Noname manuscript No.
(will be inserted by the editor)

Formal Modeling and Automatic Enforcement of *Bring Your Own Device* Policies

Alessandro Armando · Gabriele Costa · Alessio Merlo · Luca Verderame

Received: date / Accepted: date

Abstract The emerging Bring Your Own Device (BYOD) paradigm is pushing the adoption of employees' personal mobile devices (e.g., smartphones and tablets) inside organizations for professional usage. However, allowing private, general purpose devices to interact with proprietary, possibly critical infrastructures enables obvious threats. Unfortunately, current mobile OSes do not seem to provide adequate security support for dealing with them.

In this paper we present a formal modeling and assessment of the security of mobile applications. In particular, we propose a security framework for verifying and enforcing BYOD security policies on Android devices. Interestingly, our approach is non-invasive and only requires minor platform modifications at application level. Finally, we provide empirical evidence of the practical feasibility of the approach by means of a prototype which we used to validate a set of real Android applications.

Keywords Bring Your Own Device · History Expressions · Hennessy-Milner Logic · Type and Effect Systems · Partial Model Checking · Android

A. Armando
DIBRIS - University of Genova and FBK-IRST, Trento
Tel.: +39 010 353 2216
E-mail: alessandro.armando@unige.it

G. Costa
DIBRIS - University of Genova
Tel.: +39 010 353 6545
E-mail: gabriele.costa@unige.it

A. Merlo
E-Campus University and University of Genova
Tel.: +39 010 353 2344
E-mail: alessio.merlo@unige.it

L. Verderame
DIBRIS - University of Genova
Tel.: +39 010 353 6545
E-mail: luca.verderame@unige.it

1 Introduction

The pervasive spread of mobile devices is leading to a paradigm shift in the use and the management of terminal devices inside organizations. More specifically, there is a growing pressure to let devices owned by individual, private users get access to the corporate network, resources, and services. Clearly, most of these contexts should include rules defining the legal behavior of the devices. This calls for the definition and enforcement of so-called "Bring Your Own Device" (BYOD) policies.[1] Intuitively, two stakeholders are involved in the BYOD policy definition and enforcement: the device user/owner and the organization. From the security perspective, the organization is the central authority responsible for defining the policies that users accept when registering their devices to the BYOD environment. On the one hand, BYOD policies must counter the threats posed by malicious applications which could jeopardize the confidentiality and/or the integrity of corporate data. On the other hand, they should allow the users to customize their devices to their own needs and preferences by downloading and installing policy-compliant applications from public stores. The security mechanisms offered by the most popular mobile operating systems (e.g. the Android *sandbox*) only provide limited protection against this kind of threats and do not support the level of protection needed in corporate environments. To illustrate, consider the scenario where the devices must satisfy a basic security policy stating that "no application should access the internet after reading/writing the local file system". When a user, willing to comply with this policy, installs a new application, he checks (through the Android application *manifest*) what permissions it requests and refrains from installing applications that request both

[1] http://www.cmswire.com/cms/information-management/gartner-enterprises-must-develop-bringyourowndevice-byod-policies-017148.php

privileges (although he cannot be sure whether their runtime behavior violates the policy). More interestingly, the user cannot possibly take an informed decision about applications that require only one of the two permissions since no information about possible interactions between the new application and those already installed can be drawn from the manifest. As a consequence, the user might avoid installing applications that require the simultaneous installation of conflicting permissions (e.g. those using the file system and the network), without preventing the installation of maliciously interacting, namely *colluding*, ones. Colluding applications team up to overcome the limitation posed by their own set of permissions. For instance, let us consider two applications A_1 and A_2 where only A_1 has the permission to access the contact list: A_2 could trigger A_1 to retrieve some contacts information in its stead. Colluding applications have been proved to be a serious and concrete threat for the Android security model (e.g., see [12]).

In this paper we assess the issue of providing formal security guarantees over the composition of mobile applications. This result is achieved through a formal framework modeling both applications behavior and security policies. Our security framework ensures that only applications complying with a given BYOD security policy are installed on the BYOD-enabled devices. In detail, our framework has been designed to meet the following goals:

1. *Transparency*. The user experience and usage patterns must not be affected. For instance, no (or minimal) customization of the OS interfaces and workflow must be implemented.
2. *Automatic verification.* The verification of the compliance of applications against the BYOD policy must be carried out in a fully automatic way, i.e., without user intervention.
3. *Protection against colluding applications*. Before installation, an application is validated by keeping into account the whole configuration, i.e., all the installed applications, of the target device.
4. *Device configuration safety.* Changes to the configuration of a device which could lead to violations of the BYOD security policy should be prevented. Whenever the user's behavior causes irreparable conflicts with the policy, the device should be isolated from the BYOD environment and its resources.

Our approach is based on the following intuition. Let us call φ the BYOD policy that the device is expected to comply with. The security state of the device is represented by a customized security policy φ' obtained by instantiating φ according to the applications currently installed. This process can be carried out through partial model checking [3]. Checking whether an application can be installed on a device (whose current security state is represented by the policy φ') is done by *(i)* processing the code of the application and generating a model (called *history expression*) representing an over-approximation of the possible execution traces of the application and then by *(ii)* determining whether the model satisfies φ'. Moreover, in this paper we extend the original framework, introduced in [4, 6], in the following ways:

1. We enrich our policy specification framework by adopting the Hennessy-Milner logic (HML) [19] with recursion. This improvement substantially increases the expressive power of the policy language by allowing one to define properties over potentially infinite computations.
2. We provide a proof system in the style of [19] and we prove that it is sound and complete. Also, we apply partial model checking (PMC) [3] to HML, i.e., we specialize a security policy through the partial evaluation of the model of the system it is defined for.
3. We provide empirical evidence of the feasibility of the proposed approach by testing 261 free, most downloaded Android applications taken from Google Play by means of a prototype implementation of our security framework. The outcome of our experimental activity is also used to evaluate our approach and to identify possible improvements and criticalities.

The paper is structured as follows. In the next section we describe the programming framework. In Section 3 we present a type and effect system that we use to infer *history expressions* from the actual code of the applications. In Section 4 we present Hennessy-Milner logic with recursion and we apply it to security verification and policy transformation via partial model checking. In Section 5 we present *BYODroid*, a prototype implementation of the proposed security framework. Furthermore, we describe the testing setup and we analyze the experimental results. In Section 6 we discuss the related work and we conclude in Section 7 with some final remarks and future developments.

2 Programming Framework

In this section we present our programming model. Mainly, we propose a set of primitives denoting Android-specific functionalities. In order to work with a pure framework, we propose an extension of Featherweight Java (FJ) [16], i.e., a minimal, functional language having a Java-like syntax. Hence, we enrich FJ with operators supporting the definition of application-to-application communications *à la* Android, by means of intent generation and handling. An intent is a messaging object used to trigger a computation performed by another application component. For instance, they can be used to start an activity or a service.

The sender can either specify the addressee of the message (*explicit intent*) or declare the generic action to perform (*implicit intent*). In the latter case, the Android system tries to find out the appropriate receiver by comparing the intent format with the existing components. If there exist more than one suitable receivers, the end-user is required to make a choice.

Also, we include in our syntax operators denoting security-relevant, system accesses, also called system actions. A system access represents the way applications interact with the underlying platform and use its resources. The syntax of the language is given in Table 1.

Table 1 Syntax of applications and components

L ::= **class** C **extends** C' $\{\overline{D}\,\overline{f};\ K\,\overline{M}\}$	Class
K ::= $C(\overline{D}\,\overline{x})$ {**super**($\overline{x}$); **this**.$\overline{f}$:=$\overline{x}$;}	Constructor
M ::= $C\,m(Dx)$ {**return** E;}	Method
E ::= **null** | u | x | **new** $C(\overline{E})$ | $E.f$ | $E.m(E')$ | **system**$_\sigma E$ | **icast**E | **ecast**CE | $\mathrm{I}_\alpha(E)$ | E.data | $E;E'$ | $(C)E$ | **if**$(E$=$E')\{E_{tt}\}$ **else** $\{E_{ff}\}$ | **thread**$\{E\}$ **in** $\{E'\}$	Expression

A declaration of a class C, possibly extending C', contains a (possibly empty) list of typed fields $\overline{f}$, a constructor K, and a set of methods $\overline{M}$. A constructor K is denoted by its class name C and consists of a list of typed parameters $\overline{x}$ and a body. The body of the constructor contains an invocation to the constructor of the superclass **super** and a sequence of assignments of the parameters to the class fields. Methods declarations have a signature and a body. The signature identifies the method by means of its return type C (we write **void** when the returned value is irrelevant), its name m and its typed parameters $\overline{x}$. The method body contains an expression E whose value is returned. Finally, expressions can be either the void value **null**, a system resource u[2], a variable x, a newly created object **new** $C(\overline{E})$, an object field $E.f$, a method invocation E.m(E'), a system call **system**$_\sigma E$, an implicit or explicit intent propagation (**icast**E and **ecast**CE, respectively), an intent creation $\mathrm{I}_\alpha(E)$, an intent content reading E.data, a sequence of two expressions $E;E'$, a type cast $(C)E$, a conditional **if** $(E = E')$ $\{E_{tt}\}$ **else** $\{E_{ff}\}$, or the creation of a thread through the expression **thread**$\{E\}$**in**$\{E'\}$. System actions labels σ, σ' and intents labels α, α' belong to a finite set and univocally identify the corresponding element. Intuitively, we can assign an action label to certain, security-relevant (groups of) APIs of the Android runtime support, e.g., *read* for `java.io.FileInputStream.read(...)` methods. Hence, we write **system**$_{read}$(f), to denote the action of reading data from a file f, corresponding to the aforementioned Android invocations. Other actions of interest that we might consider include (but are not limited to): Bluetooth data sending/receiving, encryption primitives usage (e.g., size of encryption keys), databases access and NFC communications. Similarly, regarding intents we use I_{www}(a) for intents requesting to open a certain web address a. Thus, the label www can correspond to an actual `ACTION_VIEW` Android intent carrying an HTTP address. Other intents can include: contacts viewing and editing, email writing/sending, web and file system search.

[2] Resources belong to the class `Uri` and we can use specially formatted strings, e.g., "`http://site.com`" or "`file://dir/file.txt`", to uniquely identify them.

Example 1 Consider the following classes

```
class Browser extends Receiver {
 Browser() { super(); }
 void receive(I_www i) {
  return system_connect i.data;
 }
}
class Game {
 Game() { super(); }
 void start() {
  return (
   system_read(~/sav);
   if(UsrAct = TouchAD) {
    icast I_www("http://ad.com") }
   else {/*...play...*/ system_write(~/sav) }
  );
 }
}
```

The class `Browser` is a receiver for intents www. The method `receive`, when triggered, connects to the URL carried by the incoming intent. The class `Game` implements a stand-alone application (we assume method `start` to act as entry point). Its first step consists in reading the content of a file `~/sav`. Then, its execution can take two different branches depending on whether the last action of the user, i.e., resource `UsrAct`, was triggering an in-game advertisement, i.e., resource `TouchAD`[3]. If the user clicks on the in-game advertisement, a `www` intent containing a URL is fired. Otherwise, the game begins and the file `~/sav` is written eventually.

Operational semantics

The behavior of programs is defined through a small-step semantics. Program executions consist of sequences of reduction steps leading from arbitrary complex expressions E, E' to values v, v', i.e., atomic expressions admitting no further reductions. Before presenting the semantics rules we provide intuition of how computation is carried out in our framework by means of an example.

Example 2 Consider the classes provided in Example 1 and, specifically, the body of method `start`. According to the Java statements syntax and semantics, we expect it to behave as follows. First, the program performs a system access

[3] Since this guard is irrelevant for our purposes, we avoid using a more realistic implementation, e.g., via set/get methods.

by reading file `~/sav`. Then it evaluates whether the guard of the conditional statement is true or false. In the first case, the program sends an intent in broadcast. Otherwise, it writes file `~/sav`. In the Android application framework, we expect intents to be propagated to appropriate receivers available in the system. Hence, assuming `Browser` to be the only receiver for www intents, we expect that executing the first branch of the **if** statement triggers the execution of method `Browser.receive(`$\mathtt{I}_{www}$ `i)`. This method performs a single system action, namely `connect`, on the resource carried by the received intent.

The small-step semantics rules are given in Table 2. Rules are defined over judgements of the form $\omega, E \rightarrow \omega', E'$, where ω, ω' are sequences of system actions called *execution histories* and E, E' are expressions, meaning that configuration ω', E' can be reached from ω, E in a single execution step of the program. This representation of program executions follows the history-based approach in the style of Abadi and Fournet [1].

