Securing the “Bring Your Own Device” Policy[*]

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

Gabriele Costa[†]
Università degli Studi di Genova
Genova, Italy
gabriele.costa@unige.it

Alessio Merlo[†]
Università e-Campus
alessio.merlo@uniecampus.it

Luca Verderame
Università degli Studi di Genova
Genova, Italy
luca.verderame87@gmail.com

Abstract

The number of devices (phones, tablets, smart TVs, ...) using Android OS is continuously and rapidly growing. Together with the devices, also the amount of applications and on-line application marketplaces is increasing. Unfortunately, security guarantees are not evolving concurrently and security flaws have been reported. Far from discouraging them, more and more users and organisations rely on Android even for security critical activities. The *bring your own device* (BYOD) paradigm confirms this trend. Indeed, it allows mobile devices to join a virtual organisation (consisting of a set of federated devices) in order to access to services and functionalities. Needless to say, the basic security support offered by Android and application markets is totally inappropriate for dealing with the security requirements involved in BYOD-like scenarios.

In this work we describe a technique for guaranteeing that devices comply with a security policy. To do that, we use a type and effect system to infer behavioural models from applications implementation and we validate them against policy specification. Moreover, we define a novel approach, based on partial model checking, for partially evaluating the security policy depending on devices configurations. Finally, we present a prototype under implementation, called *BYODroid*, which concretely applies these techniques to secure the devices joining a virtual organisations in the BYOD style.

Keywords: Android security, BYOD paradigm, online marketplaces, static Analysis, partial model checking

1 Introduction

Android is currently the most widespread mobile OS and its diffusion is expected to keep growing in the near future [19]. Such level of success depends on a high customizability and on the adaptability to work over different hardware that pushed Android to be adopted by the main part of smartphone manufacturers. Besides, Android has been originally designed only for private use only, allowing users to freely customize the software installed on the smartphone, both from official and unofficial sources. Nevertheless, due to its consolidation in recent years, the Android platform is attracting organizations that plan to adopt Android for professional use. For instance, the Pentagon [27] has announced the adoption of Android on smartphones and tablets provided to employees and managers. In general, this common trend consists in using general purpose and private devices, e.g., smartphones and tablet devices, inside

[*]This work is partially supported by the EU FP7-ICT-2009-5 Project no. 257876 “SPaCIoS”.

[†]Corresponding authors.

organizations. This is due to many factors like, for instance, the growing capabilities of smartphones, their diffusion and low costs.

In this context, the “Bring Your Own Device” [25] is gaining importance. The idea of BYOD is developing a policy and enforcement mechanisms allowing persons to use their own personal device in a professional context.

However, deploying the BYOD paradigm on Android requires to address security issues that the Android Security Framework is not natively able to manage. For instance, in BYOD paradigm companies aim to define proper security policies for personal devices belonging to different groups of employees and avoid employees to violate such policies by means of enforcement mechanisms that may be both embedded on the device or hosted on trusted external server. Android is not natively able to manage such complex situations since its enforcement mechanisms are limited to the single device and are not able to interact with an external server or with other device, nor to manage custom security policies.

We argue that such narrowness is particularly critical and may impact the deployment of BYOD paradigm on Android devices. To this aim, in this paper we describe a formal framework able to provide such extended functionality to Android, allowing to check the configuration of a device, in terms of installed applications, against security policies provided by an organization. We implemented such framework in a prototype (BYODroid) which analyses the security policy and the configuration of the device, stating whether a new application may be installed according to the security policy. BYODroid has been implemented as a component wrapping original application markets.

Structure of the paper. Section 2 provides a use-case scenario for BYODroid, while Section 3 a programming framework for modeling the behavior of Android applications. In Section 4 we provide the type and effect system for the programming framework. In Section 5 we provide a policy language for security verification and management. Finally, Section 6 discusses some related works and Section 7 concludes the paper.

2 Motivating scenario

In a BYOD scenario, each user joining an organization must register the owned device. The owner of a registering device can be either a new employee, a temporary collaborator or a customer.

Typically, organizations have security policies that participants must accept and respect. Such policies are devoted to a two-fold aim, namely avoiding users to i) introduce malicious software inside the organization (outsider threats) and take outside sensitive information for the organization (insider threats).

