2011 IEEE International Conference on Privacy, Security, Risk, and Trust, and IEEE International Conference on Social Computing

YAASE: Yet Another Android Security Extension

Giovanni Russello
Create-net
Trento, Italy
giovanni.russello@create-net.org

Bruno Crispo
Università di Trento
Trento, Italy
crispo@disi.unitn.it

Earlence Fernandes
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
earlence@few.vu.nl

Yuri Zhauniarovich
Università di Trento
Trento, Italy
zhauniarovich@disi.unitn.it

***Abstract*—Three hundred and fifty thousand Android phones are activated each day. The open philosophy adopted by Google makes it easy for third-parties to develop and distribute applications. Unfortunately, the same applies to malicious applications that pose a real threat to users' privacy. The limited security model implemented on the Android Platform has failed in thwarting these attacks. In this paper, we present Yet Another Android Security Extension (YAASE) that provides a fine-grained security mechanism while protecting the user from malicious applications that attempt to leak sensitive information via network access or by privilege spreading through collusion. We have implemented YAASE and evaluated its performance overhead. Preliminary results show the approach is indeed feasible.**

I. Introduction

Android shook the world of smartphones. It created an ecosystem full of developers eager to embrace and make use of the opportunities offered by smartphones. Thanks to the open model supported by Google, developers are not forced to pay high code certification fees or sharing a significant percentage of their profit with application distribution points (a.k.a. "markets"). The fast growth of Android smartphone operating systems's market shares is the tangible proof a real demand was met [4]. Not surprisingly, also the number of applications developed for Android is knowing a similar growth rate [3].

Unleashing the smartphone application market was beneficial also for end-users who can now choose among a huge variety of applications. While most of these applications achieve their goals without abusing users' privacy, some of them have recently hit the media for being quite effective in doing the opposite [5], [1]. Furthermore, an open and popular platform as Android, provides a perfect environment to exploit and disseminate security attacks.

To prevent such undesirable situations, Android incorporates security mechanisms and features that allow partial protection of user's privacy from malicious applications. However, developing an efficient and usable security model suitable for battery powered devices which is intended for use by a wide range of people is not an easy challenge. In fact, while addressing many security issues, the Android security model has also several shortcomings [15].

Some of these have been already addressed by the many Android security extensions already proposed in the literature. Some [13], [14], [9], [18] extended run-time security features, others [6] extended the class of supported security policies. Even the problem of information flow was covered by one of the earliest extensions [13].

Up to date, all proposed extensions fall short in covering limitations related to the ability of applications, especially colluding ones, to breach user's privacy. Those addressing privacy are still unsatisfactory as discussed in the next section. This paper proposes *Yet Another Android Security Extension* to fill this gap providing in a single solution all the mechanisms users need to protect their privacy as they wish.

A. Motivations

Despite some initial attempts to address the privacy issue, the solution proposed by MockDroid[7], TISSA [23], and AppFence [16] are too coarse-grained. These approaches apply indiscriminate modifications to the whole data of a content provider. For instance in TISSA, if a user selects to apply anonymisation when an application reads her contact provider, the change will be applied to the whole set of her contacts. Another issue is related to the control of information leakage over network accesses. For instance, in AppFence it is possible to block the internet connection of an application to avoid exfiltration of data. This blocking mechanism is applied whenever the application sends sensitive data over the internet. Such a mechanism can be too restrictive since it would be more desirable to control where the application is sending the sensitive data and to block only the communications to illegitimate destinations.

Android security model mostly relies on application sandboxing. However, it has been reported that this model is vulnerable to privilege spreading attacks [13]. In this type of attacks, an unprivileged application exploits the permissions of privileged applications. If the malicious application is leveraging a vulnerability of a legitimate application then this type of attacks is often referred to as confused deputy attacks [12]. However, given that developing applications for the Android Market is quite simple (just pay a $25 fee), designing colluding applications that on purpose provide to other applications permissions without the user being aware of it is becoming increasingly popular [8], [21], [17], [11].

