COMPUTERS & SECURITY 62 (2016) 257–277

Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

A privacy enforcing framework for Android applications

Ricardo Neisse [a], Gary Steri [a,*], Dimitris Geneiatakis [b], Igor Nai Fovino [a]

[a] European Commission, Joint Research Centre (JRC), Via E. Fermi, 2749 Ispra, Italy

[b] Electrical and Computer Engineering Department, Aristotle University of Thessaloniki, GR541 24 Thessaloniki, Greece

ARTICLE INFO

Article history:
Received 15 February 2016
Received in revised form 15 June 2016
Accepted 18 July 2016
Available online 25 July 2016

Keywords:
Android
App instrumentation
Permission control
Policy enforcement
Privacy

ABSTRACT

The widespread adoption of the Android operating system in a variety type of devices ranging from smart phones to smart TVs, makes it an interesting target for developers of malicious applications. One of the main flaws exploited by these developers is the permissions granting mechanism, which does not allow users to easily understand the privacy implications of the granted permissions. In this paper, we propose an approach to enforce fine-grained usage control privacy policies that enable users to control the access of applications to sensitive resources through application instrumentation. The purpose of this work is to enhance user control on privacy, confidentiality and security of their mobile devices, with regards to application intrusive behaviours. Our approach relies on instrumentation techniques and includes a refinement step where high-level resource-centric abstract policies defined by users are automatically refined to enforceable concrete policies. The abstract policies consider the resources being used and not the specific multiple concrete API methods that may allow an app to access the specific sensitive resources. For example, access to the user location may be done using multiple API methods that should be instrumented and controlled according to the user selected privacy policies. We show how our approach can be applied in Android applications and discuss performance implications under different scenarios.

1. Introduction

Android is the dominant operating system for mobile devices; it currently has the largest installed base (IDC, 2013) mainly because (a) it supports a huge variety of different devices such as watches, tablets, TV sets, *etc.*, and (b) it provides end-users with a large variety of applications (a.k.a. apps) for accomplishing their daily needs through its official market. Due to its large adoption and every day use to perform online tasks, malicious developers/hackers have increasingly targeted this operating system. Even if the Google Bouncer (Google, 2012) security service scrutinises apps before allowing them to be published in Google Play, there is evidence (Miners, 2014) showing that malicious software (malware) can be found among legitimate apps as well. In most of the cases, the main goal of these malware apps is to access sensitive phone resources, *e.g.*, personal data, the phone billing system, geo-location information, home banking info, *etc.* Even though in this work we focus on Android, it is worth to note that similar flaws have

* *Corresponding author.*
E-mail addresses: gary.steri@jrc.ec.europa.eu (G. Steri), ricardo.neisse@jrc.ec.europa.eu (R. Neisse), dgeneiat@ece.auth.gr (D. Geneiatakis), igor.nai-fovino@jrc.ec.europa.eu (I. Nai Fovino).
http://dx.doi.org/10.1016/j.cose.2016.07.005

been reported also for other well known mobile platforms (e.g., iOS) in the literature (Damopoulos et al., 2013; Egele et al., 2011).

Android builds a part of its security on a *permission restricted access model* to provide access to sensitive resources (*e.g.* sd card, contacts). This means that to gain access to these resources, apps should declare in the manifest the required permissions, which users may grant or not. However, applications might abuse this model in order to gain access to private information. Typical examples are applications that request more permissions than what they actually need, named as *overprivileged* (Felt et al., 2011). These applications can be silently transformed into malware whenever an operating system or an app update occurs, using an attack that is called privilege escalation through updating (pileup).

The growing number of permissions[1] from the first version of Android (78) to the version 6.0 (148) does not help to solve the security issues as it represents and increase in the Android attack surface. Furthermore, the majority of users are still using previous versions that, differently from Android 6.0, does not foresee the possibility of selectively deciding which subset of the requested permissions should be granted to an app. This issue, combined with the fact that it is not very easy to understand the meaning of each requested permission, since they are too many and not clearly documented, makes the situation very dangerous for users from a security and privacy point of view.

Considering these drawbacks, in this paper we propose an expressive and fine-grain policy enforcement approach for Android that is able to selectively prevent privacy invasive app behaviour. The approach we present builds upon the Model-based Security Toolkit (SecKit) (Neisse et al., 2015), leveraging on the policy language and Policy Decision Point (PDP) component, and shows how policy refinement and policy enforcement can be achieved in the context of the Android mobile operating system. Existing approaches for enforcement of Android security policies are either hard-coded interfaces with a limited set of enforcement options (Beresford et al., 2011; Zhou et al., 2011), or flexible and fine-grain approaches using a security policy specification language focusing on low level actions (e.g. API invocations or system calls) (Rasthofer et al., 2014). The first type of approach lacks in flexibility since the set of enforcement options is limited, while the second one is too low level in order to be understandable and usable considering the complexity of policies by users. Most policy-based approaches, with the exception of AppGuard (Backes et al., 2013), do not implement modification of information, since low-level activities can only be allowed or denied. Furthermore, the supported conditions is of limited expressiveness, since they do not include context-based policy specification using event-based context situations, nor trust-based policies, but only simple context information attributes are supported (Conti et al., 2011). For example, a policy cannot be specified under specific event conditions, i.e., persons arriving or leaving their home but only considering the state when they are at home or when they are not at home.

In contrast to existing approaches, our framework includes a policy refinement step that maps abstract user-centric policies to a set of low-level enforceable policies. These abstract policies are specified in a more concise way that is more meaningful in contrast to the policies used in other approaches (e.g. Rasthofer et al., 2014), and are easier to understand without the need to consider many low-level technical details of the Android system. We are able to automatically refine policies because our security policy language is fully integrated with a reference model of the target system where the policies will be deployed. The reference model of the system and refinement relations includes the structure, behaviour, data, and identity models represented in a systematic and extensible toolkit that enables modular specification and re-use of security policy rule templates.

After the refinement, the low-level policies are enforced using a code injection mechanism that does not require the app source code to be available. All apps installed in a mobile phone must be instrumented in order to include a small code footprint that acts as a Policy Enforcement Point (PEP) and control the execution of the app. The injected PEP contacts a Policy Decision Point (PDP) component that evaluates the low-level policies and informs the PEP about the authorisation decision to allow, deny, modify, or delay the execution of specific sensitive API invocations. Our contribution in this paper is the design and implementation of this approach for specification, refinement, and enforcement of expressive security policies in Android. Our solution is suitable to all Android versions, including the latest available version 6.0, and do not pose additional constrains to the already defined minimal Software Development Kit (SDK) compatibility in the app.

From a technical perspective, all different policy enforcement solutions could be used to mitigate malware apps as well apps that have a privacy invasive behaviour. For example, considering a malware app that exploits a vulnerability in the Inter Process Communication (IPC) mechanism, a policy could be specified to deny access to the IPC mechanism preventing the vulnerability from being exploited. Although we acknowledge this possibility, our focus in this paper is on protecting the user from privacy invasive behaviour and all our example policies are only of this nature. Finally, we do not consider explicitly in this paper apps that share resources and permissions that may seem harmless in isolation but that in combination may pose a privacy risk. Our focus is simply in the specification and enforcement of policies to constrain the behaviour of a single app.

The remaining of this paper is organised as follows. Section 2 briefly overviews the Android security model. Section 3 presents a threat analysis and examples of privacy invasive behaviours. Section 4 describes in details our approach for specification, refinement, and enforcement of security policy rules. Section 5 shows the implementation details in a case study with injection of code, while Section 6 analyses our performance evaluation results including a discussion on assumptions and limitations of the proposed scheme. Section 7 discusses the related work and highlights the main differences in contrast to our approach. Section 8 concludes this work and gives pointers for future work.

2. Android security model

The security of the Android Operating System (OS) is mainly achieved by its subdivision into layers, which provides plat-

[1] According to http://developer.android.com/reference/android/Manifest.permission.html.

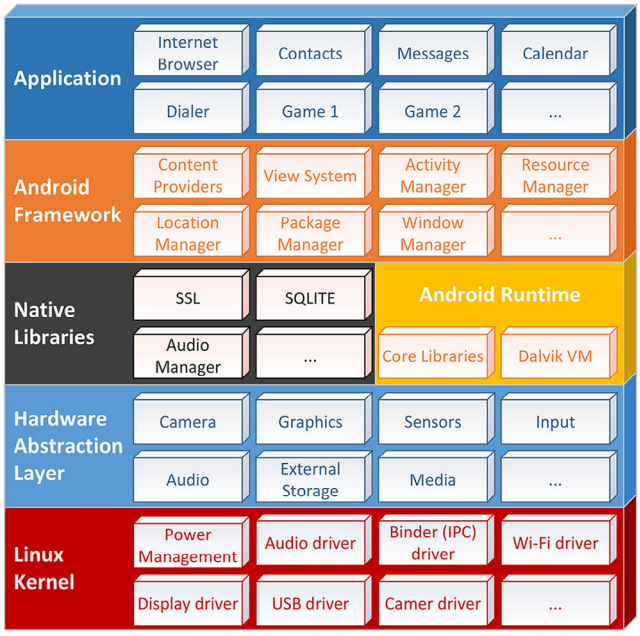

Fig. 1 – Android software stack.

form flexibility and separation of resources at the same time. This separation is reflected in the whole software implementation, shown in Fig. 1. Each level of the stack assumes that the level below is secured. In this paper we focus on security of apps, which run in the Dalvik Virtual Machine (DVM), and have their own *security environment* and dedicated filesystem.

The security mechanism for app isolation, which is also in place for native code invoked by the apps[2], is called the *Android Application Sandbox*. This sandbox is set up in the kernel, thus propagating the isolation on all the layers above and on all kinds of application. All apps running in the Android OS are assigned a low-privilege user ID, are only allowed to access their own files, cannot directly interact with each other, and have a limited access to the OS resources. The isolation is a protection against "inter-process" security flaws, meaning that a security problem in a given app will not interfere with the resources of other apps.

In the Android Software Development Kit (SDK), the functionalities an app can use are categorised and grouped in Application Programming Interfaces (APIs) that give access to resources normally accessible only by the OS. For example, among the protected APIs there are functions for SMS and MMS management, access to location information, camera control, network access, *etc*. The access to the protected APIs is regulated by a *permission mechanism*, in which a specific permission should be granted to an app in order to allow access to a particular API. Unprotected APIs do not require any special permission to be executed by the app.

More specifically, permissions in the Android OS are grouped in four different levels considering the risk level introduced to the user: *normal*, *dangerous*, *signature*, and *signature-or-system*. Normal permissions are considered of low risk to other apps, the system, or the user[3]. Dangerous permissions have a high risk of negative consequences for the users' personal data and experience. Signature permissions are used to protect exported interfaces accessible only by apps signed with the same developer key. Signature-or-system permissions are used to protect core resources available only to trusted system apps signed with the firmware key.

All permissions required by an app are declared in the *Manifest file*. Until the version 5.0 of Android, when installing an app users are notified only about the sensitive permissions required by it, and they are not given any choice on which permissions to grant: they have to accept all of them or abort the installation. However, the new recent Android-M (version 6.0) release provides *runtime* or *time-of-use* permissions as well[4] in addition to install-time permissions. Time-of-use permissions give users the possibility of denying a permission request at runtime and permanently revoking a requested permission. This new privacy feature shows that the Android community recognises the need for more advanced privacy and anonymity control for users.

Even though time-of-use permissions allow users to gain control over the restricted resources, there is a need for backward compatibility to enforce privacy control on million of devices using previous OS versions. It is worth to note that as of August 2016 less than 15% of available devices support Android's latest version features, according to the Android Dashboard[5]. In addition, the mapping of permissions to methods in the Android APIs is one to many, a characteristic that contributes to make less clear/deterministic which kind of and the actual functionalities an app really uses. Besides, the lack of protection in many sensitive APIs provides the possibility of manipulation of apps' features and services, as well as the lack of any restrictive policy-based approach to empower users to automate decisions with respect to the protection of their data, privacy, and anonymity indicates that complementary research work is needed in the Android platform.

3. Threat analysis

The evolving state of modern mobile operating systems and the proliferation of mobile services is more and more catching the attention of malicious users. Their main goal is to gain access to otherwise private information by exploiting vulnerabilities both at application and operating system level. This is the case not only of malware, but also of legitimate apps that sometimes collect an excessive amount of personal information. Many papers in the literature (Enck et al., 2010; Gibler et al., 2012; Stirparo and Kounelis, 2012; Zhou and Jiang, 2013) have shown apps with high invasion and manipulation on users' personal data. This exploitation is enabled by Android design vulnerabilities (Shabtai et al., 2010), and is triggered by the increasing value of users' personal information in digital businesses.