In words, $(\mathtt{NEW})$ rules the evaluation of the actual parameter of a class $\mathtt{C}$ constructor. Rules $(\mathtt{FLD}_1)$ and $(\mathtt{FLD}_2)$ apply to object field access operations. Note that we use functions like *fields* and *mbody* to denote lookup functions returning class data (see [16] for more details). System actions require the evaluation of the target resource (rule $(\mathtt{SYS}_1)$) before being actually executed and added to the execution trace (rule $(\mathtt{SYS}_2)$). Method invocation requires the invocation object (rule $(\mathtt{METH}_1)$) and the actual parameter (rule $(\mathtt{METH}_2)$) to be evaluated before the body of the method (provided by function *mbody*) is executed (rule $(\mathtt{METH}_3)$). Intents parameter are reduced according to rule $\mathtt{INT}$. Implicit intent casting requires the reduction of its parameter (rule $(\mathtt{IMPC}_1)$) before being passed to a valid receiver (rule $(\mathtt{IMPC}_2)$). We use a function *receiver*(α) to retrieve the finite set of receivers for an intent α. Explicit intents are handled similarly (rules $(\mathtt{EXPC}_1)$ and $(\mathtt{EXPC}_2)$). The main difference is that in rule $(\mathtt{EXPC}_2)$ the class of the receiver is bound by the intent casting operation. Similarly to class fields, intent data is accessed through rules $(\mathtt{DATA}_1)$ and $(\mathtt{DATA}_2)$. Four rules drive the execution of conditional branches. Rules $(\mathtt{IF}_1)$ and $(\mathtt{IF}_2)$ drive the reduction of the expressions appearing in the equality check. Then, according to whether the guard is satisfied (rule $(\mathtt{IF}_3)$) or not (rule $(\mathtt{IF}_4)$) the computation proceeds to the left or right branch. Parallel computation proceeds by non-deterministically reducing either the left (rule $(\mathtt{PAR}_1)$) or the right (rule $(\mathtt{PAR}_2)$) term until both reduce to values and the right one is returned (rule $(\mathtt{PAR}_3)$). Finally, type casting requires reduction of its target (rule $(\mathtt{CAST}_1)$) before checking that the actual class is a subclass (we use relation $\cdot <: \cdot$ as in [16]) of $\mathtt{C}$.

Example 3 Consider again the classes of Example 1. We show the computation for (the body of) method `start` assuming that `UsrAct = TouchAD`.

$$E \doteq \begin{array}{l} \mathbf{system}_{read}(\tilde{}/\mathtt{sav}); \\ \mathbf{if}(\mathtt{UsrAct} = \mathtt{TouchAD}) \\ \quad \{\ \mathbf{icast}\ \mathtt{I}_{www}(\text{“}\mathtt{http://ad.com}\text{”})\ \} \\ \mathbf{else} \\ \quad \{\ \mathbf{system}_{write}(\tilde{}/\mathtt{sav})\ \} \end{array}$$

Also, we start the computation from the empty history and we assume the class `Browser` to be the only receiver in the system.

$$\cdot, E \rightarrow \mathtt{system}_{read}(\tilde{}/\mathtt{sav}), \begin{array}{l} \mathtt{if}(\mathtt{UsrAct} = \mathtt{TouchAD}) \\ \ \{\mathtt{icast\,I}_{www}(\text{“}\mathtt{http://ad.com}\text{”})\} \\ \mathtt{else} \\ \ \{\mathtt{system}_{write}(\tilde{}/\mathtt{sav})\} \end{array} \tag{1}$$

$$\rightarrow \mathtt{system}_{read}(\tilde{}/\mathtt{sav}),\ \mathtt{icast\,I}_{www}(\text{“}\mathtt{http://ad.com}\text{”}) \tag{2}$$

$$\rightarrow \mathtt{system}_{read}(\tilde{}/\mathtt{sav}), \begin{array}{l} \mathtt{system}_{connect} \\ (\mathtt{I}_{www}(\text{“}\mathtt{http://ad.com}\text{”}).\mathtt{data}) \end{array} \tag{3}$$

$$\rightarrow \mathtt{system}_{read}(\tilde{}/\mathtt{sav}), \mathtt{system}_{connect}(\text{“}\mathtt{http://ad.com}\text{”}) \tag{4}$$

$$\rightarrow \mathtt{system}_{read}(\tilde{}/\mathtt{sav})\mathtt{system}_{connect}(\text{“}\mathtt{http://ad.com}\text{”}), \mathtt{null} \tag{5}$$

A few steps are collapsed in a single reduction in (1). Indeed, we apply rule $(\mathtt{SEQ}_1)$ to evaluate the first part of E. As a premise, rule $(\mathtt{SEQ}_1)$ requires $(\mathtt{SYS}_2)$ which modifies the execution trace. Also, we apply $(\mathtt{SEQ}_2)$ to simplify the sequence $\mathtt{null}; \mathtt{if} \cdots$. Reduction (2) applies rule $(\mathtt{IF}_3)$ (recall that we assumed the guard to be satisfied) and the computation takes the true branch. Step (3) is obtained by applying rule $(\mathtt{IMPC}_2)$ which causes the invocation of the `receive` method of receiver `Browser` (being the only available one). Moreover, we use rule $(\mathtt{METH}_3)$ to substitute the method call with the body of the method where the invocation parameter replaces the variable `i`. Then, reduction (4) uses rules $(\mathtt{SYS}_1)$ and, consequently, $(\mathtt{DATA}_2)$ to extract the resource carried by the received intent. Eventually, in step (5) rule $(\mathtt{SYS}_2)$ is applied again and the execution trace extended. Since the computation reduced the starting expression E to a value, i.e., **null**, no further transition can take place and the execution stops.

3 Type and Effect

We present a type and effect system for our programming language.

A type and effect system for FJ has been previously described in [27]. Some of our rules, i.e., those for the original FJ statements, resemble the rules given there. In addition, we introduce several new rules for typing Android-specific instructions, e.g., those related to intents management.

Table 2 Semantics of expressions

$$(\texttt{NEW})\ \frac{\omega, E_i \to \omega', E_i'}{\omega, \texttt{new C}(\bar{v}, E_i, \ldots) \to \omega', \texttt{new C}(\bar{v}, E_i', \ldots)} \qquad (\texttt{FLD}_1)\ \frac{\omega, E \to \omega', E'}{\omega, E.\texttt{f} \to \omega', E'.\texttt{f}} \qquad (\texttt{FLD}_2)\ \frac{\mathit{fields}(C) = \bar{D}\,\bar{f}}{\omega, \texttt{new C}(\bar{v}).\texttt{f}_\texttt{i} \to \omega', v_i}$$

$$(\texttt{SYS}_1)\ \frac{\omega, E \to \omega', E'}{\omega, \texttt{system}_\sigma\ E \to \omega', \texttt{system}_\sigma\ E'} \qquad (\texttt{SYS}_2)\ \omega, \texttt{system}_\sigma\ u \to \omega \cdot \sigma(u), \texttt{null} \qquad (\texttt{METH}_1)\ \frac{\omega, E \to \omega', E''}{\omega, E.\texttt{m}(E') \to \omega', E''.\texttt{m}(E')}$$

$$(\texttt{METH}_2)\ \frac{\omega, E' \to \omega', E''}{\omega, E.\texttt{m}(E') \to \omega', E.\texttt{m}(E'')} \qquad (\texttt{METH}_3)\ \frac{\mathit{mbody}(\texttt{m}, \texttt{C}) = \texttt{x}, E}{\omega, (\texttt{new C}(\bar{v})).\texttt{m}(v') \to \omega, E[v'/x, (\texttt{new C}(\bar{v}))/\texttt{this}]}$$

$$(\texttt{INT})\ \frac{\omega, E \to \omega', E'}{\omega, I_\alpha(E) \to \omega', I_\alpha(E')}\ (\texttt{IMPC}_1)\ \frac{\omega, E \to \omega', E'}{\omega, \texttt{icast}\ E \to \omega', \texttt{icast}\ E'}\ (\texttt{IMPC}_2)\ \frac{\texttt{new C}(\bar{v}) \in \mathit{receiver}(\alpha)}{\omega, \texttt{icast}\ I_\alpha(u) \to \omega, \texttt{new C}(\bar{v}).\texttt{receive}(I_\alpha(u))}$$

$$(\texttt{EXPC}_1)\ \frac{\omega, E \to \omega', E'}{\omega, \texttt{ecast C}\ E \to \omega', \texttt{ecast C}\ E'} \qquad (\texttt{EXPC}_2)\ \frac{\texttt{new C}(\bar{v}) \in \mathit{receiver}(\alpha)}{\omega, \texttt{ecast C}\ I_\alpha(u) \to \omega, \texttt{new C}(\bar{v}).\texttt{receive}(I_\alpha(u))}$$

$$(\texttt{DATA}_1)\ \frac{\omega, E \to \omega', E'}{\omega, E.\texttt{data} \to \omega', E'.\texttt{data}} \qquad (\texttt{DATA}_2)\ \omega, I_\alpha(v).\texttt{data} \to \omega, v \qquad (\texttt{IF}_1)\ \frac{\omega, E \to \omega', E''}{\begin{array}{l}\omega, \texttt{if}(E = E')\,\texttt{then}\,\{E_{tt}\}\,\texttt{else}\,\{E_{\mathit{ff}}\} \to \\ \quad \omega', \texttt{if}(E'' = E')\,\texttt{then}\,\{E_{tt}\}\,\texttt{else}\,\{E_{\mathit{ff}}\}\end{array}}$$

$$(\texttt{IF}_2)\ \frac{\omega, E' \to \omega', E''}{\begin{array}{l}\omega, \texttt{if}(v = E')\,\texttt{then}\,\{E_{tt}\}\,\texttt{else}\,\{E_{\mathit{ff}}\} \to \\ \quad \omega', \texttt{if}(v = E'')\,\texttt{then}\,\{E_{tt}\}\,\texttt{else}\,\{E_{\mathit{ff}}\}\end{array}} \qquad (\texttt{IF}_3)\ \omega, \texttt{if}(v = v)\,\texttt{then}\,\{E_{tt}\}\,\texttt{else}\,\{E_{\mathit{ff}}\} \to \omega, E_{tt}$$

$$(\texttt{IF}_4)\ \frac{v \neq v'}{\omega, \texttt{if}(v = v')\,\texttt{then}\,\{E_{tt}\}\,\texttt{else}\,\{E_{\mathit{ff}}\} \to \omega, E_{\mathit{ff}}} \qquad (\texttt{SEQ}_1)\ \frac{\omega, E \to \omega', E''}{\omega, E; E' \to \omega', E''; E'} \qquad (\texttt{SEQ}_2)\ \omega, v; E \to \omega, E$$

$$(\texttt{PAR}_1)\ \frac{\omega, E \to \omega', E''}{\omega, \texttt{thread}\,\{E\}\,\texttt{in}\,\{E'\} \to \omega', \texttt{thread}\,\{E''\}\,\texttt{in}\,\{E'\}} \qquad (\texttt{PAR}_2)\ \frac{\omega, E' \to \omega', E''}{\omega, \texttt{thread}\,\{E\}\,\texttt{in}\,\{E'\} \to \omega', \texttt{thread}\,\{E\}\,\texttt{in}\,\{E''\}}$$

$$(\texttt{PAR}_3)\ \omega, \texttt{thread}\,\{v\}\,\texttt{in}\,\{v'\} \to \omega, v' \qquad (\texttt{CAST}_1)\ \frac{\omega, E \to \omega', E'}{\omega, (\texttt{C})E \to \omega', (\texttt{C})E'} \qquad (\texttt{CAST}_2)\ \frac{D <: C}{\omega, (\texttt{C})\texttt{new D}(\bar{v}) \to \omega, \texttt{new D}(\bar{v})}$$

Table 3 Semantics of history expressions

$$\sigma(u) \xrightarrow{\sigma(u)} \varepsilon \qquad\qquad \alpha_\chi(u) \xrightarrow{\alpha_\chi(u)} \varepsilon$$

$$\frac{H \xrightarrow{\alpha_\chi(u)} H'' \quad \dot{H} = \sum H'\{\alpha_?(u)/h\} \text{ s.t. } \bar{\alpha}_C h.H' \in \rho(\alpha) \text{ and } \chi \succcurlyeq C}{H \xrightarrow{\cdot} \dot{H}}$$

$$\frac{H' \xrightarrow{a} H''}{H \parallel H' \xrightarrow{a} H \parallel H''} \qquad \frac{H \xrightarrow{a} H''}{H \parallel H' \xrightarrow{a} H'' \parallel H'} \qquad \frac{H \xrightarrow{a} H''}{H \cdot H' \xrightarrow{a} H'' \cdot H'}$$

$$\frac{H' \xrightarrow{a} H''}{\varepsilon \cdot H' \xrightarrow{a} H''} \qquad \frac{H \xrightarrow{a} H''}{H + H' \xrightarrow{a} H''} \qquad \frac{H' \xrightarrow{a} H''}{H + H' \xrightarrow{a} H''}$$

$$\frac{H\{\mu h.H/h\} \xrightarrow{a} H'}{\mu h.H \xrightarrow{a} H'}$$

History expressions

The type and effect system extracts *history expressions* from programs. History expressions model computational agents in a process-algebraic fashion in terms of the traces of events they produce. The syntax of history expressions is as follows.

Definition 1 (Syntax of history expressions)

$$\begin{aligned} H, H' ::= {} & \varepsilon \mid h \mid \alpha_\chi(u) \mid \bar{\alpha}_C h.H \mid \sigma(u) \mid \\ & H \cdot H' \mid H + H' \mid H \parallel H' \mid \mu h.H \end{aligned}$$

Briefly, they can be empty ε, variables h, h', intent emissions $\alpha_\chi(u)$ (with $\chi \in \{C, ?\}$), intent receptions $\bar{\alpha}_C h.H$, system accesses $\sigma(u)$, sequences $H \cdot H'$, choices $H + H'$, parallel executions $H \parallel H'$ or recursions $\mu h.H$. Their semantics is defined through a *labelled transition system* (LTS) by the rules in Table 3.

