TrUbi
Mobile Operating System Security

Miguel Rodrigues Bento Barros da Costa
miguel.d.costa@tecnico.ulisboa.pt
Instituto Superior Técnico, Lisboa, Portugal

ABSTRACT

Mobile devices, such as smartphones and tablets, are highly integrated in today's society. Due to their mobility, and wide variety of functionalities, these devices allow their users to perform a large set of diverse tasks of their daily life. However, many usage scenarios require these functionalities to be constrained in some manner. In a cinema, for example, it is in everyones' best interest that the mobile device is muted. We present TrUbi, a system that allows for temporary restrictions of mobile devices by forcing some of their specific functions to be locked for a limited amount of time, e.g. sound muted, network access blocked, etc. To provide this restriction capability, TrUbi enforces global security policies by implementing an operating system primitive named trust lease. Our TrUbi prototype, implemented on Android OS, can efficiently prevent real-world apps from accessing devices' functions constrained by TrUbi and enable new mobile utilization scenarios that are currently unsupported, to the best of our knowledge, by existing mobile operating systems. Lastly, the TrUbi prototype was also tested in terms of performance which registered a negligible impact on the overall system performance and energy consumption.

1. INTRODUCTION

Increasingly, mobile devices like smartphones and tablets pervade our everyday lives. With a wide variety of functions and uses, these devices are a helpful tool in many different scenarios of the daily life. However, certain scenarios tend to be more restrictive than usual by requiring devices' functions to be constrained in some manner. For instance, in certain mobile scenarios, specific device features must be turned off. A simple example of such a scenario is the case of movie theaters. It is in the best interest of both movie attendees and movie theater owners that devices present inside the movie screening room are muted. Another useful restriction would be disabling the camera, preventing illegal recordings of said movies. Allowing personal devices to be used within enterprise contexts—so called *bring your own device* (BYOD)—also requires blocking access to specific resources. Many working environments require employees to have restricted Internet access. In certain corporate scenarios, mobile devices' microphone must be disabled. An example of these are privacy-sensitive meetings where there must not be any information-leakage. Likewise, restrictions may apply in other privacy-sensitive scenarios where disabling location sensors might also be necessary.

1.1 Goal and Requirements

The goal of this project is thus to develop, and implement, a novel security framework for commodity mobile devices, named *TrUbi*, that enables a *restricted mode* of operation. This mode of operation enables mobile devices to be turned into special-purposed devices, restricting their functionality to a well-defined set of tasks, for a limited amount of time. Besides these restrictions, the restricted mode also provides remote parties with guarantees that said restrictions are in place. TrUbi framework follows an application centric approach. This means that applications aware of its existence, named *strapps*, are able to request the activation of said restricted mode. The implementation of *TrUbi's prototype* was on top of the *Android OS* platform. The choice of Android was motivated by the fact that this operating system is the most used in terms of scientific and academic research on the field of mobile operating systems' security.

The success of the project also depends on the quality and practicality of the implementation. With this in mind, the final implementation must guarantee the following requirements:

Flexible Security Policies. Applications, using the restricted mode, must be able to properly specify all their security needs, i.e. security policies. *TrUbi* must therefore be able to provide flexible security policies. Mechanisms such as "Allow all but...", "Deny all but...", "Allow resource X", "Deny resource Y" need to be taken into consideration.

Internal and External Assurances. The TrUbi security framework must be able to assure, strapps and external entities, that the requested security policies are being enforced.

Efficiency. TrUbi must have a low impact on the overall system performance. The system must incur in low overheads while not operating under the restricted mode and a slight loss of performance is acceptable while the mobile device is in said mode.

Developer Friendly. Application developers must be able to specify, with little effort, and in detail, the security policies required by their applications, allowing them to focus on the logic of applications rather than security mechanisms.

This solution focuses on *strapps*' security needs. *TrUbi* puts the security needs of strapps before those of the system, and of other applications. In other words, strapps may request for an application, or even for the whole mobile oper-

ating system, to have its functionalities restricted. Strapps are able to request the enforcement of a set of security policies. This enforcement specializes the device in the execution of a very specific task, while granting behavior guarantees to internal and external parties. Said security policy enforcement is named a *"Trust Lease"* [1]. When faced with a trust lease request, *TrUbi* prompts the mobile device user for his approval on such a lease. The strapp is then notified of the user's response and can proceed accordingly.

In summary, the contributions of this project are: (1) The design of TrUbi (Trusted Ubiquity), an OS independent security framework for enforcing application specific restriction policies on personal devices; (2) implementation, on an Android OS system, of a prototype of TrUbi; (3) quality and viability assessment of the prototype.

1.2 Other Solutions

For some time now, mobile devices' security has been a subject of high interest in the scientific community. There has been many different studies and implementations of new security models that target these devices. Systems such as user-centric access control systems [2, 3, 4, 5], mandatory access control systems [6, 7], privacy enhancement systems [8, 9, 10, 11], access control hooks framework systems [12, 13] and application interaction control systems [14, 15, 16] are examples of some such systems. However, none of them, to the best of our knowledge, has ever directly associated the security needs of a security sensitive application and the remaining mobile devices' applications and resources. Applications are always seen as potentially misbehaving and harmful. The needs of the system, and of its users, always precede those of applications. This is the main reason why this project's goal is not achieved by any of the related work mentioned in Section 2, in spite of their contribution in the development of this security framework.

1.3 Results

This project was successful in achieving its goals. *TrUbi* has now a concrete OS-independent architecture design, explained in Section 3. This architecture is based on trust leases which allow for the restriction of the device based on *strapps*' security needs. Restrictions are applied to mobile devices' resources and settings. These restrictions are *flexible* and can limit the access of both applications and the operating system to said resources and settings.