[2] Libraries and classes usually written in C/C++ and compiled for a specific hardware platform, which can be called by a Java application running in the Dalvik VM.

[3] http://developer.android.com/guide/topics/manifest/permission-element.html.

[4] http://developer.android.com/preview/features/runtime-permissions.html.

[5] https://developer.android.com/about/dashboards/index.html.

In this context, various solutions have been proposed to identify possible malicious apps and behaviours (Aafer et al., 2013; Arp et al., 2014; Google, 2012; Wu et al., 2012). However, most of these mechanisms are based on analysis of permissions granted to apps, *i.e.*, the set of API methods they are allowed to invoke. This type of analysis is not sufficient, because it can result in an over-approximation that rates legitimate apps as malicious (false positives) and it is not able to detect collusion attacks[6].

Furthermore, it is almost impossible to guarantee the fairness of any given app, as it has been showed that centralised security checks (*e.g.*, Google Bouncer, 2012) can be bypassed (Ducklin, 2012; Miller and Oberheide, 2012), while legitimate overprivileged apps (Geneiatakis et al., 2015) can be manipulated in order to provide access to personal data as shown in Xing et al. (2014). Therefore, even the presence of security analysis mechanisms in the Android app store does not guarantee users' privacy. Moreover, since apps consist of components namely activities, broadcast receivers, content providers, and services whose communication interfaces are clearly defined, other installed services and apps might manipulate also these interfaces to gain access to users' private information.

Consequently, the above mentioned facts show that users' personal data stored in mobile devices are at high risk. To summarise, we can classify the threats in mobile devices incorporating Android OS apps in three main categories: (1) the ones that derive from Android's architecture and mainly exploit the permissions mechanism, (2) the ones characterised by privacy invasion features that may be exploited by malware or even by legitimate apps and (3) those related to implementation vulnerabilities. In this paper we focus on the first two classes of threats. Although in this analysis we concentrate on Android OS, we believe that similar attacks could be employed for other mobile platforms like, for instance, iOS.

3.1. *Android's permission model threats*

The goal of Android's permissions model is to protect system resources from indiscriminate and unauthorised use by apps. However, this model has some inherent problems that might affect users privacy and anonymity as well. The following paragraphs describe the types of threats we have identified and that target this model, namely threats related to: pre-installed apps, permission management, permission granularity, permission notification, unused permissions, and lack of security.

First of all, pre-installed or OEM apps are automatically granted all permissions required and are considered trusted since they are part of the OS firmware. Therefore, users are not informed about the required permissions of these apps, since consent is normally granted by users during the installation process. This means that users do not have any indication on which resources are accessed by these apps and they are vulnerable to privacy invasive behaviours.

The second important point is the way permissions are managed and granted during the app life-cycle. As described in Section 2, in several cases, if a user wants to successfully install and use an app, he/she is obliged to grant all the requested permissions. Consequently, a common behaviour is just to accept all the permission requests in order to reach the end of the installation process. Besides, most of the users do not have knowledge about possible risks the requested permissions introduce toward their personal data, and the information prompted during the installation process are not really informative about the real functionalities the app is going to access and how often (e.g., regular fine grain location tracking). More knowledgeable users might try to evaluate the list of requested permissions, but even for experts it is unclear how permissions are used (Felt et al., 2012).

This is due to the fact that permissions are not a one-to-one mapping scheme with the corresponding API method calls that implement the actual functionalities. Indeed, their granularity is quite coarse, and, considering the 197 permissions of Android SDK version 17 associated to the 1259 methods with permission checks published in (Felt et al., 2011), on average a permission is associated to 7 API methods. For instance, a mobile app with CAMERA permission is allowed to take pictures or to capture videos using the `takePicture` and `MediaRecorder` methods respectively. This means that a user after granting this permission is not aware of the precise action performed by the app at any specific time since it can give access to a wider group of more or less sensitive functionalities.

As a consequence of the lack of information about permissions, users may lose track of the granted resources that might be accessed by the installed apps. The new *Android-M* version includes also *runtime* or *time-of-use* permissions,[7] giving users the possibility of denying a permission request at runtime or permanently revoking an install-time permission. However, only new models of smart-phones, high-end devices will receive this update, leaving all the rest of the installed system with the problem of dealing with the old permission model.

Another threat to users are the *normal* level permissions, which are considered of lower risk and are automatically granted to apps without asking users explicitly for consent. Even if users have the possibility to review this automatic granting, the a priori categorisation as low risk may not be perceived by all users in the same way. As a result, even though the permission granting mechanism is in place, from the user perspective this approach may be wrongly understood as if the apps are not accessing sensitive resources at all.

Some apps may also request permissions that are not used in the app implementation, and that are not actually needed for accomplishing their task (useless permissions). These apps are usually labelled as over-privileged (Geneiatakis et al., 2015), and could lead to privilege escalation problems in terms of sensitive resources they can access after an update (Xing et al., 2014). Privilege escalation may also lead to *confused deputy* attacks, when an app that has been granted a specific permission is exploited by other apps that do not have this permission in order to perform sensitive tasks (Felt et al., 2011). A classical example is an app that is allowed to send SMS messages and allows other apps to use its interfaces to send SMS messages as well. Finally, some methods in the Android API are still not protected by specific permissions and introduce a *lack*

[6] Attacks that allow apps to indirectly execute operations for which they do not have specific permission.

[7] http://developer.android.com/preview/features/runtime-permissions.html.

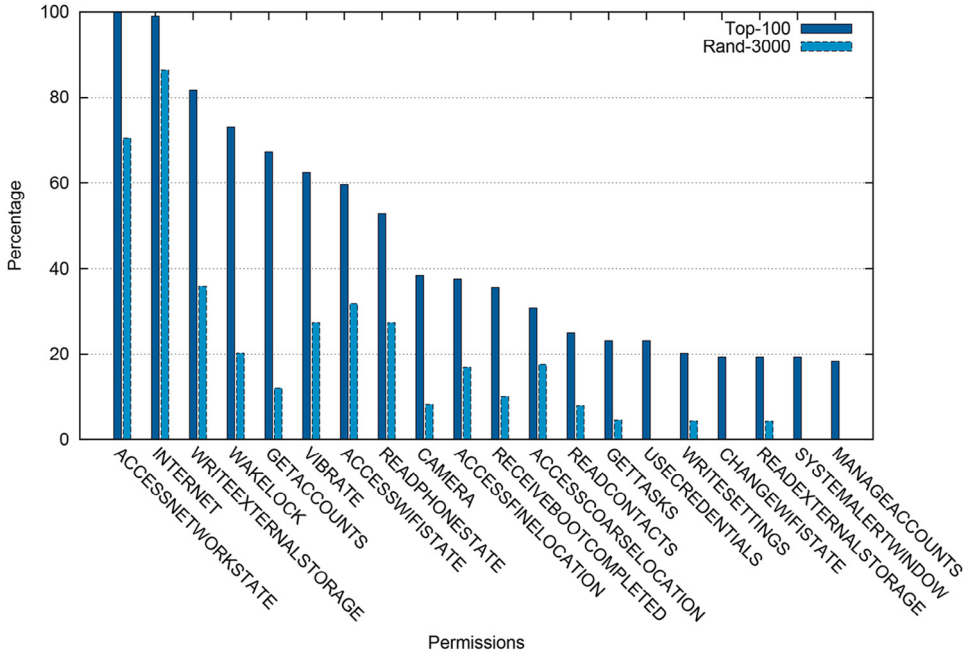

Fig. 2 – Permission distribution for 100-top versus 3000 random selected applications.

of security with respect to the sensitive resources they may allow access to. For instance, an app might use the `exec(String prog)` method to execute the process `prog` passed as parameter. This means any app could silently execute unprotected system commands in order to read system information from the `proc` filesystem, retrieve the list of installed and running apps, read the SD card contents, *etc.*

3.2. *Threats to users' privacy*

Threats to users' privacy may be posed not only by malware apps but also by legitimate apps. Many legitimate apps are characterised by a certain degree of privacy invasiveness, which is related to the permissions they request and to which use they make out of the protected methods. In this direction, TaintDroid (Enck et al., 2010) as well as other research works (Gibler et al., 2012; Stirparo and Kounelis, 2012; Zhou and Jiang, 2013) demonstrate the type of end-users' personal data manipulation performed by mobile apps. Examples of privacy invasive behaviour performed by apps are, for instance, games that request access to unique identifiers or user location that are not needed by the app to function. Ultimately, it is up to each mobile device user to judge if an app behaviour is privacy-invasive according to his/her personal perceptions.

In this direction, the Android OS provides security services to verify apps before installation, and to periodically scan the OS for harmful apps. Unfortunately, these services themselves are also potentially privacy-invasive because, according to the Android documentation, the device "may send information to Google identifying the app, including log information, URLs related to the app, device ID, your OS version, and IP address".[8] Therefore, the user-desired functionality is bound to a privacy-invasive behaviour, and users have no choice when using these services to control or restrict the personal data shared. Furthermore, the Android developers documentation[9] suggests as apps distribution options, alternative to the MarketPlace, e-mail and websites, thus exposing packages to the risk of malicious code injection. As a consequence, the existing features aimed at protecting end-users from privacy invasive applications are quite limited.

To show evidence of potential threats in terms of privacy for end-users relying their daily activities on mobile apps, we have analysed two different data sets related to Android apps. The first one is the top 100 downloaded apps in Google Play, while the second one consists of 3000 random apps from the same source. Our analysis consists in:

1. extracting static features (*i.e.*, permissions and respective invoked methods of the Android API) from apps using the Dexpler (Bartel et al., 2012) and Soot framework (Vallee-Rai et al., 1999);
2. identifying the sensitive method invocations incorporated in a given application using the permission map published in Felt et al. (2011).

Fig. 2 illustrates the first result of our analysis, showing the twenty most frequently requested permissions in the app datasets. Correspondingly, Fig. 3 depicts the twenty most frequently method invocations accessing sensitive resources incorporated in Android mobile apps. Note that, although the trend between the examined data sets are slightly different, we concentrate mainly on the features of the one hundred top apps, because users will most probably use some of them

[8] https://support.google.com/accounts/answer/2812853?hl=en.

[9] http://developer.android.com/distribute/tools/open-distribution.html.

instead of other unknown apps. However, for the sake of completeness we provide the corresponding statistics for the three thousand randomly selected apps as well.

More specifically, the results of our analysis show that almost all the top apps request network related permissions (100% for ACCESS_NETWORK_STATE and 98% for INTERNET) while 80% of the apps request permissions to write to external storage (WRITE_EXTERNAL_STORAGE), which usually is a Secure Digital (SD) card. These permissions grant an app the capability to identify the access networks of a user, and consequently infer his/her position, violating the user's privacy and anonymity. The user's position can be inferred using Internet services that identify the geolocation of an IP Address, therefore without requiring access to the GPS subsystem.

Still, considering the requested network related permissions, around 98% may invoke the openConnection method that allows connections to any arbitrary URL. This connection permission combined with the 70% of the apps that also request permissions to invoke the `getDeviceId` method show another way that can be used to violate users' anonymity. An app may use both permissions in order to monitor user online activities using a unique identifier (device ID).

Futhermore, almost 60 of the 100 top apps request access to the GET_ACCOUNTS permission. This permission allows an app to authenticate using the `getAuthToken` method without explicitly receiving consent for each specific authentication token. By calling this method an app is able to perform online activities on behalf of the end-users using any of their accounts available in their mobile device. A given application might use this permission also to access other unique identifiers (*e.g.*, e-mail) and monitor the activities of online end-users regardless of which device they are using. Therefore, without loss of generality, the impact on the different sides of end-users' privacy and anonymity depends on the type of permission an app is granted.

As additional concrete evidence of privacy invasive behaviour we show an extract of Jimple code retrieved using reverse engineering techniques from a real app that performs an invasive operation. This code is shown in Listing 1, and it accesses the device's unique identifier, i.e. the IMEI of the mobile device, through the `getDeviceId` method. Though this part of the code cannot be considered malicious as it is executed only when a fatal error occurs during the app execution, the examined app sends the device ID along with other related information to a remote server. It is questionable why an app would need to send the device ID or any other personal information when a crash occurs, and it is clear the leak of information that potentially allows user tracking.