B. Contribution

The main contribution of this paper is to provide in a coherent framework a solution for all the above issues. In particular, our approach named **YAASE (Yet Another Android Security Extension)** provides a very fine-grained

978-0-7695-4578-3/11 $26.00 © 2011 IEEE
DOI

policy enforcement mechanism where policies define several levels of control granularity over accesses to phone resources. YAASE leverages user-defined labels associated with data and enforces the correct dissemination of the tagged data for both application-to-application and application-to-internet communications. In YAASE, data tagging and tracking is performed by means of the TaintDroid tainting capability. However, we have extended TaintDroid for supporting user defined labels and for performing label modifications due to the enforcement of filtering policies on tagged data. YAASE policies define the data labels that an application is authorised to handle as defined by the user. If an application tries to access data tagged with a label for which no policy grants the access, then the application will not be permitted to access the data. The data tracking functionality in YAASE is completely transparent to the applications. Any application developed for standard Android is able to execute when YAASE is installed. This means that the data tracking functionality does not require the collaboration of applications (as in contrast with QUIRE [12]). Therefore, YAASE is able to protect the user from privilege spreading attacks even in the presence of colluding applications (hence also from the more specific confused deputy attacks). Finally, we have implemented our framework and have performed preliminary evaluations.

The rest of this paper is organised as follows. In Section II, we review existing approaches that aim at extending the security mechanism of the Android platform. In Section III, we provide an overview of the Android security model. Section IV discusses the architecture of YAASE in details. We describe here the modules introduced by YAASE and the extensions made to the Android standard components. Section V presents our policy language and provides some examples of policies written for two real application scenarios. We have implemented YAASE and performed preliminary evaluations. In particular, we have compared the overhead introduced by YAASE with the standard Android. Our findings are reported in Section VI. Finally, Section VII concludes this paper providing some final remarks and highlighting our future research directions.

II. Related Work

In Android, permissions can be requested and granted only at installation time. To remove this limitation, several approaches have been proposed to address the problem of specifying and changing fine-grained policies during runtime. Saint [19] is an install and run-time application management system for Android aiming at protecting also applications from each other. The authors built Saint to allow Android to be able to enforce application policies that allow an application A to define which application can access its interfaces, how other application use these interfaces, and to select at run time if using interfaces of other applications.

Nauman et al. [18] and Conti et al. [10] propose security extensions to support context-related policy enforcement at run-time. Bai et al. [6] has further extended the Android security model to support part of the UCON model. All these papers address some security limitations but none of them address the problems presented in this paper.

More recent papers [7], [23] concentrate on the protection of the user's private data. [7] introduces a system which can limit the access of the installed applications to the data (SMS, contacts, calendar, location and device ID) and the components of the Android OS (access to the Internet and broadcast intents). Applications are agnostic of these limitations. For instance, an application querying the contacts' provider may receive no results even if the provider is not empty. This approach is widened by Zhou et al [23]. Their work provides users with the ability to define the accuracy level of the information revealed to the application. They introduce four levels of privacy: *trusted, empty, anonymised or bogus*. For each native content provider users can set the privacy policy. *Trusted* means that for the function the real information can be provided. *Empty* means that an empty record is returned. *Anonymised* means that the information is somehow anonymised before sending it to the application. *Bogus* means that fake information is forged for the application. The paper does not solve the problem of privilege spreading and unauthorised leakage of private information to an arbitrary internet address.

TaintDroid [13] proposes dynamic taint analysis to prevent runtime attacks and data leakage. TaintDroid is capable of tracking sources of sensitive data. It assumes that third-party applications are not trusted and monitors how the applications propagate sensitive data. TainDroid limits the flow of sensitive data by tracking the taints in the outbound network connections. TaintDroid is not capable of enforcing fine-grained policies to let only specific tagged data to flow to application or to network connections. Similar to TaintDroid, Paranoid Android [20] proposes tainting of data for runtime checks. In Paranoid Android security analysis is executed by a trusted remote server, which hosts the replicas of smart phones in virtual environments. However, this approach has a severe impact on the device performance since execution traces have to be continuously sent over the remote servers.

QUIRE [12] provides a lightweight provenance system that prevents the confused deputy attacks where a malicious application abuses the interfaces of a trusted application to perform an unauthorised operation. QUIRE addresses the problem by tracing RPC chains to establish if all callers in the chain have the necessary privileges to execute the call. Tracing is realised by modifying the Android native RPCs. This however has the drawback that QUIRE solution is not transparent to application developer that need to rewrite their existing applications. Furthermore, QUIRE does not solve the problem of filtering sensitive data based on user's policies and the leakage of information to unauthorised remote sinks via internet. Also, QUIRE is not effective against colluding applications that try to spread privileges.

A solution similar to ours is AppFence [16]. By using Taintdroid's tainting capability, AppFence provides additional mechanisms to shadow sensitive data and to block exfiltration, that is the unauthorised leakage of data via network access. Shadowing allows only data anonymisation and does not

support other transformations over sensitive data. The authors present in the paper also an assessment on the side effects of such privacy enhancing mechanisms over a set of 50 applications and observe that roughly two third of them do not present any user visible side effect. Also, different from our solution, AppFence does not address the problem of privilege spreading of colluding applications.