Remote attestation is also guaranteed in TrUbi through the use of digital certificates. Essentially, TrUbi is the owner of a certified key pair which it uses to sign a digest of the currently being enforced trust lease. This allows strapps, and remote entities, to be sure of the authenticity of the digest and therefore of the enforcement of the lease.

The implementation of TrUbi in its Android OS prototype allowed for a proper *efficiency* evaluation. Section 5 presents the benchmark results that prove that the TrUbi prototype has a small impact in the overall system performance. TrUbi focus mainly on resources and settings which are only periodically used and usually through user interaction. This makes it so that the overhead of the extra lease permission verification is small and barely noticeable.

TrUbi and the corresponding prototype achieve this way all the requirements that this project proposed to fulfill, thus achieving its goal.

1.4 Document Structure

The remainder of this document proceeds as follows. Section 2 highlights the related work on mobile devices security. Section 3 describes TrUbi's model and architecture. Section 4 provides implementation details of TrUbi prototype in its current Android OS based version. Evaluation and viability aspects of the TrUbi prototype are presented and discussed in Section 5. Last, Section 6 highlights the most important aspects of the project and future work.

2. RELATED WORK

This section presents the related work, in particular, it focuses on systems aimed at improving mobile computing security. Although these systems could be coupled in several different categories, the chosen categories were so because they are aimed at very concrete security concerns of our system. This section also provides an overview of the Android operating system. This overview allows for a better comprehension of the Android system, as well as how it keeps itself and its applications safe.

2.1 Android Overview

Android is a modern mobile operating system that provides an open source platform and application environment for mobile devices. Android's application functionality is divided into four types of components: *activities*, *services*, *broadcast receivers*, and *content providers*. The activities compose the user interface of applications. Service components act as daemons, providing background processing. Broadcast receivers are components that are responsible for handling asynchronous messages. Last, the content provider components are data servers, unique to each application, that can be queried by other applications. Android allows its components to communicate with each other through the use of Binder Interprocess Communication (IPC).

To enforce security requirements Android uses *permissions*. Permissions are text strings that represent the mobile devices' resources and possible uses. Android applications are required to state, in a specific file called *AndroidManifest.xml*, which resources they require to run. These permissions are then granted to the application when it is first installed on the device and upon user confirmation. However, if the user denies to grant applications their full set of requested permissions, Android stops these applications from being installed into the user's device. The enforcement of said permissions is mostly done by authorization hooks implemented within the Activity Manager Service (AMS). The first time an application tries to use a resource, a correspondent authorization hook is triggered which checks if the requesting application has the necessary permissions to use such resource.

2.2 User-Centric Access Control Systems

These systems focus on giving the user better control over their systems' resource usage by potentially ill-intentioned applications, when compared with the traditional, off the shelf, Android operating system. Among these systems we have MockDroid[4] and AppGuard[3] which allow the user to install applications with their requested permissions but support revoking or refining such permissions later on. Apex[2] and CRePE[5] systems, besides allowing run-time revocation and refinement, also enforce run-time environment based constraints. For Apex, constraints are, for example, the

number of times a resource could be used, or the time of day when such resource usage is allowed. CRePE took these run-time constraints a little further and introduced the concept of context. Contexts, in CRePE sense, are defined by location, time of day, date, policy ownership, and other aspects that allow both a dynamic and descriptive analysis of the environment around the mobile device.

2.3 Mandatory Access Control Systems

Many of the problems affecting today's mobile devices derive from the security polices present in their operating systems. More concretely, the problem usually resides in the coarse-grained security polices associated with applications' permissions to use system's resources. Having noticed such a fact, academic studies were made and systems developed addressing such problems. These systems [6, 7] tried to adapt fine-grain Mandatory Access Control (MAC) techniques into mobile devices. The reasoning behind this was that those techniques were already successfully used in non-mobile machines when trying to enhance their base system's security. SEAndroid[6] and FlaskDroid[7] are two such systems that tried to adapt already successful MAC techniques and systems to mobile operating systems.

2.4 Application Interaction Control

Users have become used to having multiple choices and ways to fulfil their desired tasks. To allow such choices in a user friendly way, application developers have defined APIs in their applications giving other applications access to the services they provide. However, this introduced new security vulnerabilities. Ill-intentioned applications, with low system privileges, started to use trusted applications' APIs in order to fulfil actions and gather information that should not be reachable by them. This kind of attacks, called Confused Deputy Attacks, occur when an untrusted application (*Requester*) is able to trick a more privileged one (*Deputy*) into doing its own bidding. Systems like Quire[16] and IPC Inspection[15] try to solve this problem by focusing on call chains, and by lowering the permissions of called applications to those of their callers. In the other hand, Saint's[14] approach was to extend Android's security architecture with security policies aimed at restricting applications' API usage.

2.5 Privacy Enhancement Systems

Due to its mobility and ease of use, mobile devices are the usual go to when it comes to storing personal data. However, keeping such sensitive data safe can be a challenge. Most security policies make it possible for applications to ask for a resource in a legitimate way and then use it for ill-intentioned purposes. By stating that the internet resource is required for the gathering of news, and that the device's state is for notification purposes, the user would be compelled to accept such terms. However, with the two previously stated resources, one application could, and many do so, share the users' phone number, IMSI and IMEI. TaintDroid[8] focus on detecting how coarse grained policies are allowing sensitive information to leak. On the other hand, AppFence[9], TISSA[10] and Aquifer[11] have their main focus on stopping this information leakage.

