2015 IEEE 28th Computer Security Foundations Symposium

Android Permissions Unleashed

Alessandro Armando*†, Roberto Carbone*, Gabriele Costa† and Alessio Merlo†
* Security and Trust Unit, FBK-Irst, Trento, Italy
Email: surname@fbk.eu
†Dipartimento di Informatica, Bioinformatica, Robotica e Ingegneria dei Sistemi
Università degli studi di Genova
Email: name.surname@unige.it

***Abstract*—The Android Security Framework controls the executions of applications through permissions which are statically granted by the user during installation. However, the definition of security policies over permissions is not supported. Security policies must be therefore manually encoded into the application by the developer, which is a dangerous practice and may cause security breaches. We propose an improvement over the Android permission system that supports the specification and enforcement of fine-grained security policies. Enforcement is achieved by reducing policy decision problems to propositional satisfiability and leveraging a state-of-the-art SAT solver. Unlike alternative proposals, our approach does not require changes in the operating system and, therefore, it can be readily deployed in any commercial device.**

I. Introduction

Android security has been widely studied in the last years. The motivation of this interest is threefold: *(i)* Android is the most popular mobile operating system; *(ii)* most people own and routinely use smart-phones for personal reasons and—increasingly often—on the workplace; finally, *(iii)* even if the Android security framework offers a number of well-established security mechanisms (e.g., sandboxing through virtualization), it provides inadequate support to application developers for the specification of security policies. This last aspect has been often reported in the literature. For instance, after a systematic study of more than thousand applications, Enck et al. [7] remark this weakness and conclude that "many developers fail to take necessary security precautions".

Beside OS-level security, which is mostly inherited from the Linux kernel, Android application security mainly builds on a permissions system. Permissions are labels that developers must attach to their software, i.e. in the application *manifest*, if they want to access sensitive resources. When the user installs a new application, the required permissions are prompted (with a description in natural language) and she decides whether to grant them or cancel the installation. After the installation, the application can access the resources through the permissions provided by the user.

This approach has several drawbacks, e.g., the coarse-grained description of permissions and the all-or-nothing choice of the user. Among them, the most evident is the lack of guarantees that the user correctly understands the security implications of granting the permissions. This problem also arises whenever the user decides to launch a certain application. As a matter of fact, an ideal, security-aware user should consider whether it is safe to run the application in the current execution context, which might involve many other applications. More realistically, users simply neglect the security requirements of the applications.

In this paper we present an extension of the Android security support that allows developers to apply their own security policies over applications executions and interaction. This is done by slightly extending the syntax of the Android manifest so to support the specification of security policies. Moreover, we associate security policies and permissions to components (as opposed to the whole application as normally done in Android). Permissions remain unchanged, while policies are declared through a compact, yet expressive, language. The policy language mostly consists of propositional logic and few modal operators.

Our approach offers a number of advantages, the most relevant ones being listed below.

No new development skills. Application developers keep using the standard Android permission system they are already familiar with (Section II). They just need to define permissions and policies for the components having security requirements (rather than for the application as a whole, see Section III).

Fine-grained policies. Security policies are defined through a formal language that significantly extends the basic security framework. Also, policies are attached to each component, thereby allowing developers to define local security requirements (Section III).

Policy scope. Our policy language includes scope modalities. For instance, developers can define *local* as well as *global* policies. Moreover, policies can be *sticky* (i.e. affecting other components after an invocation, see Section III).

No runtime errors. Illegal interactions among components are dynamically verified before invocation and cannot generate security violations (Section IV).

No OS customization required. Although feasible, our solution does not require modifications to the Android OS. We present a prototype implementation which enables our security framework on existing Android devices. Users only need to install an extra application and developers just link a library in their code (Section VI).

No user's responsibility. When the user selects a component from a list, she cannot cause policy violations. Application permissions are checked automatically and the user is always prompted with a list of policy-compliant solutions (Section V).

Structure of the paper. In Section II we briefly recall the Android application model and security mechanisms. Section III

DOI 10.1109/CSF.2015.29

describes our enhancement of the Android application framework and Section IV details our SAT-based runtime security policy enforcement framework. In Section V we apply our methodology to a rich case study consisting of an ecosystem of applications. Then, in Section VI we describe a prototype implementation, some technical aspects and future directions, while in Section VII we survey on some related work. Finally, Section VIII concludes the paper.

II. Android Application Model

In this section we briefly describe the Android application framework and the Android permission system. Moreover, we introduce a working example which will be gradually developed in this section and along the paper. In Section V the working example will be enriched to build a complete case study.

Working example. MaplePay.com is a (hypothetical) online payment service. Registered customers can use it to transfer/receive money from/into their account. Apart from money transfers, users can also check current balance and read transfer history. The MaplePay developers want to design and implement a mobile account management application. The main purpose of the application is to allow users to perform payments from their devices. To this aim, the application must include two payment modalities, i.e. *small payments* and *normal payments*. Small payments (a.k.a. micro payments), i.e. those for amounts less or equal to 25$, can be performed without specific authorizations, in a contactless fashion. Normal payments require a specific user authorization. Moreover, MaplePay interacts with the Android environment in two ways. First, it provides an appropriate interface through which other applications can trigger the payment procedure. Secondly, users must be allowed to send their reports to their favourite document viewer.

A. Application Components

Android applications consist of collections of components, where a component can be

- an *activity*, i.e. a foreground component dedicated to the interaction with the user,
- a *service*, i.e. a background component used for asynchronous operations,
- a *content provider*, i.e. a component mediating access to resources, or
- a *broadcast receiver*, i.e. an interface component handling incoming, inter-application messages, called *intents*.

Android packages include a *manifest* file listing the components that the application exports/offers to the platform.

Example 1: Consider the application MaplePay described above. It consists of four activities (namely, MainActivity, LoginActivity, BalanceActivity, PaymentActivity), three broadcast receivers (SmallPaymentReceiver, NormalPaymentReceiver, ContactPaymentReceiver), one service (ConnectionService) and one content provider (HistoryProvider), briefly described in Table I. □

TABLE I. MaplePay components description.

Name	Description
MainActivity	Lists the application functionalities
LoginActivity	Authenticates the user via id + password
BalanceActivity	Shows the current balance
PaymentActivity	Summarizes payment data before confirmation
SmallPaymentReceiver	Receives intents for micro payments
NormalPaymentReceiver	Receives intents for normal payments
ContactPaymentReceiver	Receives intents from contacts applications
ConnectionService	Mediates the access to the MaplePay web service
HistoryProvider	Handles locally stored data

B. Execution Environment

The Android application framework relies on a customized Java virtual machine, namely the *Dalvik* virtual machine (DVM)[1]. VMs are the interpreters of an intermediate language, i.e. the bytecode. Whenever an Android application is launched, its code is loaded by a fresh instance of the VM running on a separated system process. Also, each VM process loads and executes a copy of the runtime support, i.e. Java libraries, and belongs to a distinct Linux user. This approach aims to guarantee isolation and to prevent malicious interactions among applications[2]. For instance, applications cannot access the installation directories of others. Legal interactions among applications rely on a few channels called *inter-process communications* (IPCs).

- *Intents* are used to activate three out of the four Android application components, i.e. activities, services and broadcast receivers. When an intent is fired in broadcast mode and several compatible receivers exist in the system, the user is asked to select the recipient.
- *Content resolving* is used to obtain direct access to a content provider by specifying the identifier, i.e. URI scheme, of the requested resources.
- *Service binding* allows a component to register to an existing service and receive asynchronous notifications from it.

Components appearing in the manifest are candidates for receiving intents and requests.

Example 2: Consider again our working example. From the specifications, we want MaplePay to handle some IPC messages (originated by other apps triggering the payment process). Moreover, the MainActivity of MaplePay can be launched as a standard application. Finally, MaplePay can trigger an external document viewer by firing a specific intent. The actions are schematically reported in Table II (where the column Sys indicates whether the action is natively defined in Android). The architecture of MaplePay is depicted in Figure 1. Arrows denote the interactions among components, i.e. how they invoke each other. Input and output actions are denoted by ○– and ●– respectively. □

C. Application Permissions

Android applications list the permissions they require and declare in their manifest file. Before installation, application

[1]Currently, the DVM is being replaced by a different VM, called ART. Since it is VM-independent, this transition has no effect on our proposal.

[2]Android Application Security, available at: http://source.android.com/tech/security/index.html

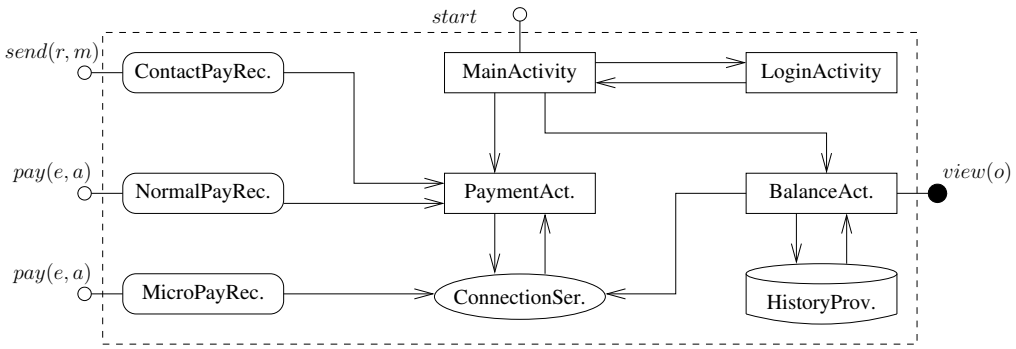


Fig. 1. The MaplePay components and interactions.