In general, security policies can cause a, possibly temporary, restriction on the usage of the device. Still, devices can change their configuration by installing new software. Figure 1 depicts this scenario.

In detail, once a personal device is registered to the organization, it gets a policy from the organization which the device configuration is initially expected to fulfil. Installation/removal of external applications by the user are likewise expected to fulfil the policy.

A policy management system for this scenario must fulfil some requirements. First of all, it must provide support for policy checking. This means that it must be always possible to verify the organization policy against a registering device, independently of its specific configuration. Policy compliance must be formally granted, e.g., through model checking or automatic theorem proving.

Moreover, it ought to provide support for configuration changes. In particular, application installation and removal must be validated and, possibly, forbidden. Clearly, such validation cannot require a complete revaluation of the policy, which would result in a massive computation.

Finally, although the system will probably need on-device support, this must be implemented avoiding customizations of the devices. Indeed, device customizations, e.g., OS modules replacement, would

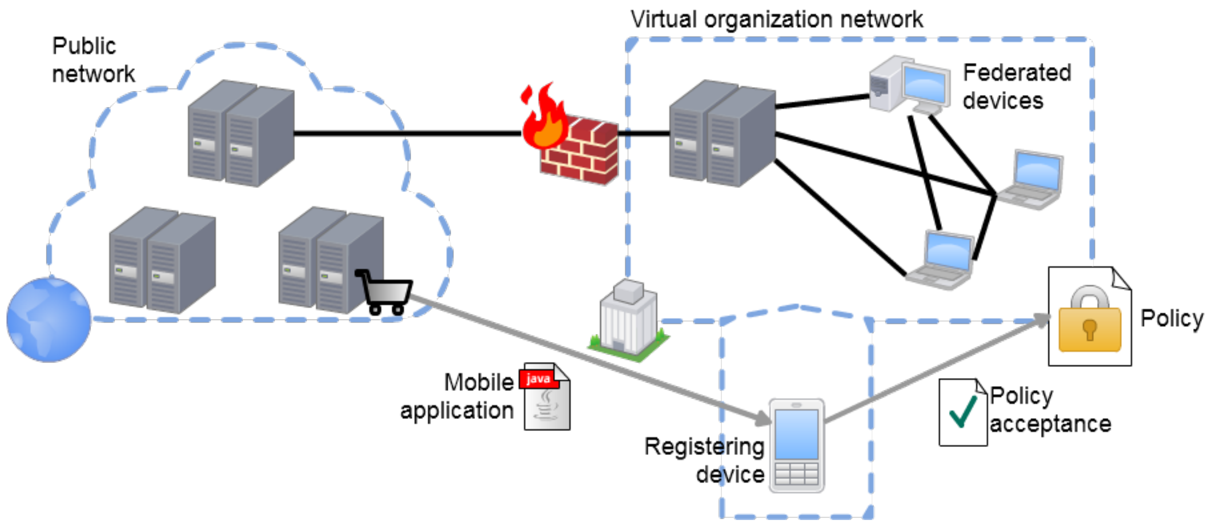

Figure 1: A BYOD scenario.

drastically reduce the applicability of the system, while a solution built on top of the existing OS is general and non-invasive.

In the remaining of this paper we will present our architecture and the adopted technologies used to implement such scenario, by extending the formal model in [3]. Finally, we introduce a prototypical implementation of an application market (BYODroid) that meets the requirements listed above, coping with the issues presented in our working example.

Example 1. *Consider a mobile device joining a virtual organisation which applies a security policy saying "devices cannot access the network after using local file system in the same session". When an Android user, willing to respect this constraint, installs a new application, he checks the application description, namely its* `manifest`*, for behavioural details. He decides to avoid applications declaring both activities (although he cannot be sure they are unsafe). However, he cannot take a decision about programs which only require one of the two permissions. As a matter of fact, no information about possible interactions between the new application and already installed ones is available. As a consequence, the user can either decide to avoid suspicious applications (i.e., those using files or network) or take the risk to install some of them (which could lead to policy violation).*

3 Programming Framework

In this section we describe our extension of Featherweight Java (FJ) [15] for modelling Android applications. Providing a minimal core calculus for Java applications, FJ supports reasoning in a pure framework and focusing on security relevant aspect without neglecting the core features of the programming language. In Section 4 we also present a type and effect system which infers *history expressions* from applications implementation. History expressions have a formal semantics binding them to labelled transition systems used to represent the applications behaviour in terms of usage of security critical APIs.