2.6 Access Control Hooks Frameworks

Most of the systems that aim to achieve better control over the resources used by applications, and still allow flexibility, realized that: it is necessary to place hooks on the systems' resources APIs in order to actually control when these are going to be used. Thus, the main drive for the frameworks of this category is to provide programmable interfaces that allow resource access control based on authorization hooks. ASM[12] and ASF[13] are two such frameworks. Although both are conceptually very similar, they differ in the approach taken. Whereas ASM seeks to ensure the existing sandboxing guarantees of Android, ASF allows third-parties to extend Android's security framework potentially breaking those guarantees. This means that ASM can only make enforcement more restrictive, i.e. fewer application permissions or less file system access.

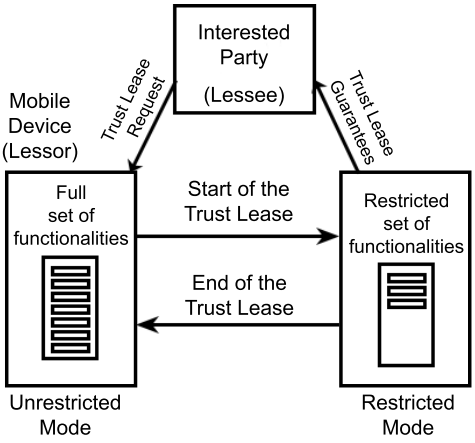


Figure 1: Trust lease technique

2.7 Summary

Although the systems mentioned in Section 2 address important security issues, none of them can achieve the goal of this project. While most scientific and academic research focus on protecting mobile devices, and their users, from ill-intentioned applications, ours focus on protecting trusted applications from ill-intentioned users and insecure applications. This focus proved to be a new paradigm in mobile security and so made impossible to find, to the best of our abilities, systems that shared the same goal. However, it is our belief that through this new paradigm for mobile security, new and improved interactions will take place between mobile devices and their users, further increasing the usefulness of mobile devices.

3. ARCHITECTURE

This section focuses on TrUbi's (Trusted Ubiquity) architecture, an OS independent security framework for enforcing application specific restriction policies on personal devices.

3.1 TrUbi - Trust Lease Model

In order to enforce restriction policies without depending on trusted device administrators, and while being able to adapt to ever changing usage scenarios, trust leases take an *application-centric* approach. This means that this model relies on *strapps* to decide which restriction policy to apply. *Strapps* are essentially applications that are aware of this security framework and are therefore able to leverage

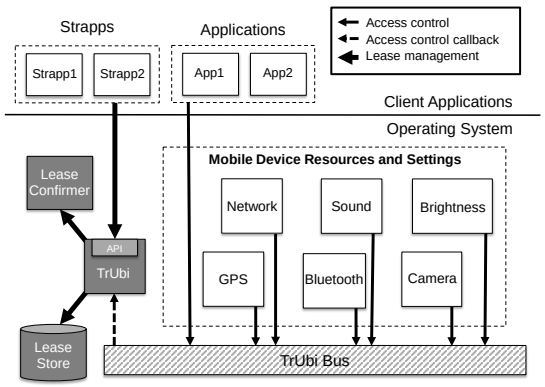


Figure 2: TrUbi high level architecture.

it into allowing them to request the enforcement of global restriction policies on the device.

The trust lease abstraction, illustrated in Figure 1, captures the process of restricting the functionality of a mobile device. It can be seen as a contract between the device's user (the *lessor*) and an interested party (the *lessee*). Lessees can be either a specific security sensitive application, or a remote entity which requires specific security guaranties. This contract defines a *restriction policy* that specifies the functions of the device that must be disabled and the condition events that cease these restrictions (typically a time limit). While a trust lease is active, the user cannot override its restrictions. To ensure that the user keeps his device under control, he must always approve trust leases before they are issued. Furthermore, the trust leases' restrictions are not permanent. The OS always defines a maximum trust lease duration that must be verified before issuing the lease.

3.2 TrUbi Architecture

Figure 2 illustrates a high level architecture of TrUbi. The so called normal *applications*, i.e., the applications that used to run on the original operating system, do not need to be altered in any way to work on top of TrUbi. For all matters and purposes, TrUbi is completely transparent to these applications. *Strapps* on the other hand, although acting as normal applications, know of the existence of TrUbi and can therefore request the enforcement of restrictions. Among other actions, these restrictions can limit the behavior of running applications, mobile devices' resources (e.g. network and camera) and their settings (e.g. sound and screen brightness). These restrictions are achieved by issuing *trust leases* on the running OS. TrUbi then uses its module named *Lease Confirmer* to present a dialog message asking the user's agreement on the lease's conditions and enforcement. If the reply is negative, the process is canceled. Otherwise, TrUbi saves the lease in its *Lease Store* module and begins its enforcement. From here on, before an application is started, a resource accessed, or a setting changed, each of these modules uses the *TrUbi Bus* to communicate with TrUbi and ask if such an action is allowed.

3.3 OS Modes: Restricted and Unrestricted

When a trust lease is instantiated, the OS applies the necessary restriction policies p, causing the device to enter a *restricted mode* state, i.e., a state where some functions (e.g. volume and brightness control) are temporarily disabled (Figure 1). When a trust lease expires, before switching back to *unrestricted mode* state, the system invokes the callback function f, which enables the strapp to perform any cleaning operations (e.g., encrypt data) deemed necessary.

To instantiate a trust lease and enter restricted mode, a strapp developer invokes:

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

This system call takes a restriction policy p and a callback function f. The restriction policy comprises: *restriction rules* (Section 3.4) for defining the functions to be disabled, and *termination rules* (Section 3.5) for specifying the termination conditions of the trust lease.