TABLE II. ACTIONS HANDLED BY MAPLEPAY APPLICATION.

Action	Abbrev.	Payload	Sys	Behaviour
ACTION_MAIN	$start$	—	✓	Launch an activity
ACTION_SEND	$send$	r, m	✓	Send message m to recipient r
ACTION_VIEW	$view$	o	✓	Trigger a viewer for object o
PAYMENT	pay	e, a		Pay amount a to entity e

permissions are shown to the user who decides whether to accept or reject to grant them all. To illustrate, consider the following fragment of an application manifest:

```
<manifest package="com.example.app">
 <uses-permission android:name="a.p.INTERNET"/>
 <application>...</application>
</manifest>
```

It states that the application requires the `INTERNET`[3] permission. If granted, the permission enables the use of the APIs for network connectivity. It is worth pointing out that *all application components* share the permissions granted to the application even though they do not necessarily use them.

Component access to resources: Optionally, components can define access policies. These constraints are defined in the manifest file through component attributes. The attributes list permissions that the caller must have to perform an invocation. For instance, the following fragment defines a service component which only accepts invocations, e.g., binding, from applications owning the `INTERNET` permission.

```
<service name="com.example.app.service"
         permission="a.p.INTERNET"> ...
</service>
```

At runtime, the IPC framework compares the requests against the access rules of the existing components and returns a list of legal ones. Typically, the user selects the actual component from the list.

According to the official documentation[4] developers use permissions to implement access control rules for IPC interaction with their application. Thus, the considered attacker model consists of one or more (possibly colluded) malicious applications attempting to illegally access sensitive data and functionalities of a target application. In this work we consider the same model.

[3]We use `a.p` as a shorthand for `android.permission`.

[4]http://developer.android.com/training/articles/security-tips.html

III. APPLICATION MODELLING AND ANALYSIS

In this section we present our security framework and we describe how we model components, permissions and security policies.

A. Modelling Components and Configurations

Components invoke each other and suspend their execution until the callee returns the control. Here we model this behaviour and formally define the execution semantics of Android components.

Component Stack and System Configuration: We define the runtime behaviour of a system by introducing the notion of *component stack*[5]. Intuitively, a component stack is a collection of frames, each of them representing a component. A certain component becomes active when the top element of a stack invokes it, i.e. through an IPC message. In that case, the new component is pushed on the stack, i.e. on top of the caller. We define frames and stacks as follows.

Definition 1: A *component frame* is a triple $F = \langle C, P, \Phi \rangle$ where C ($\in \mathcal{C}$) is a component, P is a set of permissions and Φ a set of policies. (In the following we may use C_F, P_F and Φ_F to denote the elements of a frame F.)

Thus, a frame includes the name of the invoked component, its permissions, and the associated policies.

Definition 2: A *component stack* S ($\in \mathbb{S}$) can either be empty, i.e. ε, or consists of a frame F on top of a stack S', i.e. $F :: S'$ (we write $F \in S$ whenever a frame F appears in some location on stack S and S^i for the i-th frame of S).

Intuitively, a stack denotes the invocation trace of the components involved in an execution. Still, multiple executions can coexist in a system, e.g., due to suspended applications which can be resumed or services running in background. Hence, a system configuration $\Sigma = [S_1; S_2; \ldots] \in \mathbb{S}^*$ is a finite sequence of component stacks, being $\emptyset$ the empty one, and Σ^i a shorthand for $S_i \in \Sigma$. (Again, $S \in \Sigma$ means that the configuration Σ includes the stack S.)

[5]Not to be confused with the Android stack specifically defined for browsing among the activities, namely the *activity stack*.

B. Policy Specification and Verification

Policy Language: For highlighting and motivating the features of our policy language we propose the following example.

Example 3: Consider again our working example (see Section II). We informally define some permissions and security policies which arise from the description of the expected behaviour of the application.

- **Access and contacts.** Applications can trigger a payment via the $send$ action only if they received an explicit authorization. Authorizations can be granted by both the user and the MaplePay service. Hence, we introduce permissions *User Authorized* (UAP) and *Application Payment* (APP) for them. The required policy says that a component can access if it has the APP permission or, otherwise, it must own UAP. Moreover, the operation should be allowed if and only if at least one of the "ancestors" of the accessing component has appropriate permissions for reading the contact list (permission READ_CONTACTS, RCP) and the user account information (permission GET_ACCOUNTS, GAP).
- **Payment type.** As said, MaplePay provides two payment modalities, i.e. micro and normal payments. Then, we use *Micro Payment* permission (MPP) and *Normal Payment* permission (NPP). Normal payments must be authorized by users (UAP) while small ones can also be authorized by MaplePay (APP).
- **Eavesdropping mitigation.** Password eavesdropping could be carried out by using device sensors. For instance, microphone and camera could capture the interaction of the user with the keyboard of the device. A possible countermeasure consists in *globally* preventing the existence of components which can either access the microphone or the camera. These operations require permissions RECORD_AUDIO (MIC) and CAMERA (CAM), respectively.
- **Data flow control.** Sensitive information can leave MaplePay when a balance report is visualized in an external viewer. To prevent data leakage, the viewer is required to have no access to either the internet, INTERNET (NET) permission, the external memory, WRITE_SD (WSD) permission, or the bluetooth transmitter, BLUETOOTH (BTT) permission. Furthermore, such policy should persist and be enforced on all the components which could have received a sensitive data flow. Such restriction should not apply to internal MaplePay components.

□

Definition 3: A security policy ϕ, ϕ' is a formula generated by the following syntactic rules.

$$\pi, \pi' ::= \top \mid p_i \mid \neg\pi \mid \pi \wedge \pi' \mid \pi \vee \pi' \mid \pi \rightarrow \pi'$$
$$\phi, \phi' ::= \triangledown\pi \mid \Diamond\pi \mid \Box\pi$$

where p_i are permission names ranging over the finite set $\mathcal{P} = \{p_1, \ldots, p_N\}$.

Thus, a security policy consists of a modal, scope operator followed by a propositional formula. The scope of a policy can be either *direct* ($\triangledown$), *local* ($\Diamond$), or *global* ($\Box$). Intuitively, the scope has to do with the permissions which are considered when evaluating a policy. A direct policy is compared against the permissions of a single frame (i.e. the frame below the policy's one), a local one against the permissions of an entire stack (i.e. the stack where the policy is located), and global ones against all the permissions of the configuration. Besides its scope, i.e. direct, local or global, a policy can also be *sticky*. A sticky policy behaves like a standard policy, i.e. it is satisfied under the same conditions (see below). The difference between simple and sticky policies resides in the way they *propagate* along the frames of a stack (see Section III-C). In the following, we denote that a policy $\Box\pi$, $\Diamond\pi$, $\triangledown\pi$ is sticky, by writing $\blacksquare\pi$, $\blacklozenge\pi$, $\blacktriangledown\pi$ respectively.

Policy Evaluation: The validation of a configuration requires to evaluate all the policies against the appropriate (according to the policy scope) set of permissions. Formally, we start by defining a satisfiability relation $\models$ for the propositional core of our policy language. Briefly, satisfiability of a formula π (against a set of permissions P, in symbols $P \models \pi$) follows the standard rules for propositional logic. For instance, $P \models p$ iff $p \in P$, and $P \models \pi \wedge \pi'$ iff $P \models \pi$ and $P \models \pi'$. We write (i) $F \models \pi$ whenever $P_F \models \pi$, (ii) $S \models \pi$ whenever $\bigcup_{F \in S} P_F \models \pi$ and (iii) $\Sigma \models \pi$ whenever $\bigcup_{S \in \Sigma} \bigcup_{F \in S} P_F \models \pi$. Then we define a semantic function $[\![\phi]\!]_i^j$ mapping each formula to the set of configurations satisfying it. The function is defined as follows: $[\![\triangledown\pi]\!]_i^j = \{\Sigma \mid (\Sigma^i)^{j-1} \models \pi\}$[6], $[\![\Diamond\pi]\!]_i^j = \{\Sigma \mid \Sigma^i \models \pi\}$, and $[\![\Box\pi]\!]_i^j = \{\Sigma \mid \Sigma \models \pi\}$. Slightly abusing the notation, we write $[\![\Phi]\!]_i^j$ for $\bigcap_{\phi \in \Phi} [\![\phi]\!]_i^j$.

