On the Effectiveness of Trust Leases in Securing Mobile Applications

Nuno Duarte
nuno.duarte@tecnico.ulisboa.pt
Instituto Superior Técnico, Lisboa, Portugal

ABSTRACT

In recent years we have witnessed a tremendous evolution in mobile devices' capabilities. However, there are still activities that we do not do resorting to these devices, because mobile OSes (MOSes) do not offer the necessary mechanisms to secure them. For instance, today is impossible to allow students to answer exams using their devices. Because MOSes offer no mechanisms capable of restricting Internet access and the execution of applications, students could easily cheat. In order to address this problem, we recently proposed a new OS primitive called *trust lease*. A mechanism that allows for users and third parties to negotiate OS resource and application lifetime restrictions (e.g., prevent Internet access and the execution of apps).

The goal of this work is then to develop new applications, currently unsupported by MOSes, and test the effectiveness of this new primitive in securing these applications' scenarios. Additionally, we test the effects of *trust leases* on real-world applications and spyware, and we conduct an adoption study to understand how open would users feel to adopt this solution. The results we collected encourage the use of the primitive.

1. INTRODUCTION

Over the past years, we have witnessed incredible advances in mobile technology. Smartphones and tablets emerged, equipped with powerful sensors and network interfaces that allow for everywhere connection with online services and nearby devices. Because mobile devices are fully programmable, users can enhance their functionality in various ways. There are thousands of applications (apps for short) in app stores like Google Play [2] that can be installed on devices. In addition, users have the freedom to further develop their own. Because mobile devices are lightweight, users can easily carry them everywhere. Naturally, these factors have sparked a widespread adoption of mobile technology for personal and professional purposes, changing society deeply.

Despite these advances, however, there are still plenty of activities that we simply do not do resorting to these devices. These particular activities would require strict assurances from mobile operating systems (MOSes), about the behaviour (functionality) of mobile devices, in order for them to be executed correctly. For instance, the lack of behaviour guarantees of mobile devices could be the source of security breaches, in business meetings, where the topics discussed are often privacy sensitive. Without knowing, a device could run malware that secretly taps the device, records phone calls, or captures audio signal and stealthily uploads recorded conversations to a remote site.

Mobile devices' inability to provide behavior guarantees is relevant also in other contexts. For example, in airplanes, mobile devices must be shut down or turned into flight mode; in airports' customs areas, mobile devices cannot be used to take photos or make phone calls; in museums, devices cannot be used to take pictures; in movie theaters, devices should be put in silent mode to avoid disturbing the audience, etc. However, users commonly forget to abide by these rules. Therefore, service providers lack guarantees that peoples' mobile devices have been properly configured.

In examination centers and universities, the evaluation of students could be a lot more efficient and time saving for professors by allowing examinees to fill out exams through a trusted m-exam app running on their devices. However, with today's technology, examinees could easily cheat. They could launch additional apps to obtain answers, connect to the Internet and look up Wikipedia, etc. Although current mobile technology can support the execution of such activities, the reality is that today's MOSes simply do not offer the mechanisms to do so, securely. This means that currently, there is no way to prevent students from cheating, unless resorting to human vigilance, which is still a fallible mechanism to prevent exam cheating.

So, there is a clear need for a system capable of solving these limitations. In order to overcome these issues, we recently proposed a new operating system primitive called *trust lease* [25]. This primitive introduces the concept of secure operation modes for mobile devices, allowing security sensitive applications to rely on the OS to prevent the execution of potentially dangerous operations by other applications or system services. A good example of such a secure application scenario is the mobile exam app which blocks the access to the Internet and prevents any pdf reader app from running during the exam, effectively preventing students from cheating. Recently we have developed a novel platform, called TrUbi, that uses *trust leases* to allow for the development of secure applications for Android devices. Essentially, the use of *trust leases* enable application developers to leverage TrUbi's services to enforce these secure applications' restrictions on users' devices. With this approach, the specification of the constraints an app must enforce is reduced to the lightweight task of using the *trust lease* model. The idea is for application developers to define a general-use *trust lease* to be used in their apps, but also to allow even non-technical users to define their own (e.g., a professor may define his own trust lease to enforce during his exams).

1.1 Goals & Requirements

The goal of this work is to validate the effectiveness of the *trust lease* primitive in securing mobile applications, by designing and implementing three use-case applications whose security guarantees are not supported by today's MOSes. We use them to perform a thorough evaluation of the effectiveness of *trust leases* in enforcing each of these apps' corresponding runtime constraints, through **usability tests**.

Additionally, and because all these apps require the negotiation of *trust leases* between users and third parties (e.g., students and professors), we aim at designing a **middleware layer** (Strapp Middleware) on top of TrUbi's *trust lease* API. This middleware helps application developers in using *trust leases* (i.e., negotiation and corresponding installation) transparently over network communication (sockets), and over Near Field Communication (NFC). Note that we are interested in designing a solution with globally applicable concepts; however, it is not a requirement to develop a system that is portable to all mobile devices. As a matter of fact, because TrUbi is built for Android devices we intend to develop the above mentioned apps for Android based smartphones; thus, we focus on the security mechanisms that are available in Android. In this work we also seek to test the effects of *trust leases* in **real-world applications**, i.e., existing applications available in today's application markets (e.g., Google Play [2]). Furthermore, we also plan to test the effects of *trust leases* on **real-world spyware**. Finally, we plan to conduct an **adoption study**, in order to assess how open would potential users feel about the adoption of these apps, as well as the underlying system.

In this work we aim at developing three secure applications:

- an **e-exam application** (called mExam) that allows students to answer exam questions while preventing them from cheating;
- an **enterprise meeting assistant application** (called mMeeting) that blocks certain sensitive resources (e.g., microphone or camera), in order to prevent meeting information leakage;
- a **movie theatre context aware application** (called mTicket) that restricts some mobile device's settings (e.g., volume, screen brightness), as well as the camera.