3.4 Trust Lease Restrictions

Restriction rules define states of one or more system resources or settings that cannot occur while a trust lease is active. Restrictions are application driven and for this matter TrUbi supports both white and blacklists. This means that if a whitelist is used, then all applications are denied except those on said list. On the other hand, if a blacklist is used, all applications are allowed except those on the blacklist. Depending on the kind of module the restriction rule refers to, TrUbi defines three major types of restrictions. We now explain these three major types and the envisioned restrictions for each of them.

Applications Restrictions. Applications restrictions enable blocking the execution of specific applications.

- **Process:** Prevents the execution of applications (e.g. facebook, instagram, snapchat, etc). TrUbi suspends all processes without proper authorization, as well as it blocks any new ones from launching. Upon expiration, TrUbi resumes all previously suspended processes.

Resources Restrictions. Resource restrictions control access to peripherals (e.g., network, or camera) or services (e.g., screenshot service, etc.).

- **Network:** Prevents access to the Internet. TrUbi terminates all unauthorized ongoing Internet connections, denying at the same time new ones.
- **Camera:** Denies access to the Camera. Unauthorized applications have their access revoked and future accesses are denied.
- **Bluetooth:** Prevents access to the Bluetooth resource. Unauthorized applications have their access revoked and future accesses are denied.
- **Phone Call:** Prevents applications from making or receiving phone calls. During the enforcement of a trust lease new calls are denied and ongoing calls terminated.
- **SMS & MMS:** Prevents sending and receiving SMS and MMS messages.
- **Screenshot:** Blocks the ability of a mobile device of taking screenshots. This restriction is specially useful for sensitive content sent through applications such as Snapchat.

- **Microphone:** Denies access to the microphone of the mobile device if the application has not been granted access to it by the trust lease.
- **LED:** Most mobile devices have a built in camera with a LED for flash purposes. This rule forces LED to be activated only by applications with proper authorization.

Settings Restrictions. Settings restrictions refer to system configurations, such as muting the device, enabling flight mode, etc.

- **Sound:** Sets an initial value for the device's volume and forces it to be kept unchanged, unless the application has the proper access rights.
- **Brightness:** Sets an initial value for the brightness of the screen and prevents changes to this configuration by the user or any unauthorized application.
- **Airplane Mode:** Allows the lease to set an initial value for this setting, namely *on* or *off*, which is kept unchanged throughout the duration of the lease.
- **Time:** Prevents changes to the system's date, time and timezone.

3.5 Trust Lease Termination Events

When a trust lease is active, the OS remains in restricted mode until a termination event occurs. Termination events are delivered in one of three ways:

Time Based. Typical termination rules define *time* conditions. When a trust lease is issued, the OS sets a timer that gets fired once T elapses, terminating the trust lease. With this in mind, two time based terminators can be specified:

- **Timeout:** Provides TrUbi with a relative time value for the termination of the lease. Example: $T_{out} = 1h30m$, meaning that the lease will stay active for 1 hour and 30 minutes since the start of the lease.
- **Date:** Provides TrUbi with an absolute time (day and time of day) to terminate the lease. Example: $T_{day} = 16/Oct-22:30$, meaning that the lease will stay active until the 16th of October at 22 hours and 30 minutes.

Location Based. Termination rules can also be based on *location*. Two obvious terminators were envisioned:

- **GPS:** Provides TrUbi with an absolute location (geographic coordinates) and a radius (in meters) within which the lease must be enforced. Example: Terminator $D_{Location=38N,9E} > 10m$ specifies a pivot point, the GPS coordinates, and a maximum radius of 10 meters where the trust lease is valid.
- **Wifi:** Provides TrUbi with a relative location within which the lease will remain active. Essentially TrUbi may receive the SSID of one, or more, Wifi access points which need to be in range for the lease not to be terminated. Example: $AP_{id=BaseWifi}$ is a terminator according to which the trust lease will be ended if the device is no longer in range of the access point named "BaseWifi".

Explicit Action. An alternative method for terminating a lease is by explicit invocation of `stoplease`. This method aims to allow the strapp that owns the trust lease to terminate it before the termination rules occur. The reason for disabling the lease ahead of time is application specific. To invoke `stoplease` the strapp only needs to provide the trust lease id (l_{id}) as parameter:

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

By judiciously defining termination rules and invoking `stoplease`, a strapp can generate a rich set of termination events. To prevent resources from being indefinitely locked, the OS always sets a ceiling timeout T_{max}.

3.6 Remote Attestation of Trust Leases

For some usage scenarios, strapps may need to convince an external party that the device operates in restricted mode. Taking some ideas from the trusted computing world, this need is addressed by providing a *trust lease remote attestation* mechanism, which relies on two primitives:

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

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. 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.

It is assumed that each OS is provisioned with a unique keypair and that the private key is securely stored by the OS. It is also assumed the the OS has not been tampered with, thus belonging to our TCB (Trusted Computing Base). 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.

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:

$$\mathsf{quote} = \langle m||n\rangle_{K}^{-},\ C(K^{+})$$

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, namely $C(K^{+})$. 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, which must indicate that the OS supports trust leases verifying that m returned in the quote is meaningful.

Lastly, if there are concerns about the identity of the OS, the protocol can take advantage of 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 identity of the OS.