Listing 1: An example of a Jimple code residing in a real Android application that accesses the device unique Id

Listing 1: An example of a Jimple code residing in a real Android application that accesses the device unique Id

```
$r4 = virtualinvoke $r8.<TelephonyManager: getDeviceId()>();
if $r4 == null goto label2;
$r2 = $r0.<class: java.util.Map mDeviceSpecificFields>;
$r3 = <class: class.ReportField DEVICE_ID>;
interfaceinvoke $r2.< Object put(Object,Object)>($r3, $r4);
```

Similarly, an app may access multimedia sensors (*e.g.*, microphone, camera, *etc.*) to record users' private conversations, images, or even to record real-time videos, thus violating their private sphere. More concretely, the permission RECORD_AUDIO allows the invocation of the `HandleStartListening()` method in the Android API that could be used to record the users' private conversations.

All examples presented above show that attackers can use built-in features of apps and their requested permissions to violate the end-users' anonymity and privacy. Other orthogonal approaches such as Nan et al. (2015) and Zhou et al. (2013) reveal that runtime information gathering could disclose users' different states (inside/outside of their house) and impose a real threat even for their safety.

4. Specification, refinement, and enforcement of security policies

In this section we present a flexible approach to inhibit the malicious behaviour of Android apps. The general idea is that of injecting a "control mechanism" in targeted apps, allowing the user to control or influence their behaviour. Our approach is policy-based, meaning that it uses a security policy rule language that provides flexibility by decoupling the specification of security constraints from the enforcement. In this way, users may modify their security constraints dynamically at runtime without the need to recompile or to reinstall the apps.

Fig. 4 presents a high level overview of the solution we propose. Our framework starts with the decompilation of the Android app using the ApkTool decompiler (Tumbleson and Winiewski, 2010) that reads the App .apk file (step 1) and produces the original bytecode (step 2). In parallel to the decompilation of the app our refinement engine transforms an abstract policy to a concrete policy by executing a set of refinement rules that consider the abstract and concrete models of the App behaviour (steps 4 and 5), and is deployed in the Policy Decision Point (PDP, step 6). The *Original bytecode* and the *Concrete Policy* are used as input by the *Instrumentation Engine*

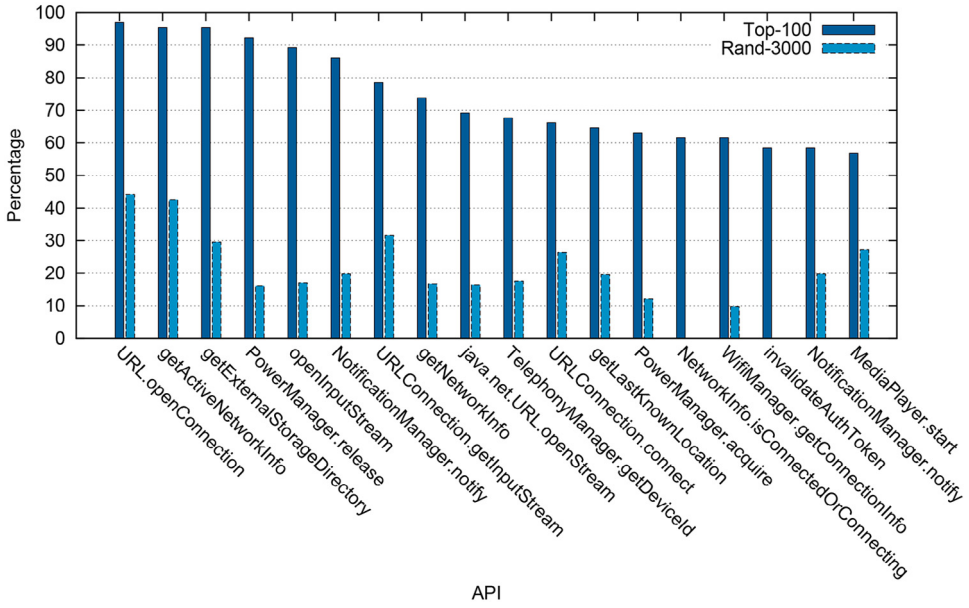

Fig. 3 – Invoked methods distribution for 100-top versus 3000 random selected apps.

that generates the *Instrumented Bytecode*, which includes additional policy enforcement bytecode (steps 3, 7, and 8). The *Instrumented Bytecode* is repackaged in a *PDP-enabled App* also using the ApkTool decompiler (steps 9 and 10). The Policy Enforcement Point (PEP) code included in the *PDP-enabled App* notifies the PDP with an event whenever an activity referenced in the concrete policy is executed (step 11), and synchronously receives an authorisation action to be enforced in response (step 12). The PDP service is installed in the phone and provides a central service interface for all instrumented apps when sensitive API invocations occur.

The following subsections report on the details of the policy specification language we adopt, the system modeling language and interaction refinement, the policy refinement rules, and the policy enforcement framework that uses instrumentation to enable policy enforcement.

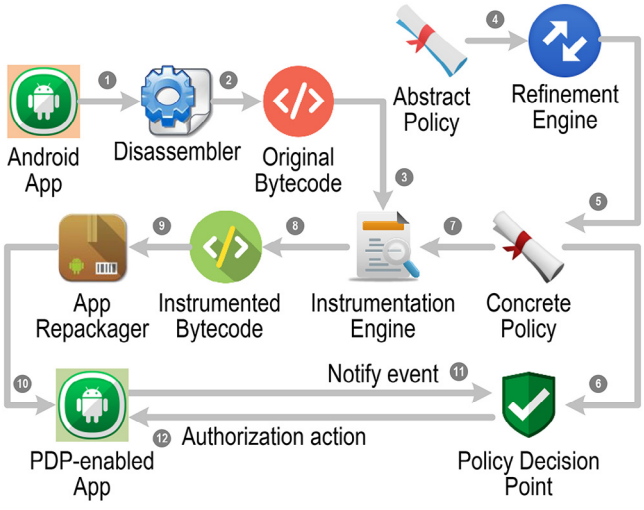

Fig. 4 – High level overview of our proposed solution.

4.1. Policy specification

Abstract and concrete policies are specified using an Event-Condition-Action (ECA) rule language. The following formalisation shows the abstract syntax of an ECA *Rule Template*. A rule engine monitors ECA rules instantiated from the templates with the following semantics: whenever the event (*EventPattern*) is observed and the condition (*Condition*) evaluates to true then the action (*Enforcement*, *TrustUpdate*, and/or *BehaviorInst*) is executed.

$$
\begin{aligned}
&[VariableName]\\
&AuthorizationResponse == Allow, Deny\\
&Modify == DataInst \rightarrow NewValue\\
&Delay == TimeAmount \times TimeUnit\\
&Enforcement == AuthorizationResponse \times \mathbb{P}\, Modify \times Delay\\
&RuleTemplate ::=\\
&\quad \mathbb{P}\, VariableName \times EventPattern \times Condition \times\\
&\quad Enforcement \times \mathbb{P}\, TrustUpdate \times \mathbb{P}\, BehaviorInst
\end{aligned}
$$

The *EventPattern* is a pattern matching an actual or tentative activity (action or interaction) in order to support respectively detective and preventive enforcement policy rules. The *Condition* corresponds to a complex conditions including event patterns, external actions, temporal, cardinality, context, role, trust, identity, and data operators. The *Enforcement* can be an authorisation response to allow or deny the execution of a tentative activity, while it may also optionally modify the activity attributes or delay the activity execution depending on the security requirements. A rule template may also instantiate a behaviour (*BehaviorInst*) in order to execute additional

activities or update (e.g, increase or decrease the degree) a trust relationship (*TrustUpdate*). A behaviour instantiation may, for example, notify the mobile phone user in case an app performs a privacy sensitive operation.

Policies alone are not sufficient, a mechanism of information flow tracking inside an app and in between apps is also necessary. Our policy language addresses information flow by assuming that an information flow tracking system is in place and by considering quantitative data identifiers (IDs) in data pattern operators. We assume an information flow tracking system is in place because it is an orthogonal problem in relation to specification and enforcement of security policies. With data pattern operators using quantitative data identifiers we can specify policies that prevent a specific data item from being sent to a remote server or even to prevent partial disclosure, for example, if an implicit information flow tracking algorithm has determined that a certain number of bits of a sensitive data may be disclosed in an interaction. The integration of information flow tracking systems and the use of data-centric operators are out of the scope of this paper.

Considering that our security policy rule conditions include cardinality and temporal operators, a policy can be specified to limit the frequency an app access the user location even when the location is obtained through a direct call to *getLastKnownLocation*. The following lists summarises additional examples of enforcement scenarios that can be implemented using our approach:

- App-centric pseudonyms: users may choose to allow apps to a pseudo device ID, which is required by some apps for purposes not always clear. However, to protect the users' privacy and prevent apps from collectively monitoring the user behaviour the device ID should be different for each app used;
- Access to partial information: user may allow apps to access their SMS messages but only see the message content, without seeing the sender. Furthermore, users may allow access to their contacts but not to the full name, only to a hash of the names and phone numbers;
- Regular access: users may allow access to their sensors (e.g., GPS) with time and cardinality constraints, for example, once a day;
- Usage statistics: our instrumentation framework in combination with our policy rule language can also be used to show users aggregated statistics about the app behaviour including temporal and cardinality constraints. For example, how often the app opens network connections and to which servers, how much data is sent, how many files the app keeps open per session, which app activities are the most used, *etc.*

More details about our policy language are presented in our case study in Section 5 with a running example. We refer the reader to Neisse et al. (2015) for a complete description of all operators and semantics of the language.

4.2. *Interaction refinement*

One of the big limitations of the use of policies to control low level behaviours is that, in general, they have to be extremely precise, requiring a non-trivial knowledge of the system, hence de-facto making impossible to a typical end-user to express by himself what an app is authorised or not to do. On the top of this, in several cases, a certain operation could be performed in different ways, by invoking different methods, requiring then to specify a huge number of different policies for the same behaviour. To solve these limitations, in our approach we provide an extensible model of high-level abstract source and sink *interactions* between the app and other system components that exchange privacy-sensitive user information. Each of these abstract interactions is refined considering the implementation options available in Android. To illustrate the refinement of sources in the following we consider a running example where a security policy rule regulates the access to the user location by an app.

Concretely, using the Android API, access to the location can be directly achieved by a call or interaction with a location manager component using the method *getLastKnownLocation*, or using a publish/subscribe mechanism where an app subscribes to a location manager as a listener with parameters, indicating the frequency of updates, and implements a callback method *onLocationChanged*. From user's centric policy specification perspective, when protecting the user location, the specific technical details are irrelevant, meaning that a user wants to have his/her location protected no matter how the app manages to obtain it.

Fig. 5 illustrates an abstract view on the *Access Location* interaction and two possible refinement alternatives for it. From an abstract point of view, *Access Location* can be defined as an interaction between an *App* and a *Location Manager* behaviour that exchange data *t* of type *Location*. We adopt a design language where behaviours are represented by rounded rectangles and the contribution or participation of each behaviour in an interaction by half ellipses, details about this language are described in Neisse et al. (2015).

In our model, an abstract interaction *AI* establishes a set of data instantiations DI_{AI} and is the outcome of data exchange between a set of roles R_{AI} representing the contribution of behaviour instantiations to the respective interaction. The

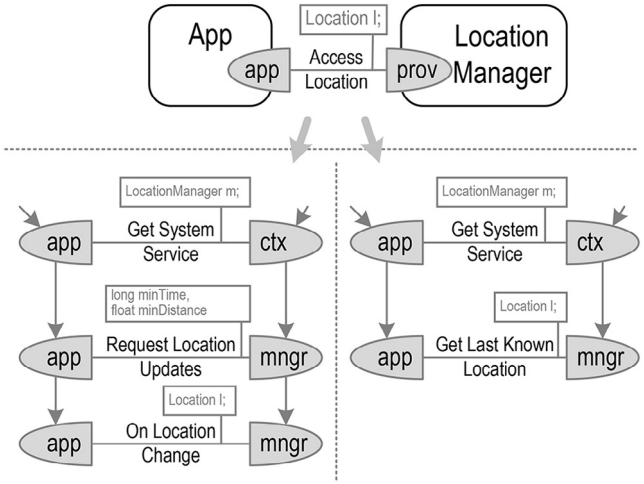

Fig. 5 – Two alternative refinements of *Access location* interaction.

