Bring Your Own Device, Securely*

Alessandro Armando
Fondazione Bruno Kessler
Trento, Italy
armando@fbk.eu

Gabriele Costa
Luca Verderame
Università degli Studi di Genova
Genova, Italy
name.surname@unige.it

Alessio Merlo
Università E-Campus
Novedrate, Italy
alessio.merlo@uniecampus.it

ABSTRACT

Modern mobile devices offer users powerful computational capabilities and complete customization. As a matter of fact, today smartphones and tablets have remarkable hardware profiles and a cornucopia of applications. Yet, the security mechanisms offered by most popular mobile operating systems offer only limited protection to the threats posed by malicious applications that may be inadvertently installed by the users and therefore they do not meet the security standards required in corporate environments. In this paper we propose a security framework for mobile devices that ensures that only applications complying with the organization security policy can be installed. This is done by inferring behavioral models from applications and by validating them against the security policy. We also present BYODroid, a prototype implementation of our proposed security framework for the Android OS.

Categories and Subject Descriptors

D.2.4 [**Operating Systems**]: Software/Program Verification; D.4.6 [**Operating Systems**]: Security and Protection—*Information flow controls*; K.4.3 [**Computer and Society**]: Organizational Impacts—*Employment*

General Terms

Programming Framework, Type and Effect System, Policy Management and Verification

Keywords

Android Security, BYOD, History Expression, BYODroid

1. INTRODUCTION

The diffusion of smartphones and tablet PCs is leading to a paradigm shift in the way terminal devices are used and administered in organizations. There is a growing pressure to let devices owned by the individual users to get access to the corporate network and thus to the definition and enforcement of so-called "Bring Your Own Device" (BYOD) policies.[1] On the one hand, BYOD policies must let the users customize their devices to their specific needs and preferences by downloading and installing applications from application stores. On the other hand, they must also counter the threat posed by malicious applications inadvertently installed by the users since they could jeopardize the confidentiality and/or the integrity of corporate data. The security mechanisms offered by the most popular mobile operating systems (e.g. sandboxing) offer only limited protection to this kind of threats and do not support the level of sophistication needed in corporate environments. To illustrate, consider the scenario where the devices must satisfy a security policy stating that "devices cannot access the network after using local file system in the same session". When a user, willing to comply with this policy, installs a new application, he checks the policy declared by the application (e.g. the application *manifest* in Android) and refrains from installing applications that declare both activities (although he cannot be sure if they are indeed unsafe). However, he cannot possibly take a decision about applications that only require 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 can only decide to avoid suspicious applications (i.e., those using the file system or the network) or take the risk to install some of them (which could lead to violation of the security policy of the organization).

In this paper we describe a security framework, previously introduced in [3], for mobile devices that ensures that only applications that comply with the organization security policy are installed on the registered devices. The framework consists of:

- a *security policy manager* that mediates access to the applications stores by *(i)* keeping track of the security state of the registered devices and *(ii)* determining whether an application can be safely added to a device, and
- an *installer application* that tells the user which applications can be safely installed, which need further scrutiny by the security policy manager, and finally which are known to violate the security policy.

The security state of a device is represented by a customized security policy P' obtained by simplifying P w.r.t. the applications currently installed on it. The simplification is carried out by partial model checking [2]. Checking whether an application can be installed on a device (whose current security state is represented by

*This work is partially funded by SPaCIoS EU project.

SAC'13 March 18-22, 2013, Coimbra, Portugal.

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

a policy P') is done by *(i)* inspecting the code of the application and computing an expression (called *history expression*) representing an overapproximation of the execution traces of the application and then by *(ii)* determining whether all execution traces in the overapproximation satisfy P'.

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 which allows us to infer *history expressions* from application implementations. In Section 4 we define a policy specification language for security verification and management and show how history expressions can be validated against policies specifications. In Section 5 we present BOYDroid, a prototype implementation of the proposed security framework. In Section 6 we discuss the related works and we conclude in Section 7 with some final remarks.

2. PROGRAMMING FRAMEWORK

In this section we present an extension of Featherweight Java (FJ) [12]. In particular, we enrich the original language with facilities for defining native Android application-to-application communications. Also, by providing a minimal core calculus for Java applications, FJ allows to reason in a pure framework and focus on security relevant aspects without neglecting the core features of the programming language.

The syntax of the language is given in Table 1.

