The difference between the number of round-trips and performance is not linear, because NFC's latency is lower than WiFi's. On the other hand, because TRB involves smaller messages, the difference between NFC and WiFi is smaller.

5.3 Impact to the user experience

We concentrate on addressing two main questions, in particular: (Q1) whether the ability of TrUbi to restrict the functionality of devices undermines users' confidence in the security of the system (which could deter the adoption of TrUbi in the future), and (Q2) whether potential users find the trust lease paradigm to be intuitive and TrUbi-based applications easy to use.

Methodology: To address these questions we carried out two studies, both of which in the context of the mExam scenario. In one case, we conducted a feasibility study consisting of two online surveys, one for students and another for professors. Together, these surveys comprised a pool of 34 questions covering a broad range of topics, such as security concerns and user incentives. We collected 20 answers from students between 18 and 28 years old; 13 male and 7 female. Additionally, we collected 10 professor answers; 9 male and 1 female.

In a second case, we conducted a hands-on usability study where a group of users interacted with a TrUbi-enabled device running mExam. For 20 minutes, users were instructed to answer a small exam, as well as trying to cheat by circumventing the trust lease mechanism. We conducted 10 tests with engineering students between the ages of 21 and 25; 7 male and 3 female; and with different levels of proficiency in mobile devices: some students can send / receive texts and phone calls, others manage apps and disk space, and some even know how to develop mobile apps. For these tests we used devices flashed with TrUbi. During the tests we aimed to find out whether users understood the trust lease model, and how they felt about its intuitiveness and ease of use. We were also interested in seeing whether users' perceptions on the trust lease model would change after a hands-on experience with a TrUbi-enabled device.

Q1. Confidence: From the feasibility study, we learned that 75% of the subjects were favorable to accepting the restrictions imposed by trust leases in the context of an exam, as they believed such restrictions were clearly required for that particular application scenario. Students felt comfortable with TrUbi's privacy guarantees because we explained that professors cannot covertly restrict settings without students' consent. Additionally, we explained professors cannot access students' private data leveraging trust leases.

From the usability study, the vast majority of students (90%) was not distressed after confirming they couldn't access other functions of the device other than the mExam strapp itself, as they understood the purpose and the time-limited duration of such restrictions. Some of these students (30%) also answered the online survey, and declared that after using mExam, their skepticism towards TrUbi's effectiveness has changed completely.

Q2. Usability: Overall, users were able to clearly understand the concept of trust leases. However, some complaints were reported mostly related with the user interface of our TrUbi implementation. First, users had difficulty interpreting the lease conditions that TrUbi displays in a pop-up window every time a trust lease is activated. We learned that the text we were using to convey these conditions was too verbose and should ideally be replaced by some suggestive icon. Second, users expressed divergent opinions with respect to the pop-up authorization windows every time a trust lease is requested by a given application. Some users expressed the opinion that, after authorizing an application to request a specific trust lease for the first time, this authorization decision remains implicit for future trust lease requests by the same application. Other users, however, declared it was important for them to be prompted for every trust lease request. Based on these different reactions, we modified our TrUbi implementation to let users configure which option they prefer. In particular, after the first trust lease authorization, users can opt to receive only notifications that the trust lease has been reactivated without the need to provide explicit input.

6 RELATED WORK

A large body of work aims to improve data security from untrusted Android applications. Some systems aim to protect users' privacy through data shadowing [27], application workflow control [13], or information flow control [7, 22]. Other proposals refine Android's permission model in order to provide users with advanced features, e.g., assignment of different permissions to apps' sub-components [24], or definition of constraints on individual app resource usage [25]. Another group of proposals improves access control mechanisms by mitigating privilege escalation, more specifically confused-deputy and collusion attacks. Existing approaches may leverage static analysis techniques [10], runtime enforcement techniques [8, 16], or both [26]. There is also extensive work in implementing mandatory access control (MAC) for Android [5, 21]. ASM [11] and ASF [3] provide middleware and kernel-level hooking APIs to implement user-level access control models. All this work is complementary to ours, since these systems assume users are fully trusted.

As for systems that impose restrictions to the device owner, we highlight Digital Rights Management (DRM) and Trusted Execution Environments (TEE). DRM systems such as OMA DRM [14] compliant systems, or Porsha [15] aim to protect copyrighted content (e.g., preventing unauthorized copies). TEE systems [12, 18, 20] enable secure execution of small pieces of app code from potentially compromised OSes. Compared to TrUbi, DRM is narrower in scope since it focuses on content protection only. TEE enforces restriction policies for app modules, whereas TrUbi covers full-blown apps and services.