The proposed apps must fulfil the following requirements:

- ensure a **secure behaviour** even in the presence of other apps that may try to circumvent the restrictions imposed (e.g., trusted parties that may have been compromised with malware trying to leak meeting information);
- provide a **user friendly experience** to the users by avoiding the need of extra-effort when compared to what happens with current apps (e.g., changing the status to vibrate, etc.);
- allow *trust lease* creators to enforce the **execution of trusted apps only** (e.g., resorting to app package name), on the target device;
- guarantee our secure apps comply with the desired set of privileges **only for the intended period of time**, after which owners regain full control of the device.

1.2 Current Solutions

Nowadays, systems and applications addressing this special type of activities have low security requirements, as they do not enforce restrictions on devices' resources. As we have said, present day MOSes do not provide the necessary mechanisms to enforce the kind of restrictions needed to secure this type of activities. Additionally, today's MOS security extensions (e.g., ASM [15], ASF [4]), come short in mitigating these limitations. In summary these extensions' main limitations are:

- **limited restrictions -** they restrict very few little OS resources, or offer no restrictions at all, focusing on other issues such as digital rights management, or app communication monitoring;
- **course-grained restrictions -** for instance, they block access to the whole Internet, and do not allow for an app to access a sub-group of domains;
- **system-wide restrictions -** for example, they block Internet access to all applications, and are not able to support a white / black-list Internet access approach;
- **weak application lifecycle restrictions -** while most extensions simply cannot prevent untrusted apps from being launched, the ones that can, are incapable of stopping already running apps;
- **low deployability -** these solutions require the modification of the Android OS source code and corresponding generation of custom system images for different devices, making the adoption for real users difficult.

1.3 Contributions

This work makes the following contributions: *(i)* design and implementation of the m-exam, meeting assistant and movie theatre context-aware apps as well as their individual communication protocols and behaviour procedures, supported on the *trust lease* primitive *(ii)* design and implementation of the Strapp Middleware, as well as its communication protocol, over sockets and NFC, leveraging TrUbi's *trust lease* API *(iii)* implementation of a webserver application, capable of providing mockup functionality (e.g. authentication) for each of the three proposed apps, *(iv)* evaluation of the effects of *trust leases* on real-world applications / spyware, *(v)* evaluation of the effectiveness of the developed apps in performing their desired tasks, through user testing, and *(vi)* publication of part of this work in a workshop [25], and submission to a conference.

1.4 Document Structure

The rest of this document is organized as follows. Section 2 describes current related solutions. Section 3 provides some background over TrUbi, and then proceeds to the specification of the Strapp Middleware and its protocol; the Section terminates with the description of the main architectural decisions of the three apps. Section 4 addresses the implementation details of the Strapp Middleware, as well as the three apps and webserver. Section 5 covers the evaluation of the effects of *trust leases* over real-world apps and spyware, and then presents the user test results over the effectiveness of the developed apps in performing their desired tasks; the Section concludes with adoption study results. Finally, Section 6 concludes this work.

2. RELATED WORK

We start this Section by covering applications similar to those we intend to develop. We provide an overall analysis of their requirements to show why existing solutions fail in providing secure ways to achieve the tasks we are interested in (Section 2.1). Then, we present mobile security mechanisms that extend Android, in order to increase users' capabilities of controlling their devices' resources and data (Section 2.2).

2.1 Applications

The field of research focused on electronic exams has been around for quite a while now and yet we have not seen a consensual system being widely adopted in schools. One of the main obstacles preventing this adoption is the lack of mechanisms capable of ensure that students cannot cheat. In fact, nowadays, academic [9] and commercial [3] e-exam systems fail in providing those guarantees, and continue to rely on invigilating professors, or on e-monitoring mechanisms incompatible with today's mobile devices.

The area of e-voting [17], shares some security requirement similarities to e-exams, but despite offering safe communication, they do not offer the possibility to apply the restrictions we need. In a more enterprise environment, e-meeting systems [14], seek to ease employees' work by providing useful context-based services. However, because these solutions have a bigger focus on context-awareness, they have more lenient concerns about security than the ones we have, rendering them incapable of solving our issue. E-ticketing systems [19], share the transactional nature of our mTicket app. Even so, they do not enforce the restrictions we need.

2.2 Mobile Security Mechanisms

A large body of work aims to improve data security from untrusted, potentially malicious, Android applications. We now present the most significant Android extensions. In the main document we provide additional information on Android extensions.

Several proposals focus on improving resource access control. Some proposals improve these mechanisms by implementing mandatory access control (MAC). TrustDroid [7] implements MAC in order to provide different trust level domain isolation (e.g., private and corporate) on Android's middleware and kernel. SEAndroid [26] taylors SELinux [20] to enforce MAC on both these layers as well. FlaskDroid [8] extends SEAndroid by synchronizing MAC on both those layers, and has a clear focus on fine-grained access control.

Other proposals improve resource access control by enabling fine-grained permissions. APEX [22] allows users to define constraints on individual app resource usage. Permission Tracker [18] allows users to revoke app permissions at runtime. MockDroid [5] allows that same revocation, and complements it with data-shadowing. Compac [29] allows users to assign different permissions to app sub-components. CRePE [10] allows trusted third parties to enforce security policies on another user's device, which resembles TrUbi's goals conceptually. However, CRePE offers no way for third parties to perform remote attestation. It also allows third parties to define policies that cannot be revoked by the user, which means they might have no control over their devices if needed. In addition, unlike TrUbi, with CRePE, a user has no way to deny the enforcement of a third party policy, and these entities may even control the device remotely, meaning that a company may send policies to a user's device while he's at home. MOSES [24] allows for the definition and enforcement of security profiles that implement different operation modes on smartphones (e.g. Work, Private). Although MOSES enforces software isolation of apps and data between modes, it has a more static approach than TrUbi because its security policies are made to be systematic, (e.g. policies can respect normal work routines, but may fail to adapt to a meeting where a host, which as no enterprise infrastructure, may define his own policies). Like CRePE, MOSES also allows for external entities to introduce new security profiles that can be enforced without user knowledge. On the other hand, no remote attestation mechanisms are offered to those entities.