Figure 2 graphically represents the evaluation contexts, i.e. the portion of configuration whose permissions must be considered, for direct, local and global policies (in a configuration consisting of three stacks). Roughly, each policy is evaluated against (the union of) the permissions of the highlighted frames. We briefly describe them from left to right. If $(\Sigma^2)^3$ carries a policy $\triangledown\pi$ it must be compared against the permissions appearing in the frame underneath. Instead, in case of $\Diamond\pi$ the union of the permissions appearing in the stack Σ^2 must be considered. Finally, for a policy $\Box\pi$ all the permissions of Σ are taken into account.

We extend the notion of validity to configurations as follows.

Definition 4: Let Σ be a configuration and let $\Phi(i,j)$ be the set of policies associated with the frame $(\Sigma^i)^j$. We say that Σ is *valid* (in symbols $\vdash \Sigma$) if and only if $\Sigma \in [\![\Phi(i,j)]\!]_i^j$ for each stack i and frame j of Σ.

In words, we say that Σ is valid if all of its stacks and frames satisfy the policies they are subject to.

Adequacy of the Policy Language: Intuitively, our policy language supports the definition of security properties involving two dimensions. On the one hand, our three modalities allow a component to discriminate on the *space* of the

[6]Where, for any i, $(\Sigma^i)^0 \models \pi$ iff $\emptyset \models \pi$.

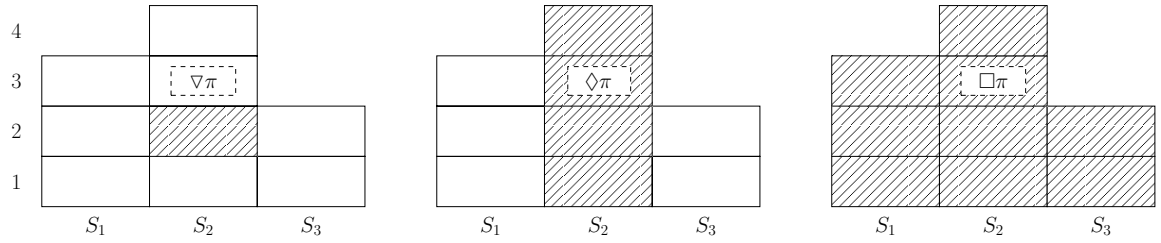


Fig. 2. A representation of the evaluation context for the three policy scopes.

possible configurations. As a matter of fact, they can filter the invocations it receives (∇), the stacks it lays on ($\Diamond$) and the whole system configurations it is involved in ($\Box$). On the other hand, sticky policies affect the changes of configuration over *time*, e.g. they persist after the disposal of the component which initially carried them. In order to compare the expressive power of our proposal against the current Android security support we need to formalize the Android-definable policies.

Definition 5: (**Android-definable policies**) Let R be a set of permissions requested by component C and P be a set of permissions owned by component D, then D can invoke C if and only if $R \subseteq P$.

The augmented expressive power is witnessed by the following statement.

Property 1: The Android-definable policies are a (proper) subset of the policy language of Section III-B (restricted to the ∇ operator).

Example 4: We provide a formalization of the security policies of Example 3.

- **Access and contacts.** Components can start a payment if they have the APP permission. Otherwise, they must own UAP. Thus, the resulting policy is $\nabla(\neg APP \rightarrow UAP)$. Instead, RCP and GAP must be present somewhere in the underlying stack when ContactPayReceiver is invoked. A suitable policy is $\Diamond(RCP \wedge GAP)$.
- **Payment type.** Normal payments can be requested by components having both NPP and UAP permissions. In symbols, $\nabla(NPP \wedge UAP)$. Instead, for micro payments, MPP permission is needed. Also, the caller must have either UAP or APP. The resulting policy is $\nabla(MPP \wedge (UAP \vee APP))$.
- **Eavesdropping mitigation.** We want to prevent LoginActivity from executing when there are components having permissions CAM or MIC. The corresponding policy is $\Box\neg$(CAM $\vee$ MIC).
- **Data flow control.** We introduce a special permission only assigned to internal components, i.e. *Authorized Component* (ACP). The resulting formula is $\blacklozenge(\neg ACP \rightarrow \neg(NET \vee WSD \vee BTT))$.

The architecture of MaplePay enriched with security annotations, i.e. permissions and policies (we use $\emptyset$ and $\top$ for the empty set of permissions and policies, respectively), is reported in Figure 3. As expected, the policies described above

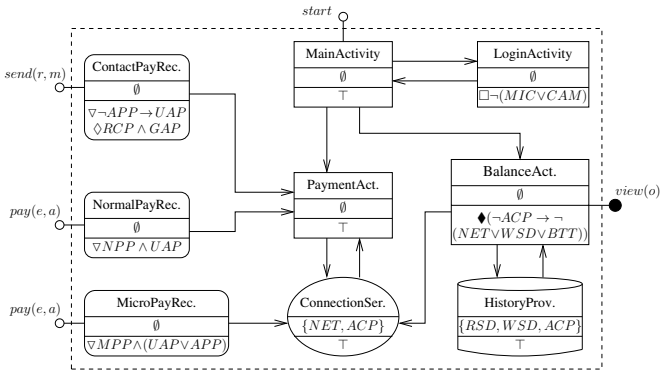


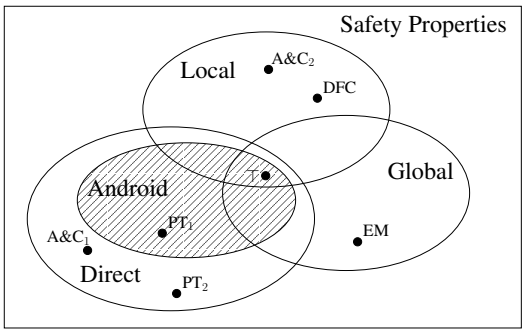
Fig. 3. The MaplePay architecture annotated with permissions and policies.

are applied to the components which need to be guarded. For instance, the policy for password eavesdropping is used in the LoginActivity. Finally, we assign permissions for accessing the internet (NET) to ConnectionService and for reading/writing the external memory (RSD and WSD) to HistoryProvider. Since they are components of MaplePay and they interact with BalanceActivity, they are also provided with the ACP permission. □

As a consequence of the previous dissertation, we can provide a characterization of the expressive power of the policy language. Trivially, our language permits to define both *safety* and *liveness* properties [1] (over the structure of configurations). Nevertheless, here we restrict to safety properties. Since our policy framework relies on dynamic verification (see Section III-C), liveness properties cannot be effectively[7] enforced [17]. Additionally, from Property 1 we know that the Android access rules are a proper subset of our direct policies. Reasonably, other families of security policies of interest may be encoded with our language. For instance, we expect that *stack inspection* [18] can be obtained through the composition of one or more local policies (as it only refers to a single call stack). Since it is not directly related to the Android application security model, we defer its formalization to future research.

The set of policies defined through the three modal operators are independent, i.e. none of them can be obtained as a composition of the other two. The resulting taxonomy is summarized by the following diagram.

[7]Although there exist enforcement models supporting liveness properties (e.g., see [13], [3]), they require to change the runtime behaviour of the applications.



We mapped in the diagram the policies of Example 4, i.e., access and contacts (A&C$_1$ and A&C$_2$), payment type (PT$_1$ and PT$_2$), eavesdropping mitigation (EM) and data flow control (DFC). Also, we use $\top$ to denote the trivial, empty policy.

C. Structural Operational Semantics

Usually, the *structural operational semantics* (SOS) describes the effect of a set of operations over the states of a system. According to Section III-A, a state consists of a configuration, being a finite collection of stacks. Hence, the basic operations that we want to model are insertions and removal of frames, i.e. *push* and *pop*. Each of the component invocations described in Section II results in a push operation. Instead, the pop operation represents the disposal of a component which concluded its life-cycle.

We start by defining few support functions as follows.

$$\Xi(\Phi, \varepsilon) = \varepsilon \qquad \Xi(\Phi, \langle C, P, \Phi' \rangle :: S) = \langle C, P, \Phi' \cup \Phi \rangle :: \Xi(\Phi, S)$$

$$PSticky(\Phi) = \{\blacktriangledown\pi, \blacklozenge\pi, \blacksquare\pi \in \Phi\}$$

Briefly, the function $\Xi(\Phi, S)$ denotes the stack obtained by adding Φ to the policies of every frame. Instead, we use $PSticky(\Phi) = \Phi'$ to denote the subset of all the sticky policies in Φ. Finally, we abbreviate with $CType(C) \in \{Activity, Service, Receiver, Provider\}$ the type of a component C.

The SOS for push and pop operations is formally defined by the rules in Table III.