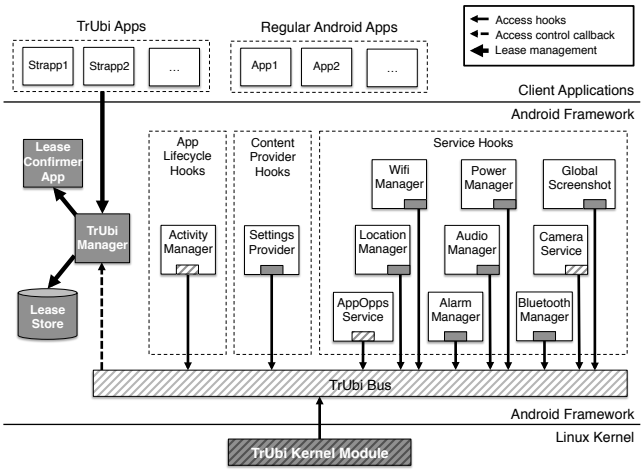


Figure 3: TrUbi prototype architecture on Android

4. IMPLEMENTATION

This section focuses upon TrUbi's implementation on the Android OS. This implementation has the goal of verifying the feasibility of the TrUbi system, as well as allowing a proper evaluation of this system, discussed in Section 5.

4.1 TrUbi's Prototype on Android OS

TrUbi's prototype architecture, Figure 3, 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 also adds a set of specific components (shown as dark shaded boxes) which implement the trust lease reference monitor.

TrUbi Manager is the core of the system, 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. To achieve this, the TrUbi Manager component is filled with the necessary trust lease logic and exposes an API (detailed in Section 4.2) for strapps to be able to use the primitives explained in Section 3.

Lease Confirmer is a system app also part of the TrUbi prototype. This app is used by TrUbi upon the request of a lease by a strapp to interface with the user and obtain trust lease authorizations. The Lease Confirmer displays the lease conditions to the user and provides buttons for accepting or declining them.

Hooks are used by TrUbi Manager to enforce access control restrictions. Spread across the system, these hooks monitor every access to restricted objects allowing TrUbi Manager to perform the corresponding control decisions. Some of these hooks are placed in the Android framework (represented in Figure 3 as small shared 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, or 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 through a dedicated system service named *TrUbi Bus*.

To implement all required hooks, it is necessary to instrument multiple Android software components, both in the framework and kernel. To reduce the complexity of this task, TrUbi leverages Android Security Modules (ASM) [12]. 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. As a result, ASM can be used to implement novel access control mechanisms, including TrUbi's. ASM provides a backbone for placing and handling hooks in both the framework and kernel.

However, ASM alone is insufficient to implement TrUbi. Various framework hooks were unsupported by ASM and had to be implemented. Among these hooks are the ones for volume control, brightness, microphone, camera LED, airplane mode and others. Modifications were also done to the ASM kernel module (to control suspension and termination of application processes). The remaining TrUbi components were built from scratch.

In this work, TrUbi's prototype is using an ASM implementation from November 2014, which requires Android KitKat. For this reason, the TrUbi prototype was built around the relatively outdated Android KitKat distribution. TrUbi source code was written in Java (10KLOC) and C (0.2KLOC).

4.2 TrUbi's API

TrUbi provides strapp developers with an API that implements the basic trust lease primitives introduced in Section 3: `startlease`, `stoplease` and `quotelease`. The `verifylease` primitive is not part of the API but is instead provided to strapps through a library implemented in TrUbi. However, to be able to implement TrUbi in a timely manner some of these primitives were slightly altered. Bellow is explained every primitive's implementation and what differs from their original architecture of Section 3.

To start a trust lease, a developer must follow the sequence of steps illustrated in the sample Java code of Listing 1: first obtain a reference to the TrUbi Manager, then read the restriction policy from a XML file into a local object instance, and finally invoke the method `startlease` with the restriction policy object. 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`. For such an action the procedure is similar to that of the start of the lease but the method that is used is `stoplease`, with no arguments.

The trust lease remote attestation primitives follow the cryptographic protocols specified in Section 3.6. For symmetric cryptography TrUbi prototype uses AES-256, and for digital signatures RSA-1024 and SHA-2. The cryptographic keys are secured in Android's Key Store. In the current implementation the integrity of the kernel is not verified with trusted computing hardware. This is because both the kernel and the operating system are part of the Trusted Computing Base of this project and their integrity is considered to be assured. However, as future work, a proper attestation of the integrity of the kernel and operating system can be

```
// 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.parseLocal();

try {
  // instantiate the trust lease in the system
  int leaseId = trubi.startLease(lease);
} catch (TrUbiRemoteException e) {
  // lease creation failed
}
```

Listing 1: Java code fragment to start a lease.

achieved through hardware with support for trusted computing primitives.

4.3 Supported Trust Lease Restrictions

This section addresses the currently supported restrictions of the TrUbi prototype and how they are implemented.

Applications Restrictions:

- **Process:** To limit applications from running, TrUbi's prototype tackles two main issues:

  (1) Preventing new applications from being launched: To do so, TrUbi uses preexisting ASM hooks placed in the Activity Manager. These hooks control: new Activities and Services, binding of Services, and broadcasting intents to applications.

  (2) Suspending already running applications: This process is not straightforward. In Android, applications may launch background processes. These background processes are not managed by the Activity Manager and therefore can only be control through the kernel module. To control these processes, TrUbi sends a `SIGSTOP` and `SIGCONT`, respectively to freeze and unfreeze background processes.

Resources Rules:

- **Network:** The two main aspects to take into account are: (1) on going connections started before the lease; (2) new connections during the lease.

  To properly enforce already ongoing connections, TrUbi overrides ASM's hooks that control new sockets to keep track of all running applications that maintain active network connections. Then, whenever a trust lease starts, TrUbi terminates all applications that have been denied network access, forcing all their network connections to be closed. While the lease is active, socket connect and accept requests will be intercepted by these hooks and validated by the TrUbi Manager.