XManDroid [8] performs runtime monitoring and analysis of communications between applications. It maintains a system state of the applications installed in a device and all the communications links (control and data flow) established among the applications. XMAnDroid monitors the ICC traffic and validates whether an ICC call can potentially lead to a spreading of privileges according to a desired system policy. The main limitation of the this approach is that cannot be used to control communication channels established outside the ICC framework, such as Internet communications. Another shortcoming of XManDriod is that it does not support fine-grained access policies, thus enabling only all-or-nothing access on the data.

III. Android Security

Google Android is a Linux-based mobile platform developed by the Open Handset Alliance (OHA) [2]. Most of the Android applications are programmed in Java and compiled into a custom byte-code that is run by the Dalvik Virtual Machine (DVM). In particular, each Android application is executed in its own address space and in a separate DVM. Android applications are built combining any of the following four basic components. *Activities* represent a user interface; *Services* execute background processes; *Broadcast Receivers* are mailboxes for communications within components of the same application or belonging to different applications; *Content Providers* store and share application's data. Application components communicate through messages called *Intents*.

Focusing on security, Android combines two levels of enforcement [15], [22]: at the Linux system level and the application framework level. At the Linux system level Android is a multi-process system. During installation, an application is assigned with a unique Linux user identifier (`UID`) and a group identifier (`GID`). Thus, in the Android OS each application is executed as a different user process within its own, isolated, address space.

At the application framework level, Android provides access control through the Inter-Component Communication (ICC) reference monitor. The reference monitor provides Mandatory Access Control (MAC) enforcement on how applications access the components. In the simplest form, protected features are assigned with unique security labels—*permissions*. Protected features may include protected application components and system services (e.g. Bluetooth). To make the use of protected features, the developer of an application must declare the required permissions in its package manifest file - `AndroidManifest.xml`.

As an example, consider an application that needs to monitor incoming SMSs, `AndroidManifest.xml` included in the application's package would specify: `<uses-permission android:name= "android.permission.RECEIVE_SMS"/>`. Permissions declared in the package manifest are granted at the installation time and can not be modified later. Each permission definition specifies a protection level which can be: `normal` (automatically granted), `dangerous` (requires the user confirmation), `signature` (requesting application must be signed with the same key as the application declaring the permission), or `signature or system` (granted to packages signed with the system key).

IV. YAASE Architecture

Our main goal is to introduce in the Android platform a flexible and powerful privacy enforcement framework transparent to the applications. To achieve this goal, we have modified part of the Android framework and core libraries, and a set of services and managers that reside outside the application VM. Figure 1 shows the architecture of our framework. In the figure, the dashed blocks represent Android components that we have modified, while gray blocks are the new components introduced by YAASE.

We want to track the data stored on the phone that is accessed by the applications and how this data is disseminated both within the phone (i.e., from one application to another) and when the data leaves the device (i.e., an application sends the data over the internet). For tracking the data, we use the TaintDroid labelling framework. We have extended TaintDroid to be able to tag sensitive information with taint values that are defined in our system in the *Labelling Store*. Each taint is represented as a 32-bit value used to define the control group, the taint label, and some extra information used for history based inspection. The control group is used to specify whether the data is coming from a system resource such as the GPS provider by means of the `''SYSTEM_SENT''` tag. Also the control group can be used to specify that the label associated with a data can be set as a consequence of a policy evaluation. This is particular useful if the taint of data needs to be augmented with labels to keep track of all the applications that have received the data.

Once the data is tainted, we need to make sure that the data is propagated according to the user's requirements. In our system, user's requirements are represented as a set of policies that can be defined per application (more on this aspect in Section V-C). Policies are stored in the *Policy Provider*. To be able to enforce policies and perform appropriate actions on the data, we need to place hooks in the components of the Android framework. For enforcing policies that control the propagation of data from providers of the Android framework to applications, we place a *Policy Enforcement Point (PEP)* in the `LibBinder` module. In this way, we are able to enforce policies for controlling access to simple resources, such as device ID (IMEI), phone number and location data, as well as complex data such as user's calendar and contact entries. In the `LibBinder`, we intercept the standard cursor from where we extract the `CursorWindow`. The `CursorWindow` provides