TABLE III. STRUCTURAL OPERATIONAL SEMANTICS.

$$\frac{CType(C) \neq Service \quad PSticky(\Phi \cup \bigcup_{F_i \in S} \Phi_i) = \Phi' \quad \vdash [\ldots; \Xi(\Phi', \langle C, P, \Phi \rangle :: S); \ldots]}{push(\langle C, P, \Phi \rangle, [\ldots; \underline{S}; \ldots]) \rightarrow [\ldots; \Xi(\Phi', \langle C, P, \Phi \rangle :: S); \ldots]} \text{ (PUSH}_{cmp}\text{)}$$

$$\frac{CType(C) = Service \quad PSticky(\Phi) = \Phi' \quad PSticky(\bigcup_{F_i \in S} \Phi_i) = \Phi'' \quad \vdash [\ldots; \Xi(\Phi', S); \ldots; \Xi(\Phi', \langle C, P, \Phi \rangle :: S); \ldots]}{push(\langle C, P, \Phi \rangle, [\ldots; \underline{S}; \ldots]) \rightarrow [\ldots; \Xi(\Phi', S); \ldots; \Xi(\Phi', \langle C, P, \Phi \rangle :: S); \ldots]} \text{ (PUSH}_{srv}\text{)}$$

$$\frac{CType(C) \neq Service \qquad \vdash [\ldots; S; \ldots]}{pop([\ldots; \underline{\langle C, P, \Phi \rangle :: S}; \ldots]) \rightarrow [\ldots; S; \ldots]} \text{ (POP}_{cmp}\text{)}$$

$$\frac{CType(C) = Service \qquad \vdash [\ldots; \varepsilon; \ldots]}{pop([\ldots; \underline{\langle C, P, \Phi \rangle :: S}; \ldots]) \rightarrow [\ldots; \varepsilon; \ldots]} \text{ (POP}_{srv}\text{)}$$

For the sake of presentation, we use $push(F, [\ldots; \underline{S}; \ldots])$ as a shorthand for $push(k, F, [\ldots; S_{k-1}; S; \ldots])$, i.e. to denote the operation consisting of pushing a frame F onto the k-th element of $\Sigma = [\ldots; S_{k-1}; S; \ldots]$ (and analogously for *pop*). In words, rule (PUSH$_{cmp}$) states that a frame $\langle C, P, \Phi \rangle$ (such that C is not a service) can be pushed on a stack S if the new configuration is valid. Analogously, rule (PUSH$_{srv}$) describes how the push operation works when applied to a frame $\langle C, P, \Phi \rangle$ where C is a service. In that case, the new configuration is obtained by allocating a fresh stack for the service frame. The new stack is initialized so to provide the execution context (that is a copy of the caller's stack) of the service. As expected, the pop operation (POP$_{cmp}$) consists of removing the first frame of a stack. Instead, when a service is popped (rule (POP$_{srv}$)) its execution context is also eliminated.

In the following, we write $\Sigma \xrightarrow{push(F,S)} \Sigma'$ (with $\Sigma = [\ldots; S; \ldots]$) in place of $push(F, [\ldots; \underline{S}; \ldots]) \rightarrow \Sigma'$. Analogously, we write $[\ldots; S; \ldots] \xrightarrow{pop(S)} \Sigma'$ for $pop([\ldots; \underline{S}; \ldots]) \rightarrow \Sigma'$. Given a configuration Σ, a stack $S \in \Sigma$ and a frame F, such that the premises of rules (PUSH) are not satisfied, we say that $push(F, \Sigma)$ *fails* and we indicate it with $\Sigma \xrightarrow{push(F,S)}\!\!\!\!\!\!/\;\;$. Moreover, we write $\Sigma \xrightarrow{pop(S)}\!\!\!\!\!\!/\;\;$, mutatis mutandis.

Example 5: Let consider a configuration consisting of a single stack $S = F :: \varepsilon$ such that $F = \langle C, \{NPP, UAP, MIC\}, \top \rangle$ (where the permissions are the same as in Example 4). Now we imagine that component C invokes the NormalPaymentReceiver of Example 1, i.e. through an intent $pay(e, a)$. This amounts to say that the following operation is performed.

$$push(\langle \text{NormalPayRec.}, \emptyset, \{\triangledown NPP \wedge UAP\} \rangle, [\underline{F :: S}])$$

Since NormalPaymentReceiver is not a service, we apply rule (PUSH$_{cmp}$). Hence, we need to check whether[8]

$$\vdash \overbrace{\left[\begin{array}{c} \langle \text{NormalPayRec.}, \emptyset, \{\triangledown NPP \wedge UAP\} \rangle \\ :: \langle C, \{NPP, UAP, MIC\}, \top \rangle \\ :: \varepsilon \end{array}\right]}^{\Sigma}$$

As only one policy appears in the configuration, it corresponds to verifying that $\Sigma \in [\![\triangledown NPP \wedge UAP]\!]_1^2$ which holds if and only if $\{NPP, UAP, MIC\} \models NPP \wedge UAP$ (trivially valid). Thus, we can write

$$[S] \xrightarrow{push(\langle \text{NormalPayRec.}, \emptyset, \{\triangledown NPP \wedge UAP\} \rangle, S)} \Sigma$$

Applying a similar reasoning we can show that

$$\Sigma \xrightarrow{push(\langle \text{LoginAct.}, \emptyset, \{\square\neg(MIC \vee CAM)\} \rangle, S)}\!\!\!\!\!\!/\;\;$$

□

IV. SAT ENCODING AND SAFE COMPONENT SELECTION

In this section we present our SAT-based framework for the encoding and verification of security policies. Briefly, we generate an instance of the Boolean satisfiability problem (SAT) which admits valid assignments if and only if a given

[8]Notice that we omit function Ξ as $PSticky(\{\triangledown NPP \wedge UAP\}) = \emptyset$.

configuration can evolve without violating any security policies. Moreover, we show how the solution of the SAT problem is used to find policy-compliant components. We define the following concepts:

(a) We say that a stack S has a permission p if there exists at least a frame $F = S^j$ (for some j) having permission p.

(b) We say that a configuration Σ has a permission p if there exists at least a stack $S = \Sigma^i$ (for some i) having permission p.

For brevity, we define *direct policies of* F the set of all the direct policies appearing in Φ_F. Similarly, we define *local policies of* S the set of all the local policies of the frames in the stack S, and *global policies of* Σ the set of all the global policies of all the frames the stacks of Σ.

As stated in Section III, a frame can be safely pushed on (popped from) a stack S if and only if the resulting configuration Σ is valid, i.e., iff $\vdash \Sigma$. In particular with reference to the definition of the function $[\![\phi]\!]_i^j$, this condition is satisfied if the following three sub-conditions are so.

(C1) The direct policies of each frame hold against the permissions of the frame underneath in the stack.
(C2) The local policies of each stack hold against the permissions of the stack.
(C3) The global policies of the configuration hold against the permissions of the configuration.

A. SAT Reduction and Frame Safety

To reduce the problem presented to SAT, preliminarily we introduce the following propositional variables:

- p_i^j to represent the statement: "the frame j in the stack i has permission p". For all i, p_i^0 stands for the propositional constant $false$.
- p_i to represent the statement: "the stack i has permission p"
- p to represent the statement: "the configuration has permission p"

Thus, we consider the conjunction of the following formula, for each frame j in the stack i having permission p:

$$p_i^j \tag{1}$$

Definitions (a) and (b) are enforced by the following propositional formulae, for each permission p. (a) is encoded as follows:

$$\bigvee_{j=1}^{n} p_i^j \leftrightarrow p_i, \tag{2}$$

for each stack i, having n frames, in the configuration. (b) is encoded as follows:

$$\bigvee_{j=1}^{m} p_j \leftrightarrow p, \tag{3}$$

where m is the number of stacks in the configuration.

Let us now consider the encoding of the policies. We denote $\pi(T)$ a propositional formula π, where T is the tuple of propositional variables occurring in π. Given a tuple of permissions $P = p1, \ldots, pn$, we denote $P_i^j = p1_i^j, \ldots, pn_i^j$, and $P_i = p1_i, \ldots, pn_i$. The following formulae encode the statements (C1), (C2) and (C3), respectively. For all i, j s.t. there exists the frame $(\Sigma^i)^j$, we consider the direct policies of each frame, and the propositional variables representing the permissions of the frame underneath in the stack.

$$\bigwedge_{\triangledown\pi\in\Phi(i,j)} \pi(P_i^{j-1}) \tag{4}$$

For each stack i, we consider the formula obtained by the conjunction of the local policies of the stack, and the propositional variables representing the permissions of the stack i:

$$\bigwedge_{F\in\Sigma^i} \bigwedge_{\Diamond\pi\in\Phi_F} \pi(P_i) \tag{5}$$

Then, we consider the conjunction of the global policies of the configuration, and the propositional variables representing the permissions of the configuration:

$$\bigwedge_{S\in\Sigma} \bigwedge_{F\in S} \bigwedge_{\Box\pi\in\Phi_F} \pi(P) \tag{6}$$

The conjunction of the conditions above can be encoded as a single instance of the Boolean satisfiability problem (SAT). Given a configuration Σ, we write $\text{SAT}(\Sigma)$ if the conjunction of formulae (1), (2), (3), (4), (5), and (6) is satisfiable. The following theorem provides an interpretation of the satisfiability of the above formula in terms of configuration validity.

Theorem 1: For all Σ if $\text{SAT}(\Sigma)$ then $\vdash \Sigma$.

In words, Theorem 1 states that we can effectively verify the validity of a configuration by checking the satisfiability of its encoding (as specified by formulae (1), (2), (3), (4), (5), and (6)). Hence, we can exploit SAT solving for the evaluation of the premises of the SOS rules of Table III.