Another group of proposals improves resource access control by mitigating privilege escalation, more specifically confused deputy and collusion attacks. Saint [23] allows app developers to specify policies defining which apps have access to their apps' APIs. QUIRE [11] forces all Binder IPC messages exchanged between apps to contain the full call chain context to determine if the original app has privileges to perform the operation currently being executed on its behalf by another app. IPC Inspection [13] follows a similar approach but instead of checking the source app's privileges, it reduces the target app's privileges, effectively preventing privilege escalation. XMandDroid [6] extends Android's monitoring mechanism to detect and prevent privilege escalation attacks. It also covers possible Android side-channels.

Some other proposals [4, 15], including ASM, improve resource access control by providing middleware and kernel-level hooking APIs to implement user-level access control models. Additionally, some systems improve this control through memory instrumentation. Instead of modifying Android source code, Aurasium [30] modifies apps in order to allow users to define their own app resource usage, a technique called inline reference monitoring (IRM). DeepDroid [28] enforces enterprise security policies by dynamically memory instrumenting Android system service processes responsible for granting apps access to OS Resources.

To improve data security, some systems focus on protecting system services and providers of high level data (e.g., location). TISSA [31] allows users to specify the information accessible to apps individually. Aquifer [21] allows developers to protect data which is used by different apps in a work chain. Other systems adopt information flow techniques to detect and / or prevent data leaks. TaintDroid [12] uses dynamic taint analysis to monitor sensitive data usage and propagation. AppFence [16] extends TaintDroid in order to perform data-shadowing, and also to prevent sensitive data from being transmitted through the network. Pebbles [27] uses TaintDroid's mechanism to create high level objects composed of different files in order to ensure that no information gets lost.

The focus of all this body of work, however, is complementary to ours. In these systems the user is fully trusted. On the other hand, they suffer either from offering coarse-grained or system-wide restrictions. Additionally, they do not offer the application lifecycle restrictions we need (e.g., prevent untrusted apps from running). For these reasons, these systems fail to provide an application developer the proper mechanisms to secure the scenarios we present in this work.

3. ARCHITECTURE

In this section we present an overview on this work's architectural landscape. We start by giving some background on the *trust lease* primitive, as well as on the new mobile operation mode paradigm we use in developing our proposed applications. Then, we describe the Android-based system that provides the *trust lease* mechanism, on top of which we develop those applications. Next, we present the Strapp Middleware, to help the transparent negotiation and consequent enforcement of *trust leases*. We then describe the architecture and workflows of mExam, mMeeting and mTicket applications separately. In the main document, we present, for each scenario, the threat model, as well as the *trust lease* specification.

3.1 Background

The *trust lease* primitive is an OS mechanism we recently proposed [25], that enables to restrict the access to several device resources, as well as the restriction of several other device functionalities (e.g., preventing untrusted apps from running). This primitive enforces a new mobile computing paradigm that enables two entities, a mobile device user and a third party (e.g., a server), to negotiate restrictions over the user's device, and the conditions under which those restrictions hold. For a matter of simplicity, we call our use case apps, *strapps*.

Operation Mode Paradigm.

The concept of *trust leases* spawned a new mobile device paradigm that introduced the concept of mobile operation mode, more specifically the *unrestricted* and *restricted* modes. This paradigm enforces the negotiation of *trust leases*, which ultimately lead to the transition of users devices from *unrestricted* to *restricted mode*, effectively constraining some of the users' device resources and / or functionalities. The *restricted mode* is a novel operation mode for mobile devices that enables a multi-purpose commodity mobile device to be turned into a special-purpose device, restricting its functionality to a well-defined set of tasks for a limited amount of time. This operation model provides a remote party the guarantees that those restrictions are in place. In order for remote party sides to control users' device states, there needs to be a way to negotiate which and for how long certain restrictions (e.g., Internet access restriction on students' phones) have to be applied on users' devices. This negotiation is essential in guaranteeing the correct behaviour of certain application-specific activities. This is supported by *trust leases*. A *trust lease* is an abstraction that captures the process of specializing a device, by transitioning it from *unrestricted* to *restricted mode*, and granting a remote party behaviour guarantees from that device. Basically the user (known as lessor) issues a *trust lease* on behalf of the remote party (known as lessee) in order to define a contract where both parties agree on:

- a set of restrictions to apply to the lessor's device (e.g., limit the access to the Internet, and the execution of other applications, during an exam);
- a set of conditions that regulate the extent of the specialization (e.g., limit the lease to the maximum duration of the exam, or up until the completion of the exam by the student).

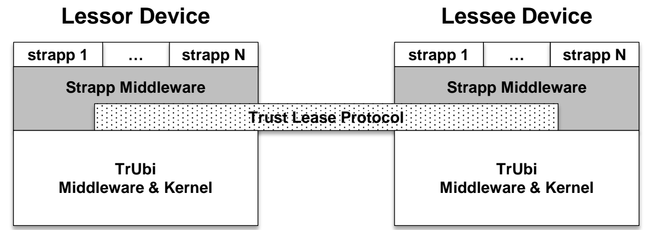

Figure 1: Strapp middleware.

Restricted Mode Threat Model.

The *restricted mode* is essential to the correct execution of the security sensitive scenarios we propose. On one hand, this mobile operation mode paradigm assumes that all lessors (users) are untrusted, as they can try to change device configurations or execute applications that can corrupt the secure execution of these applications' tasks. It is also assumed that lessors (users) may install malicious applications that try to circumvent the *restricted mode's* restrictions, in order to benefit lessors in these scenarios. So, all lessor-side third party apps are considered untrusted except the ones specifically specified by lessees in leases' whitelists. Additionally, the lessor-side's kernel and middleware services are part of the trusted computing base (TCB), and lessees' devices are also considered as trusted. Note, that this paradigm does not address second device side-channels.