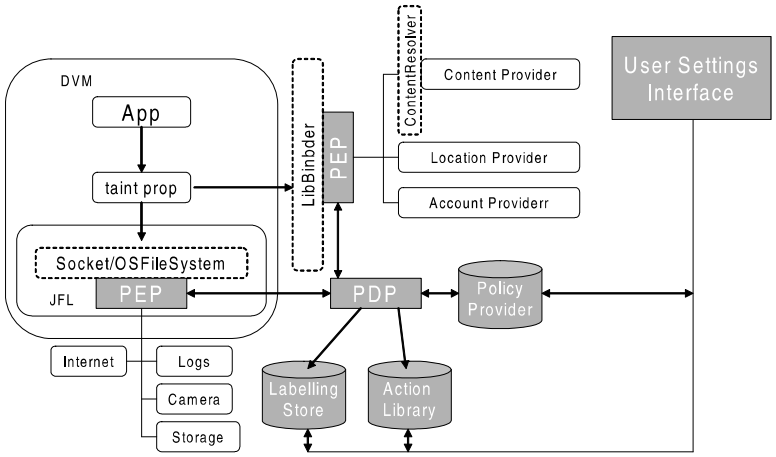

Fig. 1. The YAASE architecture.

methods that can be used for modifying the data contained in the cursor. Using the `CursorWindow` allows us to filter out from the cursor data only part of the information. In this way, our enforcement mechanism achieves a fine-grained filter capability that other solutions, such as AppFence, are not capable to achieve. For instance, AppFence is only able to substitute the content of the returned cursor with a shadowed content. In our framework, policies can perform several type of actions on the data before returning it to the requester. The actions that can be invoked by the policies are defined in the *Action Library*. More details on this feature will be provided in Section V where we present our policy language and several examples of policies. Another PEP is placed in the Java Framework Library (JFL) of the Dalvik Virtual Machine for capturing operations on the file system (such as reading and writing on the local storage as well as accessing the phone camera and microphone) and on the network stack for controlling network traffic even if sent over an encrypted socket (SSL). We have modified the `socket.open(address)` method to inspect the address to where the data is sent.

In this way, we can restrict the use of only authorised addresses or substitute the address specified by the application with an address defined by the user. By modifying the `sendStream()` we are able to intercept the data before it is sent and perform some actions, such as filtering or substitutions. Again, we would like to stress that this level of control granularity is not achieved by any other approach known to us.

When an application requests access to a resource, the PEP intercepts the request. The PEP collects information about application UID, the resource being accessed and the type of operation. Moreover, if available the PEP collects the tag information associated with the data. The PEP forwards this information to the *Policy Decision Point (PDP)*. The PDP uses the information received by the PEP to retrieve the policies relevant to the request from the Policy Provider. Based on the evaluation of the policies, the PDP might decide not to allow the request or to perform some action on the data before returning it to the requesting application. The PDP informs the PEP of the decision and then it is the responsibility of the PEP to take the necessary actions for the enforcement of such a decision.

Finally, we have included in the Android application installer the *User Settings Interface (USI)* component. This interface enables the user to define/modify policies when an application is installed. Moreover, a user can set data labels to be associated with resource providers and that are used by the tainting mechanism. The USI enables advanced users to install routines to perform actions (such as filtering, generating hash values for the phone IMEI, etc) that can be used by the policies. The USI also allows the user to change at runtime any of the security configurations defined at installation time.

V. YAASE Policy Language

In this section, we first present our policy language for controlling data dissemination in the Android platform. Afterwards, we will present two application scenarios to demonstrate how policies can be specified to control data access. Finally, we provide some details on how policies can be automatically generated at application installing time.

Figure 2 shows the syntax of a YAASE policy. Policies are identified by a name and define what `operations` a `Requester` application can execute on a `Resource`. In our framework, a resource can be represented by either the content and service providers within a phone, or other applications that expose services for other applications to be invoked. A policy can have an optional clause **`have to perform`** that specifies actions that have to be performed if this policy is enforced. In our framework, we have provided a set of libraries that can perform several actions on the data (i.e., filtering, anonymisation, generation of random values) and changing the values of the parameters of the operation being executed. The **`handle`** clause defines an expression on the labels associated

```
PolicyName: Requester can do operation on Resource
            [have to perform action]
            handle dataLabelExpression
```

Fig. 2. The syntax of our policy language.

```
1 OrgAppP1: OrgApp can do getContacts on Contacts
2             have to perform filterOut(``Pr'', returnData);
3             handle ``Pub'' and ``Contact''
4
5 OrgAppP2: OrgApp can do send on Internet
6             have to perform sendOnlyTo(aCloudUrl);
7             handle ``Pub'' and ``Contact''
```

Fig. 3. The policies specified for the OrgApp scenario.

with the data that is being passed from the `Requester` to the `Resource` if `operation` is a setter method, that is a method that does not return any data. Otherwise, if the operation is a getter method that provides data from the `Resource` to the `Requester`, the expression defined in **`handle`** clause refers to the labels associated with the returned value.