From top to bottom, a system access $\sigma(u)$ (rule *system*) and an intent generation $\alpha_\chi(u)$ (rule *intent*) cause a corresponding event and reduce to ε. An intent exchange requires a receiver $\bar{\alpha}_C h.H'$ compatible with the intent destination χ. Compatibility is checked through the relation $\cdot \succcurlyeq \cdot$ (s.t. for all C, $C \succcurlyeq C$ and $? \succcurlyeq C$). The function ρ corresponds to the function *receiver* introduced in Section 2. Concurrent history expressions $H \parallel H'$ admit a reduction either for their left or for their right component (rules *l-parallel*, *r-parallel*). A sequence $H \cdot H'$ behaves like H until it reduces to ε and then behaves like H' (rule *sequence*). The non-deterministic choice $H + H'$ can behave like either H or H' (rule *choice*). Finally, recursion $\mu h.H$ has the same behavior as H where the free occurrences of h are replaced by $\mu h.H$ (rule *recursion*). We write $H \xrightarrow{a_1 \cdots a_n}{}^{*} H'$ as a shorthand for $H \xrightarrow{a_1} \ldots \xrightarrow{a_n} H'$.

Moreover, we define a partial order relation $\sqsubseteq$ over history expressions as follows: $H \sqsubseteq H'$ iff $H \xrightarrow{a_1 \cdots a_n}{}^{*} H''$ implies there exists $\tilde{H}$ such that $H' \xrightarrow{a_1 \cdots a_n}{}^{*} \tilde{H}$. We write $H \equiv H'$ as a shorthand for $H \sqsubseteq H'$ and $H' \sqsubseteq H$ (see [7] for more details).

Type and effect system

Before presenting our type and effect system, we define *types* and *type environments*.

Definition 2 (Types and type environment)

$$\tau, \tau' ::= \mathbf{1} \mid \mathcal{U} \mid \mathcal{I}_\alpha(\mathcal{U}) \mid C \qquad \Gamma, \Gamma' ::= \emptyset \mid \Gamma\{\tau/x\}$$

In other words, types can be the unit $\mathbf{1}$, a set of resources $\mathcal{U}$, an intent type $\mathcal{I}_\alpha(\mathcal{U})$, or a class type C. A type environment is a mapping from variables to types.

We can now present the typing rules of our type and effect system. A typing judgment has the form $\Gamma \vdash E : \tau \triangleright H$ and we read it "under environment Γ, the expression E has type τ and effect H". The type and effect rules are given in Table 4. Rules (T−NULL) and (T−RES) are straightforward. Rule (T−VAR) says that the type of a variable x is given by the current type environment Γ. Rule (T−INT) states that an intent $\mathtt{I}_\alpha(E)$ has type $\mathcal{I}_\alpha(\mathcal{U})$ and effect H where $\mathcal{U}$ and H are the type and effect of E. Conversely, rule (T−DATA) types the data carried by an intent of type $\mathcal{I}_\alpha(\mathcal{U})$ to $\mathcal{U}$. Rule (T−FLD) types the field of an object according to its declaration (where `fields` is the same function used in (FLD_2) of Table 2). Implicit intent cast is typed, according to rule (T−IMPC), to $\mathbf{1}$. Also, it has effect equal to the summation of all the actions $\alpha_?(u)$ where u ranges over $\mathcal{U}$. Rules (T−EXPC) and (T−SYS) behave similarly but for the actions α_C and σ (respectively) in place of $\alpha_?$.

Typing a class constructor (rule (T−NEW)) results in a class type C and the sequence of the history expressions obtained from typing its parameters. Method invocations (rule (T−METH)) require more attention. We state that the invocation $E.\mathtt{m}(E')$ has type τ'' and effect $H \cdot H' \cdot H''$ where τ'' is the type of the return expression E'', H is the effect generated by E, H' the effect for E' and H'' for E''. Also, note that E'' is typed under the environment $[C/\mathtt{this}, \tau'/\mathtt{x}]$ and the function $\mathit{msign}(\mathtt{m}, C)$ returns the signature of a method, i.e., $\dot{\tau}' \to \tau''$ for a method declaring input type $\dot{\tau}'$ and return type τ''[4] (where the subtype relation <: is the same introduced in Section 2). A conditional (rule (T−IF)) has effect equal to the sequence the two effects generated by the expressions in its guard, i.e., H and H', and an effect generalizing those of the two branches, i.e., $\dot{H}$ (while the type is the same of E_{tt} and $E_{f\!f}$). Instead, a sequence (rule (T−SEQ)) has the type of its second expression and effect $H_1 \cdot H_2$, i.e., the concatenation of the two sub-effects. Rule (T−CAST) says that casting an expression does not alter its effect but only its type. Concurrent executions (rule (T−PAR)) evaluate to the parallel composition of two effects while the type is the one returned by the second expression. Finally, the *weakening* rule (T−WKN) allows for effect generalization. The rule says that, if we can type an expression with an effect H', then we can also type it with a more general one H, in symbols $H' \sqsubseteq H$. More details about type and effect systems can be found in [27, 26, 7].

[4] For a class C<:`Receiver` we write $\mathit{msign}(\mathtt{receive}, C) = \mathcal{I}_\alpha \to \mathbf{1}$

Traditionally, type systems guarantee certain, desirable properties of well-typed programs, e.g., they will never perform illegal memory accesses. A similar result is obtained for our type and effect system. Indeed, we show[5] that well-typed expressions do not produce erroneous computations, i.e., they always return a value or admit further reductions.

Lemma 1 *For each* closed *(i.e., without free variables) expression E, environment Γ, history expression H, type τ and trace ω, if $\Gamma \vdash E : \tau \triangleright H$ then either E is a value or $\omega, E \to \omega', E'$ (for some ω', E').*

Also well-typed expressions generate history expressions which *safely* denote the runtime behavior of expressions. The following theorem states that by typing a (closed) expression E we obtain an over-approximation of the set of all the possible executions of E.

Theorem 1 *For each closed expression E, history expression H, type τ and trace ω, if $\emptyset \vdash E : \tau \triangleright H$ and $\cdot, E \to^* \omega, E'$ then there exist H' such that $H \xrightarrow{\omega}{}^{*} H'$ and $\emptyset \vdash E' : \tau \triangleright H'$.*

We extend the type and effect system to method declarations in the following way. Given a class C and a method m (different from `receive`) such that $\mathit{mbody}(\mathtt{m}, C) = \mathtt{x}, E$, $\mathit{msign}(\mathtt{m}, C) = D \to F$ and $[C/\mathtt{this}, D/\mathtt{x}] \vdash E : F \triangleright H$, we write $\vdash C.\mathtt{m} : D \xrightarrow{H} F$ and say H to be the *latent effect* of m. Instead, if C<: `Receiver` and m = `receive` we write $\vdash C.\mathtt{m} : \mathcal{I}_\alpha \xrightarrow{\bar{\alpha}_C h.H} \mathbf{1}$.

Finally, we exploit the typing rules for the generation of the function ρ, appearing in the semantics of history expressions, which was left undefined. For each existing receiver, we type the corresponding `receive` method and we obtain

$$\rho(\alpha) = \{\bar{\alpha}_C h.H \mid \mathtt{new}\, C(\bar{v}) \in \mathit{receiver}(\alpha) \text{ and } \vdash C.\mathtt{receive} : \mathcal{I}_\alpha \xrightarrow{\bar{\alpha}_C h.H} \mathbf{1}\}$$

4 Security policies

We now focus on the specification, verification and management of security policies. We start by introducing the

[5] Proofs of this and following results can be found in Appendix.

Table 4 Typing rules

$$(\texttt{T-NULL})\ \Gamma \vdash \texttt{null} : \mathbf{1} \rhd \varepsilon \qquad (\texttt{T-RES})\ \Gamma \vdash u : \{u\} \rhd \varepsilon \qquad (\texttt{T-VAR})\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \rhd \varepsilon} \qquad (\texttt{T-INT})\ \frac{\Gamma \vdash E : \mathcal{U} \rhd H}{\Gamma \vdash \texttt{I}_\alpha(E) : \mathcal{I}_\alpha(\mathcal{U}) \rhd H}$$

$$(\texttt{T-DATA})\ \frac{\Gamma \vdash E : \mathcal{I}_\alpha(\mathcal{U}) \rhd H}{\Gamma \vdash E.\texttt{data} : \mathcal{U} \rhd H} \qquad (\texttt{T-FLD})\ \frac{\Gamma \vdash E : C \rhd H \quad \mathit{fields}(C) = \bar{D}\bar{f}}{\Gamma \vdash E.\texttt{f}_\texttt{i} : D_i \rhd H} \qquad (\texttt{T-IMPC})\ \frac{\Gamma \vdash E : \mathcal{I}_\alpha(\mathcal{U}) \rhd H}{\Gamma \vdash \texttt{icast}\, E : \mathbf{1} \rhd H \cdot \sum_{u \in \mathcal{U}} \alpha_?(u)}$$

$$(\texttt{T-EXPC})\ \frac{\Gamma \vdash E : \mathcal{I}_\alpha(\mathcal{U}) \rhd H}{\Gamma \vdash \texttt{ecast}\,\texttt{C}\, E : \mathbf{1} \rhd H \cdot \sum_{u \in \mathcal{U}} \alpha_C(u)} \qquad (\texttt{T-SYS})\ \frac{\Gamma \vdash E : \mathcal{U} \rhd H}{\Gamma \vdash \texttt{system}_\sigma\, E : \mathbf{1} \rhd H \cdot \sum_{u \in \mathcal{U}} \sigma(u)} \qquad (\texttt{T-NEW})\ \frac{\Gamma \vdash E_i : \tau_i \rhd H_i}{\Gamma \vdash \texttt{new}\,\texttt{C}(\bar{E}) : C \rhd H_1 \cdots H_n}$$

$$(\texttt{T-METH})\ \frac{\Gamma \vdash E : C \rhd H \quad \Gamma \vdash E' : \tau' \rhd H' \quad \mathit{mbody}(\texttt{m}, C) = \texttt{x}, E'' \quad [C/\texttt{this}, \tau'/\texttt{x}] \vdash E'' : \tau'' \rhd H'' \quad \mathit{msign}(\texttt{m}, C) = \dot{\tau}' \to \tau'' \quad \tau' <: \dot{\tau}'}{\Gamma \vdash E.\texttt{m}(E') : \tau'' \rhd H \cdot H' \cdot H''}$$

$$(\texttt{T-IF})\ \frac{\Gamma \vdash E : \tau \rhd H \quad \Gamma \vdash E' : \tau' \rhd H' \quad \Gamma \vdash E_{tt} : \dot{\tau} \rhd \dot{H} \quad \Gamma \vdash E_{\mathit{ff}} : \dot{\tau} \rhd \dot{H}}{\Gamma \vdash \texttt{if}(E = E')\,\texttt{then}\,\{E_{tt}\}\,\texttt{else}\,\{E_{\mathit{ff}}\} : \dot{\tau} \rhd H \cdot H' \cdot \dot{H}} \qquad (\texttt{T-SEQ})\ \frac{\Gamma \vdash E_i : \tau_i \rhd H_i}{\Gamma \vdash E_1; E_2 : \tau_2 \rhd H_1 \cdot H_2}$$

$$(\texttt{T-CAST})\ \frac{\Gamma \vdash E : D \rhd H \quad D <: C}{\Gamma \vdash (\texttt{C})E : C \rhd H} \qquad (\texttt{T-PAR})\ \frac{\Gamma \vdash E_i : \tau_i \rhd H_i}{\Gamma \vdash \texttt{thread}\,\{E_1\}\,\texttt{in}\,\{E_2\} : \tau_2 \rhd H_1 \| H_2} \qquad (\texttt{T-WKN})\ \frac{\Gamma \vdash E : \tau \rhd H' \quad H' \sqsubseteq H}{\Gamma \vdash E : \tau \rhd H}$$

policy language, then we describe how policies are used to verify and maintain the security state of the devices joining an organization. This section essentially extends the work presented in [6]. In particular, (i) we adopt a more powerful policy language, (ii) we provide a sound and complete proof system for it and (iii) we extend the PMC approach with new theoretical results.

Policy language

We use *Hennessy-Milner logic* (HML) [14] with recursion [19] for specifying security policies. Recursion extends the expressive power of HML, which is limited to finite traces in the original version. In this way we can express a fairly rich class of both safety and liveness properties. Moreover, we use parametric actions in place of simple labels. Given a finite set of identifiers $\mathbf{Id}$, we define the set of HML formulas over $\mathbf{Id}$ ($\Phi_{\mathbf{Id}}$ in symbols) as the smallest set of formulas obtained through the following syntax.

Definition 3 (Syntax of HML formulas)

$$\varphi, \varphi' ::= tt \mid \mathit{ff} \mid \varphi \wedge \varphi' \mid \varphi \vee \varphi' \mid \langle \sigma(\dot{x}) \rangle.\varphi \mid [\sigma(\dot{x})].\varphi \mid F$$

where $F \in \mathbf{Id}$.

Thus a policy can be the positive (tt) or the negative (ff) truth value, a conjunction $\varphi \wedge \varphi'$, a disjunction $\varphi \vee \varphi'$, a formula φ prefixed by the existential, namely *diamond*, $\langle \sigma(\dot{x}) \rangle$ or the universal, namely *box*, $[\sigma(\dot{x})]$ modal operator, or an identifier F. In modalities, $\dot{x}$ can be either a resource u or a variable x. We call *concrete diamond* (*box*) the operator $\langle \sigma(u) \rangle$ ($[\sigma(u)]$, resp.) and *abstract diamond* (*box*, resp.) the operator $\langle \sigma(x) \rangle$ ($[\sigma(x)]$). Identifiers are defined through a *declaration function* $\mathcal{D}$ mapping identifier names to formulas, i.e., $\mathcal{D} : \mathbf{Id} \to \Phi_{\mathbf{Id}}$.