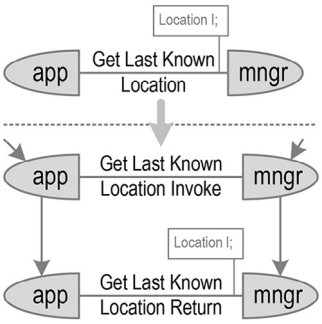

Fig. 6 – Interaction implementation pattern refinement of *Get Last Known Location* in method invocation and response.

end result of an interaction makes the set of declared data instantiation available to all behaviour roles participating in the interaction. For example, the *Access Location* interaction in Fig. 5 defines the contributions *app* and *prov* and the data instantiation *l* of type *Location*.

We consider two types of events in our policy rule language, *tentative* events triggered before an interaction takes place and *actual* events capturing the successful completion of an interaction. When a tentative event is signaled we assume the data instances are all available but have not been exchanged between the interaction participants, given the opportunity for preventive authorisation actions (e.g., allow, deny, modify, or delay) to be enforced.

An interaction *AI* may be refined into list of *k* concrete interactions CI_k, where the union of the set of established data instantiations for all concrete interactions DIC_k is a superset of the original data instantiations established by the concrete interaction $DI_{AI} \in \bigcup_{n=1}^{k} DIC_n$. Therefore, the set of concrete interactions must establish at least the same set of data instantiations of the abstract interactions, while it may include additional data instantiations if required by the concrete refinement choice. When refining an interaction a set of possible initial $CI_{start} \in CI_k$ and final $CI_{end} \in CI_k$ interactions must be also specified, representing possible choices to start and finish a concrete refined workflow of interactions. For example, the abstract *Access Location* interaction in Fig. 5 has two concrete refinement choices (bottom left and right side), where for both choices when the end interaction is completed the *l* data instantiation defined for the abstract interaction is always available.

A final refinement step is performed with respect to the interaction pattern used in the implementation. In Android interactions may map to a subscribe–notify mechanism, an invoke–return method call, and so on. For method invocations we refine further each interaction in two interactions, for example, the *Get Last Known Location* interaction is refined in *Get Last Known Location Invoke* and *Get Last Known Location Return* interaction instantiations (see Fig. 6).

Interactions are referenced in our security policy language by means of event patterns in the ECA rules. The event part (E) contains exactly one interaction pattern and in the condition part (C) zero or more interaction patterns may be combined using the policy language operators. In case the action part (A) includes authorisation actions (e.g., allow or deny) the event pattern defined in (E) must necessarily be a tentative event, since actual events represent activities that already took place and cannot be controlled anymore.

From a policy enforcement perspective an event may reference the interaction itself considering the container behaviour is enforcing the policies or it may have a focus in one of the participants. The focus in the interaction participant may be necessary when the enforcement is not possible, for example, the container behaviour represents an abstract entity outside the control. In our particular approach we do not modify the operating system, therefore the focus of enforcement is always in the app interacting with the android framework. This particular focus allows us to inject the enforcement in the app without the need to change the Dalvik VM execution.

4.3. *Policy refinement*

In order to map high-level policies to their low-level enforceable counterparts we build on the policy refinement approach introduced by Neisse and Doerr (2013). In their approach, the specification and evaluation of policy refinement rules uses as input a system model that explicitly includes abstract activities and their respective concrete refinements.

In Android, abstract activities are mapped to data sources and data sinks. For example, an app that accesses the user location (source) and further redistributes it (sink) by sending it over the network, writing it to a file, encoding it in an intent to other Android component, and so on. For instance, in case access to the user location is allowed to an app without anonymisation, a policy could also be defined to limit the redistribution of the location to other apps, using *sink* interactions. It is important in this case to define all the possible concrete redistribution interactions in Android. Policies can be defined to limit all redistribution, meaning all possible refinements, or to limit a specific more concrete redistribution such as writing to an SD card file. In this way, we can eliminate the consequences of a security flaw when an adversary exploits it.

In contrast to the refinement rules proposed in Neisse and Doerr (2013), our extension considers also the modification of events in addition to only allowing or denying the execution of activities. We use the mechanism of policy nesting for refinement in order to preserve the semantics of the original policy condition. We do not consider in our rules refinement of actions since we consider all relevant activities to be interactions (method invocations) in Android. Furthermore, in Android we do not consider refinement of behaviour types and executions, where an event pattern references an abstract interaction in an abstract behaviour or a policy rules triggers the execution of abstract activities that should be translated to concrete activities (e.g., abstract user notification translated to sending an e-mail or showing a toast notification). Refinement of behaviour types and executions are part of our future work plans. Finally, we take a more precise approach for policies modifying data by refining nested policy rules for each relevant modification in the set of concrete activities.

The following list describes the refinement rules adopted by us:

- Policies defining tentative events patterns with a deny authorisation action are refined to a set of nested policies with a *true* referencing event patterns (E) for all the possible initial concrete interactions in the set CI_{start}. With this rule we guarantee that all possible initial activities will be controlled;
- Policies defining tentative events with an allow and modify authorisation action are mapped to nested policies with a *true* condition for each concrete interaction in the set CI_k where the data instantiation to be modified is referenced in the authorisation action $DIC_k \in DI_{modify}$. The parent policy event is empty and only preserves the original policy condition. With this refinement rule we guarantee that all modifications will be preserved in the concrete activities;
- Actual event patterns in (E) or (C) part are mapped to a disjunction of the final concrete interactions CI_{end}. With this rule we guarantee that any possible completion of the abstract interaction will be captured by our policy rule.

Our refinement rules consider that an interaction enables the exchange of the specified data instantiations. Security policy rules that consider the concrete system model must be defined at the concrete level. For example, it is not possible to specify at the *Access Location* interaction level a security policy limiting the time amount in between location update subscriptions. The reason is that the abstract interaction is unaware of the concrete interaction pattern (publish–subscribe) adopted concretely.

4.4. Policy enforcement using application instrumentation

A key point in our approach to enable the enforcement of security policies is that of being able to modify any given app according to user security requirements ensuring the installation of appropriate authorisation hooks.[10]

Android apps are written in the Java language, which is compiled to a proprietary register-based bytecode format specifically designed for mobile devices having in mind their limited resources. This bytecode, called *dalvik bytecode*, can be executed in a DVM or in the Android Runtime (ART). Consequently, similar to pure Java applications, Android apps can be reverse-engineered using the appropriate tools, *i.e.*, smali/baksmali (Gruver, 2009), ApkTool (Tumbleson and Winiewski, 2010), Androguard (Desnos, 2012), Dexpler (Bartel et al., 2012) and Dex2Jar (Pan, 2012).

Using this reverse engineering tools it is possible to access an intermediate (human) readable representation of an app compiled bytecode, without having access to the original source code itself. In this way, we are able to modify the app's bytecode, without the need to recompile the original source code, and inject specific code able to enforce a given security policy. As an example, Listing 2 illustrates a part of a reverse engineered app in the *smali* format used to invoke the `getDeviceId()` API method in the `TelephonyManager` class, which can be used as an unique identifier of the mobile phone to track users.

Listing 2: A smali code example of Android mobile application invoking the getDeviceId method.

```
invoke-virtual {v5}, TelephonyManager;->getDeviceId()Ljava/
    ↪ lang/String;
move-result-object v9
iput-object v9, v5, MainActivity;->imei:Ljava/lang/String;
```

For each abstract policy referencing abstract events we provide a mapping to concrete events considering the implementation choice in Android for the abstract semantics and bytecode analysis technique that identifies the concrete implementation used in the specific Android App. Policies are enforced using instrumentation techniques by intercepting the concrete events.

The concrete policies generated using the refinement rules are deployed in a Policy Decision Point (PDP) component, which is a rule engine that evaluates the policies at runtime. The PDP component subscribes to events with the Policy Enforcement Point (PEP), which in our solution is injected in any Android application using instrumentation techniques. The following subsection discusses possible instrumentation techniques and presents the instrumentation technique adopted by us in this paper.

In our proposed approach, in order to inject the PEP code to enforce security policies in any given app we first decompile the app and perform static analysis to locate all the API method invocations that should be controlled as they handle user sensitive information. The decompilation is performed using the ApkTool (Tumbleson and Winiewski, 2010), which generates as output the smali code of all classes which are part of the app. The app's decompiled code is analyzed through our own tailored static analysis tool to identify all relevant methods and enforce the corresponding policy depending on users' security requirements. To do this, our static analysis tools extract all the methods and permission list incorporated in the

[10] For a discussion about the legal issues regarding application instrumentation we refer the reader to Schreckling et al. (2013).

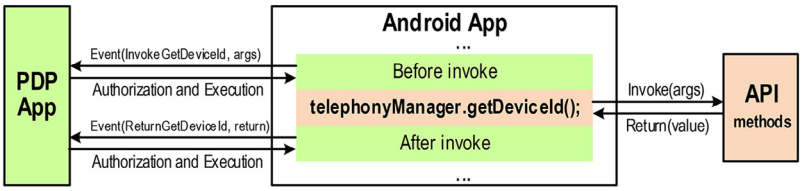

Fig. 7 – Injection of code before and after a sensitive API method invocation.

Table 1 – Example of "Sensitive" API methods as identified by our approach in a real Android application.

Permission	Sensitive API method
MANAGE_ACCOUNTS	invalidateAuthToken()
VIBRATE	NotificationManager.notify()
READ_CONTACTS	ContentResolver.openInputStream()
WAKE_LOCK	MediaPlayer.start()
	MediaPlayer.stop()
ACCESS_NETWORK_STATE	getActiveNetworkInfo()
	getNetworkInfo()
INTERNET	URL.openConnection()
	URL.connect()

examined app and correlate them with the API map published by Felt et al. (2011) to identify all the "sensitive" API method invocations. Table 1 illustrates a small sample mapping a permission to a set of instrumented sensitive invocations.

In our approach, to enforce the policies we instrument all possibly relevant API methods which may not be needed according to the users' preferences expressed in the defined security policies. At runtime when the instrumented code is reached, the app verifies if there is a policy defined for the respective method, and in case no policy is defined the execution simply continues with an almost negligible delay (see Section 6 for our performance results). With this approach, there is no need to re-instrument the application in case the policies change since we have complete coverage of all possible needed PEP code.

The injected PEP code is a simple call to the PEP library, which is automatically added to all instrumented apps. By having the PEP code in a library we minimise the amount of injected code in the instrumented app and provide higher transparency to the modified app. To do so, we identify through application manifest the appropriate package name for installing the PEP library in the appropriate path location. This means that the complete code is an additional package, while the original code should be modified mainly before or after the specific method invocations according to the given security policy, defined by the user (see Fig. 7).

The PDP enabled app contacts the PDP service before and after the invocation of the sensitive API method (see Fig. 7) and receives the appropriate authorisation actions from the PDP app.[11] The authorisation action may deny the execution, which simply ignores the method invocation, or may allow as is, allow with modifications, or delay the execution. For the authorisation action after the method execution only modifications and delays are allowed, since it is not possible to deny the method invocation anymore. This is mapped to actual and tentative events in our policy enforcement language already introduced.

For all PDP enabled apps, we instrument the *onCreate* method of the main activity to initialise the connection to the PDP service. In case the PDP service is not available, the PDP enabled app asks the user if he/she wants to retry the connection, to continue the app execution and be warned of future dangerous method invocations, or to quit the app execution. In case the user chooses to continue, the instrumented code tries to re-connect with the PDP every time a sensitive method execution is reached in the code.

When the PDP connection is established or during the app execution, the PDP service may become unavailable due to lack of resources in the mobile phone. In this case, the instrumented code prompts the user for a decision to continue the execution allowing the sensitive method to be executed, and may choose to be asked again or to allow/deny automatically any successive invocations in case the PDP continues to be unavailable. The user decision can be also generalised for all the invocations or for the specific method invoked when the PDP was not available, and it may be stored for the current execution instance of the app or made persistent for all the following execution instances. Independently of the user decision or whether the PDP was contacted or not, all sensitive invocations are logged for future verification from the user.

From a practical point of view due to copyright infringement issues apps cannot be instrumented without the consent of their developers. Therefore, we foresee that the operationalisation of our solution could only be achieved if the developers themselves adopted our solution and provided an instrumented version of their original apps, which should be signed and certified by them as well. In this way, app developers could prove that they are indeed acting in the best interest of end-users with respect to protection of their privacy, leading even to a large adoption of their apps and increased trust by end-users. It is part of our future work to provide our solution as an open platform to enable app developers in this direction.

5. Case study and implementation

In our case study we have applied our approach to specify and enforce security policies in a well-known social network app. To do so, we disassemble the app and analyze its bytecode based on our *Instrumentation Engine* in order to enforce a given policy,

[11] We made our code available as an open source project available at: https://github.com/r-neisse/SecKitRelease.