In the following, we show the features of our language by means of policy examples for different application scenarios.

A. Fine-grained Access Control Policies

We start with some examples of policies for fine-grained control over applications accessing user data. Let us consider an Organiser application (OrgApp) providing a smart way of organising the user's contacts and a back-up capability that stores the user's contacts over a cloud service. For its functionalities, the OrgApp requires access rights to the phone contact provider and to the internet service. In this scenario, the user wants that only her work contacts are accessible to the OrgApp. Moreover, the user wants to make sure that the OrgApp sends the contacts to a specific location over the internet. The user's requirements can be expressed by two policies as in Figure 3.

In particular, the policy `OrgAppP1` specifies that the OrgApp can perform the `getContacts` operation on the contact provider. We assume the user has tagged each contact entry with a label to indicate whether it is a working contact (associated with label `''Pub''`) or a private contact (associated with label `''Pr''`).[1] The contacts are returned to the OrgApp through a cursor containing all the entries. This means that the cursor will be tagged with labels "Pub" for the working entries, "Pr" for the private entries, and "Contact" to indicate it contains data coming from the contact provider. To remove from the contacts private entries, the policy invokes the `filterOut` action on the returned cursor (indicated as `returnData`) removing all the entries tagged with label `''Pr''`. After the execution of the `filterOut` action, the new cursor is now tagged with only "Pub" and "Contacts" labels. At this point, the condition specified by the **`handle`** on line 3 is satisfied and the data can be returned to the OrgApp. If the **`have to perform`** clause had not been specified, then the cursor would have been tagged also with label "Pr". Because the expression of the handle clause defines which are the only labels permitted then the policy would have denied the returning of the data to the OrgApp.

The policy `OrgAppP2` authorises the OrgApp to access the internet. However, it enforces that the OrgApp can connect only to a specific url that can be decided by the user. This is achieved by defined the value `aCloudUrl` and specifying the `sendOnlyTo` action in the **`have to perform`** clause. To make sure the data that is sent are only public contact entries, the **`handle`** clause is specified for handling only data tagged with `''Pub''` and `''Contacts''` labels. With this policy, we can prevent the application to send the user's contact to other online services. Moreover, if the application would get access to other type of data (for example call history) tagged with other labels then this policy would not allow the sending of this data over the internet.

YAASE allows the execution of very fine-grained controls that are not supported by other approaches. For instance, the data shadowing and exfiltration blocking supported by AppFence are too coarse-grained control mechanism to let the OrgApp to function correctly while guaranteeing a desirable level of user's privacy. Data shadowing is a means for providing *fake* data. In this scenario, the data shadowing approach of AppFence would have returned to OrgApp an empty list of contacts making OrgApp completely non functional for the user. If the user decides not to use data shadowing, then OrgApp would get access to the whole set of contacts, that is tagged with private and public label. When the OrgApp would send the contacts to the cloud service, AppFence would block the exfiltration of tainted data even if the OrgApp is sending the data to the url set by the user. Exfiltration blocking prevents tainted data to be written in a socket. Basically, when an output buffer contains tainted data AppFence drops the buffer *covertly*, misleading the application by indicating that the buffer has been sent, or *overtely*, by emulating the OS behaviour when the device is in airplane mode. Again, such a mechanism is too restrictive and would make the OrgApp useless to the user.

B. Preventing Access Right Spreading

One of the main security issues of the Android platform is the spreading of access permissions. Applications can implement services. These services can be invoked by other applications. In this way, the application implementing the service can allow other applications to use its permission to access phone resources. To make matter concrete, let us consider the following scenario. An application A requests permission to access the internet. The application A could pose as a simple application to provide news feeds and it would not look suspicious to the user. An application B that acts as a navigation application requires permission to access the GPS to display the user's current position. Moreover, application B implements a service that allows other applications to access the GPS through application B permission. Application A has

[1]To tag entry contacts the user can use an extended contact provider that supports such a capability.

```
PolicyA1: A can do send on Internet
            handle ``NoLabels''

PolicyB1: B can do access on FineLocationGPS
            handle ``FineLocation''
```

Fig. 4. Policies for preventing permission spreading through backdoor services.

```
PolicyA2: A can do access on LS
            handle ``NoLabels''

PolicyB2: B can do access on LS
            handle ``FineLocation''
```

Fig. 5. Policies for preventing permission spreading through file sharing.

code that invokes the service of application B to get access to the GPS without the user knowing about this. Thus, application A without asking the user for the GPS permission would be still able to get that information and it could be able to leak the user's location over the internet.