Example 6: Let us consider again the configuration Σ of Example 5. According to Formula (1), the set of permissions of each frame are encoded by the following formula:

$$NPP_1^1 \wedge UAP_1^1 \wedge MIC_1^1 \tag{7}$$

Notice that $NPP_1^2, UAP_1^2, MIC_1^2$ are not constrained here (i.e. they are not assigned a priori either to $true$ or $false$), and thus their truth values can be freely assigned by SAT solvers in such a way to satisfy the policies. The idea underlying this choice is to allow the SAT solver to suggest the user to grant new permissions in case they are necessary for a certain invocation. Then, before proceeding with the invocation of the component, the user can decide whether she agrees to provide the new permissions, or on the contrary she can block it. Still, in order to follow the *least privilege principle*, we use a heuristic provided by the SAT solvers, instructing them to prefer the assignment of the propositional variables to $false$, and thus minimizing the number of fresh permission requests.

Formulae (2) and (3) are as follows:

$$\begin{aligned}(NPP_1^1 \vee NPP_1^2) \leftrightarrow NPP_1 \quad \wedge \quad (UAP_1^1 \vee UAP_1^2) \leftrightarrow UAP_1 \\ \wedge\, (MIC_1^1 \vee MIC_1^2) \leftrightarrow MIC_1\end{aligned} \tag{8}$$

$$NPP_1 \leftrightarrow NPP \quad \wedge \quad UAP_1 \leftrightarrow UAP \quad \wedge \quad MIC_1 \leftrightarrow MIC \tag{9}$$

While, in case of LoginAct. is invoked, no direct policies are present, in case of NormalPayRec., formula (4) is as follows:

$$NPP_1^1 \wedge UAP_1^1 \tag{10}$$

Both in case of NormalPayRec. and LoginAct., no local policies are present. Considering the global policies, in case of LoginAct., Formula (6) is as follows:

$$\neg(MIC \vee CAM) \tag{11}$$

Thus, the component NormalPayRec. can be invoked, because the formula obtained by the conjunction of formulae (7), (8), (9), (10) is trivially satisfiable, e.g., by assigning to $true$ the variables NPP_1^1, UAP_1^1, MIC_1^1, NPP_1, UAP_1, MIC_1, NPP, UAP, MIC, and the other ones to $false$. On the contrary, in case of LoginAct, the conjunction of formulae (7), (8), (9), (11) is not satisfiable and thus this component cannot be invoked. □

Size of the encoding: SAT solvers require a reduction of the formulae above into the *conjunctive normal form* (CNF). The computational complexity and the time required for solving the SAT problem is mainly affected by the number of propositional variables and clauses of the CNF formula. This is the reason why we express here the size of the encoding proposed in terms of the number of variables and clauses, and it will be used for the evaluation in Section VI.

Given a configuration Σ, we indicate with $|\Sigma|_P$, $|\Sigma|_F$, $|\Sigma|_S$, the total number of permissions, frames, and stacks in Σ, respectively. As mentioned above, we introduce propositional variables to represent the permissions of each frame, each stack and the whole configuration. Thus, the total number of propositional variables considered is $|\Sigma|_P(|\Sigma|_F + |\Sigma|_S + 1)$.

As regards the number of clauses, formula (1) generates $|\Sigma|_P$ unary clauses. Apart from the permissions, the complexity of formula (2) is also dependent on the number and the size of the stacks. In particular, the number of clauses for (2) is $|\Sigma|_P(\frac{|\Sigma|_F}{|\Sigma|_S} + 1)|\Sigma|_S = |\Sigma|_P(|\Sigma|_F + |\Sigma|_S)$. While for (3), the number of clauses is $|\Sigma|_P(|\Sigma|_S + 1)$. Considering formulae (4), (5), (6), the number of clauses is dependent on the number and complexity of the policies considered. Let us call $|\Sigma|_\Phi$ the average number of clauses encoding the policies of each frame. The total number of clauses encoding formulae (4), (5), (6) is thus $|\Sigma|_\Phi|\Sigma|_F$.

V. Case Study

We propose an extended case study consisting of a small ecosystem of applications, one of them being MaplePay.

Application Ecosystem: **QRScanner.** Figure 4 shows a *Quick Response* code reader dedicated to fast payment operations. The application consists of a single component, i.e. the MainActivity. The CAM permission is required since QR codes are scanned by taking a picture of them. Moreover, it requests the MPP and UAP permissions for triggering micro payments.

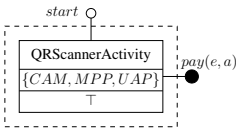


Fig. 4. The QRScanner Application.

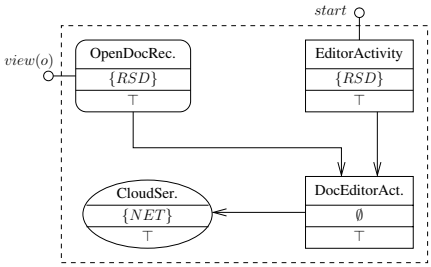


Fig. 5. The FancyEditor Application.

FancyEditor. A document editor with cloud storage support is reported in Figure 5. The user can launch the MainActivity which displays the list of documents residing in the SD card. Otherwise, the OpenDocReceiver can be activated by an action $view(o)$. In both cases, the RSD permission is requested for accessing the SD card. The DocEditorActivity shows the document to the user which can modify it. Eventually, the CloudService component is invoked for saving the modified document on some remote server (thus requesting the NET permission).

TamerReader. Figure 6 depicts a minimal, offline document viewer.

A usage scenario: Let assume the user wants to donate a small amount of money to a non-profit organization. To do that, she uses the QRScanner application (cf. Figure 4) to read a QR code from a brochure. Assuming an empty one, launching the application results in the following configuration (below we use a matrix-like notation and we omit the ε elements).

$$1 \quad \boxed{\langle QRScannerAct., \{CAM, MPP, UAP\}, \top \rangle}$$
$$S_1$$

The application decodes the QR code. As a result, it generates the action $pay(\texttt{donate@nonprofit.org}, 10\$)$. Two receivers can handle this request, i.e., MicroPayReceiver and NormalPayReceiver (cf. Figure 3). The system generates the following specification for MicroPayReceiver.

$$\bigwedge \left\{ \begin{array}{l} \text{(1) } CAM_1^1 \wedge MPP_1^1 \wedge UAP_1^1 \\ \quad \vdots \\ \text{(4) } MPP_1^1 \wedge (UAP_1^1 \vee APP_1^1) \end{array} \right\}$$

Here and in the rest of this section, the numbers in brackets refer to the general formula in Section IV used to generate the corresponding formula. For instance, the formula in the first line above refers to the formula (1) in Section IV, while the last formula encodes the direct policy of MicroPayReceiver, according to the formula (4). The specification for NormalPayReceiver is similar, being the last formula the only difference. Indeed, it is replaced by the clause $NPP_1^1 \wedge UAP_1^1$, encoding the direct policy of NormalPayReceiver.

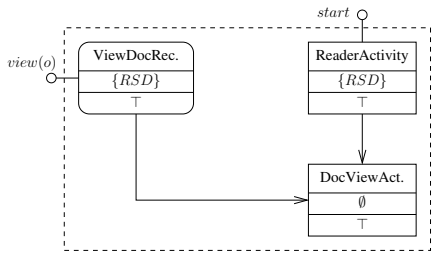
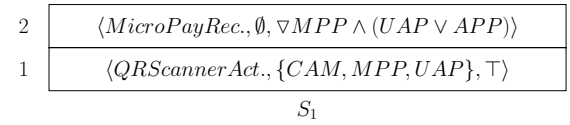


Fig. 6. The TamerReader Application.

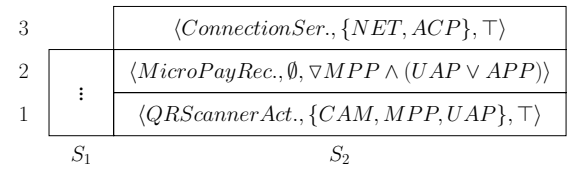
Converting the first set of clauses in CNF format and applying the SAT solver, we obtain that it is satisfied by the assignment

$$\begin{array}{rcl} CAM_1^1, MPP_1^1, UAP_1^1 & \hookleftarrow & 1 \\ APP_1^1 & \hookleftarrow & 0 \end{array}$$

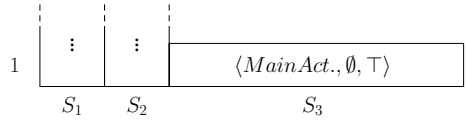
Instead, for the second specification, rather than the last assignment, we get the assignment $NPP_1^1 \hookleftarrow 1$. As a result, the user is prompted with the two choices. However, since selecting the NormalPayReceiver would require the user to grant an extra privilege, i.e., NPP to QRScannerActivity, the selection interface prioritizes the MicroPayReceiver, e.g., by hiding the second element. Assuming the user selects the MicroPayReceiver, the resulting configuration is



Then, the MicroPayReceiver triggers the ConnectionService (cf. Figure 3) for the online payment. Without showing the (trivial) evaluation steps, the configuration becomes



Let suppose now that the user wants to check her online balance. She returns to the home screen and launches the MaplePay MainActivity. Trivially, the new configuration is valid (as it introduces neither permissions nor policies) and is structured as depicted below.