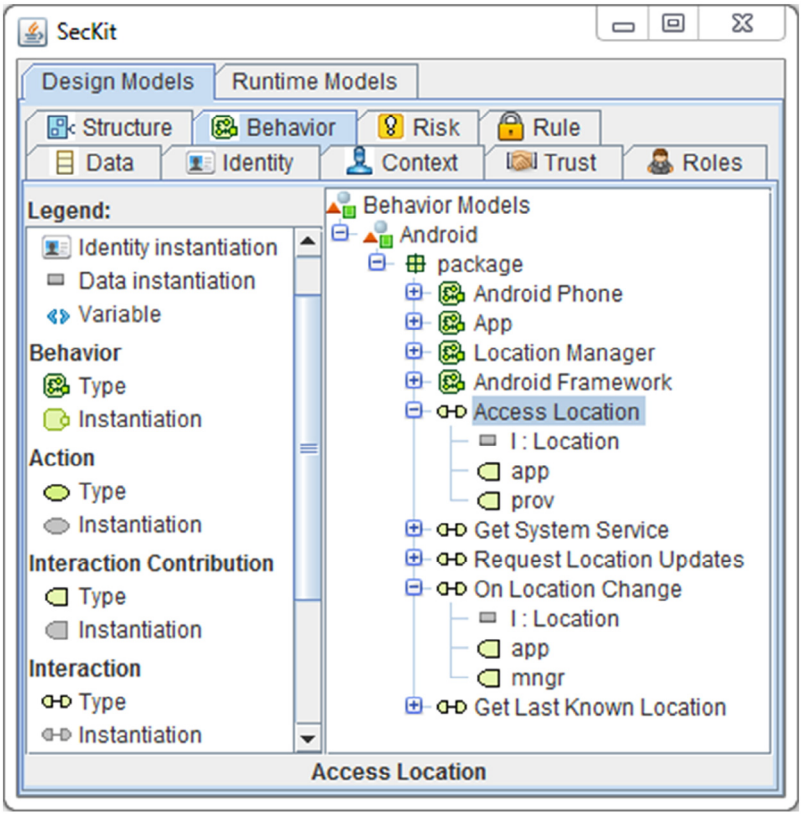

Fig. 8 – Interaction type specification.

and generate a PDP-enabled app. The app which we examine for our case study requests 58 different permissions, including the following permissions that enable the access to end-users' location:

- ACCESS_COARSE_LOCATION;
- ACCESS_FINE_LOCATION;
- ACCESS_NETWORK_STATE.

Specifically, the app gains access to end-users' location through the following API methods:

- `getCellLocation();`
- `getNeighboringCellInfo();`
- `getLastKnownLocation();`
- `requestLocationUpdates();`
- `requestLocation().`

Note that in order to identify these calls our Instrumentation Engine extracts all the possible API methods from the reverse engineered app and compare them against the permission map published in Felt et al. (2011) as mentioned in the previous section. Depending on the end-users' defined policy we introduce the appropriate hooks in order to contact the PDP, and repackaging a PDP enabled app. So for every user defined sensitive operation the PDP would be contacted.

After the analysis, instrumentation, and generation of the PDP-enabled app the abstract policy and refinement model of the Android app must be specified. Our approach has been implemented as part of the Model-based Security Toolkit (SecKit), which is an integrated approach for security engineering (Neisse et al., 2015). In our case study we show a running example considering the specification and refinement of policies focusing on the *Access Location* interaction, already introduced in the previous section.

Fig. 8 shows the SecKit Graphical User Interface (GUI) for specification of the app behaviour. In this GUI the interaction type *Access Location* is specified with one data instantiation *l* of data type *Location* and two possible interaction participants. The interaction participants are represented by the interaction contributions of an *app* and the location provider *prov*. All possible abstract and concrete interaction types considered in the specification of security policies must be defined in this GUI. The data type *Location* must be also specified, and this is done in the *Data* tab of the GUI (not shown here due to space limitations).

We assume a security expert should specify abstract interactions representing potential privacy invasions from the android app side, including possible refinements. For example, to access the user location an app may access the phone information including the connected base station and derive the location from a database of GSM antennas and coordinates. This third refinement option (see Fig. 5 for the first two) could be included and a security policy automatically derived without the need for users to change their abstract high-level policy.

Fig. 9 shows the specification of behaviour types, behaviour instantiations, and interaction instantiations. We model the *Android Phone* behaviour containing instantiations of *App*,

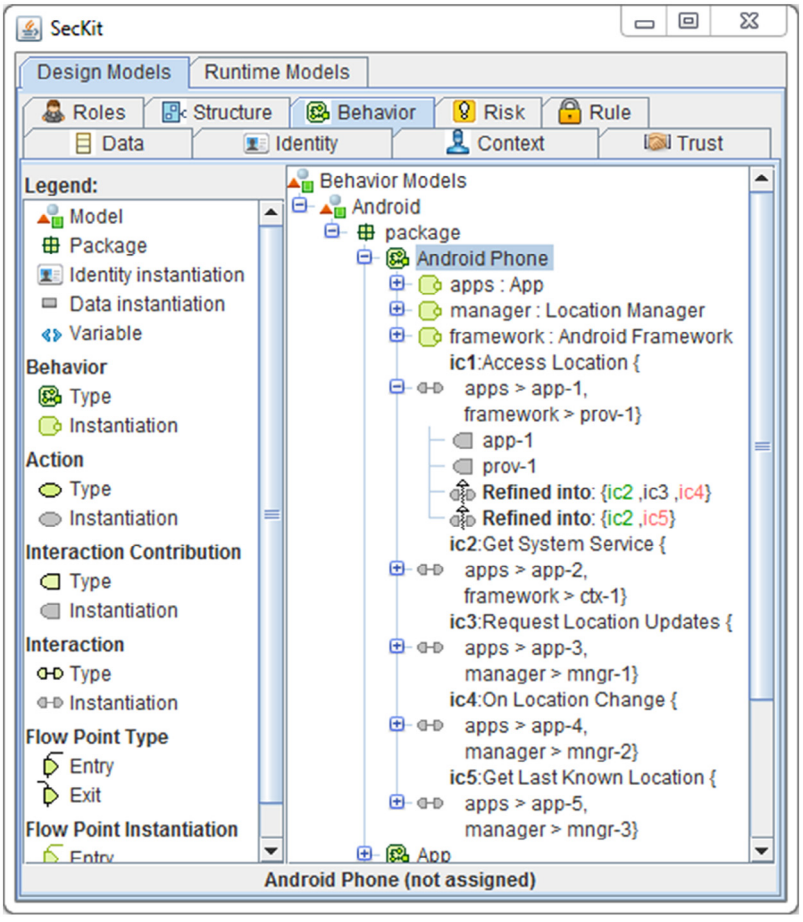

Fig. 9 – Behaviour with abstract and concrete interactions.

Location Manager, and *Android Framework* behaviours. These behaviour instantiations interact with abstract and concrete interaction instantiations, which are a relative concept, since we allow for more than one level of abstraction/refinement relations. An interaction instantiation references the respective behaviours that interact with (for example, the *ic1* instantiation of type *Access Location* represents an interaction between an *App* and the *Android Framework*). Each interaction instantiation also contains a list of possible refinements, which in the case of the *Access Location* instantiation represent two possible ways of accessing the location already introduced in Fig. 5. Start activities are shown in green and end activities are shown in red.

Fig. 10 shows the specification of security policy templates including the refined templates automatically generated using our refinement rules. The highlighted policy template *Anonymize Location* shows a simple policy that references the *Access Location* interaction instantiation *ic1*. This policy template when instantiated is triggered before the interaction is executed and modifies the data *l* to the value *anonymous*.

Fig. 10 also shows the three generated refined policy templates. According to our refinement rules, in this example, the *Access Location* tentative event is defined in a policy that allows and modifies the value of *l* exchanged in the interaction. Therefore, nested policies must be generated for all concrete interactions that instantiate *l*. As a result, the refined policy template instantiates a template that modifies *l* for the start refined activities, which in this case are *On Location Change* and *Get Last Known Location*. The container refined template of *Anonymize Location* instantiates these two generated templates with a *true* condition and the respective event pattern, while the original more complex condition is mantained in the container template with a generic event pattern.

The semantics of containment implies that a contained policy is only triggered if the trigger and condition of the container template are satisfied. Therefore, the trigger of the contained templates must be a refinement of the trigger of the container template. For example, if the container template trigger matches all *Access Location* interaction types, the container template may only further restrict this pattern matching specific instantiations of this type or matching instances with specific data values.

The SecKit implementation includes the PDP rule engine component for evaluation of the refined policy rules and a HTTP/JSON interface for notifications of events. The PDP-enabled app when executed will notify PDP every time a sensitive method is reached for execution. In the current implementation we run the PDP rule engine outside the phone, a PDP version running directly in the phone is part of our ongoing and future work.

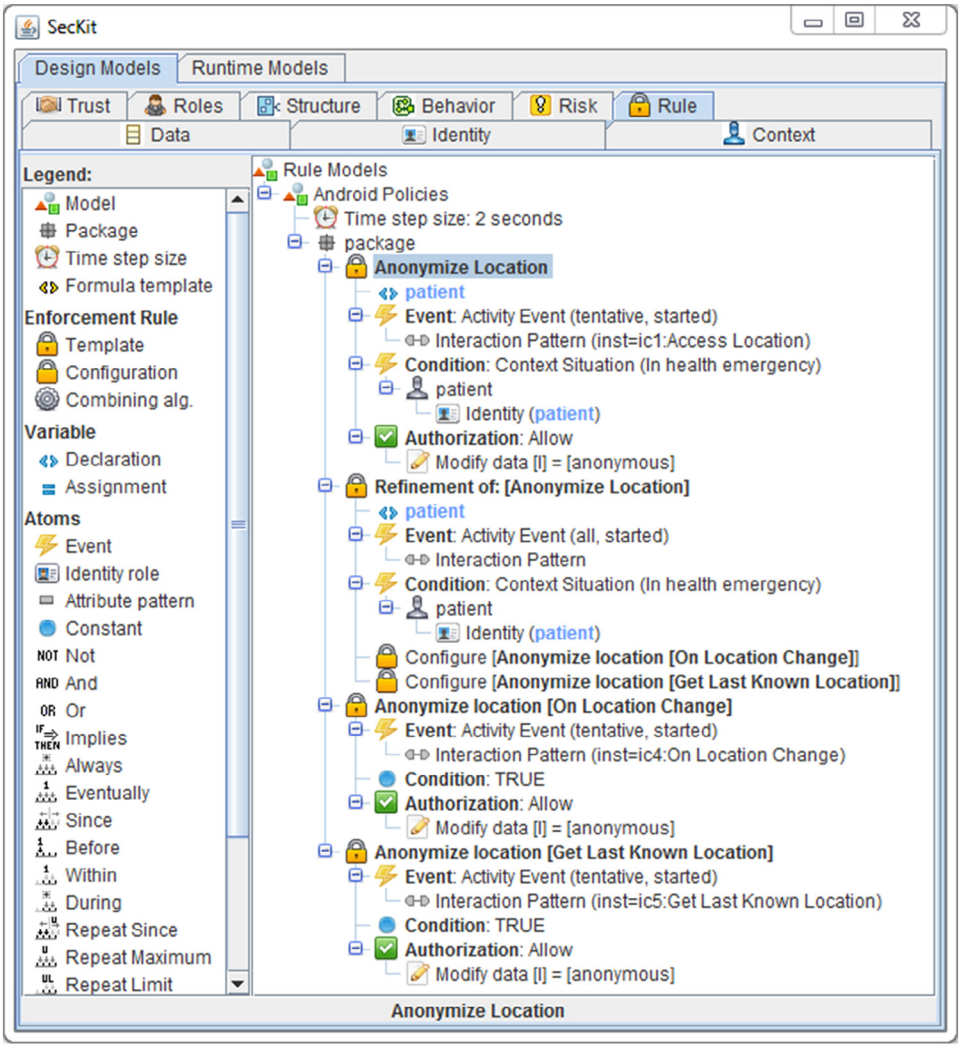

Fig. 10 – Specification of abstract policy and concrete policies automatically generated.

6. Evaluation

In this section we present the evaluation of our approach in terms of performance by analysing: (a) the overhead delay due to the interaction between PDP-enabled app with the PDP component, (b) the battery consumption due to the network interaction of the PDP-enabled app with the PDP component, and (c) the memory size of the PDP-enabled app after instrumentation.