YAASE can prevent this type of attacks by means of controlling the data flow through applications. When application A and B are installed the policies in Figure 4 will be generated.

When application A accesses the internet, `PolicyA1` would be enforced. This policy allows the application A to use the internet service. However, the data that can be sent from application A to the internet must not be tagged with any labels. `PolicyB1` takes care of letting application B access the fine location service and lets only data tagged with the ``FineLocation'' label to be propagated. When the application A accesses the service exposed by B, it will be able to access the location data. However, when A tries to send the data to the internet, this data will be still tagged with label ``FineLocation''. Therefore, `PolicyA1` will not authorise the execution of such action because it is in conflict with its **handle** clause stating that only no tagged data can be sent over the internet through application A.

Another way for application A and B to maliciously collaborate is by means of indirect flow: for instance by means of sharing data through the mobile local storage. Both applications in addition to the permissions requested before also require access to the local storage that the user is willing to grant. As a consequence of this, two new policies will be generated as shown in Figure 5. `PolicyA2` authorises application A to access the local storage. However, the data that can be written or retrieved should not have any labels. `PolicyB2` grants access to application B to the local storage only for data tagged with ``FineLocation''. When application B stores the data in the local storage, this data will be tagged with label ``FineLocation''. If application A tries to access this data from the local storage then `PolicyA2` will not authorise this action since A can access only data with no labels.

An alternative to the policy `PolicyB2` would be a policy that allows application B to only access the local storage to read and write data with no labels as shown in Figure 6.

```
PolicyB2Alt: B can do access on LS
               handle ``NoLabels''
```

Fig. 6. A policy that allows application B to use the local storage for data with no labels.

It is worth to note here that the permission spreading attack is a more general case of the confused deputy attack described in QUIRE. In the case of the confused deputy described in QUIRE, application B would be a trusted application while application A is an evil application that misuses the API of application B for accessing the fine-grained location data. To defeat a confused deputy attack, QUIRE relies on application B to correctly forwarding the complete call chain to allow the location provider to check whether application A has the required permission. However, if application B misbehaves and does not propagate the call chain then application A can still get access to the location information. In our framework, we do not rely on any application to correctly propagate the data labels. Therefore, we can prevent both types of attacks. It should also be noted that the QUIRE approach cannot prevent application A from accessing the location information from the local storage even if application B would be trusted.

C. Policy Generation

We do not expect that the average Android user is able to create policies when installing applications. For this reason, we have extended the Android installer with the User Setting Interface (USI) component (see Figure 1). When a new application is installed, the installer presents to the user the set of permissions that the application requires. These permissions are extracted from the application manifest file. The USI intercepts the permissions requested by an application and generates policies and labels according to type of permissions that the application is requesting. For instance, when the OrgApp is installed it requires access to the phone contacts (`READ_CONTACTS`) and internet (`INTERNET`). The first requirement will trigger the USI to generate a basic policy for OrgApp to access the contacts and the internet as in Figure 7. The USI checks in the Labelling Store for possible extra labels associated with the data type requested from the application. For contacts two extra labels are specified for public and private entries. The USI will prompt the user for allowing the OrgApp to access these two subtypes. The user selects only the "Pub" label to be associated with the OrgApp.

Then the USI will modify the **handle** clause adding the ``Pub'' label, and generate the **have to perform** clause for filtering out the contacts tagged with the "Pr" label. Moreover, the USI will inform the user that OrgApp could also send the contacts over the internet and whether the user is willing to grant this type of permission to the application. The user will agree to propagate the data that OrgApp has access to to the internet. The result of this is shown in Figure 8. Finally, the user decides that OrgApp has to access only one specific cloud service and selects it for the USI. This will allow the USI to

```
OrgAppP1: OrgApp can do getContacts
     on Contacts
          handle ``Contact''

OrgAppP2: OrgApp can do send on
     Internet
```