Table 1: Syntax of applications and components

L	::= $\texttt{class C extends C}'\ \{\bar{\texttt{D}}\ \bar{\texttt{f}};\ \texttt{K}\ \bar{\texttt{M}}\}$	Class
K	::= $\texttt{C}(\bar{\texttt{D}}\ \bar{\texttt{x}})\{\texttt{super}(\bar{\texttt{x}})\,;\ \texttt{this.}\bar{\texttt{f}} := \bar{\texttt{x}}\,;\}$	Constructor
M	::= $\texttt{C m}(\texttt{D x})\{\texttt{return}\ E;\}$	Method
E	::= $\texttt{null} \mid u \mid x \mid \texttt{new C}(\bar{E}) \mid E.\texttt{f} \mid$	Expressions
	$E.\texttt{m}(E') \mid \texttt{system}_\sigma\ E \mid E;E' \mid (\texttt{C})E \mid$	
	$\texttt{if}\ (E = E')\{E_{tt}\}\ \texttt{else}\ \{E_{ff}\} \mid \texttt{thread}\ \{E\}\ \texttt{in}\ \{E'\} \mid$	
	$I_\alpha(E) \mid \texttt{icast}\ E \mid \texttt{ecast C}\ E \mid E.\texttt{data}$	

A class C, possibly extending $\texttt{C}'$, declaration contains a list of typed ($\bar{\texttt{D}}$) fields $\bar{\texttt{f}}$, a constructor K and a set of methods $\bar{\texttt{M}}$. A constructor K consists of its class C, a list of typed parameters $\bar{\texttt{x}}$ 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 no object is returned), its name m and its typed parameters $\bar{\texttt{x}}$. Instead, the method body contains an expression E whose computation provides the return value. Finally, expressions can be either the void value null, a system resource u[2], a variable x, an object creation $\texttt{new C}(\bar{E})$, a field access $E.\texttt{f}$, a method invocation $E.\texttt{m}(E')$, a system call $\texttt{system}_\sigma\ E$, a sequence of two expressions $E;E'$, a type cast $(\texttt{C})E$, a conditional branch $\texttt{if}\ (E = E')\{E_{tt}\}\ \texttt{else}\ \{E_{ff}\}$ or a thread creation $\texttt{thread}\ \{E\}\ \texttt{in}\ \{E'\}$. Android-specific expressions are defined, namely the intent creation $I_\alpha(E)$, implicit or explicit intent propagation ($\texttt{icast}\ E$ and $\texttt{ecast C}\ E$, respectively), an intent content reading $E.\texttt{data}$. Intents correspond to messages that applications exchange in order to communicate.