- **Camera:** TrUbi leverages a pre-existing ASM hook from the Camera Service. This hook is executed the first time an application accesses the camera. TrUbi overrides this hook to keep track of applications that have gained prior access to the camera. If TrUbi needs to revoke their permissions when the trust lease starts, then it terminates such applications.

- **Phone Call:** TrUbi leverages an already existing ASM hook which allows for a proper access control before a call is actually issued. For ongoing calls, and when a lease is activated, TrUbi checks if said call should be allowed or not. If not, TrUbi sends an end call event to the TelephonyService. To control phone call reception events TrUbi uses a broadcast receiver that listens to incoming calls and rejects them if need be.

- **SMS & MMS:** This restrictor's implementation is straightforward. ASM provides hooks that are invoked every time an application sends or receives an SMS / MMS, thus, all of these operations can be intercepted.

- **Screenshot:** Android provides three ways for taking screenshots of the device's screen: (1) an API call that applications can use to capture an image of their own screen, (2) the System UI service, which can be activated by the user (e.g., by pressing the power and volume down buttons), and (3) through the Android Debug Bridge (ADB) interface. TrUbi ignores mechanism 1 and enables trust leases to disable mechanisms 2 and 3. TrUbi has a third party ASM hook in the System UI service which intercepts and blocks accesses denied by the trust lease. To disable the ADB interface, TrUbi controls the execution of *screencap*, an external program used by Android's ADB server to take screenshots.

- **Microphone:** TrUbi implementation of this restriction is done by placing a new hook in the Audio Manager's function responsible for toggling the microphone mute function. This hook intercepts all requests to unmute the microphone and aborts them if a trust lease is enabled.

- **LED:** TrUbi implements this restriction by placing a new hook in the Camera Manager's function where the LED mode is changed.

- **Bluetooth:** To implement this restriction, TrUbi placed new hooks in the Bluetooth Manager service which prevent modification of bluetooth configurations by the user through the System Settings or by applications with the required permissions.

Settings Rules:

- **Sound:** TrUbi placed new hooks in all entry points to the Audio Manager where the sound volume can be changed.

- **Brightness:** To implement the brightness restriction, TrUbi has three new hooks in the entry points of all relevant system services where the screen settings are controlled: Window Manager, System Settings, and Power Manager.

- **Airplane Mode:** TrUbi provides a specific restrictor for keeping airplane mode on/off until the trust lease terminates. Since airplane mode can only be toggled by the System Settings (since Android 4.2 onwards), it was only necessary to place one third party hook.

- **Time:** TrUbi implements this through new hooks in two system services: System Clock and Alarm Manager. These services are reached through the System Settings, which is the only application that is authorized to modify the time settings of the system.

4.4 Supported Trust Lease Terminators

TrUbi's implementation on Android supports the previously mentioned terminators of Section 3.5. We now explain how each one of these is implemented.

Time Based Terminators

- **Timeout:** When TrUbi starts a lease with the time based terminator it launches a thread that sleeps for the duration of the lease; upon waking up, it terminates the lease.

- **Date:** TrUbi uses the system time and absolute date from the lease specification to calculate a timeout and proceeds as if it had been given a `timeout` terminator.

Note: When both terminators are applied in a single lease, TrUbi calculates the shortest of both and only launches one thread for that one.

Location Based Terminators

- **GPS:** Leveraging ASM's location hooks, whenever the mobile device's location changes, TrUbi verifies if the mobile device is still within the lease enforcement range. In case it is, nothing happens, in case it has left the enforcement range, then TrUbi terminates the lease.

- **Wifi:** Upon the enforcement of the lease, TrUbi launches a thread that checks the nearby Wifi access points every once in a while checking if the specified access points are in range and proceeding accordingly.

4.5 Trust Lease Persistence

The TrUbi Manager is responsible for keeping track of active trust leases. However, keeping this information in volatile memory alone opens some vulnerabilities. In particular, if the device is rebooted, information of active trust leases would be lost, resulting in a potential violation of their assurances. TrUbi's prototype addresses this problem by backing up all relevant trust lease state data persistently. Essentially, as soon as TrUbi receives a lease request, followed by the corresponding user approval, it writes into persistent memory the whole lease specification. On every single boot, TrUbi always checks first for any lease specifications that it might have stored in a previous session. This way, if for some unforeseen reason the mobile device reboots, either due to technical reasons, or malicious ones, the lease is always assured to be properly enforced. Upon the termination of the lease, TrUbi deletes from storage the correspondent lease specification.

5. EVALUATION

This section focuses on the evaluation of the TrUbi prototype. As mentioned before, TrUbi's prototype is required to be efficient and secure. In this section TrUbi is measured in terms of *security efficiency*, proving its viability in applying security policies, and in *performance and energy consumption efficiency*, proving the fulfillment of its efficiency *requirement*.

Resources\OS	AOSP	TrUbi
Graphics (Softweg)		
Draw opacity bitmap (MPix per sec)	6.7 ± 0.03	6.7 ± 0.01
Draw transparent bitmap (MPix per sec)	6.2 ± 0.01	6.2 ± 0.02
CPU Whetstone (Softweg)		
MWIPS DP (DP)	296.2 ± 2.84	298.0 ± 2.78
MWIPS SP (SP)	317.5 ± 1.51	312.5 ± 2.71
MFLOPS DP (SP)	83.7 ± 0.49	86.1 ± 0.57
MFLOPS SP (SP)	80.8 ± 0.21	82.0 ± 0.29
VAX MIPS DP (DP)	236.9 ± 0.84	235.6 ± 1.51
VAX MIPS SP (SP)	244.9 ± 1.11	236.0 ± 1.37
Memory (Softweg)		
Copy memory (Mb/sec)	2201.3 ± 43.21	2229.8 ± 20.45
Filesystem (Softweg)		
Creating 1000 empty files (sec)	0.3 ± 0.01	0.3 ± 0.01
Deleting 1000 empty files (sec)	0.2 ± 0.01	0.2 ± 0.01
Write 1M into file (M/sec)	111.6 ± 0.39	112.4 ± 0.46
Read 1M from file (sM/sec)	512.4 ± 2.08	505.0 ± 1.62

Table 1: Results of Softweg experiments.