Fig. 7. Basic generation of policies

```
OrgAppP1: OrgApp can do getContacts
     on Contacts
          handle ``Pub'' and ``
               Contact''

OrgAppP2: OrgApp can do send on
     Internet
          handle ``Pub'' and ``
               Contact''
```

Fig. 8. Extending the policies

```
OrgAppP1: OrgApp can do getContacts
     on Contacts
          have to perform
               filterOut(``Pr'',
               returnData);
          handle ``Pub'' and ``
               Contact''

OrgAppP2: OrgApp can do send on
     Internet
          have to perform
               sendOnlyTo(aCloudUrl
               );
          handle ``Pub'' and ``
               Contact''
```

Fig. 9. Final policies.

generate the **have to perform** clause in the OrgAppP2 policy as shown in Figure 9.

Similarly to the case of the READ_CONTACTS permission, the USI checks whether the user associates data with extra labels for READ_CALENDAR, READ_LOGS, READ_SMS, and GET_ACCOUNTS permission requests. For permissions that enable applications to generate data traffic, such as INTERNET, SEND_SMS, CALL_PHONE and BLUETOOTH_ADMIN, the USI enables the user to set specific destinations or to black list addresses and phone numbers. Moreover, when these permissions are combined with permission for access data then the USI always prompts the user for explicit consent to allow the application to combine the permissions.

The user can change at runtime the permissions granted to an application by disabling the respective policy. YAASE enforcement mechanism overwrites the check permission of the standard Android. YAASE enforces a negative-by-default policy meaning that if there is no policy associated with an application request then the request is not granted.

VI. Performance Evaluation

In this section, we performed a preliminary evaluation of our solution. Since the time overhead is a fundamental issue to have a usable policy enforcing system, we aim here at evaluating the time overhead of YAASE as compared to standard Android. All simulations were run on the Android Emulator with the portable VM (written in C). To measure the time overhead, we hooked a call to System.nanoTime() before and after the event to be measured, to compute the time (in nanoseconds) required by the single operations.

TABLE I
Performance of reading operations for contacts with stock Android and with YAASE filtering policies. Time in seconds.

Configuration	Read 100 Contacts
Stock Android	0.048
YASSE filtering 0	1.9
YASSE filtering 20	2.0
YASSE filtering 40	2.1
YASSE filtering 60	2.1
YASSE filtering 80	2.4

We evaluated the effects of YAASE on the throughput performance of the Android platform. We performed read operations on 100 contacts with the stock Android and with YAASE where filtering policies where activated for removing private data. For the case when YAASE is activated, we have changed the number of private contacts from 0 to 80. The results shown in Table I are averaged over 50 executions. The second row represents the execution time for the stock Android for performing a read of 100 contacts. The third row reports the time values for performing the same operations but this time with YAASE framework active. We have inserted a policy that filters out private data. However, since all contacts are set as public (0 private contacts) then no filtering is perform although the **have to perform** clause is executed. From the fourth row, we have increased the number of private contacts from 20 up to 80 (last row). We can see that the overhead introduced by YAASE is quite high. However, these experiments have been executed on non-optimised data structures.

In the second set of experiments, we have evaluated the enforcement of filtering policies over data sent on the internet. In particular, we have used location data that was sent over the internet using the sentStream method to write data over a socket. We have changed the size of the packets sent over the stream using 8, 16, and 32 bytes. For each packet size, we have measured the sending time in stock Android. Then, we activated YAASE and used two different policies. One policy was dropping the packet when data tagged with the location label was in it (YAASE dropping). The second policy instead was filtering the location data by replacing the bytes. Although the overhead introduced by YAASE is still 10 folds than standard Android, the sending operations are still performed within 100 milliseconds. Moreover, it stands to reason that on an actual device running the ASM interpreter, these values will only decrease. Supported by these results, we can conclude that YAASE's overhead would not be noticed by the user.

VII. Conclusion and Future Work

In this paper, we have presented YAASE an Android security extension that supports fine-grained access control policies. YAASE makes use of the TaintDroid taint mechanism for tagging data and enforcing security decisions on how data has to be disseminated within the device (application

TABLE II
PERFORMANCE OF SENDING LOCATION DATA OVER THE INTERNET WITH STOCK ANDROID AND WITH YAASE FILTERING POLICIES. TIME IN MILLISECONDS.