[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)
          then {icast I_www("http://ad.com") }
          else {/*...play.../* system_write ~ /sav; };
  }
}
```

The class Browser *is a receiver for intents* www. *Method* receive*, when triggered, connects to the url carried by the incoming intent. 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* $\sim$ /sav*. Then, its execution can take two different branches (for our purposes the terms in the guard are irrelevant). If users clicks on an in-game advertisement, an* www *intent containing a url is fired. Otherwise, the game begins and, eventually, the file* $\sim$ /sav *is written.* □

Operational semantics

The behavior of programs follows a small step semantics. We report some of the semantic rules in Table 2. Rules are of the type $\omega, E \rightarrow \omega', E'$ where ω, ω' are sequences of system calls performed by the execution, namely *execution histories*, and E, E' are expressions. A rule $\omega, E \rightarrow \omega', E'$ says that a configuration ω, E can perform a computation step and reduce to ω', E'.

Table 2: Semantics of expressions (fragment)

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

$$(\texttt{PAR}_3)\ \omega, \texttt{thread}\ \{v\}\ \texttt{in}\ \{v'\} \rightarrow \omega, v'$$

$$(\texttt{SYS}_2)\ \omega, \texttt{system}_\sigma\ u \rightarrow \omega \cdot \sigma(u), \texttt{null}$$

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

$$(\texttt{EXPC}_2)\ \frac{\texttt{new}\ C(\bar{v}) \in receiver(\alpha)}{\omega, \texttt{ecast C}\ I_\alpha(u) \rightarrow \omega, (\texttt{new}\ C(\bar{v})).\texttt{receive}(I_\alpha(u))}$$

$$(\texttt{IMPC}_2)\ \frac{\texttt{new}\ C(\bar{v}) \in receiver(\alpha)}{\omega, \texttt{icast}\ I_\alpha(u) \rightarrow \omega, \texttt{new}\ C(\bar{v}).\texttt{receive}(I_\alpha(u))}$$

Intuitively, rule ($\texttt{PAR}_1$) says that the first of two threads E can make a step and reduce to E'' extending the shared history ω to ω'. Needless to say, rule ($\texttt{PAR}_2$) (not reported here) is symmetric to ($\texttt{PAR}_1$). Rule ($\texttt{PAR}_3$) can be applied when both the threads have reduced to a value and says that only one of them is returned, i.e., v'. When a system call is performed (rule ($\texttt{SYS}_2$)), the current history ω is extended by appending the symbol $\sigma(u)$. A method invocation requires the target object and the actual parameters to be reduced to values, then rule ($\texttt{METH}_3$) can be applied. Briefly, it reduces the method invocation to the evaluation of the method body E where the formal parameters have been replaced by the actual ones (also including the special variable this). Note that the function *mbody*,

returning the body and parameters names of a method, also deals with methods lookup (see [12] for more details). Finally, we have rules for explicit ($\texttt{EXPC}_2$) and implicit ($\texttt{IMPC}_2$) intents propagation. Firing an explicit intent causes the system to check whether there exists a receiver $\texttt{new}\, C(\bar{v})$ of the specified class, then the configuration reduces to the execution of the special method `receive`[3]. Implicit intents are handled in a similar way. The only difference is that the receiver's class C is not fixed by the rule and the system is responsible for generating it. In both cases, if no suitable receiver exists, the expressions reduce to `null`.

We informally describe the behavior of the other expressions under the rules of our operational semantics. The expression $\texttt{new C}(\bar{E})$ accepts reductions for the expressions in $\bar{E}$ (in the specified order), until they all reduce to values. A field access $E.\texttt{f}$ reduces along with E and, when E is an object value, reduces to the corresponding field value. Intents $I_\alpha(E)$ can reduce until their data expression E is a value which can be accessed through $E.\texttt{data}$. A sequence $E; E'$ behaves like E as long as it is not a value (which is discharged) and then behaves like E'. Class cast $(\texttt{C})E$ can reduce along with E and, when E reduces to an object, the cast operator can be removed if it is an instance of a subclass of C. Finally, the conditional branch admits reductions for the expressions making its guard (E before E') and then reduces to one between E_{tt}, if the equality check succeeds, and E_{ff}, otherwise.

EXAMPLE 2. *Consider again the classes of Example 1. We simulate the computation of*

$$E \doteq \texttt{icast I}_{\texttt{www}}(\texttt{"http://site.org"})$$

starting from the empty history and assuming the class `Browser` *to be the only receiver in the system.*

$$\begin{array}{ll} \cdot, \texttt{icast I}_{\texttt{www}}(\texttt{"http://site.org"}) & \rightarrow \\ \cdot, \texttt{system}_{\texttt{connect}}\ \texttt{I}_{\texttt{www}}(\texttt{"http://site.org"}).\texttt{data}; & \rightarrow \end{array} \quad (1)$$

$$\begin{array}{ll} \cdot, \texttt{system}_{\texttt{connect}}\ \texttt{"http://site.org"}; & \rightarrow \\ \texttt{system}_{\texttt{connect}}\ \texttt{"http://site.org"}, \texttt{null} & \end{array} \quad (2)$$

□