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

Table 1: Syntax of applications and components.

3.1 Applications and components

In Table 1 we report the syntax of the elements of our framework. Basically, we extend FJ with new expressions representing some of the native features of Android, mainly for intents creation and exchange. We also include expressions for the concurrent execution of threads.

A class C, possibly extending C′, declaration contains a list of typed 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 (e.g. 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.), a variable x, an object creation $\texttt{new}\,\texttt{C}(\bar{E})$, a field access $E.\texttt{f}$, a method invocation $E.\texttt{m}(E')$, a system call $\texttt{system}_\sigma E$, an implicit or explicit intent propagation ($\texttt{icast}\,E$ and $\texttt{ecast}\,\texttt{C}\,E$, respectively), an intent creation $I_\alpha(E)$, an intent content reading $E.\texttt{data}$, 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 $\texttt{thread}\,\{E\}\,\texttt{in}\,\{E'\}$.

Example 2. *Consider the following classes*

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

In word, the class Browser *is a receiver for intents* www*. The method* receive*, when triggered, connects to the url carried by the incoming intent.*

Also Class Game *extends* Receiver*. When an intent* play *is received, the execution can take two different branches. If the intent carries a resource* AD *(for* advertisement*) the game is saved by writing a file* ∼ /sav *and a* www *intent containing a url is fired. Otherwise, the game data is loaded by reading file* ∼ /sav*.*

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

$$(\texttt{PAR}_1)\ \frac{\omega, E \to \omega', E''}{\omega, \texttt{thread}\,\{E\}\,\texttt{in}\,\{E'\} \to \omega', \texttt{thread}\,\{E''\}\,\texttt{in}\,\{E'\}} \qquad (\texttt{PAR}_3)\ \omega, \texttt{thread}\,\{v\}\,\texttt{in}\,\{v'\} \to \omega, v'$$

$$(\texttt{SYS})\ \omega, \texttt{system}_\sigma\, u \to \omega \cdot \sigma(u), \bot \qquad (\texttt{METH}_3)\ \frac{\mathit{mbody}(\texttt{m}, \texttt{C}) = \texttt{x}, E}{\omega, (\texttt{new}\,\texttt{C}(\bar{v})).\texttt{m}(v') \to \omega, E[v'/x, (\texttt{new}\,\texttt{C}(\bar{v}))/\texttt{this}]}$$

$$(\texttt{EXPC})\ \frac{E \in \mathit{receiver}(\alpha) \quad E : C}{\omega, \texttt{ecast}\,\texttt{C}\,I_\alpha(u) \to \omega, E.\texttt{receive}(I_\alpha(u))} \qquad (\texttt{IMPC})\ \frac{E \in \mathit{receiver}(\alpha)}{\omega, \texttt{icast}\,I_\alpha(u) \to \omega, E.\texttt{receive}(I_\alpha(u))}$$

Table 2: Semantics of expressions (fragment).

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 $(\texttt{SYS})$), the current history ω 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 [15] for more details). Finally, we have rules for explicit $(\texttt{EXPC})$ and implicit $(\texttt{IMPC})$ intents propagation. Firing an explicit intent causes the system to check whether there exists a receiver E of the specified class C, then the configuration reduces to the execution of the special method `receive` (We reserve the keyword $\texttt{I}_\alpha$ for the type of the parameter of method `receive`). Implicit intents are handled in a similar way. The only difference is that the receiver E is not specified and the system is responsible for retrieving it. In both cases, if no suitable receiver exists, the expressions reduce to $\bot$.