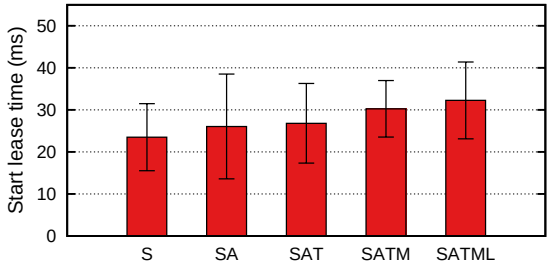
Hardware Testbed: Throughout the benchmarks present in this section, TrUbi's implementation was tested in a Nexus 4 smartphone. This device features a quad-core CPU of 1.5 GHz, 2GB of RAM, 16 GB of memory with no SD-Card support, 802.11 WiFi interface, 768 x 1280 display, an 8 MP camera with 3264 x 2448 pixels and a LED flash. The running operating system is an Android 4.4.1 AOSP build.

5.1 Restrictions Enforcement Efficiency

TrUbi, as a security framework, has its main concern on security. For this reason the first major concern is to test each and every implemented restriction to prove that it is working as intended. With this goal in mind, we use 10 representative applications for each of the supported restrictions. The applications were chosen based on the resources they use and on their ratting on Google Play [17]. However, the *screenshot*, *airplane mode* and *time* restrictions could not be tested with 10 different applications. This is due to the fact that these resources and settings have special ways of being used and not every application can have access to them. Nevertheless, in all cases, every restriction was successfully tested and properly blocked.

5.2 System Overall Performance

This section evaluates the overall impact of TrUbi's implementation. The software used for this benchmark is an Android application named *Softweg* [18]. Softweg was chosen because it was used to benchmark other security systems such as SE Android [6]. This application measures the throughput of graphics, CPU, memory and filesystem. The measurements are taken while stress testing each of these components several times (the number of times is user configurable). For a proper evaluation of the prototype, this software was executed 6 times, with 200 readings per resource. In terms of methodology the device was first flashed with a clean setup of the OS being tested. The executions



Legend: S: sound; A: airplane; T: time; M: SMS; L: LED

Figure 4: Execution time of start lease primitive.

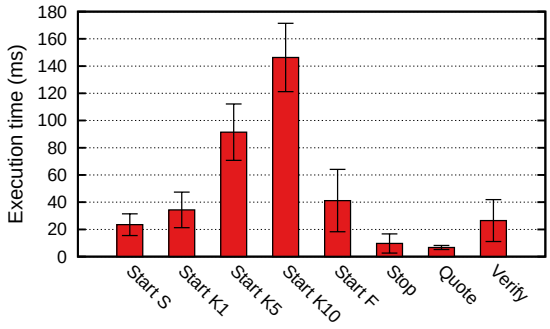


Figure 5: Execution time of API calls.

were then taken in succession of each other while the device was always connected to the power plug to prevent performance fluctuations due to power management settings. Table 1 illustrates the readings provided by the benchmark.

By analyzing Table 1 it is possible to verify that the impact of TrUbi, when not enforcing any leases, is low and well within the standard deviation values. This is due to the fact that most of TrUbi's hooks are not related with the type of operations used in these system performance benchmarks. TrUbi has low impact on GPU and CPU intensive tasks, as well as memory and filesystem. Instead, TrUbi focuses more on resources and settings that are usually accessed through user interaction. This makes overheads of a few milliseconds negligible when compared to the average user reaction time. Furthermore, camera, microphone and sound are not usual targets for performance benchmarks. This is mainly due to the fact that resources like these are usually benchmarked in terms of quality, i.e., image quality or sound sharpness, which is not affected by TrUbi.

5.3 Trust Lease Primitives Performance

This section focuses on measuring the elapsed time for each of the trust lease primitives that the prototype implements. Besides analyzing all four primitives, different scenarios are also taken into account within the same primitive. Each of the experiments was done 50 times and measured for average values and standard deviation.

- **Start Lease - No Action:** Figure 4 shows the execution times of `startlease` with restrictions: sound, airplane mode, time, SMS & MMS and LED. These restrictions have the particular characteristic that they can be issued without requiring a state change of the system's resources or settings. For this reason they register the lowest measured values for `startlease`, $23.5ms$. It is also possible to verify that `startlease` has a fixed overhead of about $21ms$, which increases, on average, around $2ms$ per restriction added. Another important aspect of these measurements is the standard deviation of $8.73ms$. Such a high value for the standard deviation was traced back to Android's garbage collector. We noticed that every time the garbage collector performed an action the registered values would spike.

- **Start Lease - Application Killing:** Some leases require TrUbi to terminate running applications. This happens when a lease with the network or camera restrictions is issued. For this benchmark TrUbi prototype was tested with scenarios of 1, 5 and 10 running camera applications, respectively K1, K5 and K10 (Figure 5). The values of 1, 5 and 10 were chosen to allow us to understand if the time increased linearly as the number of applications terminated did, which they did. Registered values were $34.32ms$, $91.46ms$ and $146.3ms$.