	Packet Size		
Configuration	8 bytes	16 bytes	32 bytes
Stock Android	8.7	8.6	7.9
YAASE Dropping	101	93	91
YAASE Filtering	80	93	82

to application) or the outside world (through internet connections). Compared to other approaches, YAASE is able to filter out data tagged with user-defined labels (such as public, private, confidential, etc.). In this way, applications can still access the data without reaching for user's sensitive information. Moreover, YAASE enables the user to control internet connection only allowing applications to communicate with specific URLs. YAASE policy enforcing mechanism is able to contrast privilege spreading attacks by controlling which labels can be handled by each application. Such a mechanism is also very effective against confused deputy attacks (which are more specific cases of privilege spreading attacks). We have implemented YAASE and analysed its performance in comparison with stock Android. Our results show that while the overhead for accessing un-optimised data structure is quite noticable, in the case of filtering data sent over the internet the overhead is quite acceptable.

As a future work direction, we are planning to extend the capabilities for the USI component to make simpler for the average user to express security requirements. One way of doing this would be to let the YAASE system learn about the user decisions in order to automatically define security policies.

As a final note, we observe that while patching Android security by extensions can be a useful paradigm for researchers, it may not be the best way for consumers. Thus, our hope is not to come in future with a *yet* yet another security android extension but rather to see some of the ideas proposed by security researchers working in this area, implemented in the next official release.

ACKNOWLEDGEMENT

The work of this paper is partly funded by the EU project NESSOS contract no. FP7-256980. The work of the third author is supported by the project S-MOBILE, contract VIT.7627 funded by STW - Sentinels, The Netherlands.

REFERENCES

[1] Android malware steals info from one million phone owners. http://nakedsecurity.sophos.com/2010/07/29/android-malware-steals-info-million-phone-owners/.

[2] Android Project. http://www.android.com.

[3] ARM Trustzone Technology. http://www.arm.com/products/processors/technologies/trustzone.php.

[4] Gartner says android to command nearly half of worldwide smartphone operating system market by year-end 2012. http://www.gartner.com/it/page.jsp?id=1622614.

[5] These 26 Android Apps Will Steal Your Phone's Information. http://www.businessinsider.com/up-to-120000-android-phones-have-been-infected-with-malware-2011-5.

[6] Guangdong Bai, Liang Gu, Tao Feng, Yao Guo, and Xiangqun Chen. Context-aware usage control for android. In *Proc. SecureComm 2010*, pages 326–343, 2010.

[7] Alastair R Beresford, Andrew Rice, and Nicholas Skehin. MockDroid: trading privacy for application functionality on smartphones. In *Proc. HotMobile '11*, 2011. to be published.

[8] 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, D-64293 Darmstadt, Germany, June 2011. Available at: http://www.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_TRUST/PubsPDF/xmandroid.pdf.

[9] M. Conti, V.T.N. Nguyen, and B. Crispo. Crepe: context-related policy enforcement for android. In *Proc. ISC '10*, pages 331–345, 2010.

[10] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. Crepe: context-related policy enforcement for android. In *Proceedings of the 13th international conference on Information security*, ISC'10, pages 331–345, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th international conference on Information security*, ISC'10, pages 346–360, Berlin, Heidelberg, 2011. Springer-Verlag.

[12] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *20th USENIX Security Symposium*, 2011.

[13] 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 *Proceedings of OSDI 2010*, October 2010.

[14] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proc. CCS '09*, pages 235–245, 2009.

[15] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding android security. *IEEE Security and Privacy*, 7(1):50–57, 2009.

[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. Technical report, April 2011. Available at: http://appfence.org/appfence.pdf.

[17] Anthony Lineberry, David Luke Richardson, and Tim Wyatt. These aren't the permissions you're looking, 2010. Available at: http://dtors.files.wordpress.com/2010/08/blackhat-2010-slides.pdf.

[18] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proc. ASIACCS '10*, pages 328–332, 2010.

[19] Machigar Ongtang, Stephen McLaughlin, William Enck, , and Patrick McDaniel. Semantically rich application-centric security in android. In *Proc. ACSAC '09*, pages 73–82, 2009.

[20] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid android: Zero-day protection for smartphones using the cloud. Technical report, 2010. Available at: http://www.cs.vu.nl/~herbertb/papers/trpa10.pdf.

[21] Roman Schlegel, Kehuan Zhang, Xiaoyong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *Proceedings of the 18th Annual Network & Distributed System Security Symposium*, NDSS '11, pages 17–33, 2011.

[22] Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, Shlomi Dolev, and Chanan Glezer. Google android: A comprehensive security assessment. *IEEE Security and Privacy*, 8:35–44, 2010.

[23] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and V.W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proc. TRUST 2011*, 2011. to be published.