We informally describe the behaviour of the other expressions under the rules of our operational semantics. A `null` expression does not modify the current history and reduces to the void value $\bot$. The expression $\texttt{new}\,\texttt{C}(\bar{E})$ accepts reductions for the expressions in $\bar{E}$ (in the specified order), until they all reduce to values (i.e., closed expressions accepting no further reductions). 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 until 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 3. *Consider again the classes of Example 2. We simulate the computation of*

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

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

$$\sigma(u) \xrightarrow{\sigma(u)} \varepsilon \qquad \alpha_\chi(u) \xrightarrow{\alpha_\chi(u)} \Lsh \qquad \frac{H \xrightarrow{a} H''}{H \parallel H' \xrightarrow{a} H'' \parallel H'} \qquad \frac{H' \xrightarrow{a} H''}{H \parallel H' \xrightarrow{a} H' \parallel H''} \qquad \frac{H \xrightarrow{\alpha_\chi(u)} H'' \quad \chi \succcurlyeq C}{H \parallel \bar{\alpha}_C h.H' \dot{\rightarrow} H'' \parallel H'\{\alpha_?(u)/h\}}$$

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

Table 3: Semantics of history expressions

(also, we use $(\cdot)$ and $\{\cdot\}$ to group histories and expressions, respectively).

$$\cdot, \texttt{icast}\ \texttt{I}_{\texttt{www}}(\text{"}\texttt{http://site.org}\text{"}) \rightarrow \cdot, \texttt{system}_{\texttt{connect}}\ \texttt{I}_{\texttt{www}}(\text{"}\texttt{http://site.org}\text{"}).\texttt{data}; \rightarrow \tag{1}$$

$$\cdot, \texttt{system}_{\texttt{connect}}\ \text{"}\texttt{http://site.org}\text{"}; \rightarrow \texttt{system}_{\texttt{connect}}\ \text{"}\texttt{http://site.org}\text{"}, \bot \tag{2}$$

The initial reduction steps (1) correspond to an implicit intent casting operation. Since `Browser` *is the only valid receiver for* `www`*, the execution reduces to the body of method* `receive` *(see Example 2) 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`.

4 Type and Effect

Like normal type systems, type and effect systems consist of a set of rule for inferring types from programs. Moreover, they also produce effects, i.e., expressions describing the program in terms of its side effects. A type and effect system for FJ has been previously described in [24]. We extend it with rules for handling Android-specific instructions. Among them intents, i.e., Android application-to-application communications, management plays a central role.

History expressions The type and effect system extracts *history expressions* from programs. History expressions model computational agents in a process-algebraic fashion. This means that they denote traces of security-relevant events that computations produce. The syntax of history expressions follows.

Definition 1. *(Syntax of history expressions)*

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

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 $\Lsh$ (basically, $\Lsh$ behaves like ε but for sequences which are interrupted by $\Lsh$, i.e., $\varepsilon \cdot H \equiv H$ while $\Lsh \cdot H \equiv \varepsilon$). Concurrent agents $H \parallel H'$

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

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

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

Table 4: Typing rules (fragment).

admit either a reduction for each component or a intent exchange (rules *l-parallel*, *r-parallel* and *s-parallel*). Intent exchange requires the receiver $\bar{\alpha}_C h.H'$ to be compatible with the intent destination χ. Compatibility is checked through the relation $\cdot \succcurlyeq \cdot$ (s.t. for all C, $C \succcurlyeq C$ and $? \succcurlyeq C$). 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 behaviour 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 $H \xrightarrow{a_1} \dots \xrightarrow{a_n} H'$ 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 $H \xrightarrow{a_1} \dots \xrightarrow{a_n} H''$ implies there exists $\tilde{H}$ s.t. $H' \xrightarrow{a_1} \dots \xrightarrow{a_n} \tilde{H}$. 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)*

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

Types can be the unit one $\mathbf{1}$, a set of resources $\mathscr{U}$, a intent type $\mathscr{I}_\alpha(\mathscr{U})$ 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 judgement 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 $H \cdot \sum_{u\in\mathscr{U}} \alpha_?(u)$ where H is obtained by typing the intent expression E and $\sum_{u\in\mathscr{U}} \alpha_?(u)$ is an abbreviation for $\alpha_?(u_1) + \dots + \alpha_?(u_n)$ with $\mathscr{U} = \{u_1, \dots, u_n\}$. The rule for explicit intents is similar, but it uses the receiver class C instead of the wildcard ? to label the intents.