We say that *a history expression* H *satisfies a policy* φ iff $H \models \phi$, where $H \models \phi$ is inductively defined as follows:

- (true) $H \models tt$;
- (conjunction) $H \models \varphi \wedge \varphi'$ iff $H \models \varphi$ and $H \models \varphi'$;
- (disjunction) $H \models \varphi \vee \varphi'$ iff $H \models \varphi$ or $H \models \varphi'$;
- (c-diamond) $H \models \langle \sigma(u) \rangle.\varphi$ iff there exists H' such that $H' \xrightarrow{\sigma(u)} H'$ and $H' \models \varphi$;
- (a-diamond) $H \models \langle \sigma(x) \rangle.\varphi$ iff there exist u and H' such that $H \xrightarrow{\sigma(u)} H'$ and $H' \models \varphi\{u/x\}$;
- (c-box) $H \models [\sigma(u)].\varphi$ iff for all H' if $H \xrightarrow{\sigma(u)} H'$, then $H' \models \varphi$;
- (a-box) $H \models [\sigma(x)].\varphi$ iff for all u and H' if $H \xrightarrow{\sigma(u)} H'$, then $H' \models \varphi\{u/x\}$;
- (identifier) $H \models F$ iff $H \models \bigvee_{n \geqslant 0} f^n(\mathit{ff})$, where $f(\varphi) = \mathcal{D}(F)\{\varphi/F\}$.

Informally, tt is valid for every history expression H (rule *true*), a conjunction is satisfied by history expressions valid for both sub-formulas (rule *conjunction*) and disjunction is satisfied when at least one of the sub-clauses is so (rule *disjunction*). Existential modality referring to a resource u is satisfied if the history expression admits at least one corresponding transition and the resulting expression satisfies the sub-formula φ (rule *c-diamond*). Similarly, diamonds with a variable x check whether a valid reduction exists (rule *a-diamond*). However, in this case the history expression H' is checked against a sub-formula φ where the free instances of x have been replaced by the referred resource u. Complementary, universal modality (rules *c-box* and *a-box*) is satisfied when each transition $\sigma(u)$ ($\sigma(x)$, respectively) leads to a history expression satisfying φ ($\varphi\{u/x\}$, resp.).

Example 4 Although real policies might refer to domain-specific aspects, we propose a few specifications that could be useful in a generic BYOD environment. We briefly list them with a short explanation in natural language.

1. SSL socket connections with the corporation servers must be eventually closed.

$$[\texttt{openSSL}(\text{“}\texttt{corp.com}\text{”})].F$$

where
$F \triangleq [\texttt{recvData}()].F \vee \langle \texttt{closeSSL}(\text{“}\texttt{corp.com}\text{”})\rangle.tt$

2. Standard socket connections with the corporation servers are prohibited.

$$[\texttt{openSocket}(\text{“}\texttt{corp.com}\text{”})].ff$$

3. Do not save corporate emails on public cloud storage services.

$$[\texttt{connect}(\text{“}\texttt{mail.corp.com}\text{”})].[\texttt{write}(\texttt{dropbox/})].ff$$

Notice that, the first policy exploits the recursion on F to define a *liveness* property. Instead, the second and third policies define *safety* properties. The importance of these two categories of properties is well known [18].

Policy verification

Here we present a proof system (in the style of [19]) for the verification of compliance between a history expression and a HML formula. Then we prove our proof system to be sound and complete with respect to the definition of validity given above.

The proof system is defined by the following rules.

$$\text{(true)}\quad H \Vdash tt \qquad (\wedge)\ \frac{H \Vdash \varphi \qquad H \Vdash \varphi'}{H \Vdash \varphi \wedge \varphi'}$$

$$(\vee_L)\ \frac{H \Vdash \varphi}{H \Vdash \varphi \vee \varphi'} \qquad (\vee_R)\ \frac{H \Vdash \varphi'}{H \Vdash \varphi \vee \varphi'}$$

$$(\langle u\rangle)\ \frac{H' \Vdash \varphi}{H \Vdash \langle\sigma(u)\rangle.\varphi} \quad \text{with } H \xrightarrow{\sigma(u)} H'$$

$$(\langle x\rangle)\ \frac{H' \Vdash \varphi\{u/x\}}{H \Vdash \langle\sigma(x)\rangle.\varphi} \quad \text{with } H \xrightarrow{\sigma(u)} H'$$

$$([u])\ \frac{H_1 \Vdash \varphi \qquad H_2 \Vdash \varphi \qquad \cdots}{H \Vdash [\sigma(u)].\varphi}$$

$$\text{with } H_i \in \{H' \text{ s.t. } H \xrightarrow{\sigma(u)} H'\}$$

$$([x])\ \frac{H_1 \Vdash \varphi\{u_1/x\} \qquad H_2 \Vdash \varphi\{u_2/x\} \qquad \cdots}{H \Vdash [\sigma(x)].\varphi}$$

$$\text{with } H_i \in \{H' \text{ s.t. } H \xrightarrow{\sigma(u_i)} H'\}$$

$$\text{(Rec)}\ \frac{H \Vdash \varphi}{H \Vdash F} \quad \mathcal{D}(F) = \varphi$$

We propose a brief example to clarify the usage of the rules given above.

Example 5 Consider the history expression defined below.

$$H = \texttt{read}(\tilde{}/\mathsf{sav}) \cdot \begin{pmatrix} \texttt{www}_?(\text{“}\texttt{http://ad.com}\text{”}) \\ +\texttt{write}(\tilde{}/\mathsf{sav}) \end{pmatrix}$$

and the formula

$$\varphi = \langle \texttt{read(x)}\rangle.(\langle \texttt{connect(y)}\rangle.tt \wedge [\texttt{read(x)}].tt)$$

For simplicity, here we assume that the only existing receiver for `www` is `Browser`[6]. This amounts to say that $\rho(\texttt{www})$ $= \{\overline{\texttt{www}}_{\texttt{Browser}}\texttt{h.connect}(\text{“}\texttt{http://ad.com}\text{”})\}$. We show that $H \models \varphi$.

$$(\langle x\rangle)\ \frac{(\wedge)\ \dfrac{(\langle x\rangle)\ \dfrac{\varepsilon \Vdash tt}{H' \Vdash \langle\texttt{connect(y)}\rangle.tt} \qquad \dfrac{\varepsilon \Vdash tt}{H' \Vdash [\texttt{read}(\tilde{}/\mathsf{sav})].tt}\ ([x])}{H' \Vdash \langle\texttt{connect(y)}\rangle.tt \wedge [\texttt{read}(\tilde{}/\mathsf{sav})].tt}}{H \Vdash \langle\texttt{read(x)}\rangle.(\langle\texttt{connect(y)}\rangle.tt \wedge [\texttt{read(x)}].tt)}$$

where $H' = \texttt{www}_?(\text{“}\texttt{http://ad.com}\text{”}) + \texttt{write}(\tilde{}/\mathsf{sav})$.

Clearly, the compact formula proposed in Example 5 is too simplistic for modeling a realistic policy and we consider it only for the sake of presentation.

The proof system presented here is sound and complete as stated by the following theorem.

Theorem 2 $H \models \varphi$ *if and only if* $H \Vdash \varphi$

Hence, by applying the given proof system we obtain an effective procedure for verifying the validity of an history expression against a formula.

Partial model checking

In [3] *partial model checking* (PMC) is presented as a technique for the partial evaluation of a formula against a model. When applied to an instance of the model checking problem, PMC returns a new, equivalent one in which pieces of information, i.e., part of the denoted behavior, have been moved from the model to the formula. Under our assumptions, PMC allows to represent the whole configuration of a mobile device, i.e., the set of installed applications, by means of a single, partially evaluated policy.

In practice, PMC uses reduction rules for transferring information from the model to the formula it must satisfy. Although originally defined for the equational modal μ-calculus, we can apply PMC to HML by redefining the equivalence rules. To this end, we define the operator $\cdot_{/\!/\cdot}$ for the partial evaluation against parallel composition.

$$tt_{/\!/H} = tt \qquad (\varphi \wedge \varphi')_{/\!/H} = \varphi_{/\!/H} \wedge \varphi'_{/\!/H}$$

$$ff_{/\!/H} = ff \qquad (\varphi \vee \varphi')_{/\!/H} = \varphi_{/\!/H} \vee \varphi'_{/\!/H}$$

$$(\langle\sigma(u)\rangle.\varphi)_{/\!/H} = \langle\sigma(u)\rangle.\varphi_{/\!/H} \vee \bigvee_{H \xrightarrow{\sigma(u)} H'} \varphi_{/\!/H'}$$

$$(\langle\sigma(x)\rangle.\varphi)_{/\!/H} = \langle\sigma(x)\rangle.\varphi_{/\!/H} \vee \bigvee_{H \xrightarrow{\sigma(u)} H'} \varphi\{u/x\}_{/\!/H'}$$

$$([\sigma(u)].\varphi)_{/\!/H} = [\sigma(u)].\varphi_{/\!/H} \wedge \bigwedge_{H \xrightarrow{\sigma(u)} H'} \varphi_{/\!/H'}$$

$$([\sigma(x)].\varphi)_{/\!/H} = [\sigma(x)].\varphi_{/\!/H} \wedge \bigwedge_{H \xrightarrow{\sigma(u)} H'} \varphi\{u/x\}_{/\!/H'}$$

$$F_{/\!/H} = F_H \text{ and if } F_H \not\in dom(\mathcal{D}) \text{ then } \mathcal{D}(F_H) = \mathcal{D}(F)_{/\!/H}$$

[6] Real Android devices natively install some receivers of this kind.

Intuitively, the tt and ff formulas keep unchanged while for conjunction and disjunction PMC applies to the sub formulas, recursively. A formula $\langle\sigma(u)\rangle.\varphi$ reduces to the disjunction between (left) the formula obtained by recursively applying PMC to φ and (right) the finite disjunction among the PMC of φ against all the possible history expressions H' (obtained after a $\sigma(u)$ step from H). The rule for $\langle\sigma(x)\rangle.\varphi$ is similar. The main difference is that the right disjunction also causes the instantiation of x to u (being a resource appearing in some σ-transition) in φ. Instead, box modality $[\sigma(u)].\varphi$ transforms to a conjunction of the formula where the PMC operator is applied recursively (left) and the conjunction of all the PMC of φ against H' where H' is the target of a $\sigma(u)$ transition from H. PMC of a formula $[\sigma(x)].\varphi$ generates a conjunction analogous to the previous case but for the sub formulas on the right-hand side where x is replaced by u (appearing in any σ-transition). Finally, an identifier F is replaced by a new identifier F_H. Also, if F_H is fresh, i.e., it is not defined in $\mathcal{D}$, a new declaration for it is added to $\mathcal{D}$. The formula associated to F_H is obtained through PMC of the formula for F.

Policy compliance is preserved by the PMC operator as stated by the following theorem.

Theorem 3 $H \models \varphi_{/\!/H'}$ *if and only if* $H \parallel H' \models \varphi$.

In words, this property grants that the compliance of two (or more) concurrent components against a policy is logically equivalent to the compliance of one of them against a variant of the the original policy obtained by specializing it w.r.t. the other component. Clearly, this theorem is particularly useful when new (models of) applications compose with the set of previously installed ones. An example can better illustrate this aspect.

Example 6 Consider the simple, sandboxing policy

$$\varphi = [\mathtt{read(x)}].[\mathtt{connect(y)}].\mathit{ff} \wedge [\mathtt{write(x)}].[\mathtt{connect(y)}].\mathit{ff}$$

and the history expressions

$$H_U = (\mu h_1.(\mathtt{www}_?(\text{“}\mathtt{home.com}\text{”})) + h_1)$$
$$H_G = (\mathtt{read}(\tilde{}/\mathrm{sav})\cdot(\mathtt{write}(\tilde{}/\mathrm{sav}) + \mathtt{www}_?(\text{“}\mathtt{ad.com}\text{”})))$$
$$H_B = (\overline{\mathtt{www}}_B h_2.\sum_u \mathtt{connect}(u))$$

That is, the history expression for the `Browser` receiver (H_B), the `Game` activity (H_G) and a (simplified) user interface (H_U). Terms H_G and H_B are obtained by typing methods `receive` and `play` of `Browser` and `Game`, respectively. Instead, H_U represents the user's behavior (for instance, we could obtain it by typing the user interface component). Informally, the history expression H_U says that the user can iteratively open the browser on the page “`home.com`” ($\mathtt{www}_?(\text{“}\mathtt{home.com}\text{”})$).

It is easy to verify that, assuming $\rho(\mathtt{www}) = \emptyset$, $H_U \parallel H_G \models \varphi$. Indeed, none of the two history expressions H_U and H_G can perform a `connect` action, needed to violate the policy. Moreover, assuming $\rho(\mathtt{www}) = \{H_B\}$, also $H_U \models \varphi$ is trivially satisfied (neither `read` nor `write` can take place). However, by using PMC, we show that under $\rho(\mathtt{www}) = \{H_B\}$, $H_U \parallel H_G \not\models \varphi$. In our context, this corresponds to showing that the installation of `Game` is not allowed by φ on a system already running `Browser` (and the user interface).