The initial reduction step (1) corresponds to an implicit intent casting operation. Since `Browser` is the only valid receiver for `www`, the execution reduces to the body of its method `receive` (see Example 1) where the formal parameter has been replaced by the actual one. Finally, in (2) the data carried by the intent is extracted (left side) and used (right side) to fire a new system call, i.e., `connect`.

3. TYPE AND EFFECT

A type and effect system for FJ has been previously described in [19]. We extend it with rules for handling Android-specific instructions, e.g., for intents management.

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 follows.

DEFINITION 1. *(Syntax of history expressions)*

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

[3]We reserve the keyword $\texttt{I}_\alpha$ for the type of the parameter of method `receive`

Table 3: Semantics of history expressions

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

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

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

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

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 $\mu h.H$.

The semantics of history expressions is defined through a *labelled transition system* (LTS) according to the rules in Table 3.

Roughly, a system access $\sigma(u)$ (rule *system*) fires a corresponding event and reduces to ε. Similarly, a intent generation $\alpha_\chi(u)$ (rule *intent*) causes a event and reduces to ε. Intent exchange requires a receiver $\bar{\alpha}_C h.H'$ to be compatible with the intent destination χ. Compatibility is checked through the relation $\cdot \not\geqslant \cdot$ (s.t. for all C, $C \not\geqslant C$ and $? \not\geqslant C$). The function ρ is analogous to *receiver* and we will describe how it is structured below. Concurrent agents $H \parallel H'$ admit either a reduction for left or right component (rules *l-parallel*, *r-parallel*). A sequence $H \cdot H'$ behaves like H until it reduces to ε and then reduces to H' (rule *sequence*). Instead, 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 instances of h are replaced by $\mu h.H$ (rule *recursion*). When necessary, we also write $H \xrightarrow{\omega}{}^{*} H'$ as a shorthand for $H \xrightarrow{a_1} \ldots \xrightarrow{a_n} H'$ with $\omega = a_1 \cdots a_n$.

Moreover, we define a partial relation order over history expressions, denoted by $\sqsubseteq$ and defined as: $H \sqsubseteq H'$ iff $H \xrightarrow{a_1} \ldots \xrightarrow{a_n} H''$ implies there exists $\tilde{H}$ s.t. $H' \xrightarrow{a_1} \ldots \xrightarrow{a_n} \tilde{H}$. Also, we can write $H \equiv H'$ as a shorthand for $H \sqsubseteq H'$ and $H' \sqsubseteq H$ (see [6] for more details).

Type and effect system

Before presenting our type and effect system, we need to introduce the two preliminary definitions of *types* and *type environment*.

DEFINITION 2. *(Types and type environment)*

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

Types can be the unit one $\mathbf{1}$, a set of resources $\mathcal{U}$, a intent type $\mathcal{I}_\alpha(\mathcal{U})$ or a class type C. Instead, 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".

Table 4 reports some rules of interest. Implicit intent casting has type unit and effect $H \cdot \sum_{u \in \mathcal{U}} \alpha_?(u)$ where H is obtained by typing the intent expression E and $\sum_{u \in \mathcal{U}} \alpha_?(u)$ is an abbreviation for $\alpha_?(u_1) + \ldots + \alpha_?(u_n)$ with $\mathcal{U} = \{u_1, \ldots, u_n\}$. The rule for explicit intents is similar, but it uses the receiver class C instead of

Table 4: Typing rules (fragment)

$$(\texttt{T}-\texttt{IMPC})\ \frac{\Gamma \vdash E : \mathcal{I}_\alpha(\mathcal{U}) \triangleright H}{\Gamma \vdash \texttt{icast}\ E : \mathbf{1} \triangleright H \cdot \sum_{u \in \mathcal{U}} \alpha_?(u)} \qquad (\texttt{T}-\texttt{EXPC})\ \frac{\Gamma \vdash E : \mathcal{I}_\alpha(\mathcal{U}) \triangleright H}{\Gamma \vdash \texttt{ecast C}\ E : \mathbf{1} \triangleright H \cdot \sum_{u \in \mathcal{U}} \alpha_C(u)}$$

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

$$(\texttt{T}-\texttt{SYS})\ \frac{\Gamma \vdash E : \mathcal{U} \triangleright H}{\Gamma \vdash \texttt{system}_\sigma\ E : \mathbf{1} \triangleright H \cdot \sum_{u \in \mathcal{U}} \sigma(u)} \qquad (\texttt{T}-\texttt{PAR})\ \frac{\Gamma \vdash E : C \triangleright H \quad \Gamma \vdash E' : C' \triangleright H'}{\Gamma \vdash \texttt{thread}\,\{E\}\,\texttt{in}\,\{E'\} : C' \triangleright H \parallel H'} \qquad (\texttt{T}-\texttt{WKN})\ \frac{\Gamma \vdash E : C \triangleright H' \quad H' \sqsubseteq H}{\Gamma \vdash E : C \triangleright H}$$

the wild card ? to label the intents.

Method invocations, rule (T−METH) require more attention. We state that the invocation $E.\texttt{m}(E')$ has type F' and effect $H \cdot H' \cdot H''$ where F' is the 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 $[C/\texttt{this}, D'/\texttt{x}]$ and the function $msign(\texttt{m}, C)$ returns the signature of a method, i.e., $C \to D$ for a method declaring input type C and return type D[4] (for the definition of the subtype relation $<:$ see [12]).