The TrUbi System.

TrUbi is a system that supports the recently proposed *trust lease* primitive [25], for Android devices. This system extends both Android's kernel and middleware level resource access control mechanisms (e.g., access to Internet, location). TrUbi also provides features that can better accommodate the application lifecycle requirements that both the mExam and mMeeting *strapps* require (e.g., prevent unauthorized apps from running). In order to control device resources, system services, and application lifecycles, TrUbi essentially monitors these components' access / execution decisions, and each time the OS performs some operation of interest to TrUbi, a callback is triggered in TrUbi, which it then uses to decide whether that operation may or may not proceed accordingly with the lease currently active. In TrUbi, *strapps*, run in the application layer, just like any other app. The interactions between these *strapps* and TrUbi are performed through the API provided by the TrUbi Middleware component. This API essentially offers four *trust lease* handling primitives:

- *startlease* - used on lessors' side to signal TrUbi to start the enforcement of a lease;
- *stoplease* - used on lessors' side to signal TrUbi to stop the enforcement of a lease;
- *quote* - used on lessors' side to request TrUbi to compute the hash of the lease being enforced;
- *verifylease* - used on lessees' side to compare the hash calculated on lessors' side with the one expected, generated locally.

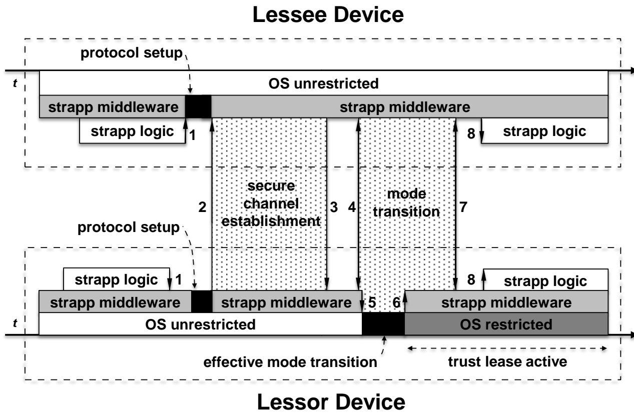

Figure 2: Trust Lease protocol workflow.

3.2 Strapp Middleware

The application scenarios proposed for the mExam, mMeeting, and mTicket *strapps* require that users' devices make a transition from *unrestricted* to *restricted mode*. This transition is transversal to these new application scenarios now supported by TrUbi, effectively establishing a new pattern. The profuseness of this pattern proves that the development of a middleware layer on top of the TrUbi API, capable of providing a protocol that can transparently perform this mode transition can bring tremendous benefits for *strapp* developers. For this reason, we present the Strapp Middleware which provides a new protocol called Trust Lease Protocol (TLP for short). A representation of this middleware can be found in Figure 1. Basically *strapps* on the lessor and lessee side leverage the Strapp Middleware to negotiate leases, over TLP, which are then enforced by TrUbi on the lessor's device. We now present TLP's design.

3.2.1 Trust Lease Protocol

The Trust Lease Protocol is a developer-friendly mechanism that transparently mediates the Trust Lease negotiation between lessors and lessees. The goal of this protocol is thus to negotiate a lease and make the transition of the lessor's device from *unrestricted* to *restricted mode* accordingly. This protocol comprises two main phases: *(1) secure channel establishment*, and *(2) mode transition*. Figure 2 gives a high-level representation of TLP's workflow. In the main document, we detail the design of the *secure channel establishment* and *mode transition* phases.

The protocol starts with both the lessee-side and lessor-side *strapps* setting up the necessary protocol configurations (step 1), using an API that allows *strapps* to manage when they wish to negotiate the protocol (i.e., in a simple start-stop state machine). In step 2 the lessor's protocol side initiates the *secure channel establishment*, which yields a session key when that phase is completed (step 3). This key is then used from there on in the encryption of every communication message of the protocol. Step 4 represents the bi-directional communication where lease conditions are presented and accepted by the lessor. In step 5 the control goes from the Strapp Middleware to the OS so that the lease can be activated, and in step 6 the control returns from the OS back to the Strapp Middleware. Step 7 shows the attestation step where the lessee is effectively notified of the lessor's device mode transition success. Finally in step 8, the Strapp Middleware returns the control to *strapps*.

3.3 Use Case Strapps

In this section, we detail the design and workflows of the mExam, mMeeting and mTicket *strapps*.

3.3.1 mExam

The mExam *strapp* aims to allow students to answer exams using their devices, while preventing them form cheating. In our context, exams must still be answered in examination rooms in the presence of a supervisor, which must also verify that each student takes only one mobile device into the room. To prevent students from cheating, their devices will be restricted to execute a single trusted application (mExam) on their devices while answering the exam. Since the mExam *strapp* runs on the devices exclusively, students cannot use alternative applications to access the network, or retrieve locally available material.

Figure 3 a) shows mExam's workflow. In the day of the exam, already in the class room, the invigilating professor starts by retrieving the list of all enrolled students from the school server (step 1). This list contains the ids, photos and also a token customized for each student. As students enter the room, they communicate with the professor, using NFC (represented in the figure with a dashed line), to identify themselves (step 2). After confirming the identity of a student, the professor sends him the exam *trust lease* (still step 2), which the student accepts, and by doing so, restricts his device (step 3). Then the professor then sends the token to the student (step 4), already through WiFi. The student then uses the token to prove to the school's server he's in fact in the room, and to retrieve the exam statement from the server (step 5). At that point, the student signals the professor he's ready to start the exam. Once the professor identifies all students in the room, he broadcasts a message signalling the start of the exam to students. When the student's exam time runs out, he can no longer perform any changes, as neither the *strapp* or the school server will allow him to do so. Once the student finishes his exam, he makes his complete exam submission to the server (step 6a). If the invigilating professor detects, during the exam, that the student is cheating he may also force the abortion of that student's exam (step 6b). Independently of the way the exam ends, the scenario terminates with the transition from *restricted mode*, back to *unrestricted mode* (step 7).