Method invocations, rule ($\mathtt{T-METH}$) require more attention. We state that the invocation $E.\mathtt{m}(E')$ 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 $[C/\mathtt{this}, D'/\mathtt{x}]$ and the function $\mathit{msign}(\mathtt{m}, C)$ returns the signature of a method, i.e., $C \to D$ for a method declaring input type C and return type D. Instead, for a class $C <: \mathtt{Receiver}$ we write $\mathit{msign}(\mathtt{receive}, C) = \mathscr{I}_\alpha \to \mathbf{1}$ (for the definition of the subtype relation $<:$ see [15]).

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 generalisation. 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 behaviour 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 $\mathscr{I}_\alpha(\mathscr{U})$ (where $\mathscr{U}$ 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 generalising 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 [24, 23, 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) By induction over the structure of E. □

Also well-typed expressions generate history expressions which *safely* denote the runtime behaviour 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 $\omega = \Delta\omega'$ (where Δ is a function removing the intent events from a trace).*

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

The original statements of these properties (together with formal proofs) can be found in [3].

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

5 Policy verification and management

Below we describe our proposal for 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 a virtual organization.

5.1 Policy language and partial model checking

We use *Hennessy-Milner logic* (HML) [14] for specifying security policies. The HML is a classical specification language with existential and universal modalities which, also, served as the bases for several extensions and other policy languages. Here, we adopt the HML version with negation and we use parametric actions in place of simple labels. The resulting syntax is

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

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* $\langle \sigma(x) \rangle$.

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

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

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

$$\bar{\alpha}_C h.H \models \varphi \iff \forall u.H\{\alpha_?(u)/h\} \models \varphi \text{ (receiver)} \qquad H \models \varphi \iff H \xrightarrow{\alpha_\chi(u)} H' \text{ implies } H' \models \varphi \text{ (intent)}$$

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. Finally, a receiver is valid w.r.t. a policy if and only if it satisfies the policy for each possible incoming intent (rule *receiver*) and intents emission does not affect policy validity (rule *intent*).

Example 4. *Consider the history expression $H = \mu h.((\sigma(u) \cdot h) + \varepsilon)$ and the formula $\varphi = \langle \sigma(x) \rangle.\langle \sigma(x) \rangle.tt$; we show that $H \models \varphi$ (we use HE to denote the steps using a property of history expressions and LV for HML validity rules).*

$$\begin{array}{lcr}
\mu h.((\sigma(u) \cdot h) + \varepsilon) \models \langle \sigma(x) \rangle.\langle \sigma(x) \rangle.tt & \iff & \textit{(HE recursion rule)} \\
(\sigma(u) \cdot \mu h.((\sigma(u) \cdot h) + \varepsilon)) + \varepsilon \models \langle \sigma(x) \rangle.\langle \sigma(x) \rangle.tt & \iff & \textit{(HE choice rule)} \\
\sigma(u) \cdot \mu h.((\sigma(u) \cdot h) + \varepsilon) \models \langle \sigma(x) \rangle.\langle \sigma(x) \rangle.tt & \iff & \textit{(LV a-diamond)} \\
\mu h.((\sigma(u) \cdot h) + \varepsilon) \models \langle \sigma(u) \rangle.tt & \iff & \textit{(HE recursion rule)} \\
(\sigma(u) \cdot \mu h.((\sigma(u) \cdot h) + \varepsilon)) + \varepsilon \models \langle \sigma(u) \rangle.tt & \iff & \textit{(HE choice rule)} \\
\sigma(u) \cdot \mu h.((\sigma(u) \cdot h) + \varepsilon) \models \langle \sigma(u) \rangle.tt & \iff & \textit{(LV c-diamond)} \\
\mu h.((\sigma(u) \cdot h) + \varepsilon) \models tt & & \textit{(LV true)}
\end{array}$$

which concludes the proof.

Moreover, we introduce some, standard abbreviations defined as follows.

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

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 exploits a set of 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.

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

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

$$\varphi_{//H} = \varphi_{//H'} \text{ if } H \xrightarrow{\alpha_\chi(u)} H' \wedge H \overset{\sigma(u')}{\not\longrightarrow} \qquad \varphi_{//\bar{\alpha}_C h.H} = \varphi_{//\sum_u H\{\alpha_?(u)/h\}} \qquad \varphi_{//H} = \varphi_{//H'} \text{ if } H \equiv H'$$

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. Moreover, we state that the PMC operator is neutral w.r.t. intents emissions and history expressions equivalence. Finally, for intent receivers, we can reduce $\bar{\alpha}_C h.H$ to H where the free instances of h are replaced by all the possible incoming intents.