- **Start Lease - Application Freezing:** The *process* restriction, as mentioned before, allows to limit the execution of applications. In order to test this mechanism the prototype was measured issuing a lease with such a restriction. The measured values (Figure 5, Start F) had an average of $41.18ms$ with a standard deviation of $22.89ms$. Android's garbage collector also caused spikes to happen on these tests. Because the applications were being frozen, sometimes Android's garbage collector would perform a cleanup of their resources. Note that this value includes freezing every non-essential Android application and service. The registered value is thus considered the lowest value one can achieve with such a lease.

- **Stop Lease:** Terminating a lease is simple and quite straightforward. As so, the time registered for this trust lease primitive during these benchmarks was the lowest. The `stoplease` primitive was executed with an average time of $9.66ms$ and a high value of standard deviation of $7.04ms$, once again due to Android's garbage collector cleanup actions.

- **Quote & Verify Lease:** The `quotelease` and `verifylease` are the two last benchmarks present in this section. In Figure 5 is noticeable that there is a big discrepancy between these two primitives. The measured values were $6.7ms$ and $26,54ms$, respectively `quotelease` and `verifylease`. This is due to the fact that `quotelease` performs a single digital signature, while `verifylease` has two signatures to verify: from the quote and the certificate (see Section 3.6).

5.4 Energy Consumption

This section compares the energy consumption of TrUbi's prototype and the unpatched version of Android 4.4.1 AOSP. The benchmark was done using a software named Trepn profiler 6.0 [19] by Qualcomm. Trepn obtains power consumption samples every $100ms$. It can report accurate values by accessing the power management integrated circuit available in some devices, including the testbed's device: Nexus

Benchmark	AOSP	TrUbi	Overhead
Consumption (J)	2207.8 ± 90.3	2218.8 ± 123.7	0.5 %

Table 2: Energy consumption of TrUbi prototype.

4. This benchmark was done in the same test environment as that of the performance benchmark, only difference being that the device is no longer connected to the charger. The test was done running the Trepn software 10 times while it evaluated the Softweg benchmark which was configured to do 50 repetitions per resource. Running Trepn while performing a Softweg benchmark allowed both systems to be stress tested in the same testing environments achieving accurate results. Table 2 shows a slight increase in power consumption by 0.5%, which is well within the standard deviation. The slight increase in power consumption of 0.5% was already expected as TrUbi's overall performance is just slightly inferior than that of the original Android.

6. CONCLUSIONS

This project presented TrUbi, a system that provides trust lease support for mobile devices, as well as its implementation on Android. Trust lease is a general abstraction that allows applications to enforce global functional restrictions on mobile devices. Trust leases allow for devices to be used in new scenarios which are currently unavailable. To the best of our knowledge, TrUbi is the first implemented system that explores the potential of this abstraction. TrUbi provides Android developers with simple API and expressive policy specification features. We demonstrate TrUbi's potential through proper evaluation and benchmarks.

6.1 Future Work

Some interesting aspects still need to be studied and polished in order to achieve a proper public release. In the future we will keep on developing and enhancing TrUbi with the goal of releasing TrUbi to the scientific community. The most important aspects to be worked on are: (1) researching the microphone and bluetooth routines on Android OS, allowing us to turn their restrictions into fine-grained restrictions; (2) implementing a callback function to be called upon the termination of the lease, like the one described in Section 3.3; (3) writing detailed documentation of the TrUbi architecture and implementation, for the scientific community; (4) developing an "Hello World" strapp, with proper documentation, for application developers. Furthermore, there are also more complex enhancements that can still be done to the TrUbi project. The most compelling among these being the reduction of the TCB through the use of trusted computing primitives and hardware.

References

[1] 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 *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, 2015.

[2] M. Nauman, S. Khan, and X. Zhang, "Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints," in *Proc. of ASIACCS*, 2010.

[3] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "AppGuard: Enforcing User Requirements on Android Apps," in *Proc. of TACAS*, 2013.

[4] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "MockDroid: Trading Privacy for Application Functionality on Smartphones," in *Proc. of HotMobile*, 2011.

[5] M. Conti, V. T. N. Nguyen, and B. Crispo, "CRePE: Context-Related Policy Enforcement for Android," *Information Security*, vol. 6531, pp. 331–345, 2011.

[6] S. Smalley and R. Craig, "Security enhanced (se) android: Bringing flexible mac to android." in *NDSS*. The Internet Society, 2013.

[7] 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.

[8] 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.

[9] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These ArenâĂŹt the Droids YouâĂŹre Looking For: Retrofitting Android to Protect Data from Imperious Applications," in *Proc. of CCS*, 2011.

[10] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming Information-Stealing Smartphone Applications (on Android)," in *Proc. of TRUST*, 2011.

[11] A. Nadkarni and W. Enck, "Preventing accidental data disclosure in modern operating systems," in *Proc. of SIGSAC*, 2013.

[12] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi, "ASM: A Programmable Interface for Extending Android Security," in *Proc. of USENIX Security*, 2014.

[13] 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.

[14] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically Rich Application-Centric Security in Android," *Security and Communication Networks*, vol. 5, no. 6, pp. 658–673, Jun. 2012.

[15] 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.

[16] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "QUIRE: Lightweight Provenance for Smart Phone Operating Systems." in *Proc. of USENIX Security*, 2011.

[17] "Google Play," https://play.google.com/store. Accessed July 2015.

[18] "Softweg. Benchmark." https://play.google.com/store/apps/details?id=softweg.hw.performance. Accessed July 2015.

[19] "QUALCOMM Trepn Profiles," https://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-profiler. Accessed July 2015.