First we compute $\varphi_{/\!/H_U}$ as follows.

$$\varphi_{/\!/H_U} = \overbrace{([\mathtt{read(x)}].[\mathtt{connect(y)}].\mathit{ff})_{/\!/H_U}}^{A} \wedge \underbrace{([\mathtt{write(x)}].[\mathtt{connect(y)}].\mathit{ff})_{/\!/H_U}}_{B}$$

Since H_U does not admit transitions of the kind $\xrightarrow{\mathtt{read}(\cdot)}$, by applying the PMC rules for box operator to A we obtain

$$A = [\mathtt{read(x)}].([\mathtt{connect(y)}].\mathit{ff})_{/\!/H_U}$$

However, we observe that

$$H_U \equiv \mathtt{connect}(\text{“}\mathtt{home.com}\text{”}) + \mathtt{connect}(\text{“}\mathtt{ad.com}\text{”}) + H_U$$

Hence, we have that $([\mathtt{connect(y)}].\mathit{ff})_{/\!/H_U}$ reduces to

$$[\mathtt{connect(y)}].\mathit{ff} \wedge \mathit{ff}$$

which is trivially equivalent to ff. Symmetrically, we apply the same reasoning to B and obtain $\varphi_{/\!/H_U} = [\mathtt{read(x)}].\mathit{ff} \wedge [\mathtt{write(x)}].\mathit{ff}$. Eventually, since every trace of H_G starts with a $\mathtt{read}(\cdots)$ action, we easily verify that $H_G \not\models \varphi_{/\!/H_U}$.

5 BYODroid

We implemented a prototype, namely *BYODroid*, based on the techniques introduced in previous sections. Apart from the PMC procedure, the BYODroid workflow is fully implemented. At the best of our knowledge, no usable tools for partial model checking are publicly available (a proof-of-concept implementation was presented in [2], but no actual prototype was released). Hence, we needed to develop one from scratch. Although we already have a prototype implementation, it has not been integrated yet. This activity required further investigations, laying out of the scope of the current work, that we plan to present in future publications. For the time being, the prototype handles device configurations by means of models composition. As discussed above, we expect better performances when integrating the PMC step.

We used BYODroid to assess the compliance of a set of actual Android applications against sample BYOD policies.

We now briefly illustrate the technologies we used in the development of *BYODroid* and discuss their relation withe the techniques we introduced in the previous sections. We then present the results of an experimental evaluation obtained by running *BYODroid* to check a number of Android applications against the policies of Example 6 and Example 4.

Technological background.

BYODroid works on a model of the target mobile application H and a security policy φ. Even though we can assume φ to be statically defined, *BYODroid* must build a model of the mobile application. The type and effect system we presented in Section 3 can be used to extract history expressions from the application code. However, Android applications are packaged in APK archives consisting of Android bytecode, called Dalvik bytecode. Since the instruction set of the Dalvik bytecode is very large, building a type and effect system for it would be a daunting task. For this reason we opted for generating history expressions from the *Control Flow Graph* (CFG) of the application[7]. CFGs provide a substantial simplification w.r.t. the Dalvik bytecode representation of the application. Nodes of the CFGs can be either simple (representing a linear, jump-free sequence of instructions) or branching (representing conditional or unconditional jump instructions). Following [8] we obtain a history expression by converting the CFG. This is done by mapping the elements of the CFG in operators and terms of the history expressions. For instance, a sequence of simple nodes can be represented through the $\cdot$ operator, while a branching node can result in a $+$ operator. Similarly, loops can lead to recursive μ terms.

It is worth pointing out that BYODroid also carries out a number of optimizing transformations on the CFG. For instance, it prunes branching nodes carrying no security-relevant information, thereby reducing the size of the model. The resulting CFG is then converted to a suitable history expression by means of the procedure sketched above.

Verification is the second key procedure of our proposal. Nowadays, model checking is a mature technology for the automatic verification of concurrent systems. Among the existing model checkers, SPIN [15] is a major proposal. Encoding our verification problem $H \models \varphi$ as a SPIN specification, namely *Promela*, requires to translate both H and φ to Promela agents. Since history expressions are endowed with a LTS semantics, the first part is straightforward. Moreover, by exploiting the operational semantics provided in [28, 17] we translate a formula φ to a corresponding transition system and implement an equivalent Promela agent. In particular, we follow the approach of Martinelli and Matteucci [20] for generating a *maximal controller*, i.e., the most general agent (according to a simulation relation) enforcing a given security policy.

Once we have agents for both the application and the policy we encode them in Promela through parallel composition. Then, we use unsatisfiable assertions to label policy-violating states. Thus, if some of these states are reachable, the model checker reports a policy violation.

BYODroid Architecture.

The BYODroid solution consists of the *BYODroid Server*, in charge of history expressions extraction and policy verification, and the *BYODroid Installer*, a client application that each mobile device must install before joining the organization.

The organization installs the BYODroid Server in a suitable location of its infrastructure. Then, it defines the BYOD security policy regulating the access to the critical assets. Finally, it asks the users to install the BYODroid Installer on their own device. The BYODroid Installer interacts with the BYODroid Server to handle the installation and removal of applications. Once installed, it replaces the native system installer, analyzes the device configuration (i.e. the currently installed applications), and then sends it to the BYODroid Server for the registration process. If the new configuration satisfies the policy of the BYOD Server, the device is allowed to enter and the registration procedure successfully completes. Otherwise, the user is notified about the applications causing the rejection and is asked whether she wants to remove them and try again. A user interacts with the BYODroid Installer by means of a GUI that presents a view of applications available for installation. The BYODroid Server is responsible for providing the BYODroid Installer this list. Also, it manages the BYOD policy, the authorized mobile devices and mediates the access to application stores (e.g. Google Play and Samsung Store). Clearly, the BYODroid Server only permits a connected device to install applications that comply with the BYOD policy.

At the core of the BYODroid Server lies the *security policy manager* which mainly maintains the security state of each device. The security policy manager builds a Direct Acyclic Graph (DAG), rooted in the BYOD policy φ. The DAG is defined as $G = \langle N, r, A, E \rangle$ where N is a finite set of nodes with $r \in N$ the root node, A is a finite set of application names and $E \subseteq N \times A \times N$ is a set of edges.

Nodes $n, m \in N$ are triples of the form $n = \langle \varphi_n, R_n, D_n \rangle$, where φ_n is the node policy, $R_n \subseteq A$ is a set of proved *unsafe* applications, i.e. those found to not satisfy φ_n, and D_n is a set of identifiers of devices associated with n. Intuitively if $\delta \in D_n$, then each new installation request coming from

[7] Tools capable to automatically build a CFG of any given APK application already exist and are publicly available. In our implementation we use Androguard (`https://code.google.com/p/androguard/`) to this end.

δ must be validated against φ_n, that is φ_n represents the security configuration of δ.

A policy φ_n is obtained through partial model checking (see Section 4) of the policy φ_m where m is the parent of n[8]. Hence, an edge $\langle n, a, m\rangle$ denotes that for a device $\delta \in D_n$ an application a has been proved to be *safe* for installation.

Whenever a new device δ registers to the BYODroid Server, the BYODroid Installer generates the set I of installed applications. Then, starting from node r, the security policy manager looks for a path $\langle r, a_1, n_1\rangle \cdots \langle n_{k-1}, a_k, n_k\rangle$ such that $I = \{a_1, \ldots, a_k\}$. Note that if no such path exists, fresh nodes are added to create it (see below).

When δ queries for a view of installable applications, the security policy manager behaves as follows. First δ is located in G, i.e. the system finds the node n such that $\delta \in D_n$. Meanwhile, the BYOD Server queries the application stores for applications that can be installed on δ; let be B the set of retrieved applications. Then for each a appearing in B we have three possibilities:

- $\exists m.\langle n, a, m\rangle \in E$. In this case application a has been already verified against φ_n. Thus, an entry a:"safe" is added to the application view.
- $a \in R_n$. This means that a is known to violate φ_n and, consequently, it is discharged.
- Otherwise, the system has no suitable data about a and an entry a:"unchecked" is added to the view.

If δ decides to install an application a labeled as "safe", the BYODroid Server retrieves the package of a and sends it to δ. Then the BYODroid Installer running on δ simply installs it. Eventually δ is moved from node n to m, i.e. $D_n \setminus \{\delta\} \mapsto D_n$ and $D_m \cup \{\delta\} \mapsto D_m$.

The request for an "unchecked" application requires more attention. As a matter of fact, the validity of a against the configuration of δ must be verified. Then we proceed as follows. The BYODroid Server extracts the CFG from application a and converts it to a corresponding history expression H.

Then the system creates a suitable Promela encoding and uses SPIN to check whether $H \models \varphi_n$. If $H \not\models \varphi_n$, the installation of a is rejected and a is added to R_n.

Otherwise the installation of a continues as described before and the graph G is updated as follows: a new node $m = \langle \varphi_{n/\!\!/H}, R_n, \{\delta\}\rangle$ is created for the new configuration of δ and δ is removed from D_n. Also an edge $\langle n, a, m\rangle$ is added to E to denote that now a is known to be safe for installation in n. The whole workflow is depicted in Figure 1.

Moreover, a device δ is allowed to remove an installed application a. In this case, the BYODroid Server *(i)* locates δ in G, i.e. it retrieves the node m s.t. $\delta \in D_m$, *(ii)* looks for a transition $e \in E$ s.t. $\langle n, a, m\rangle$, *(iii)* moves δ from m to n and finally *(iv)* triggers the BYODroid Installer for the deletion of a.

[8] Since the graph is built by iteratively adding nodes to an existing graph, being $\langle\{r\}, r, A, \emptyset\rangle$ the starting one, we can always assume a node $n \neq r$ to have a parent.

Experimental Evaluation.

We have tested the BYODroid architecture against the security policies of Example 6 and Example 4[9], using 261 real-world Android applications. The dataset of applications has been acquired from the top free chart of the *Google Play Store*.

The experiments have been carried out on a Debian 6.0.7 Server running on a Intel Core i7-870 @2.93 GHz with 8 GB RAM system. Table 5 shows an extract of our experimental results[10].

For each application, the BYODroid Server first extracts its CFG. In Table 5 we report the dimension of the application package, the time needed to extract the model (T_{ext}) and the number of nodes and edges belonging to the CFG, i.e., its size. For instance, the GO Launcher application (line 14) has a 0.23 MB package, required 0.18 seconds for the extraction which generated an empty (i.e., 0 nodes/edges) CFG[11].

CFGs are then turned into Promela agents. The encoding times are reported in column T_{enc}. Finally, the BYODroid Server executes the verification procedure using SPIN. The verification consists of three steps. First, SPIN generates a C file containing the verification procedure, then a standard C compiler is used to build a corresponding executable which is eventually launched. Since we do not admit unbounded execution times, we forced the verification program (third step) to terminate within a given threshold (i.e. one minute). The model checking execution time is reported in column T_{mc} of Table 5. We write TO for computations that caused a time out by reaching the one minute threshold.

The output of the model checker is given in column Valid. We reported *YES* whenever $H \models \varphi_n$, *NO*, in case of a policy violation or a time out. When present, we also provide the length of the execution trace causing the policy violation between parentheses.

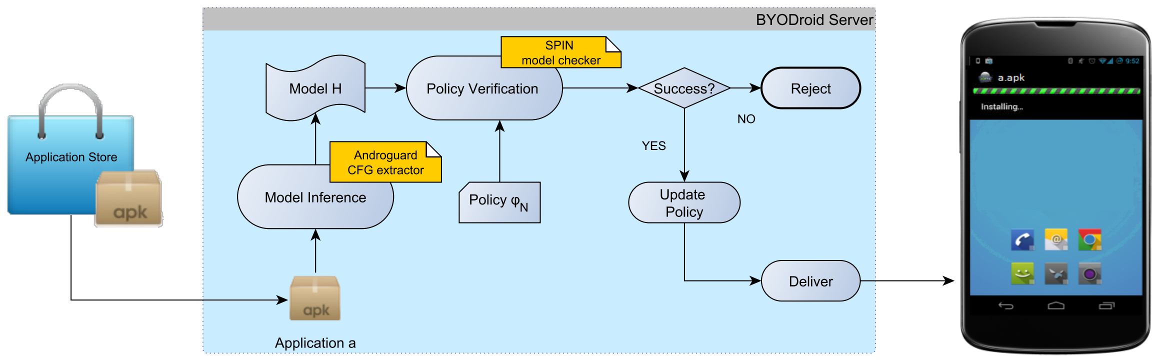
The global values in Table 5 summarize the overall statistics of our experiments. BYODroid could successfully evaluate the 88.9% of the applications in the dataset with an average time of 2 minutes and 54 seconds ($\mu T_{ext} + \mu T_{enc} + \mu T_{enc}$). Notice that this overhead is paid only once when installing a new application.

[9] Where abstract action names, e.g., `openSSL(...)`, have been mapped into corresponding, concrete Java/Android APIs, e.g., `SSLSocketFactory.generateSocket(...)`.

[10] The complete list of experimental results is available at: `http://www.ai-lab.it/byodroid/experiments_IJIS.html`

[11] Empty CFGs arise when an application contains no security-relevant operations/invocations

Fig. 1 Workflow of the installation of an unchecked application a



Summing up, the experimental results look promising and show applicability of the BYODroid architecture to a real scenario. Comparing the structure of our prototype and the outcome of our experiments with the purposes of our framework, we can make the following observations.