System calls (T−SYS) have unit type $\mathbf{1}$ and their effect correspond to effect for their parameter E followed by the choice among all the valid instantiation of the access event σ. Instead, concurrent executions have the same type of the second expression, i.e., E', and effect equal to the parallel composition of the effects of the two sub-processes. Finally, we also reported the rule called *weakening* (T−WKN) which allows for effect generalization. In words, 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 $H' \sqsubseteq H$.

Without showing the corresponding rules, we briefly describe the behavior of the type and effect system for the other expressions of our language. Intuitively, `null` has unit type and empty effect, a resource u has empty effect and type equal to the singleton $\{u\}$, the type of a variable x is assigned by the type environment Γ, an object $\texttt{new}\,C(\bar{E})$ has the type of its class C and effect equal to the sequence of the effects produced by the instantiation parameters $\bar{E}$. Field access $E.\texttt{f}$ has the type of `f` and the effect for E while intent creation $I_\alpha(E)$ has type $\mathcal{I}_\alpha(\mathcal{U})$ (where $\mathcal{U}$ is the type of E) and the effect of E. Similarly to field access, intent content reading $E.\texttt{data}$ has the type of the intent (denoted by E) data and effect equal to that of E. Finally, the effect of sequences is the sequence of the two sub-effects for E and E' (while the type is that of E') and the effect of choices is the sequence of the two effects generated by the expressions in the guard and an effect generalizing those of the two branches, i.e., E_{tt} and E_{ff} (while the type is the same of E_{tt} and E_{ff}). More details about these and other similar rules for types and effects can be found in [19, 18, 6].

As expected, 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').*

PROOF. (Sketch[5]) By induction over the structure of E. □

Also well-typed expressions generate history expressions which *safely* denote the runtime behavior of expressions. In particular, the following theorem states that 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' and ω' such that $H \xrightarrow{\omega'}{}^* H'$, $\emptyset \vdash E' : \tau \triangleright H'$ and $\omega = \omega'$.*

PROOF. (Sketch) We start by proving that the property holds for single-step reductions. Then, we proceed by induction on the derivations length. □

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

Finally, we exploit the typing rules for the generation of the function ρ described above. For each existing receiver, we type the corresponding `receive` method and so to obtain

$$\rho(\alpha) = \left\{ \bar{\alpha}_C h.H \;\middle|\; \begin{array}{l} \texttt{new}\,C(\bar{v}) \in receiver(\alpha) \quad \wedge \\ \vdash C.\texttt{receive} : \mathcal{I}_\alpha \xrightarrow{\bar{\alpha}_C h.H} \mathbf{1} \end{array} \right\}$$

4. POLICY VERIFICATION AND MANAGEMENT

We now focus on the management of security policies. We start by introducing the policy language, then we describe how the policy is used to verify and maintain the security state of the devices joining an organization.

Policy language

We use *Hennessy-Milner logic* (HML) [11] for specifying security policies. In particular, we adopt the HML version with negation and we use parametric actions in place of simple labels. The resulting syntax is

$$\varphi, \varphi' ::= tt \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \langle \sigma(\dot{x}) \rangle . \varphi$$

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

[5]Technical proofs can be found in Appendix.

Roughly, a policy can be the positive truth value tt, a negation $\neg\varphi$, a conjunction $\varphi \wedge \varphi'$ or a formula prefixed by the existential modal operator, namely *diamond*, $\langle\sigma(\dot{x})\rangle.\varphi$. In modalities, $\dot{x}$ can be either a resource u or a variable x. We call *concrete diamond* the operator $\langle\sigma(u)\rangle$ and *abstract diamond* $\langle\sigma(x)\rangle$.

The validity of a policy φ is verified against a history expression H, in symbols $H \models \varphi$ according to the following rules.

$$H \models tt \text{ (true)} \qquad H \models \neg\varphi \iff H \not\models \varphi \text{ (negation)}$$

$$H \models \varphi \wedge \varphi' \iff H \models \varphi \text{ and } H \models \varphi' \text{ (conjunction)}$$

$$H \models \langle\sigma(u)\rangle.\varphi \iff H \xrightarrow{\sigma(u)} H' \text{ and } H' \models \varphi \text{ (c-diamond)}$$

$$H \models \langle\sigma(x)\rangle.\varphi \iff \exists u.H \xrightarrow{\sigma(u)} H' \text{ and } H' \models \varphi\{u/x\} \text{ (a-diamond)}$$

In words, tt is valid for every history expression H (rule *true*), $\neg\varphi$ is satisfied by the history expressions violating φ (rule *negation*) and conjunction is satisfied by history expressions valid for both the sub-clauses (rule *conjunction*). Modalities referring to a resource u are satisfied if the history expression admits at least one corresponding transition and the resulting expression satisfies the sub formula φ (rule *c-diamond*). Similarly, modalities 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.