6.1. *Increase of app execution delay due to PDP interaction*

In order to evaluate the introduced interaction delay we employed a test-bed architecture where a PDP-enabled app is deployed in an Android OS emulator and the PDP component runs in the host machine of the emulator. We do not evaluate scalability issues and the delay introduced in the PDP due to a high number of deployed policies, we simply have one accept rule deployed.

The evaluation of the delay introduced exclusively by the PDP component has to be evaluated separately because the PDP is a back-end component that can used for evaluation of policies enforced using different technologies. We have performed a complete isolated evaluation of the policy evaluation delay and scalability issues in a previous publication by some of the co-authors (Neisse et al., 2015).

With respect to an estimate number of policies for a user a worst case scenario could be the case where each installed app requests all possible APIs associated with all permissions. According to Callaham (2014) users have an average of 95 apps installed in their Android phones, and according to Au et al. (2012) there are around 750 API methods associated with the available permissions in Android version 4.3.1. This average number of apps and possible API methods would be equivalent to around 71 thousand policies, which according to the PDP evaluation results published in Neisse et al. (2015) would correspond to a response time of around 100 ms. This is just an indicative number since policies may be complex and make reference to multiple API methods, or multiple policies could reference the same API method due to different requirements.

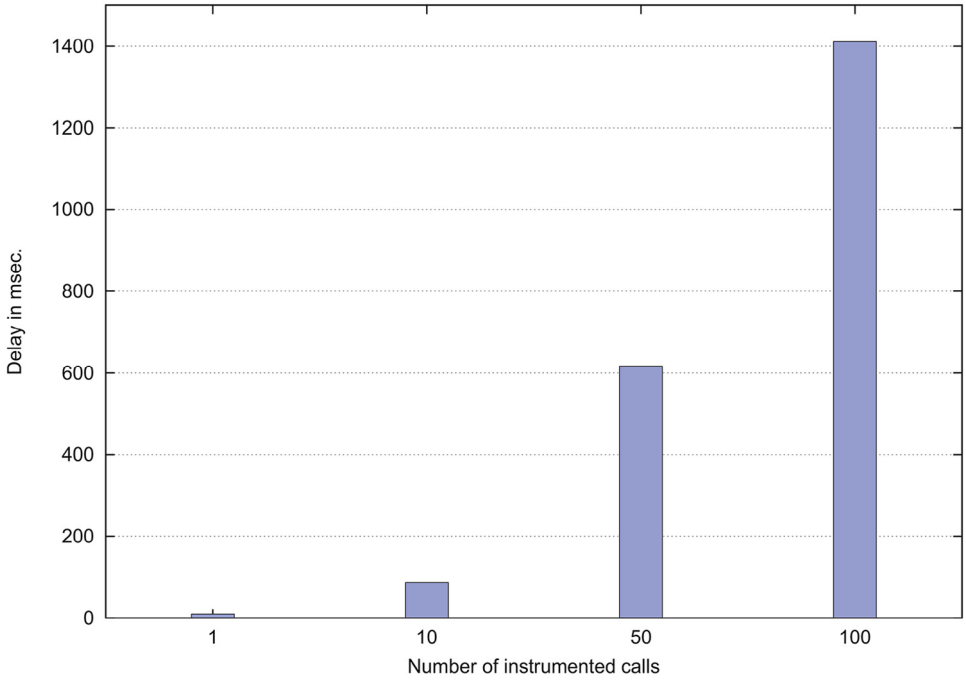

Fig. 11 – PDP enabled application average delay considering different number of instrumented APIs.

In order to measure the app execution delay we developed a custom made benchmark in which we simulated an instrumentation of one (1), ten (10), fifty (50) and one hundred (100) API calls of a given application according to the preferences of the end-user. This was done to assess application's responsiveness from end-users' perspective under different cases. All the experiments were repeated 1000 times, while the execution was automated by using the Android's Monkey test suite.[12]

Fig. 11 gives an overview of our evaluation results. This graph shows that using our proposed framework for empowering user's privacy the total delay introduced in the app was of 1.400 msec when instrumenting 100 API method invocation to call the PDP, while it was only 9.9 msec for instrumentation of one method invocation.

6.2. *Increase of battery consumption due to PDP interaction*

In order to evaluate the battery consumption of our solution we have developed a *Timer* app that runs for 3 hours, keeping the phone screen always on with a fixed brightness level, and updates every second the main activity screen displaying the current battery level. The tests were run in a Samsung Galaxy S5 mobile phone, running Android version 5 (Lollipop), using WiFi or 4G networking, and with automatic updates disabled in order to prevent app updates that could cause additional battery consumption during our tests. In order to establish a baseline we ran this app for 3 hours after fully charging and restarting the phone, resulting in an average battery consumption of 17% for 3 runs with WiFi or 4G only enabled.

After establishing this baseline we instrumented the *Timer* app including one call to the PDP every second when the battery level was displayed in the main activity screen, resulting in a total of around 10 thousand PDP calls. These number of PDP calls in a 3 hours period is definitely a worst case scenario since it is very unlikely that during normal usage an app will request access to a sensitive/instrumented API methods every second.

We measured the increased battery consumption using WiFi or 4G communication with the PDP running in a secure cloud server accessible through an encrypted connection (HTTPS) using a valid server certificate.[13] The average battery consumption after running the tests 3 times was of 21% using WiFi and 24% using the 4G connection, meaning and increased battery consumption of 4% and 7% respectively for WiFi and 4G. In summary, considering the large number of PDP calls performed during our tests and the observed battery consumption we believe our solution is feasible to be used by users since it does not pose a significant overhead. Table 2 summarises our evaluation results.

6.3. *Increase of app size due to instrumentation*

With regard to the impact on the application size, the main overhead is due to the PDP library incorporated in the application to enforce the corresponding policy. For example, the original size of the examined application was 835.882KB, and after injecting the PDP code the size increased only by 8224 bytes. It is worth noticing that the instrumentation of additional calls would add a very small overhead since, as shown

[12] https://developer.android.com/studio/test/monkeyrunner/index.html.

[13] The PDP was located at the server https://seckit.eu.

Table 2 – Battery consumption.

PDP enabled	WiFi	4G	Battery	Delta
N	Y	N	17%	Baseline
N	N	Y	17%	+0%
Y with HTTPS	Y	N	22%	+5%
Y with HTTPS	N	Y	24%	+7%

Table 3 – PDP enabled application size considering different number of instrumented API methods invocations.

Instrumented calls	Size (KB)	Diff (bytes)
–	835.882	
1	844.106	8224
10	844.112	8230
50	844.118	8236
100	844.132	8250

by our analysis, the difference between instrumenting 100 method invocations or only 1 is of 26 bytes (see Table 3). This is an effect of the Android APK optimisation that takes place during the app repackaging stage.

7. Related work

In this section we provide an overview of the existing related works focusing on the security policies enforcement in Android for enhancing users' security and privacy levels against apps with a privacy invasive behaviour. Approaches like Damopoulos et al. (2014) and Grace et al. (2012) that focus specifically on malware detection is out of the scope of this paper. We present these solutions in chronological order, briefly analysing the functionality proposed, the implementation details, limitations, and contrast to our proposal.

Kirin, proposed by Enck et al. (2009), is a security service running on the phone which analyses the requested permissions of an app and detects potential security flaws. When an app is about to be installed, Kirin evaluates, using a rule based engine, if there is a match between the set of requested permissions and the signatures defined in the rule engine. Note that the signatures defined in the rule engine represent possible attack vectors, for example, RECORD_AUDIO and INTERNET permissions define a rule in the signature set. Kirin results in a high number of false positives since legitimate apps that follow the defined signature pattern are characterised as malicious. In contrast to our proposal, Kirin does not provide enforcement capabilities, it is only a solution to inform users about possible risks.

Ongtang et al. (2009) propose Saint as an extension of Kirin. In addition to the analysis performed by Kirin at install time, Saint monitors apps also at the runtime Inter-Component Communication (ICC) flows, *e.g.,* activities initialisation, components binding to services, access to content providers, *etc.* The policies defined in Saint are static and similarly to our approach define conditions to control the runtime behaviour. For example, when a specific activity can bind with a specific content provider considering the allowed permissions, signature, or package name. Saint is implemented as a modified Android middleware, while our proposal relies on app instrumentation for policy enforcement. Furthermore, we propose a more flexible architecture and expressive policy language with the possibility of deploying and changing policies at runtime without requiring changes to the middleware or instrumented app.

Also Apex, introduced by Nauman et al. (2010), focuses on policy enforcement for regulating ICC flows. In contrast to Saint, Apex supports more complex policy conditions using dynamic attributes including cardinality and time dimension constrains, *i.e.,* restricting the maximum number of SMS messages sent by an app. Policy rules must be defined to manage the initialisation, updating, and resetting of dynamic attributes. Both Saint and Apex support authorisation actions to allow or deny an ICC flow, without the possibility of modifying or obfuscating a flow which is supported in our framework.

Orthogonally to our proposal, Dietz et al., 2011) propose QUIRE as a solution to protect android apps manipulation by other malicious apps or services. Their proposal is to enable apps to reason about access to sensitive data through call chain validation. To achieve this goal the authors propose the modification of the underlying OS IPCs mechanism in order to pass the appropriate information between IPCs.

In the same direction, Porscha, proposed by Ongtang et al. (2010), introduces a Digital Rights Management (DRM) framework for Android phones that mediates the access to protected content between different Android components. For example, it can regulate the access of an app to the content of an SMS message. The Porscha mediator supports constraints on devices, apps, and on the use (e.g., cardinality) of the protected data. Porscha mediates ICC flows, with extensions including a policy filed, and it has been implemented as a modified Android firmware that is considered to be trusted. Our proposal does not require changes to the Android firmware, therefore, our solution could be adopted straightforwardly in all different firmware versions.

CRePE, introduced by Conti et al. (2011)., is also a customised Android OS system able to enforce fine-grained security policies considering time and location features. Policies in this system intercept authorisation requests before the standard Android permission checks, so that if the request is allowed by CRePE the standard permission check may still deny it. In addition to the standard permission checks, it also intercepts and enforces policies when activities are started. Policies in CRePE consist of propositional conditions of allow or deny actions, which are less expressive than our policy rules that also support modifications and delays as enforcement actions.

Bai et al. (2010) propose a context-aware usage control that focus on a user basis mechanism for granting and revoking permissions, similar to the approach introduced in the latest android OS version. To do so, authors enable users to define policies related to application permission grants in order to protect access to users' sensitive resources. However, the use of this approach requires the modification of underlying Android OS services. In a similar direction, Sun et al. (2012) introduce a design that requires the modification of Android sandbox as well, in order to monitor access to sensitive information. In this approach the hook points are installed before the actual permission check occurs by Android OS. On the contrary,

SecureDroid (Arena et al., 2013) extends Android OS security manager service to control access also for user defined sensitive URIs.

Shabtai et al. (2010) first proposed the use of SELinux in Android to implement low-level Mandatory Access Control (MAC) policies. From Android 4.3 on, SELinux is used by default to further define apps in permissive mode, only logging permission denials. From Android 5.0, SELinux is used in full enforcement mode, in which permission denials are logged and enforced according to the specified policies.

Batyuk et al. (2011) introduced Androlyzer, a server based solution that focuses mainly on informing users about apps potential security and privacy risks. To do this, Androlyzer first does the reverse engineering of the app, and then a static analysis to determine possible flaws. In addition, Androlyzer provides an approach to mitigate the identified flaws by modifying the examined app based on users' preferences. However, Androlyzer does not use an expressive policy language to support users in enforcing their security requirements into an app.

Papamartzivanos et al. (2014) propose a cloud-based crowdsourcing architecture where users share any locally logged information about the app of interest. The authors' goal is to use the exchanged logs to calculate the app's privacy exposure level considering the exchanged information between the various participants in the system. The authors use the Cydia Substrate, which can only be installed in rooted devices to hook code in method invocations and object creations. A user may decide to always allow, deny, or be asked about what to do every time a hooked method is invoked by the running app.

TISSA, proposed by Zhou et al. (2011), introduces a *privacy mode* functionality in Android with coarse-grained control over the behaviour of an app. Using TISSA users can have more fine-grained control over private information like location, phone identity, contacts, call log, *etc.* TISSA is implemented as a modified OS with proxy content providers for each controlled information that are responsible for retrieving and enforcing the corresponding policies. TISSA's policies are hard-coded and restricted to a static set of authorisation options without support for complex conditions. A very similar approach with slightly less control on private information is introduced by Beresford et al. (2011) in their solution named MockDroid. Complementary, AppFence proposed by Hornyack et al. (2011)., also implemented as a modified OS on the basis of TaintDroid, shadows and ex-filtrates users' private data according to their preferences.