3.3.2 mMeeting

The mMeeting *strapp*'s goal is to prevent sensitive information leaks in business meetings. Our main concern is the presence of applications on the devices that can cause data leaks, namely: spyware (i.e., malware that senses the surrounding environment through the device's sensors and transmits the collected data to untrusted parties), or legitimate user-installed applications that cause undesired side-effects (e.g., automatic data backup to the cloud). To prevent data exfiltration in business meetings, mMeeting implements a *private mode* policy that prevents all but a set of pre-defined trusted applications (e.g., e-mail client, contacts list) from accessing device's sensors and communication interfaces, in particular: microphone, telephony, SMS, network, Bluetooth, camera, and GPS. Untrusted applica-

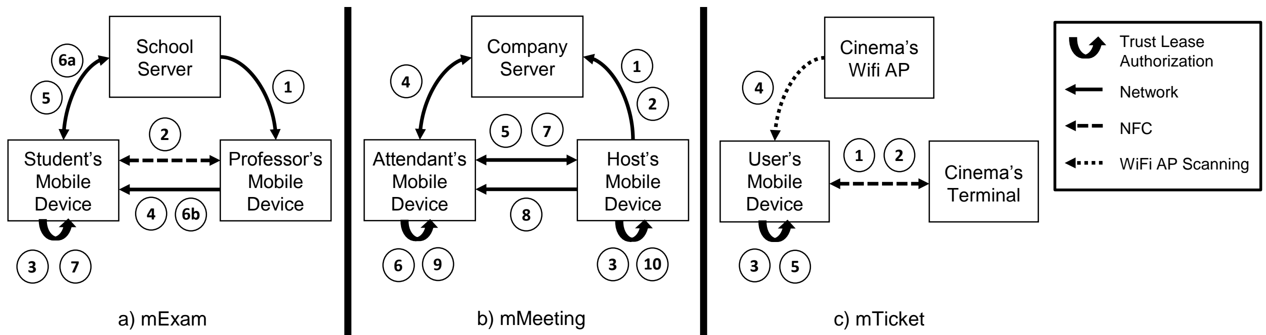

Figure 3: Use Case Strapps: mExam, mMeeting, and mTicket.

tions will be denied access to these resources. To ensure that all attendants' devices run in private mode, the meeting host will be responsible for coordinating the *trust lease* enforcement in everyone's device.

Figure 3 b) shows mMeeting's workflow. Before the meeting starts, the leader registers his IP, and submits the list of people allowed to attend the meeting to the company server (step 1). Then, he signals the beginning of the meeting (step 2) to the company server, and restricts his own device with the meeting's *trust lease* (step 3). Once attendants arrive, they retrieve the host's IP from the server (step 4), and then identify themselves to the host (step 5). After identifying an attendant, the host then sends the meeting *trust lease* in return (still step 5), which the attendant accepts, and by doing so, restricts his device (step 6). During the meeting hosts and attendants can attest each others' devices (step 7). Once the meeting is over, the host signals the end of the meeting to every meeting attendant (step 8), which triggers their devices' transitions from *restricted* back to *unrestricted* mode (step 9). After every attendant's device makes that transition, the host makes that same transition himself, concluding the scenario (step 10). Note however, that if an attendant wishes to leave earlier he's free to terminate the restrictions on his device (step 9).

3.3.3 mTicket

The mTicket *strapp* aims to improve the viewing experience of cinema attendees by restricting their mobile devices' sound during movie screenings. We are mostly concerned about careless or inattentive users, rather than intentionally-driven ones. In addition to ensuring that mobile devices remain silent, we aim to reduce light pollution by keeping screens' brightness dimmed and the cameras' LED off. Restrictions must apply roughly for the movie duration. Figure 3 c) shows mTicket's workflow. It is assumed that before step 1, the user has acquired a movie ticket. In step 1, the cinema NFC terminal validates the user's ticket, and after doing so, it immediately sends the lease conditions to be enforced on the user's device during the movie (step 2). After accepting the lease's terms, the user's device transitions to *restricted mode* (step 3). If a user decides to leave in the middle of the movie, he would automatically regain control of his device (optional step 4), by stepping outside the cinema's WiFi signal range. At the end, either because the movie ended or because the user left the movie earlier, the user's device transitions back to *unrestricted mode* (step 5), terminating the scenario.

```
public abstract class TLProtocolAdapter {
  ...
  public abstract void start();
  public abstract void restart();
  public abstract void stop();
}
```

Listing 1: TLP adapter API.

4. IMPLEMENTATION

In this section we provide some implementation details of this work. Because of space limitations, we focus on the API provided by the Strapp Middleware, and give some details on the network and NFC implementations of TLP. Additionally, we also cover our *strapps*' implementation.

4.1 Strapp Middleware API

The Strapp Middleware provides TLP under a developer-friendly programming paradigm. In order to use TLP, developers must use protocol adapters, whose root class is the `TLProtocolAdapter`. Essentially the protocol adapter family is divided into network and NFC adapters, where each of these branches have their corresponding lessee and lessor adapters, which must be used complementarily in order to successfully negotiate TLP. These adapters, the ones developers must use, depend on their side's corresponding interfaces, (i.e., `TLProtocolLesseeHandshakeListener` or `TLProtocolLessorHandshakeListener`) which developers must implement in their application components, in order for their *strapps* to communicate with the Strapp Middleware. These adapters use a very simple API that can be seen in Listing 1. As the method names imply, both `start()` and `stop()` methods are used to enable and disable the device from being able to negotiate TLP respectively, while the `restart()` method is used to reset this negotiating ability in certain situations (e.g., `stop()` must be called if an activity responsible for handling an adapter must be sent to the background, but in order to reload it back to the foreground, `restart()` must be called instead of `start()`).