Moreover, we introduce some standard abbreviations defined as follows.

$$ff \triangleq \neg tt \qquad \varphi \vee \varphi' \triangleq \neg(\neg\varphi \wedge \neg\varphi') \qquad [\sigma(\dot{x})].\varphi \triangleq \neg\langle\sigma(\dot{x})\rangle.\neg\varphi$$

EXAMPLE 3. *Consider the history expression*

$$H = \texttt{read}(\sim \texttt{/sav}) \cdot (\texttt{www}_?(\texttt{"http://ad.com"}) + \texttt{write}(\sim \texttt{/sav}))$$

and the formula $\varphi = \langle\texttt{read(x)}\rangle.\langle\texttt{connect(y)}\rangle.ff$.

Assuming $\rho(\texttt{www}) = \{\overline{\texttt{www}}_{\texttt{Browser}}\texttt{h.connect}(\texttt{"http://ad.com"})\}$ *we show that* $H \not\models \varphi$ *(we use "HE" to denote the steps using a property of history expressions).*

$\texttt{read}(\sim \texttt{/sav}) \cdot (\texttt{www}_?(\texttt{"http://ad.com"}) + \texttt{write}(\sim \texttt{/sav}))$
$\models \langle\texttt{read(x)}\rangle.\langle\texttt{connect(y)}\rangle.ff \iff$ *(a-diamond)*

$\texttt{www}_?(\texttt{"http://ad.com"}) + \texttt{write}(\sim \texttt{/sav}) \models \langle\texttt{connect(y)}\rangle.ff \implies$ *(HE choice)*

$\texttt{www}_?(\texttt{"http://ad.com"}) \models \langle\texttt{connect(y)}\rangle.ff \implies$ *(HE intent)*

$\texttt{connect}(\texttt{"http://ad.com"}) \models \langle\texttt{connect(y)}\rangle.ff \iff$ *(a-diamond)*

$\varepsilon \models ff$

which is trivially false. Hence, $H \not\models \varphi$. □

Partial model checking

In [2] *partial model checking* (PMC) is presented as a technique for the partial evaluation of a formula against a model. Intuitively, PMC uses reduction rules for transferring information from the model 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. In particular, we show the operator $\cdot_{//\cdot}$ for the partial evaluation against parallel composition.

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

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

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

Intuitively, the tt formula keeps unchanged while for negation and conjunction 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 in φ.

Policy compliance is granted by the following theorem.

THEOREM 2. $H \models \varphi_{//H'} \implies H \parallel H' \models \varphi$

PROOF. (Sketch) By induction on the PMC rules. □

Intuitively, this property states that we can prove the compliance of two (or more) concurrent components against a policy if we can check the compliance of one of them against a "specialization" of the original policy. Clearly, this is particularly useful under our assumptions in which new applications join a set of previously installed ones.

EXAMPLE 4. *Consider the policy*

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

and the following history expression

$$\left.\begin{array}{l}(\mu h_1.(\texttt{www}_?(u)) \cdot h_1) \\ \parallel (\texttt{read}(\sim \texttt{/sav}) \cdot \\ \qquad (\texttt{write}(\sim \texttt{/sav}) + \texttt{www}_?(\texttt{"ad.com"}))) \\ \parallel (\overline{\texttt{www}}_B h_3.\textstyle\sum_u \texttt{connect}(u))\end{array}\right\} H_1 \parallel H_2 \parallel H_3$$

It consists of (the parallel composition of) the history expression for the Browser *receiver* (H_3)*, the* Game *activity* (H_2) *and a (simplified) user interface* (H_1)*. Terms* H_2 *and* H_3 *are generated by typing the methods* receive *and* play *of* Browser *and* Game*, respectively. Instead,* H_1 *represents the user's behaviour (for instance, we could obtain it by typing the user interface component). Roughly,* H_1 *says that, iteratively, the user can open the browser on the homepage* u ($\texttt{www}_?(u)$)*.*

It is easy to verify that both $H_1 \parallel H_2 \models \varphi$ *and* $H_1 \parallel H_3 \models \varphi$*. As a matter of fact,* φ *is only violated by traces containing either a* read *or* write *(both missing in* $H_1 \parallel H_3$*) followed by a* connect *(missing in* $H_1 \parallel H_3$*). However, we prove (using PMC) that* $H_1 \parallel H_2 \parallel H_3 \not\models \varphi$*. We show that the installation of* Game *is not allowed by* φ *on a system running* Browser*. First we compute* $\varphi_{//H_1} = \varphi$ *(trivial since* H_1 *contains no system actions) and* $\varphi' = \varphi_{//H_2}$*.*

$$\varphi' = \left(\begin{array}{l}\neg\big(\langle\texttt{read(x)}\rangle.\langle\texttt{connect(y)}\rangle.tt \vee \langle\texttt{connect(y)}\rangle.tt\big) \wedge \\ \neg\big(\langle\texttt{write(x)}\rangle.\langle\texttt{connect(y)}\rangle.tt \vee \langle\texttt{connect(y)}\rangle.tt\big)\end{array}\right)$$

Finally, we just need to verify that $H_3 \not\models \varphi'$*. Without showing the proof, we can trivially observe that* $\texttt{connect}(u)$ *is a trace of* H_3 *that violates* φ'. □

5. BYODROID

We implemented a prototype (*BYODroid*) using the techniques described above for guaranteeing organizations against security violations in BYOD scenarios. BYODroid consists of two main components: the *BYODroid Market* and the *BYODroid Installer*. The