Immediately, MainActivity attempts to invoke the LoginActivity for authenticating the user. As a consequence, the following specification must be verified[9].

[9]For brevity, we omit writing few clauses for S_2.

$$\bigwedge \left\{ \begin{array}{ll} (a) & (1)\ CAM_1^1 \wedge MPP_1^1 \wedge UAP_1^1 \\ & (1)\ CAM_2^1 \wedge MPP_2^1 \wedge UAP_2^1 \wedge NET_2^3 \wedge ACP_2^3 \\ (b) & (2)\ CAM_1^1 \leftrightarrow CAM_1 \\ & (2)\ MPP_1^1 \leftrightarrow MPP_1 \\ & (2)\ UAP_1^1 \leftrightarrow UAP_1 \\ & (2)\ CAM_2^1 \leftrightarrow CAM_2 \\ & \vdots \\ & (2)\ NET_2^3 \leftrightarrow NET_2 \\ & (2)\ ACP_2^3 \leftrightarrow ACP_2 \\ (c) & (3)\ (CAM_1 \vee CAM_2) \leftrightarrow CAM \\ & (3)\ (MPP_1 \vee MPP_2) \leftrightarrow MPP \\ & (3)\ (UAP_1 \vee UAP_2) \leftrightarrow UAP \\ & (3)\ NET_2 \leftrightarrow NET \\ & (3)\ ACP_2 \leftrightarrow ACP \\ & (4)\ MPP_1^1 \wedge (UAP_1^1 \vee APP_1^1) \\ & (4)\ MPP_2^1 \wedge (UAP_2^1 \vee APP_2^1) \\ (d) & (6)\ \neg(MIC \vee CAM) \end{array} \right\}$$

However, the specification is not satisfiable. As a matter of fact, clauses (a), (b) and (c) can be satisfied only by assignments such that $CAM \hookleftarrow 1$, whereas clause (d) is satisfied if and only if $CAM \hookleftarrow 0$. Hence, before proceeding, the system automatically disposes[10] the inactive stacks S_1 and S_2 by popping their frames. When only S_3 appears in the configuration, the LoginActivity can be safely invoked. After the user successfully logs in, the LoginActivity is popped and the BalanceActivity invoked. The specification arising from such invocation is simply:

$$(5)\ \neg ACP_3 \rightarrow \neg(NET_3 \vee WSD_3 \vee BTT_3)$$

Which is trivially satisfied by the assignment:

$$ACP_3, NET_3, WSD_3, BTT_3 \quad \hookleftarrow \quad 0$$

Finally, let consider the case in which the user wants to open a report using one of the document viewer, i.e., either FancyEditor or TamerReader. In that case, the BalanceActivity component fires a corresponding action, e.g., $view(\texttt{file://tmp/report.pdf})$. The system reacts by checking whether one of the existing receivers, i.e., OpenDocReceiver (cf. Figure 5) and ViewDocReceiver (cf. Figure 6), can safely handle the request. For both of them, the resulting specification is

$$\bigwedge \left\{ \begin{array}{l} (1)\ RSD_3^3 \\ \vdots \\ (5)\ \neg ACP_3 \rightarrow \neg(NET_3 \vee WSD_3 \vee BTT_3) \end{array} \right\}$$

The satisfying assignment is

$$\begin{array}{rcl} RSD_3^3 & \hookleftarrow & 1 \\ ACP_3, NET_3, WSD_3, BTT_3 & \hookleftarrow & 0 \end{array}$$

Let assume the user selects the OpenDocReceiver. In this case, the configuration evolves to

[10]Recall that, according to the Android specification, disposing a component is not an extraordinary measure. In fact it is a standard OS operation, e.g., used for deallocating locked resources.

Then, the OpenDocReceiver invokes the DocEditorActivity. Clearly, the invocation succeeds (since no new permissions/policies are involved) so leading to the configuration

4	$\langle DocEditorAct., \emptyset, \blacklozenge(\neg ACP \rightarrow \neg(NET \vee WSD \vee BTT))\rangle$
3	$\langle OpenDocRec., \{RSD\}, \blacklozenge(\neg ACP \rightarrow \neg(NET \vee WSD \vee BTT))\rangle$
2	$\langle BalanceAct., \emptyset, \blacklozenge(\neg ACP \rightarrow \neg(NET \vee WSD \vee BTT))\rangle$
1	$\langle MainAct., \emptyset, \blacklozenge(\neg ACP \rightarrow \neg(NET \vee WSD \vee BTT))\rangle$

S_3

Finally, the DocEditorActivity attempts to activate the CloudService component. The resulting specification is

$$\bigwedge \left\{ \begin{array}{l} (1)\ RSD_3^3 \wedge RSD_4^3 \wedge NET_4^5 \\ (2)\ NET_4^5 \leftrightarrow NET_4 \\ (2)\ ACP_4^5 \leftrightarrow ACP_4 \\ \quad \vdots \\ (5)\ \neg ACP_3 \rightarrow \neg(NET_3 \vee WSD_3 \vee BTT_3) \\ (5)\ \neg ACP_4 \rightarrow \neg(NET_4 \vee WSD_4 \vee BTT_4) \end{array} \right\}$$

which is satisfied by the assignment

$$\begin{array}{rcl} RSD_3^3, RSD_4^3, NET_4^5, NET_4, ACP_4^5, ACP_4 & \leftarrow\!\shortmid & 1 \\ ACP_3, NET_3, WSD_3, BTT_3, WSD_4, BTT_4 & \leftarrow\!\shortmid & 0 \end{array}$$

However, the solution assigns 1 to ACP_4^5. This amounts to say that a new permission, i.e., ACP, must be granted by the user to the CloudService component in stack 4. Hence, the system prompts the user who may as well decide to block the invocation.

VI. Implementation and Discussion

In this section we introduce the prototype implementation of our framework and we discuss some technical aspects. Moreover, we outline some possible future directions which are currently under evaluation.

A. Prototype

Our approach has been implemented and a working prototype, called *Safe Component Provider* (SCP), is publicly available[11]. The prototype includes two elements: SCPcore and SCPlib. Users install the SCPcore application for enabling their device with our security framework. Instead, developers include the SCPlib in their applications. SCPlib contains classes which resemble and mimic the standard Android IPC library, e.g., it defines a ScpIntent to be used in place of Intent. When an application including the SCPlib is installed on a device mounting SCPcore, its components are registered together with their permissions and policies. Below we outline the differences between the classical Android IPC mechanism (Figure 7) and our prototype (Figure 8). Briefly, application A starts an interaction with an external component through an Intent, that is a class provided by the runtime support of the Android VM (e.g. Dalvik). As a result of firing an intent, a kernel module, called Binder, is invoked to find a suitable component. Finally, the Binder locates a component D provided by application B and activates it through the runtime support of the VM running it.

[11]https://github.com/SCPTeam/Safe-Component-Provider

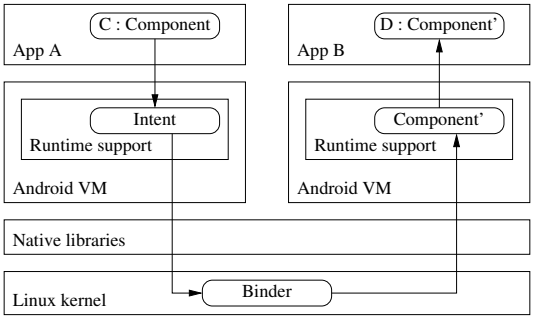


Fig. 7. Android IPC component invocation.

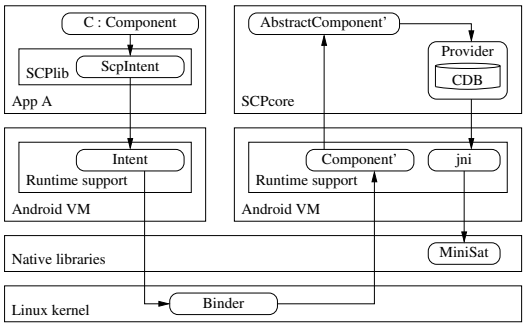


Fig. 8. SCP component invocation.

Instead, SCP-enabled applications work as follows. When an interaction is requested, an ScpIntent is activated. This class converts the request into a standard Intent, but it replaces the requested component with an abstract one, only provided by the SCPcore application. In this way, the Binder is forced to dispatch the intent to SCPcore. When triggered, SCPcore queries its components database (CDB) and retrieves a list of available components. Then, their permissions and policies are converted into CNF specifications and submitted to the SAT-solver for deciding the actual receiver of the intent[12]. SCPcore relies on MiniSat [8], a minimalistic, state-of-the-art solver.

Comparing the two pictures, we can notice that modifications only appear at the top, application level. Since, the prototype requires no modifications of the lower levels, e.g., OS kernel, it can be installed on any existing device.