Policy compliance is granted by the following property.

Lemma 2. $H \models \varphi_{//H'} \Longrightarrow H \parallel H' \models \varphi$

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

Example 5. *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*

$$\overbrace{(\mu h_1.(\texttt{www}_?(u) + \texttt{play}_?(\texttt{null}) \parallel h_1))}^{H_1} \parallel \overbrace{(\overline{\texttt{play}}_A h_2.\texttt{read}(\sim/\texttt{sav}) + \texttt{write}(\sim/\texttt{sav}) \cdot \texttt{www}_?(\text{"ad.com"}))}^{H_2} \parallel \overbrace{(\overline{\texttt{www}}_B h_3.\sum_u \texttt{connect}(u))}^{H_3}$$

It consists of the history expression for the `Browser` *receiver (H_3),* `Game` *receiver (H_2) and a (simplified) user interface (H_1). Terms H_2 and H_3 are generated by typing the methods* `receive` *of* `Browser` *and* `Game`*. 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 either open the browser on the homepage u (*$\texttt{www}_?(u)$*) or start a game (*$\texttt{play}_?(\texttt{null})$*).*

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 $H_1 \parallel H_2 \parallel H_3 \not\models \varphi$. First we compute $\varphi_{//H_1} = \varphi$ (trivial since H_1 contains no system actions) and then we prove that $H_2 \models \varphi$ as follows.*

$$\begin{array}{lrl}
\textit{1.} & \overline{\texttt{play}}_A h_2.\texttt{read}(\sim/\texttt{sav}) + \texttt{write}(\sim/\texttt{sav}) \cdot \texttt{www}_?(\text{"ad.com"}) \models & \neg\langle\texttt{read(x)}\rangle.\langle\texttt{connect(y)}\rangle.tt \wedge \neg\langle\texttt{write(x)}\rangle.\langle\texttt{connect(y)}\rangle.tt \quad \Leftrightarrow \\
\textit{2.} & \texttt{read}(\sim/\texttt{sav}) + \texttt{write}(\sim/\texttt{sav}) \cdot \texttt{www}_?(\text{"ad.com"}) \models & \neg\langle\texttt{read(x)}\rangle.\langle\texttt{connect(y)}\rangle.tt \wedge \neg\langle\texttt{write(x)}\rangle.\langle\texttt{connect(y)}\rangle.tt \quad \Leftrightarrow \\
\textit{3.} & \texttt{read}(\sim/\texttt{sav}) + \texttt{write}(\sim/\texttt{sav}) \cdot \texttt{www}_?(\text{"ad.com"}) \models & \left\{\begin{array}{ll} (4) & \neg\langle\texttt{read(x)}\rangle.\langle\texttt{connect(y)}\rangle.tt \quad \text{and} \\ (6) & \neg\langle\texttt{write(x)}\rangle.\langle\texttt{connect(y)}\rangle.tt \end{array}\right. \quad \Leftrightarrow \\
\textit{4.} & \texttt{read}(\sim/\texttt{sav}) + \texttt{write}(\sim/\texttt{sav}) \cdot \texttt{www}_?(\text{"ad.com"}) \models & \neg\langle\texttt{read(x)}\rangle.\langle\texttt{connect(y)}\rangle.tt \quad \Leftrightarrow \\
\textit{5.} & \varepsilon \not\models & \langle\texttt{connect(y)}\rangle.tt \quad \blacksquare \\
\textit{6.} & \texttt{read}(\sim/\texttt{sav}) + \texttt{write}(\sim/\texttt{sav}) \cdot \texttt{www}_?(\text{"ad.com"}) \models & \neg\langle\texttt{write(x)}\rangle.\langle\texttt{connect(y)}\rangle.tt \quad \Leftrightarrow \\
\textit{7.} & \texttt{www}_?(\text{"ad.com"}) \not\models & \langle\texttt{connect(y)}\rangle.tt \quad \Leftrightarrow \\
\textit{8.} & \varepsilon \not\models & \langle\texttt{connect(y)}\rangle.tt \quad \square
\end{array}$$