Feth and Pretschner (2012) employ information flow tracking as well. Their framework uses an expressive policy language to describe users' preferences to content providers, intents, and certain data sinks like the network, file system and IPC in order to eliminate access to private data. Jung et al. (2013) extend the work of Feth and Pretschner with context-aware policy rules. In this direction, Andriatsimandefitra et al. (2012) introduce an approach for determining data flows, which is an important aspect for realising a policy enforcement tool. In contrast to all these approaches for policy enforcement in Android we are the only ones to propose the use of policy refinement techniques to simplify the management of the security policies by end-users.

In an alternative approach, Xu et al. (2012) introduce an additional "sandbox" security service for protecting users' against apps malicious behaviour, namely Aurasium. In contrast to other similar approaches including ours, Aurasium enforces its security policies in the Android libc level through interposition as a middleware between Android kernel and user space layer. This means that the original app is repackaged to a new app which includes the appropriate code enabling Aurasium to control access to sensitive sources.

Constroid, introduced by Schreckling et al. (2012), also defines a management framework for employing data-centric security policies of fine granularity. To do this, Constroid adopts the $UCON_{ABC}$ model. However, in contrast to our contribution in this paper only the abstract model is detailed and no concrete example of policy is provided.

Zefferer and Teufl (2013) propose a solution for device security assessment based on user defined preferences. The use of security policies guarantees that each application that integrates the developed service can define and assess its own critical aspects. In order for such a service to be employed in a given device is required by a third party app to integrate the appropriate controls to the app through the corresponding API. However, researchers have shown that programmers are not taking into consideration in most of the cases security features.

SEDalvik, introduced by Bousquet et al. (2013), proposes a MAC mechanism to regulate information flows between apps objects building on the advantages of Dalvik internal debugger. Specified policies define which interactions are allowed to take place in a given context.

Schreckling et al. (2013) introduce Kynoid, a solution that extends Taintdroid with security policies at the variable level. Kynoid retains the taint propagation performed by Taintdroid and maintains a dependency graph where a direct edge represents a security requirement (a.k.a. policy) between two objects. In this way, Kynoid provides a fine grained control to sensitive flows. The focus of Kynoid is on information flow policies, which are not explicitly supported in our framework and are part of our future work.

AppGuard, introduced by Backes et al. (2013), is an app instrumentation framework that runs directly in users' device and allows user-centric security policies customisations. AppGuard computes a risk score for each app considering the number of dangerous permissions and provides the option of instrumenting the app to control the access to "dangerous" calls. For example, in an app with NETWORK permission, a user can choose to enable/disable the corresponding functionality. The solution presented by Bartel et al. (2012) follows the same direction. In contrast to our proposal these solutions do not support context-based policy specification and policy refinement in order to simplify the policy management by end-users.

Zhauniarovich et al. (2014) propose MOSES, which enforces context-based policy specification at the kernel level, meaning that MOSES requires a modification to the underlying OS. In this approach users can define a security profile that could be applied in a specific context, *i.e.*, at a specific time and location for a given app. Note that if a security profile is not linked to an app, then MOSES does not allow access to any "sensitive" resource since by default employs a negative authorisation policy. MOSES security profile consists of allow or deny rules according to user's requirement.

IdentiDroid, proposed by Shebaro et al. (2014), is a customised version of Android which gives to the user the possibility to switch in an anonymous modality that shadows sensitive data and block permissions at runtime. Even though there is no a complex policy definition and refinement, the IdentiDroid Profile Manager allows users to define different profiles specifying which applications can access or not sensitive data and resources.

DroidForce proposed by Rasthofer et al. (2014) relies on the Soot framework for analysing and instrumenting an app to enforce a security policy. This approach considers PEPs injected in multiple applications with a single PDP running as an app in the phone, addresses information flow intra-app statically and inter-app at runtime, and uses an expressive policy language with cardinality and temporal constraints. Their policies allow or deny an activity, while do not support modification/obfuscation of values. Complementary, Jing et al. (2015) propose DeepDroid, which in contrast to DroidForce performs instrumentation at the native level with the possibility of intercepting system calls in addition to methods invocation to regulate the access to sensitive resources. However, DeepDroid does not consider information flow tracking nor uses any expressive policy language for enforcement.

Bagheri et al. (2015) in DroidGuard introduce a framework for modelling inter-app vulnerabilities and employing the appropriate protection mechanism to enhance user's privacy and security. Briefly, DroidGuard analyses statically a set of given apps to foresee security flaws realising through apps intercommunication. The generated model is used as a policy to be employed as a proactive countermeasure.

More similar to our approach Cotterell et al. (2015) introduce a solution to enable users to install policies for controlling sensitive data; however, the policies are not based on access to sensitive resources. Instead, the authors focus on known malicious activities for defining a policy with a more explicit focus on malware apps.

8. Conclusions and future work

Mobile devices are today the primary interface used by end-users to access online services. They are the mean to perform several different operations and they are the repository of a huge amount of sensitive and personal information. Hence, their security and privacy should be the building blocks to enhance trust in digital services. In the mobile world, Android is the dominant operating system as it is an open platform that supports different types of hardware devices. However, due to its popularity and its innate design properties, it is also one of the main targets of attackers that want to access to users' private data. In this paper we analysed different attack vectors leading to private data leakages, and we proposed a policy based approach for enhancing users' privacy by empowering them in controlling the access to sensitive resources.

Our solution provides flexibility and transparency both to users and apps, by decoupling the specification of security constraint from the enforcement. Performance evaluation outcomes show that the enforcement overhead in terms of processing time is limited, and thus we believe that our solution provides a balance between users' privacy and apps "unrestrained" access. The presented approach, at the moment, foresees the enforcement of the policy through app instrumentation. Even if instrumentation can be easily automated to make the operation accessible to the average end-users, we recognise that code injection poses some questions related to liability issues of the resulting new "instrumented app". A far more logical approach would be that of encouraging the mobile app community in developing "privacy by design applications" integrating by default the PEP (policy enforcement point) into their code, giving back to the end-users full control of the behaviour of their mobile devices.

As future work we plan to include in our refinement approach the support for behaviour types, actions types, and execution of abstract actions by policies. By addressing behaviour types we could specify an abstract policy for a phone that would be refined to all installed apps and components. Furthermore we plan to investigate the integration of explicit and implicit information flow tracking tools with quantitative data policies. We also plan to launch a community-based release of our tool where users can contribute with abstract policies and refinement models of privacy sensitive activities, for example, integrating the results of other Android security approaches (e.g. Rasthofer et al., 2014) that cannot be easily reused at the moment.

REFERENCES

Aafer Y, Du W, Yin H. Droidapiminer: mining api-level features for robust malware detection in android. In: Zia T, Zomaya A, Varadharajan V, Mao M, editors. Security and Privacy in Communication Networks, vol. 127. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer International Publishing; 2013. p. 86–103.

Andriatsimandefitra R, Geller S, Tong VVT. Designing information flow policies for android's operating system, in: 2012 IEEE international conference on communications (ICC), 2012, pp. 976–81. doi:10.1109/ICC.2012.6364161.

Arena V, Catania V, Torre GL, Monteleone S, Ricciato F. Securedroid: an android security framework extension for context-aware policy enforcement, in: 2013 international conference on privacy and security in mobile systems (PRISMS), 2013, pp. 1–8. doi:10.1109/PRISMS.2013.6927185.

Arp D, Spreitzenbarth M, Hubner M, Gascon H, Rieck K. Drebin: effective and explainable detection of android malware in your pocket, in: 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23–26, 2013, 2014.

Au KWY, Zhou YF, Huang Z, Lie D. Pscout: analyzing the android permission specification, in: Proceedings of the 2012 ACM conference on computer and communications security, CCS '12, ACM, New York, NY, USA, 2012, pp. 217–28.

Backes M, Gerling S, Hammer C, Maffei M, von Styp-Rekowsky P. Appguard enforcing user requirements on android apps. In: Piterman N, Smolka S, editors. Tools and Algorithms for the Construction and Analysis of Systems, vol. 7795. Lecture Notes in Computer Science. Springer Berlin Heidelberg; 2013. p. 543–8 <http://dx.doi.org/10.1007/978-3-642-36742-7_39>.

Bagheri H, Sadeghi A, Jabbarvand R, Malek S. Automated dynamic enforcement of synthesized security policies in android, Tech.

Rep. Technical Report GMU-CS-TR-2015-5, George Mason University, 2015.
Bai G, Gu L, Feng T, Guo Y, Chen X. Security and Privacy in Communication Networks: 6th International ICST Conference, SecureComm 2010, Singapore, September 7–9, 2010. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, Ch. Context-Aware Usage Control for Android, pp. 326–43. http://dx.doi.org/10.1007/978-3-642-16161-2_19.
Bartel A, Klein J, Monperrus M, Allix K, Traon YL. Improving privacy on android smartphones through in-vivo bytecode instrumentation, CoRR abs/1208.4536. <http://arxiv.org/abs/1208.4536>; 2012 [accessed 07.16].
Bartel A, Klein J, Le Traon Y, Monperrus M. Dexpler: converting android Dalvik bytecode to Jimple for static analysis with soot, in: Proceedings of the ACM SIGPLAN international workshop on state of the art in Java program analysis, SOAP '12, ACM, New York, NY, USA, 2012, pp. 27–38.
Batyuk L, Herpich M, Camtepe SA, Raddatz K, Schmidt A-D, Albayrak S. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications, in: Proceedings of the 2011 6th international conference on malicious and unwanted software, MALWARE '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 66–72. http://dx.doi.org/10.1109/MALWARE.2011.6112328.
Beresford AR, Rice A, Skehin N, Sohan R. Mockdroid: trading privacy for application functionality on smartphones, in: Proceedings of the 12th workshop on mobile computing systems and applications, HotMobile '11, ACM, New York, NY, USA, 2011, pp. 49–54. http://doi.acm.org/10.1145/2184489.2184500.
Bousquet A, Briffaut J, Clevy L, Toinard C, Venelle B. Mandatory access control for the android Dalvik virtual machine, in: Presented as part of the 2013 workshop on embedded self-organizing systems, USENIX, Berkeley, CA, 2013. <https://www.usenix.org/conference/esos13/workshop-program/presentation/Bousquet>; [accessed 07.16].
Callaham J. Yahoo Aviate data shows 95 apps are installed on the average android device. <http://www.androidcentral.com/yahoo-aviate-data-shows-95-apps-are-installed-average-android-device>; 2014 [accessed 07.16].
Conti M, Nguyen VTN, Crispo B. Crepe: context-related policy enforcement for android, in: Proceedings of the 13th international conference on information security, ISC'10, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 331–45. <http://dl.acm.org/citation.cfm?id=1949317.1949355>.
Cotterell K, Welch I, Chen A. An android security policy enforcement tool. Int J Electron Telecomm 2015;61(4):311–20. doi:10.1515/eletel-2015-0040.
Damopoulos D, Kambourakis G, Anagnostopoulos M, Gritzalis S, Park JH. User privacy and modern mobile services: are they on the same path? Pers Ubiquitous Comput 2013;17(7):1437–48. doi:10.1007/s00779-012-0579-1. <http://dx.doi.org/10.1007/s00779-012-0579-1>.
Damopoulos D, Kambourakis G, Portokalidis G. The best of both worlds: a framework for the synergistic operation of host and cloud anomaly-based IDS for smartphones, in: Proceedings of the seventh European workshop on system security, EuroSec '14, ACM, New York, NY, USA, 2014, pp. 6:1–6. http://doi.acm.org/10.1145/2592791.2592797.
Desnos A. Androguard – reverse engineering, malware and goodware analysis of android applications and more (ninja !) – Google project hosting. <http://code.google.com/p/androguard/>; 2012 [accessed 07.16].
Dietz M, Shekhar S, Pisetsky Y, Shu A, Wallach DS. QUIRE: lightweight provenance for smart phone operating systems, in: 20th USENIX Security Symposium, San Francisco, CA, USA, August 8–12, 2011, Proceedings, 2011. <http://static.usenix.org/events/sec11/tech/full_papers/Dietz7-26-11.pdf>; [accessed 07.16].
Ducklin P. Apple's app store bypassed by Russian hacker, leaving developers out of pocket. <http://nakedsecurity.sophos.com/2012/07/14/apple-app-store-bypassed-by-russian-hacker-leaving-developers-out-of-pocket/>; 2012 [accessed 07.16].
Egele M, Kruegel C, Kirda E, Vigna G. Pios: detecting privacy leaks in ios applications, in: Proceedings of the network and distributed system security symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011, 2011. <http://www.isoc.org/isoc/conferences/ndss/11/pdf/9_2.pdf>.
Enck W, Ongtang M, McDaniel P. On lightweight mobile phone application certification, in: Proceedings of the 16th ACM conference on computer and communications security, CCS '09, ACM, New York, NY, USA, 2009.
Enck W, Gilbert P, Chun B-G, Cox LP, Jung J, McDaniel P, et al., Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones, in: Proceedings of the 9th USENIX conference on operating systems design and implementation, OSDI'10, USENIX Association, Berkeley, CA, USA, 2010, pp. 1–6.
Felt AP, Chin E, Hanna S, Song D, Wagner D. Android permissions demystified, in: Proceedings of the 18th ACM conference on computer and communications security, CCS '11, ACM, New York, NY, USA, 2011, pp. 627–38. http://doi.acm.org/10.1145/2046707.2046779.
Felt AP, Hanna S, Chin E, Wang HJ, Moshchuk E. Permission re-delegation: attacks and defenses, in: In 20th Usenix Security Symposium, 2011.
Felt AP, Ha E, Egelman S, Haney A, Chin E, Wagner D. Android permissions: user attention, comprehension, and behavior, in: Proceedings of the eighth symposium on usable Privacy and security, SOUPS '12, ACM, New York, NY, USA, 2012, pp. 3:1–14.
Feth D, Pretschner A. Flexible data-driven security for android, in: Proceedings of the 2012 IEEE sixth international conference on software security and reliability, SERE '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 41–50.
Geneiatakis D, Fovino IN, Kounelis I, Stirparo P. A permission verification approach for android mobile applications. Comput Secur 2015;49(C):192–205. <http://dx.doi.org/10.1016/j.cose.2014.10.005>.
Gibler C, Crussell J, Erickson J, Chen H. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale, in: Proceedings of the 5th international conference on trust and trustworthy computing, TRUST'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 291–307.
Google, Android and security: Google bouncer. <http://googlemobile.blogspot.it/2012/02/android-and-security.html>; 2012 [accessed 07.16].
Grace M, Zhou Y, Zhang Q, Zou S, Jiang X. Riskranker: scalable and accurate zero-day android malware detection, in: Proceedings of the 10th international conference on mobile systems, applications, and services, MobiSys '12, ACM, New York, NY, USA, 2012, pp. 281–94. <http://doi.acm.org/10.1145/2307636.2307663>.
Gruver B. Smali/baksmali assembler/disassembler for the dex format used by Dalvik. <https://github.com/JesusFreke/smali/wiki>; 2009 [accessed 07.16].
Hornyack P, Han S, Jung J, Schechter S, Wetherall D. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications, in: Proceedings of the 18th ACM conference on computer and communications security, CCS '11, ACM, New York, NY, USA, 2011, pp. 639–52. http://doi.acm.org/10.1145/2046707.2046780.
IDC, Worldwide smartphone os market in 4q12, may 2013.