Performances: For the time being, our prototype is still under evaluation as a systematic analysis requires to set up realistic usage patterns (possibly obtained from real users) and security policies. We mounted our prototype on a Android 4.4.4 LG Nexus 7 device, mounting a 1.2 GHz, quad-core ARM processor with 1 GB of RAM. Figure 9 reports the execution times for the steps of our working example.

Each column reports the time required for generating the list of candidate components of one of the steps presented in Section V (from the beginning to the first request of LoginActivity). The columns represent the invocation of QRScannerAct. (QRA), MicroPayRec. (MPR), ConnectionSer. (CS), MainActivity (MA) and LoginActivity (LA). The components

[12]Notice that few irrelevant details, e.g., user selection, have been omitted.

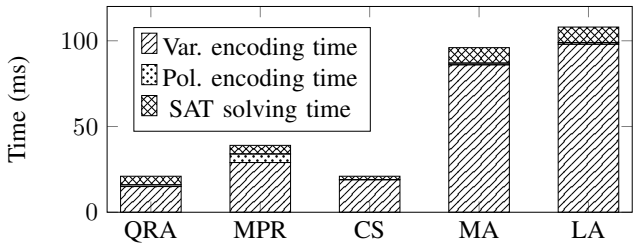


Fig. 9. Case study execution times.

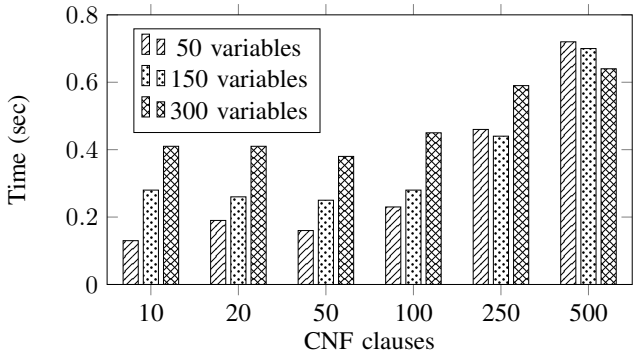


Fig. 10. Benchmark execution times.

list generation time includes (i) variables and (ii) policies encoding plus (iii) solving time. We recall that every step performs one instance of our encoding and solving procedure for each available component of the requested type (e.g., QRA verifies all the 9 activities of our case study).

We also used benchmark SAT problems for checking the performances of MiniSat on a real mobile device. Since the SAT-solving process is central, showing that it can scale over reasonably big specifications is fundamental for the feasibility of our approach. Hence, we used Tough SAT[13] for generating a heterogeneous set of CNF specifications. Then, we submitted them to MiniSat running on our testing device. The results are depicted in Figure 10.

Although preliminary, the results appear to be promising as, even with large specifications, MiniSat takes less then 0.7 seconds. To provide an intuition about the size of actual specifications, consider that the larger formula of our working example consists of 23 variables and 43 clauses.

Manifest and compatibility: In our model application policies and permissions are defined through an extension of the manifest file. Also, applications using our security features need to be compiled with a special runtime support library, i.e., SCPlib. Hence, a platform running our system experiences a separation between the two classes of applications: those enabled with our security mechanism and the standard ones. In practice, this results in an actual isolation of secured applications that can neither invoke nor being invoked by normal applications. Such isolation can be effectively enforced by validating the apps bytecode (see below).

[13]https://toughsat.appspot.com/

Bytecode validation: Clearly, our approach relies on the fact that applications cannot use both standard intents and SCP ones. Currently, our prototype does not include code checking steps, but few alternatives exist. A viable approach for ensuring the isolation is through a reference monitor (e.g., see [17]). Nevertheless, runtime monitoring generates an extra computational effort. To avoid this, an alternative solution consists of a class loader which verifies the class files before injecting them in the execution environment. The class loader just needs to (syntactically) check that no illegal invocations appear in the application code. A similar verification can also be carried out at installation time.

Reflection: Another open issue is Java reflection. Briefly, reflection is a powerful technique for dynamically accessing and modifying Java elements like classes and methods. Many analysis techniques exploited in the implementation of security enforcement frameworks simply do not cope with it (e.g., see [14]). Hence, a common solution, also viable for SCP, is to simply disable reflection. Nevertheless, code modified through reflection does not avoid permission checks in Android. Thus, our approach still applies as far as reflection does not introduce the IPC/SCP primitives where not allowed. In this case, a reference monitor could dynamically check how these primitives are used, while a class loader would be ineffective (as reflected code is not reloaded).

Dynamic permissions: Android also provides APIs for dynamically assigning and revoking access permissions. An application owning a resource `res` can invoke `grantUriPermission(app_id, res)` and `revokeUriPermission(app_id, res)` to modify the access rights of `app_id`. Although not currently implemented, a similar behaviour could be obtained in our approach by allowing applications to edit their own entries in the components database. In this way, according to runtime values, an application could decide to enforce a different policy on one or more of its components. Such improvement would require to reconsider our operational semantics, but not the verification process.

B. Policy granularity

Below we list few examples of security policies that Android cannot enforce and we show how to encode them.

Confused deputy. An application A owns some security-relevant resources and allows the access only to applications having permission p. Application B (legally) owns p and invokes A. Then, the unprivileged application X exploits an interaction with B to access to the resources of A. With our support A can declare a sticky policy $\blacktriangledown p$ which guarantees all the future invocations of B to request permission p.

Application collusion. An application X owns permission p needed to read data owned by A. However, A releases the data only if the caller has no internet access (`NET` permission). Since X has no other permissions but p, it legally accesses. However, after collecting the data, X sends it to application Y, having `NET` (thus violating A's requirement). In our framework, the policy of A can be expressed by $\Box p \rightarrow \neg NET$.

Permission re-delegation. In [10] the authors describe the permission re-delegation problem and propose an enforcement

mechanism for preventing it. Briefly, when at time t a component D invokes C, the permissions of C are updated according to the formula $P_t(C) = P_{t-1}(C) \cap P_{t-1}(D)$. In words, the permissions of C are restricted (through set intersection) before each IPC interaction. Clearly, removing permissions can lead to runtime errors. Although this cannot happen with our framework, a similar policy is to prevent interactions with underprivileged components. A possible encoding is given by the formula $\blacktriangledown \bigwedge_{p \in P_C} p$, i.e., C can only be invoked by components owning a superset of its permissions. By leveraging a sticky policy we can apply a restriction on future interactions without revoking previously granted permissions.

C. Future developments

Securing SmartCampus Applications: The SmartCampus[14] platform offers a collection of Android applications aiming at enhancing the quality of services provided by the city of Trento to citizens and students. Each (open source) application provides access to a specific service, e.g., public transportation schedule and event organization news. Moreover, new services arise from the interaction of applications, e.g., users can plan a bus trip to reach an event location. Developers contribute by publishing their own applications. We plan to apply our approach to the SmartCampus ecosystem by considering the specific security requirements of each application.

Policy language extension: A further improvement would be extending the policy language to first-order logic. As a matter of fact, permissions could be considered as predicates (rather than propositions) over data objects. In this way we could model more comprehensive security policies also including data flow and resource usage. Our approach could be adapted by replacing SAT-solving with SMT-solving [2]. Clearly, the extended expressiveness would impact on the runtime performances and new experiments would be necessary to grant the sustainability of the new language.

VII. Related Work

Several authors investigated the Android application security framework, often exposing its limitations. Both Egners et al. [6] and Felt et al. [9] discuss the potential misuse of Android permissions. The former, shows that Android permissions can hide unexpected vulnerabilities that can be easily exploited by properly-crafted malicious applications. In the latter, the authors empirically show that Android apps are generally over-privileged and their interactions can lead to privilege escalation attacks (i.e. an app can trigger another app exploiting its permissions). These analyses do not charge the Android permissions with the responsibility of the identified issues; on the contrary, they show that many security concerns derive from the misunderstanding of the role and purpose of the permissions by both developers and users.

Nonetheless, many proposals have been put forward to enhance the native set of permissions. In [16] the authors propose a novel kind of app-centric permissions that are suited to the needs of single apps. App-centric permissions are built upon basic Android ones but require to instrument apps. In [12], authors propose a set of fine-grained permissions and present RefineDroid, a tool that automatically infers them from apps by means of static analysis techniques. Also the authors of [11] propose a refinement of the existing Android permissions. In particular, they focus on a subset of critical permissions and analyse the most common pattern involving them. As a result, they redefine a subset of the existing permissions. Although more permissions can ideally provide a better security characterization of the applications, they are not sufficient to solve permission-related security issues without relying on a well-defined policy language for supporting the definition of security properties that the Android system is expected to meet. To this aim, more structural approaches have been proposed to reconsider the Android security facilities rather than the set of permissions only. For instance, Felt et al. [10] propose IPC Inspection. Briefly, their framework restricts the actual permissions of an application whenever it receives an IPC stimulus. The restriction amounts to the intersection of its current permissions and those of the invoker. Clearly, dynamic restrictions of the permissions can cause runtime errors which cannot happen with our proposal. Indeed, in our framework, when an invocation does not fulfil the requirements, it is simply avoided. In [5] an approach to detect malicious operations performed by apps without the user consent is proposed. The idea is to build up Permission Event Graphs (PEG) aimed at keeping track of the order in which app permissions are exploited by applications. PEGs are analysed by Pegasus, a model checking-based tool aimed at verifying whether a PEG complies with a given security property, expressed as a boolean formula. Bauer et al. [4] present a runtime verification mechanism allowing to enforce temporal properties, expressed in a LTL-like syntax, over Android applications execution. Similarly, in [15] the authors propose Apex as a policy enforcement system for Android applications allowing the user to define her own security policies. Unlike our proposal, these approaches are meant to replace the current Android runtime security support. Thus, their implementation requires a substantial modification of the operating system. Enforcing permission at component-level is a very recent proposal [19]. Here, authors propose a solution allowing both users and developers to selectively assign permissions to single app components. However, such solution does not support the definition and enforcement of security policies on sets of permissions, we do in this paper.