In order for applications to communicate with the Strapp Middleware to negotiate TLP, they must implement in their components, their corresponding TLP sides' interfaces. In Listing 2 we can see the methods called by the TLP adapters, through their sides' APIs. It's through these methods that these adapters can notify applications of several different events. As can be seen in those listings, both interfaces provide a `handshakeCompleted()` method, which as the name implies, is used to notify applications of the result of the

```
public interface TLProtocolLessorHandshakeListener {
  void lessorHandshakeCompleted(boolean success, String msg);
  void leaseTimeNotification(int timeToFinish);
}

public interface TLProtocolLesseeHandshakeListener {
  boolean validateLessorId(String lessorId);
  void lesseeHandshakeCompleted(boolean success, String msg);
  void attestationCompleted(boolean success, String msg);
}
```

Listing 2: Lessor and lessee TLP interfaces.

TLP negotiation. The lessor interface provides a `leaseTimeNotification()` method, that is used to notify the application about the time left on a lease. This mechanism must be explicitly specified when setting up the adapter. On the other hand, the lessee interface provides an `attestationCompleted()` method, which similarly to the `handshakeCompleted()` method, is used to notify applications of an attestation request result. For instance, in a mExam app, the result of this method could determine if the student could or could not continue to answer his exam. The `validateLessorId()` method is used as the name implies for developers to design the way they wish their lessee applications to validate the identity of a lessor.

4.2 TLP Network and NFC Implementations

The development of TLP over network communication was done using sockets (`java.net` package), a mechanism used extensively, with plenty of documentation and examples available. For cryptographic operations, we used both the `javax.crypto` and `java.security` packages, for which there is also plenty of information available. For the *mode transition phase* of TLP we leveraged the API provided by TrUbi to start the enforcement of the *trust lease* and to provide the attestation step that give lessee guarantees of a correct lease installation. This implementation uses the adapters and interfaces described in Section 4.1.

The work surrounding cryptographic operations and the installation of *trust leases* in NFC was similar to the one for the network implementation. On the other hand, the work on the communication part of NFC, was much more complex. Because Android's Beam mechanism offers peer-to-peer communication, with the obligation of making the user confirm each message before sending it, we had to rely on a lower level solution that could handle these message round trips transparently to the user. In order to do so, we adopted the Host Card Emulation (HCE) mode, for lessors, and the NFC reader mode for lessees. Similar to the network implementation, the NFC implementation also uses the adapters and interfaces described in Section 4.1. While the network communication relies on a fixed set of messages, exchanged in a certain order, the NFC implementation is based on commands, meaning that the state and corresponding data at each step of the protocol must be handled differently (i.e., it must be kept in memory and reset when the protocol finishes or aborts). This *command-based* approach forced us to create a specific TLP NFC message structure. Another issue from this approach is the message payload max size, which was limited to a maximum of 261 bytes, because of our devices' chip capabilities. Because of the mTicket scenario we also implemented TLP in a Java application, to support the communication with a real NFC reader, mimicking the cinema's NFC terminal.

4.3 Use Case Strapps

The implementation of the three scenarios involved the creation of four applications: *(1a)* mExam *strapp* for students, *(1b)* mExam *strapp* for professors, *(2)* mMeeting *strapp* usable by both meeting hosts and meeting participants, and *(3)* mTicket *strapp* for movie spectators. Their implementation is entirely in Java following the Android API level 19, and their structure is solely based on Activities. In terms of user interface, all these apps use Android custom resource files and their theme is Android's own Theme.AppCompat.

Given that lessees may use the very same *trust lease* in different instances of the same scenario (e.g., a professor only needs to change the exam's start time when administering multiple exams), with the need for very short to no *trust lease* specification changes, we made it so that the creation of *trust leases* in both mExam and mMeeting *strapps* to be saved locally. In practice this mechanism allows for the definition of policy profiles providing a simpler and smoother handling of *trust leases*, reducing lessees' efforts when specifying their needs. By offering this mechanism we also help enforcing our user-friendliness requirement. To support this mechanism we created a set of UI elements (i.e., Activities) that are used in the mExam professor and mMeeting *strapps*, to populate an instance of TrUbi's own `Lease` object. Additionally, we also created a XML writer and parser to save leases in XML format, and to load them back into a `Lease` object, so they can be edited using these UI elements, or used for TLP negotiation. In order to communicate with the central servers, these *strapps* use JSSE to specify their SSLSocketFactories, as well as their KeyManagers and TrustManagers. Given that our webserver communicates through HTTPS, our apps also use Apache's HttpClient to establish those secure connections. The transition of lessors' devices to *restricted mode* is done using the two TLP implementations. mExam lessors and lessees negotiate TLP through NFC using their devices. mMeeting participants rely on the network implementation of TLP for the *trust lease* negotiation on their devices. mTickets lessors use the NFC implementation between their devices and the NFC reader.

5. EVALUATION

In this section, we present the evaluation methods and corresponding results, conducted throughout this work. We start by presenting the effects of *trust leases* on real world applications, and on a real world spyware. Afterwards, we evaluate the performance of TLP focusing on the times of its two phases, and also on the times of the network and NFC implementations. We conclude by analysing the results of our *strapps*' usability tests and adoption study.

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

5.1 Trust Lease Effects on Applications

In these sections we first present the effects of *trust leases* in real world applications, downloaded from Google Play. Then we present the impact of using *trust leases* against a

real world spyware [1], already identified by a major antivirus company.

5.1.1 Real World Applications

Our application study aims to: (1) evaluate the effectiveness of the enforcement of *trust leases* on real applications, and (2) study how such applications react when subjected to constraints while accessing devices' resources.

Methodology.

To achieve our goals, we collected a test suite of unmodified real world applications and observe the effects of restricting such applications using *trust leases.* In our experiments, we have to contemplate three access patterns:

- **Access before lease (ABL)** – an application starts accessing a resource, and a *trust lease* is activated. Once the *trust lease* starts, access to the resource must be blocked.
- **Access during lease (ADL)** – the *trust lease* is active, and an app accesses a restricted resource. Access must be denied.
- **Access after lease (AAL)** – an application has been previously restricted, and the *trust lease* terminates. The application must be able to regain access to the resource.