1. Most of the security analysis procedure is transparent to the user. Nevertheless, users can notice a delay at install-time. Although, we can expect that they are correctly informed about this process, real implementations should reduce as much as possible this overhead. Since our prototype admits further optimizations, we are optimistic on the possibility of reducing it.
2. The verification process is fully automatic. Both users and organizations are not required to take any critical decision.
3. Application collusion is disabled. Security policies are evaluated against models in which the notion of application is abstracted away. Model composition is obtained by simply juxtaposing the corresponding Promela encodings. Clearly, model checking performances can suffer from the model size increment. In this respect, PMC may provide important simplifications that deserve to be evaluated through further experiments.
4. Standard users cannot compromise the safety of their device with respect to the BYOD policy. Nevertheless, malicious (clumsy) users could attempt to tamper the device intentionally (by mistake). Hence, monitoring facilities must be considered for preventing dangerous actions like the manual installation of application packages.

6 Related work

Due to the wide diffusion of smartphones, a substantial amount of research in security is now focusing on the existing mobile platforms. Most of this activity addresses the security of mobile OSs and, in particular, on Android, being the most common one. Most of the literature about Android security consists of proposals for *(i)* extending the native security policy, *(ii)* enhancing the Android Security Framework (ASF) with new tools for specific security-related checks, and *(iii)* detecting vulnerabilities and security threats. Regarding the first category, in [24] the (informal) Android security policy is analyzed in terms of efficacy and some extensions are proposed. Besides, in [21] the authors propose an extension to the basic Android permission systems and some new policies built on top of the extended version. Moreover, in [29] new privacy-related security policies are proposed for addressing security problems related to users' personal data.

Related to ASF enhancement, many proposals have been made to extend the native Android security mechanisms. For instance, [22] proposes a method for monitoring the Android permissions system by properly customizing the Android stack. Authors in [13] present Scandroid, an application certification tool verifying whether the actual application's data flows are coherent with its manifest. A similar approach [12] assesses the actual privileges of Android applications by analyzing their intent-based communications.

Also, Android-specific malware detection tools have been proposed. For instance, Crowdroid [10] is a malware detector executing a dynamic analysis of the applications' behavior, while DroidRanger [30] is an application stores analyzer that combines a footprint-based detection engine of known malware with a heuristic detection engine for zero-day malware attacks. Besides, other work has successfully detected vulnerabilities related to DoS attacks [5], covert channels [23] and privilege escalation attacks [9].

The effectiveness and soundness of previous approaches have been advocated through empirical assessments, without formal bases. Clearly, without strong, mathematical foundation there is no guarantee about the effectiveness of an approach. Recently, researchers focused on the formal modeling and analysis of the Android platform and its security aspects. In [25] the authors formalize the permission scheme of Android. Briefly, their formalization consists of a state-

Table 5 Computation times and outcomes per application when applying the policy of Example 6.

#	Application	Size (MB)	T_{ext} (s)	Nodes	Edges	T_{enc} (s)	T_{mc} (s)	Valid
1	Adobe Reader	6.63	14.23	44	158	0.85	0.40	NO (63)
2	Angry Birds	33.80	197.7	232	807	13.00	TO	NO (-)
3	Angry Birds Rio	32.61	189.83	232	807	13.06	TO	NO (-)
4	Angry Birds Seasons	42.27	190.77	251	837	13.22	0.51	NO (73)
5	ANSA	1.93	56.19	71	258	1.79	0.42	YES
6	Browser Chrome - Google	24.6	29.76	65	267	2.35	0.58	NO (84)
7	Dropbox	5.59	32.52	79	295	2.25	0.44	YES
8	Facebook	15.08	24.7	26	108	0.51	0.36	NO (61)
9	FB Messenger	11.98	145.43	112	449	4.85	0.43	NO (67)
10	Firefox	23.13	35.16	63	216	1.59	0.46	YES
11	Fruit Ninja	18.33	69.34	120	420	3.82	0.98	NO (129)
12	Gmail	3.56	64.67	98	381	3.62	0.48	YES
13	Google Street View	0.25	2.87	13	54	0.21	0.36	YES
14	GO Launcher	0.23	0.18	0	0	0.03	0.36	YES
15	Instagram	14.87	47.91	56	223	1.56	0.48	NO (199)
16	LinkedIn	6.55	136.78	170	626	9.61	0.38	YES
17	Mobile Banking Unicredit	2.32	13.33	20	93	0.42	0.38	YES
18	Skype	14.73	54.82	82	277	1.86	0.38	NO (62)
19	Spotify	3.64	17.07	49	186	1.06	0.39	YES
20	Tiny Flashlight	1.28	61.36	112	374	2.92	0.40	YES
Global values								
# Applications		$\mu Size$	μT_{ext}	μN	μE	μT_{enc}	μT_{mc}	V/TO (%)
261		11.41	156.27	139.88	484.94	8.39	9.18	88.9/11.1

based model representing entities, relations and constraints over them. Also, they show how their formalism can be used to automatically verify that the permissions are respected. Unlike our proposal, their language only describes permissions and obligations and does not capture application interactions which we infer from actual implementations. In particular, their framework provides no notion of interaction with the platform, while we represent it through system calls. Similarly to the present work, Chaudhuri [11] proposes a language-based approach to infer security properties from Android applications. Moreover, this work presents a type system that guarantees that well-typed programs respect user data access permissions. The type and effect system that we presented here exceeds the proposal of [11] as it also infers/verifies history expressions. History expressions can denote complex interactions/behaviors and they allow for the verification against a rich class of security policies [1]. To the best of our knowledge, our proposal is the first one to tackle the problem of modeling and validating the behavior of Android applications against customized security policies in a formal, non-invasive way.

7 Conclusion

In this work we presented a framework for modeling the behavior of Android applications and verifying their compliance w.r.t. security policies. We applied this method in a corporate context, in order to allow organizations to define and enforce BYOD security policies on employees' personal devices. The models we produce, namely history expressions, are safe in the sense that they correctly represent all the possible runtime computations of the applications they come from. Moreover, we significantly improved the theoretical results previously proposed in [6] by extending the type and effect system, the HML-based policy language and the PMC rules. Also, we have completed the implementation of BYODroid, a prototypical implementation of the proposed security framework, by adopting state-of-the-art solutions and tools. Finally, we used BYODroid to empirically assess the feasibility of the approach by testing a BYOD policy against 261 free Android applications.

Future developments will be focused on three main open issues: (i) the theoretical results of the PMC technique must be effectively implemented in an actual system, (ii) the performance of the analysis must be evaluated on groups of applications (i.e., representing device configurations) instead of single applications, and (iii) techniques for instrumenting non-compliant applications must be investigated.

References

1. Abadi M, Fournet C (2003) Access control based on execution history. In: Proceedings of the 10th annual Network and Distributed System Security Symposium, pp 107–121
2. Andersen H, Lind-Nielsen J (1996) MuDiv: A Tool for Partial Model Checking. In: CONCUR

3. Andersen HR (1995) Partial model checking (extended abstract). In: In Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, pp 398–407
4. Armando A, Costa G, Merlo A (2012) Formal modeling and reasoning about the Android security framework. In: Proceedings of Seventh International Symposium on Trustworthy Global Computing
5. Armando A, Merlo A, Migliardi M, Verderame L (2012) Would you mind forking this process? A denial of service attack on Android (and some countermeasures). In: Proc. of the 27th IFIP International Information Security and Privacy Conference (SEC 2012), pp 13–24
6. Armando A, Costa G, Merlo A (2013) Bring your own device, securely. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, ACM, New York, NY, USA, SAC '13, pp 1852–1858, DOI 10.1145/2480362.2480707
7. Bartoletti M, Degano P, Ferrari GL (2005) History-based access control with local policies. In: FoSSaCS, pp 316–332
8. Bartoletti M, Costa G, Degano P, Martinelli F, Zunino R (2009) Securing Java with Local Policies. Journal of Object Technology 8(4):5–32
9. Bugiel S, Davi L, Dmitrienko A, Fischer T, Sadeghi AR (2011) Xmandroid: A new Android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Univ. Darmstadt, URL `http://www-infsec.cs.uni-saarland.de/~bugiel/publications/pdfs/XManDroid-tr-2011-04.pdf`
10. Burguera I, Zurutuza U, Nadjm-Therani S (2011) Crowdroid: behavior-based malware detection system for Android. In: Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM'11)
11. Chaudhuri A (2009) Language-based security on Android. In: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, ACM, New York, NY, USA, PLAS '09, pp 1–7
12. Chin E, Felt AP, Greenwood K, Wagner D (2011) Analyzing inter-application communication in Android. In: Proceedings of the 9th international conference on Mobile systems, applications, and services, ACM, New York, NY, USA, MobiSys '11, pp 239–252, DOI 10.1145/1999995.2000018
13. Fuchs AP, Chaudhuri A, Foster JS (2011) Scandroid: Automated security certification of Android applications. Tech. rep., URL `http://www.cs.umd.edu/~avik/projects/scandroidascaa/`
14. Hennessy M, Milner R (1980) On Observing Nondeterminism and Concurrency. In: Proceedings of the 7th Colloquium on Automata, Languages and Programming, Springer-Verlag, London, UK, pp 299–309
15. Holzmann G (2003) The Spin model checker: primer and reference manual, 1st edn. Addison-Wesley Professional
16. Igarashi A, Pierce BC, Wadler P (1999) Featherweight Java: A Minimal Core Calculus for Java and GJ. In: ACM Transactions on Programming Languages and Systems, pp 132–146
17. Janin D, Walukiewicz I (1995) Automata for the modal mu-calculus and related results. In: Wiedermann J, Hájek P (eds) MFCS, Springer, Lecture Notes in Computer Science, vol 969, pp 552–562
18. Lamport L (1977) Proving the correctness of multiprocess programs. IEEE Trans Softw Eng 3(2):125–143, DOI 10.1109/TSE.1977.229904
19. Larsen KG (1988) Proof system for Hennessy-Milner logic with recursion. In: Dauchet M, Nivat M (eds) CAAP, Springer, Lecture Notes in Computer Science, vol 299, pp 215–230
20. Martinelli F, Matteucci I (2007) Through modeling to synthesis of security automata. Electronic Notes in Theoretical Computer Science 179:31 – 46
21. Nauman M, Khan S, Zhang X (2010) 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, ACM, New York, NY, USA, ASIACCS '10, pp 328–332, DOI http://doi.acm.org/10.1145/1755688.1755732
22. Ongtang M, Mclaughlin S, Enck W, Mcdaniel P (2009) Semantically rich application-centric security in Android. In: In ACSAC '09: Annual Computer Security Applications Conference
23. Schlegel R, Zhang K, Zhou X, Intwala M, Kapadia A, Wang X (2011) Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In: Proceedings of the 18th Annual Network & Distributed System Security Symposium
24. Shabtai A, Fledel Y, Kanonov U, Elovici Y, Dolev S, Glezer C (2010) Google Android: A comprehensive security assessment. Security Privacy, IEEE 8(2):35 –44, DOI 10.1109/MSP.2010.2
25. Shin W, Kiyomoto S, Fukushima K, Tanaka T (2010) A Formal Model to Analyze the Permission Authorization and Enforcement in the Android Framework. In: Proceedings of the 2010 IEEE Second International Conference on Social Computing, IEEE Computer Society, Washington, DC, USA, SOCIALCOM '10, pp 944–951
26. Skalka C, Smith S (2004) History effects and verification. In: Second ASIAN Symposium on Programming

Languages and Systems (APLAS), Springer, pp 107–128

27. Skalka C, Smith S, Van Horn D (2005) A Type and Effect System for Flexible Abstract Interpretation of Java. Electronic Notes in Theorical Computer Science 131:111–124
28. Streett RS, Emerson EA (1989) An automata theoretic decision procedure for the propositional mu-calculus. Inf Comput 81(3):249–264
29. Zhou Y, Zhang X, Jiang X, Freeh VW (2011) Taming information-stealing smartphone applications (on Android). In: Proceedings of the 4th international conference on Trust and trustworthy computing, TRUST'11, pp 93–107
30. Zhou Y, Wang Z, Zhou W, Jiang X (2012) Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In: Proceedings of the 19th Annual Network & Distributed System Security Symposium

Proofs

Lemma 1 *For each* closed *(i.e., without free variables) expression E, environment Γ, history expression H, type τ and trace ω, if $\Gamma \vdash E : \tau \triangleright H$ then either E is a value or $\omega, E \to \omega', E'$ (for some ω', E').*

Proof By induction over the structure of E.