Hence, we compute $\varphi_{//H_2}$ as in the following way.

$$\neg\langle\texttt{read(x)}\rangle.\langle\texttt{connect(y)}\rangle.tt \wedge \neg\langle\texttt{write(x)}\rangle.\langle\texttt{connect(y)}\rangle.tt_{//\overline{\texttt{play}}_A h_2.\texttt{read}(\sim/\texttt{sav})+\texttt{write}(\sim/\texttt{sav})\cdot\texttt{www}_?(\text{"ad.com"})} \quad =$$

$$\neg\langle\texttt{read(x)}\rangle.\langle\texttt{connect(y)}\rangle.tt \wedge \neg\langle\texttt{write(x)}\rangle.\langle\texttt{connect(y)}\rangle.tt_{//\texttt{read}(\sim/\texttt{sav})+\texttt{write}(\sim/\texttt{sav})\cdot\texttt{www}_?(\text{"ad.com"})} \quad =$$

$$\bigwedge\begin{cases} \neg\langle\texttt{read(x)}\rangle.\langle\texttt{connect(y)}\rangle.tt_{//\texttt{read}(\sim/\texttt{sav})+\texttt{write}(\sim/\texttt{sav})\cdot\texttt{www}_?(\text{"ad.com"})} \\ \neg\langle\texttt{write(x)}\rangle.\langle\texttt{connect(y)}\rangle.tt_{//\texttt{read}(\sim/\texttt{sav})+\texttt{write}(\sim/\texttt{sav})\cdot\texttt{www}_?(\text{"ad.com"})} \end{cases} \quad =$$

$$\bigwedge\begin{cases} \neg(\langle\texttt{read(x)}\rangle.(\langle\texttt{connect(y)}\rangle.tt_{//\texttt{read}(\sim/\texttt{sav})+\texttt{write}(\sim/\texttt{sav})\cdot\texttt{www}_?(\text{"ad.com"})}) \vee (\langle\texttt{connect(y)}\rangle.tt_{//\varepsilon})) \\ \neg(\langle\texttt{write(x)}\rangle.(\langle\texttt{connect(y)}\rangle.tt_{//\texttt{read}(\sim/\texttt{sav})+\texttt{write}(\sim/\texttt{sav})\cdot\texttt{www}_?(\text{"ad.com"})}) \vee (\langle\texttt{connect(y)}\rangle.tt_{//\texttt{www}_?(\text{"ad.com"})})) \end{cases} \quad =$$

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

Finally, we just need to verify that $H_3 \not\models \varphi'$ which is obtained through the derivation below.

1. $\overline{\texttt{www}}_B h_3.\sum_u \texttt{connect}(u) \models \Big(\neg(\langle\texttt{read(x)}\rangle.\langle\texttt{connect(y)}\rangle.tt \vee \langle\texttt{connect(y)}\rangle.tt)\Big) \wedge \Big(\neg(\langle\texttt{write(x)}\rangle.\langle\texttt{connect(y)}\rangle.tt \vee \langle\texttt{connect(y)}\rangle.tt)\Big) \Rightarrow$
2. $\overline{\texttt{www}}_B h_3.\sum_u \texttt{connect}(u) \models \neg(\langle\texttt{read(x)}\rangle.\langle\texttt{connect(y)}\rangle.tt \vee \langle\texttt{connect(y)}\rangle.tt) \Rightarrow$
3. $\overline{\texttt{www}}_B h_3.\sum_u \texttt{connect}(u) \models \neg\langle\texttt{read(x)}\rangle.\langle\texttt{connect(y)}\rangle.tt \wedge \neg\langle\texttt{connect(y)}\rangle.tt \Rightarrow$
4. $\overline{\texttt{www}}_B h_3.\sum_u \texttt{connect}(u) \models \neg\langle\texttt{connect(y)}\rangle.tt \Rightarrow$
5. $\sum_u \texttt{connect}(u) \models \neg\langle\texttt{connect(y)}\rangle.tt \Rightarrow$
6. $\sum_u \texttt{connect}(u) \not\models \langle\texttt{connect(y)}\rangle.tt$ ■

However, the last relation is trivially false for each non-empty summation $\sum_u \texttt{connect}(u)$. Indeed, choosing an arbitrary element u' in the summation, we obtain $\texttt{connect}(u') \models \langle\texttt{connect(y)}\rangle.tt$.