For our test suite, we selected applications that perform operations that can be constrained by different *trust lease* restrictions: execution of applications (cpu), network, camera, phone call, SMS, microphone, sound, brightness, LED, and Bluetooth. To test each type of restriction, we use a mix of: system apps from the vanilla Android distribution (e.g., System Camera), and third-party apps from the Google Play app market [2]. We chose third-party apps that were popular and highly rated by March 2015. Each type of restriction was tested with 10 apps, except for the restrictions on: airplane mode, time, and screenshot. The reason for not testing these restrictions with multiple apps is that they can be triggered only through a limited set of entrypoints, such as the system settings app, a combination of the power and volume down buttons, or through the Android Debug Bridge. In total, we collected 96 different third-party apps. Most of these apps were used to test a single type of restriction. However, since Android has no specific permissions for constraining the execution of applications, we reused 9 of these apps to test the cpu restriction. We executed our test suite on ABL, ADL, and AAL.

Findings.

For all tested applications, *trust leases* were effective at enforcing the intended restrictions, i.e., whenever a *trust lease* is active, applications have no access to the resources that are constrained by the *trust lease.* Table 1 presents detailed information of our findings for: ABL, ADL, and AAL.

- **Under ABL -** applications follow 3 policies: they either stop having access to the resource (policy 1), are killed (policy 2), or are frozen (policy 3). Policy 1 applies when TrUbi can intercept all operations on a given resource performed by applications. 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 (policy 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 (policy 3).

Resources	Size	ABL	ADL	AAL
	S / 3 / T	**B / K / F**	**C / I / D**	**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 third-party 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 1: Effects of *trust leases* on real mobile apps.

- **Under ADL -** different apps react differently to the access revocation of their resource / functionality of need. For example, for the camera, 9 out of 10 apps terminate gracefully; 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.

  Note that when applications are denied access to a particular resource, side-effects can occur affecting the consistency between the app's internal state and its visual interface (GUI). In 38 apps, we observed such inconsistencies. For example, if the user turns on the Bluetooth and a lease restricting Bluetooth is active, an application will misleadingly show Bluetooth as enabled. In contrast, 61 apps update their visual interfaces consistently. For instance, all phone dialer apps refresh their GUI properly by disabling calls. Apps not allowed to run are considered inconsistent, because their state is frozen. Under ADL, one app crashed due to a null pointer exception.

- **Under AAL -** in most cases, applications seamlessly regain access to a resource (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. For instance 6 apps require explicit refresh action by users,

Trust Lease Protocol Times				
		SEC phase	MT phase	Total
Network	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 2: Performance of *trust lease* installer.

in order to detect Internet connection availability. In more extreme cases, apps must be entirely restarted. This occurs in 13 cases: 10 camera apps, and 3 apps handling microphone, brightness, and SMS.

5.1.2 Real-World Malware / Spyware

Inspired by the mMeeting use case, we wanted to test if *trust leases* can effectively mitigate real-world spyware. With this in mind, we tested *trust leases* with the Dendroid trojan horse [1]. It consists of a seemingly inoffensive Android application that periodically connects to a remote web server listening for specific commands, which enable an attacker to stealthily execute operations on the victim's phone: record audio, take pictures, make phone calls, control sound volume, open web pages, launch apps, and access private information, such as contacts, text messages, and personal files. We created leases and observed that they mitigate Dendroid's malicious activities by blocking access to sensitive resources, preventing communication with the server, or suspending Dendroid's processes.

5.2 Trust Lease Protocol Performance

The measurements of the performance of TLP were conducted for both network (i.e., WiFi) and NFC communication. For benchmarking purposes the lease terms were accepted instantaneously. Table 2 shows the total execution time for each setup: 1.1 and 1.59 seconds for network and NFC, respectively. To better understand these numbers, we break them down into two constituent parcels: *secure channel establishment* (SEC) and *mode transition* (MT) phases. SEC is the first phase of TLP, which takes 2 round trips to execute over network communication, and 8 round trips over NFC. This difference relates to the need of breaking the high-level messages into shorter NFC specific messages. The results show that this need leads to a performance penalty, where NFC takes over two times longer to complete the SEC phase, when compared with network. The difference between the number of round-trips and performance is not linear, because NFC's latency is lower than the network's. MT ensues the SEC phase and comprises 4 round trips to execute over network, and 11 round trips over NFC. In this case, the results from the table show that the time difference between network and NFC are shorter, when compared to the ones from SEC, due to the smaller difference of the round trips involved. Additionally, Table 2 shows that while SEC takes about 29% of the total time of the network implementation, it takes 44% of the NFC's total time. The reason behind this difference is the connection setup time, which is higher for NFC, but also due to the difference in the number of round trips, i.e., 2 - 4 for network, and 8 - 11 for NFC.

5.3 Application User Tests

In order to evaluate whether our *strapps* are effective in supporting their corresponding scenarios, and also to assess how *trust leases* would be received and handled by real users, we conducted usability tests with potential *strapp* users. Due to the relatively simple menus and interactions required by users to complete the tasks we defined for the tests, users generally found the apps intuitive and easy to use. The *trust lease* primitive proved to be effective in keeping the scenarios safe, i.e., they prevented users from accessing unauthorized resources, and from running unauthorized apps. In general, the biggest criticisms had to do with the lease conditions, which according to the users, had to much or not clear enough information, which caused confusion. Additionally, the lack of system notifications capable of informing the user about the reasons behind the inability to perform certain actions, was also called to our attention. In general, users understood the benefits of these apps, and although some shown privacy concerns about the restrictions imposed, they generally would adopt these apps.