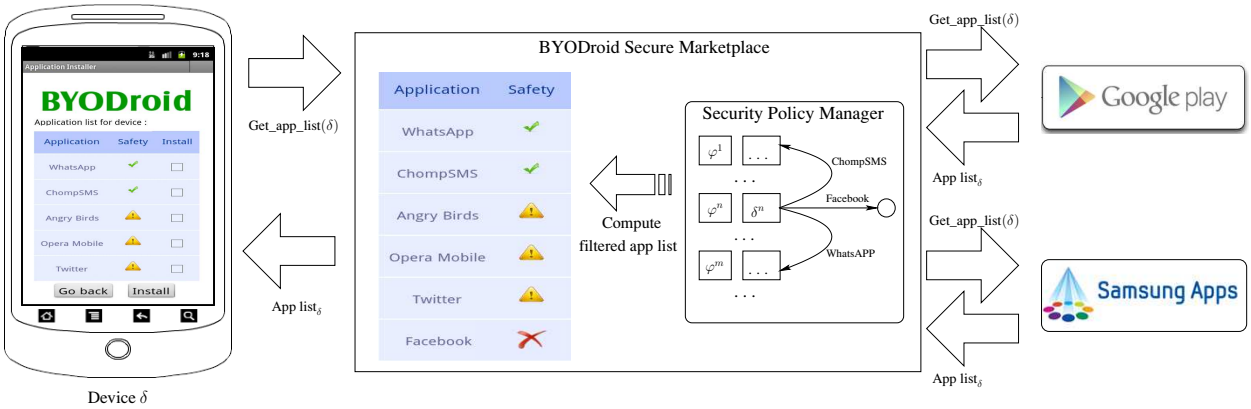

Figure 1: The BYODroid marketplace generating an applications list.

BYODroid Installer allows users to access the BYODroid Market and get new applications respecting the policy of the organization. Once a user requires to register its device to the organization, the BYODroid Installer replaces the native system installer. After installation, the BYODroid installer analyzes the device configuration, i.e., the installed applications, and sends it to the BYODroid Market. If the configuration is a legal one, the registration successfully completes and the device is connected to the organization.

The BYODroid Market is responsible for holding the security state of the devices registered to the organization and for mediating the access to other application markets by only allowing a connected device to install legal applications. Basically, the BYODroid Market contains a *security policy manager* which maintains a *graph of policies dependency*, rooted in the organization policy. Each node of the graph contains information about the security configuration of registered devices. In detail, a node N is labeled with a policy φ_N and contains a list of device identifiers and has a finite set of outgoing edges labeled with application identifiers. When a new device δ successfully registers, its identifier is associated to a node N in the graph such that the path from the root to N is labeled with the list of δ's applications (if no such state exists it is created by using the node creation procedure below). The policy φ_N is obtained through partial evaluation (see Section 4) of the root policy against the (model of the) applications installed on δ. In this way we create the premises for applying Theorem 2.

When the registered device δ (appearing in node N) accesses the market service, the BYODroid Market retrieves the list of applications and generates a view for the BYODroid installer. The view is a list of applications $A_1, \ldots, A_m$ such that if an edge from N labeled with A_i exists, then A_i is marked *safe*. Otherwise, A_i is marked *unchecked*. If δ (being in node N) installs a safe applications A_s labeling an edge to N', its device identifier is moved to N'. Instead, if δ aims at installing an unchecked application A_u, the following procedure is applied: the BYODroid Market downloads A_u and applies the type and effect system in Section 3 to infer H_u. Then, it verifies whether $H_u \not\models \varphi_N$. If so, the installation is blocked and the user alerted. Otherwise, $\varphi_{N\,//H_u}$ (see Section 4) is computed. Such value is used to label a fresh node $\bar{N}$, adding an edge from N to $\bar{N}$, labeled by A_u, to the graph. Finally, δ is moved to $\bar{N}$ and the installation can proceed.

In general, each node can also store a black list of failed installations, i.e., applications that do not fulfil the node policy. Applications in the black list do not appear in the list of available installations for the registered devices being in the corresponding node.

The prototype structure and behavior are described in Figure 1 which also shows the outputs of our prototype.

According to the BYOD paradigm, a user is free to use his personal device outside the organization. The BYODroid installer supports such situation by allowing the user to install any application when the device is not connected to the organization. However, each time the user tries to rejoin the organization, the current device configuration is checked against the organization policy; if this is not satisfied due to the presence of forbidden applications, the user is prompted with the list of applications to remove in order to get access to the organization.

We implemented our prototype BYODroid marketplace as a local service running on a desktop PC. When a request arrives, the service uses the procedure described above to produce a web page that customers visualize and use to select and install new applications. Also, we used Android simulator to create virtual devices running the BYODroid Installer and connecting to the service.

6. RELATED WORK

Due to the high diffusion of smartphones, a substantial part of the current research in security is turning towards mobile platforms. Most of such research is focused on security aspects of mobile OSs. On this tread, the research is focused on the Android platform, due to its open source, hardware-independence and general purpose nature. In particular, current literature on Android security contains 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 [16] the (informal) Android security policy is analyzed in terms of efficacy and some extensions are proposed. Besides, in [13] authors propose an extension to the basic Android permission systems and some new policies built on top of the extended permission system. Moreover, in [20] 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, [14] proposes a method for monitoring the Android permissions system by properly customizing the Android stack. Malware detection approaches also exist, e.g., XManDroid [9]. Besides, other works have successfully detected vulnerabilities related to DoS attacks [4], and covert channels [15]. The efficacy and soundness of previous approaches have been proved through empirical assessments, without any formal basis. Recently, researchers focused on the formal modeling and analysis of the Android platform and its security aspects. In [17] the