– Case $\texttt{null}$, u, x. Trivial.
– Case $\texttt{new}\,\texttt{C}(\bar{E})$. If each $E_i = v_i$ then E is a value and the property holds. Otherwise it suffices to apply the inductive hypothesis to the first $E_i \neq v$.
– Case $E'.\texttt{f}$. If $E' \neq v$ we apply the inductive hypothesis. Otherwise, by $\texttt{T}{-}\texttt{FLD}$ we know that $E' = \texttt{new}\,\texttt{C}(\bar{v})$ and we can conclude by applying $\texttt{FLD}_2$.
– Case $E'.\texttt{m}(E'')$. If both $E' = v'$ and $E'' = v''$ we assume the premises of $\texttt{T}{-}\texttt{METH}$ and we can apply $\texttt{METH}_3$. Instead if either $E' \neq v'$ or $E'' \neq v''$ we can directly apply the inductive hypothesis to $\texttt{METH}_1$ and $\texttt{METH}_2$.
– Case $\texttt{system}_\sigma\, E'$. If E' is not a value, we apply the inductive hypothesis and rule $\texttt{SYS}_1$. Otherwise, by $\texttt{T}{-}\texttt{SYS}$ we know that $\Gamma \vdash E' : \mathcal{U} \triangleright H'$ and the only suitable value for E' is a resource u. Thus, we conclude by applying $\texttt{SYS}_2$.
– Case $\texttt{icast}\, E'$ and $\texttt{ecast}\,\texttt{C}\, E'$. Similar to the previous step.
– Case $I_\alpha(E')$. If $E' = v$ then E is also a value. Instead, if E' is not a value, we apply the inductive hypothesis and rule $\texttt{INT}$.
– Case $E'.\texttt{data}$. If E' is a value, by $\texttt{T}{-}\texttt{DATA}$ it must be $E' = I_\alpha(v)$. Hence, we conclude by applying rule $\texttt{DATA}_2$. Otherwise, we simply apply the inductive hypothesis and rule $\texttt{DATA}_1$.
– Case $\texttt{if}\ (E' = E'')\ \texttt{then}\,\{E_{tt}\}\,\texttt{else}\,\{E_{f\!f}\}$. If one between E' and E'' is not a value, we conclude by applying the inductive hypothesis. Otherwise, we can apply either $\texttt{IF}_3$ or $\texttt{IF}_4$. In both cases, the property holds.
– Case $E'; E''$. Here we have two possibilities: either E' is a value or not. In both cases, E admits reduction (through $\texttt{SEQ}_1$ and $\texttt{SEQ}_2$, respectively).
– Case $(\texttt{C})\texttt{E}'$. If E' is not a value, it suffices to apply the inductive hypothesis and rule $\texttt{CAST}_1$. On the other hand, by rule $\texttt{T}{-}\texttt{CAST}$ we know that $E' = \texttt{new}\,\texttt{D}(\cdots)$ and $D <: C$. Thus, we can apply $\texttt{CAST}_2$ and conclude.
– Case $\texttt{thread}\,\{E'\}\,\texttt{in}\,\{E''\}$. Again, if either E' or E'' are not values, we use the inductive hypothesis and rule $\texttt{PAR}_1$ or $\texttt{PAR}_2$. Instead, if $E' = v'$ and $E'' = v''$ we conclude by applying rule $\texttt{PAR}_3$.

□

Definition 4

$$[\![H]\!] = \{\omega \mid \exists H'.H \xrightarrow{\omega}{}^{*} H'\}$$

Lemma 2 *For each* closed *expression E, environment Γ, history expression H, type τ and trace ω, if $\Gamma \vdash E : \tau \triangleright H$ and $\omega, E \to^* \omega', E'$ then $\Gamma \vdash E' : \tau \triangleright H'$ and for all $\dot{\omega}' \in [\![H']\!]$ there exists $\dot{\omega} \in [\![H]\!]$ such that $\omega\dot{\omega} = \omega'\dot{\omega}'$.*

Proof We first (1) prove the property for a single step and then (2) we extend it to arbitrary long derivations.

1. By induction on the structure of E.
   – Case $\texttt{null}$, u, x. Trivial.
   – Case $\texttt{new}\,\texttt{C}(\bar{E})$. If each $E_i = v_i$ then E is a value and the property holds. Otherwise we type E in this way

$$\frac{\Gamma \vdash E_i : \tau_i \triangleright H_i}{\Gamma \vdash \texttt{new}\,\texttt{C}(\bar{E}) : \tau \triangleright \varepsilon \cdots H_j \cdots}$$

   where E_j is the first non-value in $\bar{E}$. Applying Lemma 1 we know that $\omega, E_j \to \omega', E'_j$. Hence, we apply the inductive hypothesis obtaining $\Gamma \vdash E'_j : \tau_j \triangleright H'_j$ and $\forall\omega'_j \in [\![H'_j]\!].\exists\omega_j \in [\![H_j]\!].\omega\omega_j = \omega'\omega'_j$. Then, by instantiating rule $\texttt{NEW}$ we have

$$\omega, \texttt{new}\,\texttt{C}(\bar{v}, E_j, \ldots) \to \omega', \texttt{new}\,\texttt{C}(\bar{v}, E'_j, \ldots)$$

   Since $A = [\![\varepsilon \cdots H_j \cdots]\!] = \{\omega_j\omega_{j+1} \cdots \mid \omega_j \in [\![H_j]\!] \wedge \omega_{j+1} \in [\![H_{j+1}]\!] \wedge \cdots\}$ and $B = [\![\varepsilon \cdots H'_j \cdots]\!] = \{\omega'_j\omega_{j+1} \cdots \mid \omega'_j \in [\![H'_j]\!] \wedge \omega_{j+1} \in [\![H_{j+1}]\!] \wedge \cdots\}$ (where the $\cdots$ parts are equal) we can conclude by observing that $\forall\dot{\omega}' = \omega'_j \cdots \in B$ we can find a trace $\dot{\omega} = \omega_j \cdots \in B$ such that $\omega\dot{\omega} = \omega'\dot{\omega}'$
   – Case $E'.\texttt{f}$. If $E' \neq v$ we just apply the inductive hypothesis. Otherwise, by $\texttt{T}{-}\texttt{FLD}$ we know that $E' = \texttt{new}\,\texttt{C}(\bar{v})$ and by applying $\texttt{FLD}_2$ we obtain $\omega, \texttt{new}\,\texttt{C}(\bar{v}).\texttt{f} \to \omega, v_f$ which trivially satisfies the property.
   – Case $E'.\texttt{m}(E'')$. If either $E' = v'$ or $E'' = v''$ we assume the premises of $\texttt{T}{-}\texttt{METH}$ and we can apply the inductive hypothesis and $\texttt{METH}_1$/$\texttt{METH}_2$. Instead if both $E' = \texttt{new}\,\texttt{C}(\bar{v})$ and $E'' = v'$ we apply $\texttt{METH}_3$ and we have

$$\frac{\mathit{mbody}(\texttt{m}, \texttt{C}) = \texttt{x}, E_m}{\omega, (\texttt{new}\,\texttt{C}(\bar{v})).\texttt{m}(v') \to \omega, E_m[v'/x, (\texttt{new}\,\texttt{C}(\bar{v}))/\texttt{this}]}$$

   However, typing the two sides of this transition we obtain the same history expression. Indeed, for each history expression H_m produced by typing the right side, we have for the left part $\varepsilon \cdot \varepsilon \cdot H_m$ which is trivially equivalent.
   – Case $\texttt{system}_\sigma\, E'$. If E' is not a value, we apply the inductive hypothesis and rule $\texttt{SYS}_1$. Otherwise, by $\texttt{SYS}_2$ we have $\omega, \texttt{system}_\sigma\, u \to \omega \cdot \sigma(u), \texttt{null}$. Also by $\texttt{T}{-}\texttt{SYS}$ we have $\Gamma \vdash E : \mathbf{1} \triangleright \sigma(u)$. Since $\Gamma \vdash \texttt{null} : \mathbf{1} \triangleright \varepsilon$, we have to show that $\exists\dot{\omega} \in \{\sigma(u)\}.\omega\sigma(u) = \omega\dot{\omega}$ which trivially holds.
   – Case $\texttt{icast}\, E'$. Similarly to the previous step, if E' is not a value, we can simply apply the inductive hypothesis. Otherwise, by $\texttt{T}{-}\texttt{IMPC}$ we must have $E' = I_\alpha(u)$ and $H = \alpha_?(u)$. Also, by $\texttt{IMPC}_2$ we have

$$\frac{\texttt{new}\, C(\bar{v}) \in \mathit{receiver}(\alpha)}{\omega, \texttt{icast}\, I_\alpha(u) \to \omega, \texttt{new}\, C(\bar{v}).\texttt{receive}(I_\alpha(u))}$$

and we type $\Gamma \vdash \texttt{new}\, C(\bar{v}).\texttt{receive}(I_\alpha(u)) : \mathbf{1} \triangleright \tilde{H}$. Hence we must prove that $\forall \dot{\omega}' \in [\![\tilde{H}]\!].\exists \dot{\omega} \in [\![\alpha_?(u)]\!]$ such that $\omega\dot{\omega} = \omega\dot{\omega}'$. However, by the semantics of history expressions we have

$$\frac{\alpha_?(u) \xrightarrow{\alpha_?(u)} \varepsilon \qquad \dot{H} = \sum H'\{\alpha_?(u)/h\} \text{ s.t. } \bar{\alpha}_C h.H' \in \rho(\alpha) \text{ and } \chi \not\succcurlyeq C}{H \dashrightarrow \dot{H}}$$

Then, by definition of ρ, we have that $\tilde{H} \sqsubseteq \dot{H}$ which suffices to conclude.

- Case $\texttt{ecast}\,\texttt{C}\,E'$. Analogous to the previous case.
- Case $I_\alpha(E')$. If $E' = v$ then E is also a value. Instead, if E' is not a value, we apply the inductive hypothesis and rule $\texttt{INT}$.
- Case $E'.\texttt{data}$. If E' is a value, by $\texttt{T-DATA}$ it must be $E' = I_\alpha(v)$. Hence, we conclude by applying rule $\texttt{DATA}_2$. Otherwise, we simply apply the inductive hypothesis and rule $\texttt{DATA}_1$.
- Case $\texttt{if}\,(E' = E'')\,\texttt{then}\,\{E_{tt}\}\,\texttt{else}\,\{E_{\mathit{ff}}\}$. If either E' or E'' are not values, we conclude by applying the inductive hypothesis. Otherwise, we can apply either $\texttt{IF}_3$ or $\texttt{IF}_4$ and the inductive hypothesis to E_{tt} and E_{ff}, respectively.
- Case $E'; E''$. Here we have two possibilities: either E' is a value or not. In both cases, E admits reduction (through $\texttt{SEQ}_1$ and $\texttt{SEQ}_2$, respectively). The property holds by the inductive hypothesis.
- Case $\texttt{(C)E}'$. If E' is not a value, it suffices to apply the inductive hypothesis and rule $\texttt{CAST}_1$. On the other hand, by rule $\texttt{T-CAST}$ we know that $E' = \texttt{new}\,\texttt{D}(\cdots)$ and $D <: C$. Thus, we can apply $\texttt{CAST}_2$ and conclude.
- Case $\texttt{thread}\,\{E'\}\,\texttt{in}\,\{E''\}$. If either E' or E'' are not values, we use the inductive hypothesis and rule $\texttt{PAR}_1$ or $\texttt{PAR}_2$. Instead, if $E' = v'$ and $E'' = v''$ we conclude by applying rule $\texttt{PAR}_3$.

2. By induction on the derivation length.
   - Base case. For zero-long derivations the property is trivially satisfied by the identity $\omega\dot{\omega} = \omega\dot{\omega}$.
   - Induction. We assume the property holds for n-long derivations and we apply (1) to prove that the property is preserved by a further step.

□

Theorem 1 *For each* closed *expression E, history expression H, type τ and trace ω, if $\emptyset \vdash E : \tau \triangleright H$ and $\cdot, E \rightarrow^* \omega, E'$ then there exist H' and ω' such that $H \xrightarrow{\omega'}{}^* H'$, $\emptyset \vdash E' : \tau \triangleright H'$ and $\omega = \omega'$.*

Proof A corollary of Lemma 2. □

Theorem 2 $H \models \varphi \iff H \Vdash \varphi$

Proof We prove the two directions $\Leftarrow$ (soundness) and $\Rightarrow$ (completeness) separately.

$\Leftarrow$) By induction over the definition of $\Vdash$.
- Case (true). Trivial.
- Case ($\wedge$). We simply assume the premises of the rule and apply the inductive hypothesis to them.
- Cases ($\vee_L$) and ($\vee_R$). In both cases we assume the rule premise and apply the inductive hypothesis.
- Case ($\langle u \rangle$). We assume the premises of the rule and we apply the inductive hypothesis to $H' \Vdash \varphi$. Hence there exists H' such that $H \xrightarrow{\sigma(u)} H'$ and $H' \models \varphi$ which allows us to apply rule (c-diamond).
- Case ($\langle x \rangle$). We apply a similar reasoning and we obtain that there exist u and H' such that $H \xrightarrow{\sigma(u)} H'$ and $H' \models \varphi\{u/x\}$ so that we can apply (a-diamond) and conclude.
- Case ($[u]$). We assume the premises of the rule and apply the inductive hypothesis to them. By definition, if $H \xrightarrow{\sigma(u)} H'$ then H' appears in one of the premises of the rule. This suffices to conclude by applying (c-box).
- Case ($[x]$). As in the previous step, we assume the premises and we apply to them the inductive hypothesis. Again, if we have $H \xrightarrow{\sigma(u)} H'$ (for some u and H'), then $H' \Vdash \varphi\{u/x\}$ is among the rule premises. Hence, we obtain the right hand side of (a-box).
- Case (Rec). We assume the premise of the rule and apply the inductive hypothesis to obtain $H \models \mathcal{D}(F)$. Also we can write $H \models \mathcal{D}(F) = F\{\mathcal{D}(F)/F\}$. Then, by rule (identifier) we have $H \models \bigvee_{n \geqslant 0} f^n(\mathit{ff})$ where
$$f(\varphi) = \mathcal{D}(F)\{\mathcal{D}(F)/F\}\{\varphi/F\}$$
We rewrite $f(\varphi) = \mathcal{D}(F)\cancel{\{\mathcal{D}(F)/F\}}\{\varphi/F\}$ and we apply rule (identifier) to obtain $H \models F$.