5.4 Strapp Adoption Study

In order to get a clearer perspective on the openness of real users in adopting a system and applications that could offer the functionalities we propose in this work, we developed a set of adoption questionnaires, for the six different types of entities involved in our three scenarios: students, professors, meeting organizers, meeting attendants, movie spectators, and movie theater managers. In this document we focus on the answers from students, attendants, and spectators. Regarding the mExam, mMeeting, and mTicket *strapps*, users generally identify themselves with the issues these apps address, and the majority see the benefits of these apps, and admit they would adopt them. When presented with the need for the restrictions for each scenario in particular, users from the mExam, and mTicket scenarios, seem to better understand them. Although they have concerns about allowing an OS to provide the restrictions we need, to certain apps, they admit they feel more comfortable about the restrictions, when applied to these scenarios in particular, as they understand their benefits in securing these scenarios. On the other hand, mMeeting users feel these restrictions are too strict, and are therefore less comfortable with them. In terms of the possibility of a lessee restricting more than he should, or access personal data, users generally feel worried.

6. CONCLUSIONS

In this work, we focused on evaluating the effectiveness of a new OS primitive, called *trust lease*, in supporting certain mobile device usage scenarios not supported by today's MOSes. In order to do so, we leveraged TrUbi, an Android-based system that offers the primitive for Android apps. We designed and implemented a middleware layer, that allows for users and third parties to negotiate *trust leases*, and also three use case apps: *(1)* mExam, an app that allows students to answers exams in their devices without cheating, *(2)* mMeeting, an app that prevents meeting information leaks, and *(3)* mTicket, an app that prevents movie audience disturbances. We shown that the primitive does not trigger erratic behaviours in real-world apps, and that it can even be a good mechanism to stop spyware. Additionally, we analysed the performance of the negotiation protocol over NFC and network communication. Then, we performed user tests

on our apps, that show they are easy to use and intuitive, but also show the OS should do a better job at informing users about the restrictions. We concluded with an adoption study, that shows that although users see the benefits of the apps, the restriction model still faces some user scepticism.

References

[1] Dendroid. `http://www.symantec.com/security_response/writeup.jsp?docid=2014-030418-2633-99`. Accessed October 2015.

[2] Google Play. `https://play.google.com/store`. Accessed October 2015.

[3] Think Exam. `http://www.thinkexam.com/index.html`. Accessed October 2015.

[4] Michael Backes, Sven Bugiel, Sebastian Gerling, and Philipp von Styp-Rekowsky. Android Security Framework: Enabling Generic and Extensible Access Control on Android. In *Proc. of ACSAC*, 2014.

[5] Alastair R Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proc. of HotMobile*, 2011.

[6] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. Xmandroid: A New Android Evolution to Mitigate Privilege Escalation Attacks. Technical report, Technische Universität Darmstadt, Technical Report TR-2011-04, 2011.

[7] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. Practical and Lightweight Domain Isolation on Android. In *Proc. of SPSM*, 2011.

[8] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In *Proc. of USENIX Security*, 2013.

[9] Jordi Castellà-Roca, Jordi Herrera-Joancomarti, and Aleix Dorca-Josa. A Secure e-Exam Management System. In *Proc. of ARES*, 2006.

[10] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. CRePE: Context-Related Policy Enforcement for Android. *Information Security*, 6531:331–345, 2011.

[11] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S Wallach. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *Proc. of USENIX Security*, 2011.

[12] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. of OSDI*, 2010.

[13] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission Re-Delegation: Attacks and Defenses. In *Proc. of USENIX Security*, 2011.

[14] Tor-Morten Gronli, Jarle Hansen, and Gheorghita Ghinea. A Context-Aware Meeting Room: Mobile Interaction and Collaboration Using Android, Java ME and Windows Mobile. In *Proc. of COMPSAC*, 2010.

[15] Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. ASM: A Programmable Interface for Extending Android Security. In *Proc. of USENIX Security*, 2014.

[16] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *Proc. of CCS*, 2011.

[17] Rui Joaquim, Paulo Ferreira, and Carlos Ribeiro. EVIV: An End-to-end Verifiable Internet Voting System. *Computers & Security*, 32:170–191, February 2013.

[18] Michael Kern and Johannes Sametinger. Permission Tracking in Android. In *Proc. of UBICOMM*, 2012.

[19] Hu Linli, Wang Yuhao, and Li Dong. Uniticket: A Third Party Universal e-Ticket System Based on Mobile Phone. *Wireless Engineering and Technology*, 2(3):157–164, 2011.

[20] Peter Loscocco. Integrating Flexible Support for Security Policies Into the Linux Operating System. In *Proc. of FREENIX Track*, 2001.

[21] Adwait Nadkarni and William Enck. Preventing accidental data disclosure in modern operating systems. In *Proc. of SIGSAC*, 2013.

[22] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *Proc. of ASIACCS*, 2010.

[23] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically Rich Application-Centric Security in Android. *Security and Communication Networks*, 5(6):658–673, June 2012.

[24] Giovanni Russello, Mauro Conti, Bruno Crispo, and Earlence Fernandes. Moses: Supporting operation modes on smartphones. In *Proc. of SACMAT*, 2012.

[25] Nuno Santos, Nuno O. Duarte, Miguel B. Costa, and Paulo Ferreira. A case for enforcing app-specific constraints to mobile devices by using trust leases. In *Proc. of HotOS*, 2015.

[26] Stephen Smalley and Robert Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proc. of NDSS*, 2013.

[27] Riley Spahn, Jonathan Bell, Michael Lee, Sravan Bhamidipati, Roxana Geambasu, and Gail Kaiser. Pebbles: Fine-Grained Data Management Abstractions for Modern Operating Systems. In *Proc. of OSDI*, 2014.

[28] Xueqiang Wang, Kun Sun, Yuewu Wang, and Jiwu Jing. Deepdroid: Dynamically enforcing enterprise policy on android devices. In *Proc. of NDSS*, 2015.

[29] Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. Compac: Enforce Component-Level Access Control in Android. In *Proc. of CODASPY*, 2011.

[30] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proc. of USENIX Security*, 2012.

[31] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proc. of TRUST*, 2011.