The closest systems to ours aim to allow for the maintenance of security policies on mobile devices by trusted third parties. Mobile Device Administration (MDA) systems, such as Samsung KNOX [18], Android for Work [2], Android's Device Administration API [1], or DeepDroid [23], enable IT managers to install and maintain security policies on company-owned devices. Such policies constrain the kind of operations that users are allowed to perform, and are usually employed within organizations to prevent security breaches. Recent work [4] allows for ARM TrustZone-based mobile devices to be regulated by centralized servers deployed within restricted spaces, such as federal or corporate offices, or examination halls. CRePE [6] is a related system in which multiple trusted third parties can maintain different sets of policies on a given device. MOSES [17] allows for the definition and enforcement of security profiles that implement different operation modes on smartphones (e.g. Work, Private, etc). In all these systems, however, users must give away full control of their devices on behalf of one or more third parties which may hinder their adoption. TrUbi complements these systems by providing the ability to negotiate restrictions and termination conditions dynamically (through trust leases) without relying on a trusted administrator. TrUbi is, therefore, more suitable for personal mobile scenarios than existing systems. Lastly, although the concept of trust lease has been introduced in prior work [19], TrUbi fully validates this new OS primitive by presenting a full system design, and demonstrating its practical implementation on a commodity mobile operating system.

7 CONCLUSIONS

This paper presented TrUbi, a system providing trust lease support for Android devices. Trust lease is a general abstraction that allows applications to enforce functional restrictions on mobile devices. To the best of our knowledge, TrUbi is the first implemented system that explores the potential of this abstraction. TrUbi provides Android developers with a simple API and expressive policy specification. We demonstrate TrUbi's potential by developing a use case application showing a novel restricted mobile scenario. Moreover, through extensive empirical evaluation, we show that TrUbi is efficient and introduces low overheads to both system and applications.

Acknowledgments: We would like to thank the anonymous reviewers for their comments. This work was partially supported by the EC through project H2020-645342 (reTHINK), and project H2020-635266 (TRACE), as well as by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 (INESC-ID).

REFERENCES

[1] Android Device Administration API Overview. http://developer.android.com/guide/topics/admin/device-admin.html. Accessed Jun 2017.
[2] Android for Work. https://www.google.com/work/android. Accessed Jun 2017.
[3] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky. Android Security Framework: Enabling Generic and Extensible Access Control on Android. In *Proc. of ACSAC*, 2014.
[4] F. Brasser, D. Kim, C. Liebchen, V. Ganapathy, L. Iftode, and A.-R. Sadeghi. Regulating ARM TrustZone Devices in Restricted Spaces. In *Proc. of MobiSys*, 2016.
[5] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In *Proc. of USENIX Security*, 2013.
[6] M. Conti, V. T. N. Nguyen, and B. Crispo. CRePE: Context-Related Policy Enforcement for Android. *Information Security*, 6531:331–345, 2011.
[7] 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 *Proc. of OSDI*, 2010.
[8] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission Re-Delegation: Attacks and Defenses. In *Proc. of USENIX Security*, 2011.
[9] Google Play. https://play.google.com/store. Accessed Jun 2017.
[10] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proc. of NDSS*, 2012.
[11] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi. ASM: A Programmable Interface for Extending Android Security. In *Proc. of USENIX Security*, 2014.
[12] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proc. of EuroSys*, 2008.
[13] A. Nadkarni and W. Enck. Preventing accidental data disclosure in modern operating systems. In *Proc. of SIGSAC*, 2013.
[14] OMA. Enabler Release Definition for DRM V2.0.1, 2008.
[15] M. Ongtang, K. Butler, and P. Mcdaniel. Porscha: Policy Oriented Secure Content Handling in Android. In *Proc. of ACSAC*, 2010.
[16] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. *Security and Communication Networks*, 5(6):658–673, June 2012.
[17] G. Russello, M. Conti, B. Crispo, and E. Fernandes. Moses: Supporting operation modes on smartphones. In *Proc. of SACMAT*, 2012.
[18] Samsung KNOX. https://www.samsungknox.com/en. Accessed Jun 2017.
[19] N. Santos, N. O. Duarte, M. B. Costa, and P. Ferreira. A Case for Enforcing App-Specific Constraints to Mobile Devices by Using Trust Leases. In *Proc. of HotOS*, 2015.
[20] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using ARM TrustZone to Build a Language Runtime for Mobile Applications. In *Proc. of ASPLOS*, 2014.
[21] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proc. of NDSS*, 2013.
[22] M. Sun, T. Wei, and J. Lui. TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime. In *Proc. of CCS*, 2016.
[23] X. Wang, K. Sun, Y. Wang, and J. Jing. Deepdroid: Dynamically enforcing enterprise policy on android devices. In *Proc. of NDSS*, 2015.
[24] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du. Compac: Enforce Component-Level Access Control in Android. In *Proc. of CODASPY*, 2011.
[25] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proc. of USENIX Security*, 2012.
[26] W. Xu, F. Zhang, and S. Zhu. Permlyzer: Analyzing Permission Usage in Android Applications. In *Proc. of ISSRE*, 2013.
[27] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proc. of TRUST*, 2011.