VIII. Conclusion

In this paper we presented an extension of the Android permission system and security framework. Our proposal is non-invasive as it relies on existing resources and can be implemented on top of standard Android distributions. Moreover, we enhanced the basic security support with an effective policy enforcement mechanism. Under these settings, developers can specify fine-grained security policies that will not be violated at runtime. We released a prototype implementation and we carried out some preliminary experiments which appear to confirm the feasibility of our approach.

Acknowledgment

This work has partially been supported by the PRIN project "Security Horizons" (no. 2010XSEMLC) funded by MIUR, and by the Activity "FIDES" (no. 15383) funded by EIT ICT Labs.

[14] www.smartcampuslab.it

REFERENCES

[1] Bowen Alpern and Fred B. Schneider. Defining Liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[2] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[3] D. Basin, V. Jugé, F. Klaedtke, and E. Zălinescu. Enforceable security policies revisited. *ACM Trans. Inf. Syst. Secur.*, 16(1):3:1–3:26, June 2013.

[4] A. Bauer, J. Küster, and G. Vegliach. Runtime Verification Meets Android Security. In Alwyn Goodloe and Suzette Person, editors, *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 174–180. Springer, 2012.

[5] K. Z. Chen, N. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. Magrino, E. X. Wu, M. Rinard, and D. Song. Contextual Policy Enforcement in Android Applications with Permission Event Graphs. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'13)*, San Diego, USA, 2013.

[6] A Egners, U. Meyer, and B. Marschollek. Messing with Android's Permission Model. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*, pages 505–514, June 2012.

[7] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[8] N. Eén and N. Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[9] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, 2011.

[10] Adrienne Porter Felt, Steven Hanna, Erika Chin, Helen J. Wang, and Er Moshchuk. Permission re-delegation: Attacks and defenses. In *In 20th Usenix Security Symposium*, 2011.

[11] Elli Fragkaki, Lujo Bauer, Limin Jia, and David Swasey. Modeling and Enhancing Android's Permission System. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Computer Security – ESORICS 2012*, volume 7459 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin Heidelberg, 2012.

[12] J. Jeon, K. Micinski, J. Vaughan, A. Fogel, N. Reddy, J. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Application. In *Proceedings of 2nd Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2013.

[13] J. Ligatti, L. Bauer, and D. Walker. Edit automata: enforcement mechanisms for run-time security policies. *Inter. J. of Information Security*, 4(1-2):2–16, 2005.

[14] Benjamin Livshits, John Whaley, and MonicaS. Lam. Reflection Analysis for Java. In Kwangkeun Yi, editor, *Programming Languages and Systems*, volume 3780 of *Lecture Notes in Computer Science*, pages 139–160. Springer Berlin Heidelberg, 2005.

[15] M. Nauman, S. Khan, and X. Zhang. 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, pages 328–332, New York, NY, USA, 2010. ACM.

[16] N. Reddy, J. Jeon, J. Vaughan, J. Foster, and T Millstein. Application-centric security policies on unmodified Android. Technical Report 10017, UCLA Computer Science Department, 2011.

[17] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[18] Dan S. Wallach and Edward W. Felten. Understanding Java Stack Inspection. In *Security and Privacy - 1998 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 3-6, 1998*, pages 52–63, 1998.

[19] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du. Compac: Enforce Component-level Access Control in Android. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, pages 25–36, New York, NY, USA, 2014. ACM.

APPENDIX

Property 1: The policy language of Section III-B (strictly) includes all the Android-definable policies.

Proof. We show that all the Android-definable policies can be encoded in our language. First we notice that any Android-definable policy can be expressed by means of the policy $\phi = \triangledown \bigwedge_{p \in R} p$. As a matter of fact, $P \models \phi$ if and only if $\forall p \in R.P \models p$. By definition, this is equivalent to $\forall p \in R.p \in P$ which trivially corresponds to $R \subseteq P$.

We have strict inclusion since some policies cannot be represented as a finite set of permissions R. For instance, $\phi = \Diamond p \vee q$ (for some p and q) is satisfied by both $P = p$ and $P' = q$. Hence the only possible R for this case is $\emptyset$ which trivially does not represent ϕ. □

Lemma 1: $P \models \pi$ if and only if $\text{SAT}(\pi \wedge \bigwedge_{p \in P} p)$.

Proof. Trivially, $P \models \pi$ if and only if there exists an assignment from variables to truth values $\sigma : \mathcal{P} \rightarrow \{0, 1\}$ satisfying both π and $\bigwedge_{p \in P} p$. □

Theorem 1: For all Σ if $\text{SAT}(\Sigma)$ then $\vdash \Sigma$.

Proof. We proceed by contradiction. Hence, we assume that there exists a Σ such that $\text{SAT}(\Sigma)$ and $\nvdash \Sigma$. By definition, $\nvdash \Sigma$ implies $\exists i, j.\Sigma \notin [\![\Phi(i, j)]\!]_i^j$ or, unfolding the notation $\exists \phi, i, j$ s.t. $\phi \in \Phi(i, j) \wedge \Sigma \notin [\![\phi]\!]_i^j$. Depending on the modality of ϕ, we can have three branches.

1) if $\phi = \triangledown \pi$. In this case we have $\Sigma \notin [\![\triangledown \pi]\!]_i^j$ which implies $F \nvdash \pi$ (where $F = (\Sigma^i)^{j-1}$), and $P_F \not\models \pi$. Applying Lemma 1, this entails $\neg\ \text{SAT}(\pi \wedge \bigwedge_{p \in P_F} p)$. By a straightforward variable renaming, this also implies $\neg\ \text{SAT}(\pi(P_i^{j-1}) \wedge \bigwedge_{p \in P_F} p_i^{j-1})$. Hence, we have that also the formula (1) $\wedge$ (4) is unsatisfiable and therefore (by $\wedge$ introduction), we find that $\neg \text{SAT}(\Sigma)$, which contradicts the hypothesis.
2) if $\phi = \Diamond \pi$. Here $\Sigma \notin [\![\Diamond \pi]\!]_i^j$ which implies $\bigcup_{F \in S_i} P_F \not\models \pi$ (where $S_i = \Sigma^i$). By Lemma 1 this holds iff $\neg\ \text{SAT}(\pi \wedge \bigwedge_{F \in S_i} \bigwedge_{p \in P_F} p)$. Again, by variable renaming, this holds iff $\neg\ \text{SAT}(\pi(P_i) \wedge \bigwedge_{F \in S_i} \bigwedge_{p \in P_F} p_i)$. However, from $\text{SAT}(\Sigma)$ and (5) we know that $\text{SAT}(\pi(P_i))$. Hence, there must be at least a $\bar{p}_i$ such that $\pi(P_i)$ is only satisfied by assignments mapping $\bar{p}_i$ to 0. By construction, $\bar{p}$ appears in stack S_i. Let assume $\bar{p} \in S_i^k$. By (1) (and $\text{SAT}(\Sigma)$), we know that $\bar{p}_i^k$ must be set to 1. Nevertheless, by (2) also $\bar{p}_i$ must evaluate to 1, that is the contradiction.
3) if $\phi = \Box \pi$. We apply a similar argument. From $\Sigma \notin [\![\Box \pi]\!]_i^j$ we infer that $\bigcup_{S \in \Sigma} \bigcup_{F \in S} P_F \not\models \pi$. By Lemma 1 this happens if and only if $\neg\ \text{SAT}(\pi \wedge \bigwedge_{S \in \Sigma} \bigwedge_{F \in S} \bigwedge_{p \in P_F} p)$. From $\text{SAT}(\Sigma)$ and (6) we know that $\text{SAT}(\pi(P))$. Again, there must be at least a $\bar{p}$ (in one of the frames of one of the stacks) such that $\pi(P)$ is only satisfied by assignments mapping $\bar{p}$ to 0. Let assume $\bar{p} \in (\Sigma^k)^h$. By (1) (and $\text{SAT}(\Sigma)$), we know that $\bar{p}_k^h$ must be set to 1. Consequently, by (2) also $\bar{p}_k$ must evaluate to 1. Finally, by (3) this implies that also $\bar{p}$ evaluates to 1 and we conclude.

□