$\Rightarrow$) By induction over the definition of $\models$.
- Case (true). Trivial.
- Case (conjunction). By inductive hypothesis we have $H \Vdash \varphi$ and $H \Vdash \varphi'$ which suffice to apply rule ($\wedge$).
- Case (disjunction). We have two symmetric cases. If $H \models \varphi$ then we obtain $H \Vdash \varphi$ and apply ($\vee_L$). Otherwise, we follow the same reasoning with φ'.
- Case (c-diamond). $\langle \sigma(u) \rangle.\varphi$ implies that $\exists H'.H \xrightarrow{\sigma(u)} H'$ and $H' \models \varphi$. Hence, we apply the inductive hypothesis and rule ($\langle u \rangle$).
- Case (a-diamond). We proceed similarly to the previous case. The only difference is that we apply the inductive hypothesis to $\varphi\{u/x\}$ and we use rule ($\langle x \rangle$).
- Case (c-box). Here we have that (*) $\forall H' \cdot H \xrightarrow{\sigma(u)} H'$ implies $H' \models \varphi$. We then consider the set $\{H' \text{ s.t. } H \xrightarrow{\sigma(u)} H'\}$ and its elements H_i. Hence, the premise (*) can be applied to every H_i which suffices to apply the inductive hypothesis to all of them and conclude with ($[u]$).
- Case (a-box). We proceed in a similar way. We build the set $\{H' \text{ s.t. } H \xrightarrow{\sigma(u)} H'\}$ and apply the inductive hypothesis to each of its elements H_i. Then we conclude by applying the inductive hypothesis and ($[x]$).
- Case (identifier). We have $H \models \bigvee_{n \geqslant 0} f^n(\mathit{ff})$ where $f(\varphi) = \mathcal{D}(F)\{\varphi/F\}$. This implies that there exists φ^* such that $H \models \varphi^*$, φ^* is equivalent to $\mathcal{D}(F)$. Then we apply the inductive hypothesis to φ^* to obtain $H \Vdash \varphi^*$ and, by logical equivalence, $H \Vdash \mathcal{D}(F)$. Thus we conclude by applying (Rec).

□

Theorem 3 $H \models \varphi_{/\!/H'} \iff H \parallel H' \models \varphi$

Proof $\Rightarrow$) By induction over the structure of φ.
- Cases tt and ff. Trivial.
- Case $\varphi_1 \wedge \varphi_2$. By PMC rules we have $H \models \varphi_{1/\!/H'} \wedge \varphi_{2/\!/H'}$ then we apply the inductive hypothesis to both $H \models \varphi_{1/\!/H'}$ and $H \models \varphi_{2/\!/H'}$ so as to obtain $H \parallel H' \models \varphi_1$ and $H \parallel H' \models \varphi_2$ and, by rule (conjunction), $H \parallel H' \models \varphi_1 \wedge \varphi_2$.
- Case $\varphi_1 \vee \varphi_2$. By PMC rules we have $H \models \varphi_{1/\!/H'} \vee \varphi_{2/\!/H'}$ and, by (disjunction), we can assume either $H \models \varphi_{1/\!/H'}$ or $H \models \varphi_{2/\!/H'}$. In both cases, we apply the inductive hypothesis and the (disjunction) rule.
- Case $\langle \sigma(u) \rangle.\varphi$. By PMC rule we have
$$H \models \overbrace{\langle \sigma(u) \rangle.\varphi_{/\!/H'}}^{A} \vee \overbrace{\bigvee_{H' \xrightarrow{\sigma(u)} H''} \varphi_{/\!/H''}}^{B}$$

If A holds then by (c-diamond) rule both (i) $H \xrightarrow{\sigma(u)} \dot{H}$ and (ii) $\dot{H} \models \varphi_{/\!\!/H'}$ hold. Hence, by inductive hypothesis on (ii) we have that $\dot{H} \parallel H' \models \varphi$ and by concurrency rule using (i) premise we have that $H \parallel H' \xrightarrow{\sigma(u)} \dot{H} \parallel H'$. These two facts suffice to conclude that $H \parallel H' \models \langle\sigma(u)\rangle.\varphi$. Instead, if B holds, there must exist at least one H'' such that $H' \xrightarrow{\sigma(u)} H''$ and $H \models \varphi_{/\!\!/H''}$. The latter implies (by inductive hypothesis) that $H \parallel H'' \models \varphi$, the former implies $H \parallel H' \xrightarrow{\sigma(u)} H \parallel H''$. These two facts imply (c-diamond) $H \parallel H' \models \langle\sigma(u)\rangle.\varphi$.

– Case $\langle\sigma(x)\rangle.\varphi$. We proceed similarly to the previous case. Applying the PMC rule to $H \models (\langle\sigma(x)\rangle.\varphi)_{/\!\!/H'}$ we have

$$H \models \overbrace{\langle\sigma(x)\rangle.\varphi_{/\!\!/H'}}^{A} \vee \overbrace{\bigvee_{H' \xrightarrow{\sigma(u)} H''} \varphi\{x/u\}_{/\!\!/H''}}^{B}$$

If A holds then by a-diamond rule there exists u such that (i) $H \xrightarrow{\sigma(u)} \dot{H}$ and (ii) $\dot{H} \models \varphi\{x/u\}_{/\!\!/H'}$. Hence, by inductive hypothesis on (ii) we have that $\dot{H} \parallel H' \models \varphi\{x/u\}$ and by concurrency rule using (i) premise we have that $H \parallel H' \xrightarrow{\sigma(u)} \dot{H} \parallel H'$. These two facts suffice to conclude that $H \parallel H' \models \langle\sigma(x)\rangle.\varphi$. Instead, if B holds, there exist H'' and u such that $H' \xrightarrow{\sigma(u)} H''$ and $H \models \varphi\{x/u\}_{/\!\!/H''}$. The latter implies (by inductive hypothesis) that $H \parallel H'' \models \varphi\{x/u\}$, the former implies that $H \parallel H' \xrightarrow{\sigma(u)} H \parallel H''$. Applying (a-diamond) to these two facts we obtain $H \parallel H' \models \langle\sigma(u)\rangle.\varphi$, that is, the thesis.

– Case $[\sigma(u)].\varphi$. By PMC rule we have

$$H \models \overbrace{[\sigma(u)].\varphi_{/\!\!/H'}}^{A} \wedge \overbrace{\bigwedge_{H' \xrightarrow{\sigma(u)} H''} \varphi_{/\!\!/H''}}^{B}$$

By rule (c-box) we must prove that $H \parallel H' \models [\sigma(u)].\varphi$ iff $\forall H''.H \parallel H' \xrightarrow{\sigma(u)} H''$ implies $H'' \models \varphi$. By history expressions semantics we know that either $H \parallel H' \xrightarrow{\sigma(u)} \dot{H} \parallel H'$ or $H \parallel H' \xrightarrow{\sigma(u)} H \parallel \dot{H}'$. In the first case we apply the inductive hypothesis so obtaining $\dot{H} \parallel H' \models \varphi$. In the second case, we use B to show that for each $\dot{H}''$ such that $H' \xrightarrow{\sigma(u)} \dot{H}''$ holds that $H \models \varphi_{/\!\!/\dot{H}''}$. By inductive hypothesis this implies that for each $\dot{H}''$ $H \parallel \dot{H}'' \models \varphi$. By composing the two cases, we obtain that for each $\sigma(u)$ transition $H \parallel H' \models \varphi$ which suffices to conclude.

– Case F. Here we have $H \models F_{/\!\!/H'}$. Hence, by rule identifier and PMC we infer that $H \models \bigvee_{n \geqslant 0} f^n(f\!f)$ with $f(X) = \mathcal{D}(F)_{/\!\!/H'}\{X/F\}$. However, we can easily check that F does not appear as a free identifier in $\mathcal{D}(F)_{/\!\!/H'}$. Thus, we can say that $H \models \mathcal{D}(F)_{/\!\!/H'}$. By inductive hypothesis, $H \parallel H' \models \mathcal{D}(F)$ which is equivalent to $H \parallel H' \models F\{F/\mathcal{D}(F)\}$ and $H \parallel H' \models F$, that is, the thesis.

$\Leftarrow$) By induction over the structure of φ.

– Cases tt and $f\!f$. Trivial.

– Case $\varphi_1 \wedge \varphi_2$. By (conjunction) rule we have $H \parallel H' \models \varphi_1$ and $H \parallel H' \models \varphi_2$. Thus, by inductive hypothesis we have $H \models \varphi_{1/\!\!/H'}$ and $H \models \varphi_{2/\!\!/H'}$ which suffice to conclude.

– Case $\varphi_1 \vee \varphi_2$. Similarly to the previous case, we apply (disjunction) and obtain that either $H \parallel H' \models \varphi_1$ or $H \parallel H' \models \varphi_2$ hold. In both cases, we apply the inductive hypothesis and conclude.

– Case $\langle\sigma(u)\rangle.\varphi$. We have $H \parallel H' \models \langle\sigma(u)\rangle.\varphi$. By applying the rule for parallel history expressions and (c-diamond) we obtain that either

$$(A) \exists \dot{H}.H \xrightarrow{\sigma(u)} \dot{H} \wedge \dot{H} \parallel H' \models \varphi$$

or

$$(B) \exists \dot{H}'.H' \xrightarrow{\sigma(u)} \dot{H}' \wedge H \parallel \dot{H}' \models \varphi$$

If (A) holds then by inductive hypothesis $\exists \dot{H}.\dot{H} \models \varphi_{/\!\!/\dot{H}'}$ and, by (c-diamond), $H \models \langle\sigma(u)\rangle.\varphi_{/\!\!/H'}$ Instead, if (B) holds, $\exists \dot{H}'.H' \xrightarrow{\sigma(u)} \dot{H}'$ and $H \models \varphi_{/\!\!/\dot{H}'}$. This implies that $H \models \bigvee_{H' \xrightarrow{\sigma(u)} \dot{H}'} \varphi_{/\!\!/\dot{H}'}$. Composing the two cases we obtain the thesis.

– Case $\langle\sigma(x)\rangle.\varphi$. The premise is $H \parallel H' \models \langle\sigma(x)\rangle.\varphi$. By applying the rule for parallel history expressions and (a-diamond) we obtain that either

$$(A) \exists u, \dot{H}.H \xrightarrow{\sigma(u)} \dot{H} \wedge \dot{H} \parallel H' \models \varphi\{u/x\}$$

or

$$(B) \exists u, \dot{H}'.H' \xrightarrow{\sigma(u)} \dot{H}' \wedge H \parallel \dot{H}' \models \varphi\{u/x\}$$

If (A) holds, by inductive hypothesis we know that $\exists u.\dot{H} \models \varphi\{u/x\}_{/\!\!/H'}$ which implies $H \models \langle\sigma(x)\rangle.\varphi_{/\!\!/H'}$. Otherwise, applying the inductive hypothesis to (B) we have that (there exist u and $\dot{H}'$ s.t.) $H \models \varphi\{\{u/x\}\}_{/\!\!/\dot{H}'}$ which implies that $H \models \bigvee_{H' \xrightarrow{\sigma(u)} \dot{H}'} \varphi\{\{u/x\}\}_{/\!\!/\dot{H}'}$. Then, by rule (disjunction) we obtain the thesis.

– Case $[\sigma(u)].\varphi$. Here we can assume that

$$(A) \forall \dot{H}.H \xrightarrow{\sigma(u)} \dot{H} \Longrightarrow \dot{H} \parallel H' \models \varphi$$

and

$$(B) \forall \dot{H}'.H' \xrightarrow{\sigma(u)} \dot{H}' \Longrightarrow H \parallel \dot{H}' \models \varphi$$

Applying the inductive hypothesis to (A) we have that $\dot{H} \models \varphi_{/\!\!/H'}$ which implies $H \models [\sigma(u)].\varphi_{/\!\!/H'}$. From (B), by inductive hypothesis, we have that (for each $\dot{H}'$) $H \models \varphi_{/\!\!/\dot{H}'}$. Hence, we can infer that $H \models \bigwedge_{H' \xrightarrow{\sigma(u)} \dot{H}'} \varphi_{/\!\!/\dot{H}'}$. We conclude by applying rule (conjunction) and the PMC definition.

– Case $[\sigma(x)].\varphi$. Here we can assume that

$$(A) \forall u, \dot{H}.H \xrightarrow{\sigma(u)} \dot{H} \Longrightarrow \dot{H} \parallel H' \models \varphi\{u/x\}$$

and

$$(B) \forall u, \dot{H}'.H' \xrightarrow{\sigma(u)} \dot{H}' \Longrightarrow H \parallel \dot{H}' \models \varphi\{u/x\}$$

Applying the inductive hypothesis to (A) we have that $\dot{H} \models \varphi\{u/x\}_{/\!\!/H'}$ and then $H \models [\sigma(x)].\varphi_{/\!\!/H'}$. From (B), by inductive hypothesis, we have that (for each u and $\dot{H}'$) $H \models \varphi\{u/x\}_{/\!\!/\dot{H}'}$. Then, we obtain that

$$H \models \bigwedge_{H' \xrightarrow{\sigma(u)} \dot{H}'} \varphi\{u/x\}_{/\!\!/\dot{H}'}$$

By applying (conjunction) we reduce to the definition of PMC applied to abstract box.

- Case F. In this case we have $H \parallel H' \models F$ which implies $H \parallel H' \models \mathcal{D}(F)$. Then we apply the inductive hypothesis and we obtain $H \parallel H' \models \mathcal{D}(F)_{/\!\!/H'}$ which is by definition equivalent to $H \parallel H' \models F_{H'}$ where $F_{H'} = F_{/\!\!/H'}$.

$\square$