authors formalize the permission scheme of Android. Briefly, their formalization consists of a state-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 [10] proposes a language-based approach to infer security properties from Android applications. Moreover, this work proposes 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 [10] 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. a security policies. Furthermore, we applied this method to a specific context, i.e., securing BYOD-based organizations, and we detailed the features of a prototype under implementation. The models we produce are safe in the sense that they correctly represent all the possible runtime computations of the applications they come from. Moreover, the history expressions representing each single application can be combined together in order to create a global model for a specific Android platform. History expressions, originally proposed by Bartoletti et al. [8], have been successfully applied to the security analysis of Java applications [5] and web services [7], and we extended most of their results to the Android framework. Moreover, we introduces a new, PMC-based approach for the partial evaluation of security policies which we fruitfully exploited in the definition of the BYODroid prototype.

8. REFERENCES

[1] M. Abadi and C. Fournet. Access control based on execution history. In *Proc. of the 10th annual Network and Distributed System Security Symposium*, pages 107–121, 2003.

[2] H. R. Andersen. Partial model checking (extended abstract). In *Proc. of 10th Annual IEEE Symposium on Logic in Computer Science*, pages 398–407. IEEE Computer Society Press, 1995.

[3] A. Armando, G. Costa, and A. Merlo. Formal modeling and reasoning about the Android security framework. In *Proc. of 7th International Symposium on Trustworthy Global Computing*, 2012. (To appear).

[4] A. Armando, A. Merlo, M. Migliardi, and L. Verderame. 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)*, pages 13–24.

[5] M. Bartoletti, G. Costa, P. Degano, F. Martinelli, and R. Zunino. Securing Java with Local Policies. *Journal of Object Technology*, 8(4):5–32, 2009.

[6] M. Bartoletti, P. Degano, and G. L. Ferrari. History-based access control with local policies. In *FoSSaCS*, pages 316–332, 2005.

[7] M. Bartoletti, P. Degano, and G. L. Ferrari. Planning and verifying service composition. *Journal of Computer Security (JCS)*, 17(5):799–837, 2009.

[8] M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino. Types and effects for resource usage analysis. In *Proc. of the 10th International Conference on Foundations of Software Science and Computation Structures*, pages 32–47, 2007.

[9] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Univ. Darmstadt, Apr 2011.

[10] A. Chaudhuri. Language-based security on Android. In *Proc. of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 1–7, New York, NY, USA, 2009. ACM.

[11] M. Hennessy and R. Milner. On Observing Nondeterminism and Concurrency. In *Proc. of the 7th Colloquium on Automata, Languages and Programming*, pages 299–309, London, UK, 1980. Springer-Verlag.

[12] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *ACM Transactions on Programming Languages and Systems*, pages 132–146, 1999.

[13] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proc. of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 328–332, New York, NY, USA, 2010. ACM.

[14] M. Ongtang, S. Mclaughlin, W. Enck, and P. Mcdaniel. Semantically rich application-centric security in android. In *In ACSAC '09: Annual Computer Security Applications Conference*, 2009.

[15] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *Proc. of the 18th Annual Network & Distributed System Security Symposium*, 2011.

[16] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google android: A comprehensive security assessment. *Security Privacy, IEEE*, 8:35–44, march 2010.

[17] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka. A Formal Model to Analyze the Permission Authorization and Enforcement in the Android Framework. In *Proc. of the 2010 IEEE Second International Conference on Social Computing*, SOCIALCOM '10, pages 944–951, Washington, DC, USA, 2010. IEEE Computer Society.

[18] C. Skalka and S. Smith. History effects and verification. In *Second ASIAN Symposium on Programming Languages and Systems (APLAS)*, pages 107–128. Springer, 2004.

[19] C. Skalka, S. Smith, and D. Van Horn. A Type and Effect System for Flexible Abstract Interpretation of Java. *Electronic Notes in Theorical Computer Science*, 131:111–124, May 2005.

[20] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on android). In *Proc. of the 4th international conference on Trust and trustworthy computing*, TRUST'11, pages 93–107, 2011.