Jing J, Sun K, Wang Y, Wang X. Deepdroid: dynamically enforcing enterprise policy on android devices, in: NDSS 2015, 2015.
Jung C, Feth D, Seise C. Context-aware policy enforcement for android, in: 2013 IEEE 7th international conference on software security and reliability (SERE), 2013, pp. 40–9. doi:10.1109/SERE.2013.15.
Miller C, Oberheide J. Dissecting the android bouncer. <http://jon.oberheide.org/blog/2012/06/21/dissecting-the-android-bouncer/>; 2012 [accessed 07.16].
Miners Z. Report: Malware-infected android apps spike in the Google play store. <http://www.pcworld.com/article/2099421/report-malwareinfected-android-apps-spike-in-the-google-play-store.html>; 2014 [accessed 07.16].
Nan Z, Kan Y, Muhammad N, Xiaoyong Z, XiaoFeng W. Leave me alone: app-level protection against runtime information gathering on android, in: IEEE Symposium on Security and Privacy, 2015.
Nauman M, Khan S, Zhang X. Apex: extending android permission model and enforcement with user-defined runtime constraints, in: Proceedings of the 5th ACM symposium on information, computer and communications security, ASIACCS '10, ACM, New York, NY, USA, 2010.
Neisse R., Doerr J., Model-based specification and refinement of usage control policies, 11th international conference on privacy, security and trust (PST), 2013.
Neisse R, Steri G, Fovino IN, Baldini G. Seckit: a model-based security toolkit for the internet of things. Comput Secur 2015;<http://dx.doi.org/10.1016/j.cose.2015.06.002>, <http://www.sciencedirect.com/science/article/pii/S0167404815000887>.
Ongtang M, McLaughlin S, Enck W, McDaniel P. Semantically rich application-centric security in android, in: Computer Security Applications Conference, 2009. ACSAC '09. Annual, 2009.
Ongtang M, Butler K, McDaniel P. Porscha: policy oriented secure content handling in android, in: Proceedings of the 26th annual computer security applications conference, ACSAC '10, ACM, New York, NY, USA, 2010, pp. 221–30. http://doi.acm.org/10.1145/1920261.1920295.
Pan B. ModifyApkWithDexTool – dex2jar – modify apk with dex-tools – tools to work with android.dex and java.class files – Google project hosting. <http://code.google.com/p/dex2jar/wiki/ModifyApkWithDexTool>; 2012 [accessed 07.16].
Papamartzivanos D, Damopoulos D, Kambourakis G. A cloud-based architecture to crowdsource mobile app privacy leaks, in: Proceedings of the 18th Panhellenic conference on informatics, PCI '14, ACM, New York, NY, USA, 2014, pp. 59:1–6. http://doi.acm.org/10.1145/2645791.2645799.
Rasthofer S, Arzt S, Bodden E. A machine-learning approach for classifying and categorizing android sources and sinks, Proceedings of the network and distributed system security symposium (NDSS) 2014.
Rasthofer S, Arzt S, Lovat E, Bodden E. Droidforce: enforcing complex, data-centric, system-wide policies in android, in: 2014 ninth international conference on availability, reliability and security (ARES), 2014, pp. 40–9. doi:10.1109/ARES.2014.13.
Schreckling D, Posegga J, Hausknecht D. Constroid: data-centric access control for android, in: Proceedings of the 27th annual ACM symposium on applied computing, SAC '12, ACM, New York, NY, USA, 2012, pp. 1478–85. http://doi.acm.org/10.1145/2245276.2232012.
Schreckling D, Kstler J, Schaff M. Kynoid: real-time enforcement of fine-grained, user-defined, and data-centric security policies for android. Inf Secur Tech Rep 2013;17(3):71–80.
Schreckling D, Huber S, Hhne F, Posegga J. Uranos: user-guided rewriting for plugin-enabled android application security. In: Cavallaro L, Gollmann D, editors. Information Security Theory and Practice. Security of Mobile and Cyber-Physical Systems, vol. 7886. Lecture Notes in Computer Science. Springer Berlin Heidelberg; 2013. p. 50–65 <http://dx.doi.org/10.1007/978-3-642-38530-8_4>.
Shabtai A, Fledel Y, Elovici Y. Securing android-powered mobile devices using SELinux. Security Privacy, IEEE 2010;8(3):36–44. doi:10.1109/MSP.2009.144.
Shabtai A, Fledel Y, Kanonov U, Elovici Y, Dolev S, Glezer C. Google android: a comprehensive security assessment. IEEE Secur Privacy, 2010;8(2):35–44.
Shebaro B, Oluwatimi O, Midi D, Bertino E. Identidroid: android can finally wear its anonymous suit. Trans Data Priv 2014;7(1):27–50. <http://dl.acm.org/citation.cfm?id=2612163.2612165>.
Stirparo P, Kounelis I. The mobileak project: forensics methodology for mobile application privacy assessment, 2012 international conference for internet technology and secured transactions, 2012, pp. 297–303.
Sun L, Huang S, Wang Y, Huo M. Application policy security mechanisms of android system, in: 2012 IEEE 14th international conference on high performance computing and communication 2012 IEEE 9th international conference on embedded software and systems (HPCC-ICESS), 2012, pp. 1722–5. doi:10.1109/HPCC.2012.258.
Tumbleson C, Winiewski R. A tool for reverse engineering android APK files. <http://ibotpeaches.github.io/Apktool/>; 2010 [accessed 07.16].
Vallee-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V. Soot – a Java bytecode optimization framework, 1999.
Wu D-J, Mao C-H, Wei T-E, Lee H-M, Wu K-P. Droidmat: android malware detection through manifest and api calls tracing, in: Proceedings of the 2012 seventh Asia joint conference on information security, ASIAJCIS '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 62–9.
Xing L, Pan X, Wang R, Yuan K, Wang X. Upgrading your android, elevating my malware: privilege escalation through mobile os updating, in: Proceedings of the 2014 IEEE symposium on security and privacy, SP '14, IEEE Computer Society, Washington, DC, USA, 2014, pp. 393–408. http://dx.doi.org/10.1109/SP.2014.32.
Xu R, Saïdi H, Anderson R. Aurasium: practical policy enforcement for android applications, in: The 21st USENIX security symposium (USENIX Security 12), USENIX, Bellevue, WA, 2012, pp. 539–52. <https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/xu_rubin>; [accessed 07.16].
Zefferer T, Teufl P. Policy-based security assessment of mobile end-user devices an alternative to mobile device management solutions for android smartphones, in: 2013 international conference on security and cryptography (SECRYPT), 2013, pp. 1–8.
Zhauniarovich Y, Russello G, Conti M, Crispo B, Fernandes E. Moses: supporting and enforcing security profiles on smartphones. IEEE Trans Dependable Secure Comput 2014;11(3):211–23. doi:10.1109/TDSC.2014.2300482.
Zhou X, Demetriou S, He D, Naveed M, Pan X, Wang X, et al., Identity, location, disease and more: inferring your secrets from android public resources, in: Proceedings of the 2013 ACM SIGSAC conference on computer & communications security, ACM, 2013, pp. 1017–28.
Zhou Y, Jiang X. Detecting passive content leaks and pollution in android applications, in: 20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24–27, 2013, 2013. <http://internetsociety.org/doc/detecting-passive-content-leaks-and-pollution-and-roid-applications>; [accessed 07.16].
Zhou Y, Zhang X, Jiang X, Freeh VW. Taming information-stealing smartphone applications (on android), in: Proceedings of the 4th international conference on trust and

trustworthy computing, TRUST'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 93–107. <http://dl.acm.org/citation.cfm?id=2022245.202225>.

Ricardo Neisse received the PhD degree in Computer Science from the University of Twente, Enschede, The Netherlands, in 2012. He is currently a Scientific/Technical Project Officer at the Joint Research Centre (JRC) of the European Commission in Ispra, Italy. His research interests include security engineering for the Internet of Things, mobile, and enterprise systems.

Gary Steri is a postdoctoral researcher at the Joint Research Center of the European Commission, in Italy. He received his master's degree in Information Technologies in 2006 and his PhD in Computer Science in 2011. His research activity first focused on security and authentication in wireless networks and sensor networks, in particular on wearable inertial measurement units. Currently he is working on security aspects of Internet of Things, Device-to-Device communication and intelligent transport systems.

Dimitris Geneiatakis holds a PhD in the field of Information and Communication Systems Security from the Department of Information and Communications Systems Engineering of the University of Aegean, Greece. His current research interests are in the areas of security mechanisms in Internet telephony, smart cards, intrusion detection systems, and network and software security. Currently, he is postdoctoral researcher at Joint Research Centre of European Commission. Previously, was within Columbia University as a postdoctoral researcher. He is an author of more than thirty refereed papers in international journals and conference proceedings.

Igor Nai Fovino holds a PhD in computer security. His research fields belong to the area of ICT Security of industrial systems and Smart Grids, Intrusion Detection Techniques, Cryptography and Secure Network Protocols. He is an author of more than sixty scientific papers published on international journals, books and conference proceedings. He is member of the IFIP Working group 11.10 for the Protection of Critical Infrastructures and serves as International Expert within the ERNCIP European Expert Group on the security of Energy Smart Grids. Currently, he is within the Joint Research Centre of the European Commission as a scientific project manager.