5.2 Prototype description

We started the implementation of a prototype, called *BYODroid*, using the techniques described above for guaranteeing virtual organisations against security violations in BYOD scenarios. BYODroid consists of two main components: the *BYODroid Market* web service and an the *BYODroid Installer*.

The BYODroid Installer allows users to access the BYODroid Market for obtaining new applications that cannot generate security violations on their devices. Simply, it is installed when the user register his device to the virtual organisation. After registration it communicates the device configuration, i.e., the installed applications, to the BYODroid Market. If the configuration is a legal one, the registration successes and BYODroid Installer replaces the system application installer.

The BYODroid Market is responsible for holding the security state of registered devices and mediating the access to other application markets by only allowing a device to install legal applications. Basically, the BYODroid Market contains a *security policy manager* which maintains a policies dependency graph rooted in the virtual organisation policy. Each node of the graph contains information about devices security configuration. In particular, a node N is labelled with a policy φ_N, contains a list of device identifiers and has a finite set of outgoing edges labelled with application identifiers. When a new device D registers, its identity is associated to a node N in the graph such that the path from the root to N is labelled with the list of D's applications (if no such state exists it is created with using the node creation procedure below).

When a registered device D (appearing in node N) accesses the market service, the BYODroid Market retrieves the list of applications and generates a view for the BYODroid installer application. The view is a list of applications $A_1,\ldots,A_m$ such that if an edge from N labelled with A_i exists, then A_i is marked *safe*. Otherwise, A_i is marked *unchecked*. If a device D (being in node N) installs a safe applications A_s labelling an edge to N', its identity token is moved to N'. Instead, if D wants to install an unchecked application A_u, we proceed as follows. The BYODroid Market downloads A_u and applied the type and effect system of Section 4 to infer H_u. Then, we verify whether $H_u \not\models \varphi_N$. If it is the case, the installation is interrupted and the user alerted. Otherwise, we compute $\varphi_{N//H_u}$ (see Section 5), we use it to label a fresh node $\bar{N}$ and we add to the graph the edge from N to $\bar{N}$ labelled by A_u. Finally, D is moved to $\bar{N}$ an the installation can proceed.

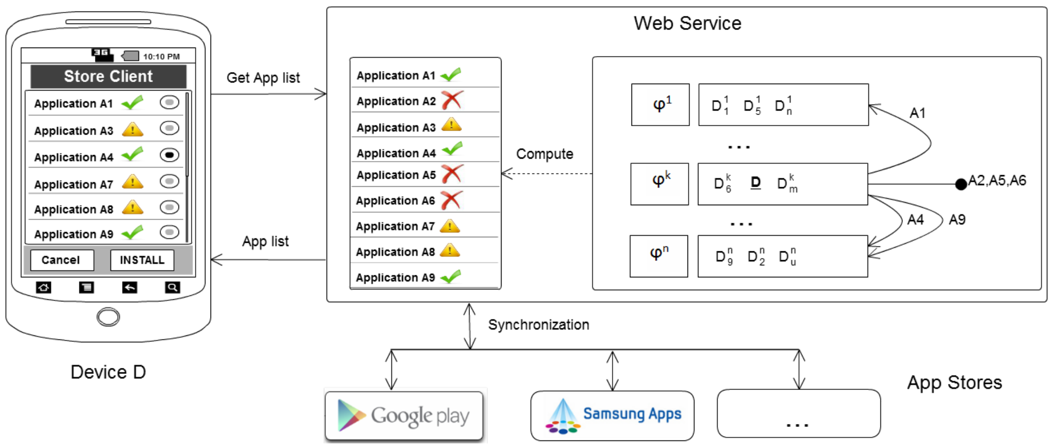

Figure 2:

In general, each node can also keep in memory a black list of failed installations, i.e., those for applications that do not fulfil the node policy. The applications in the black list simply do not appear in the list of available installations for devices being in the corresponding node.

The prototype structure and behaviour are sketched in Figure 2. Briefly, a device D requests the list of applications available for a new installation. The system checks the node containing D and generates a corresponding applications list that the user visualises.

6 Related work

Most of the early literature related to Android security tackles security and privacy issues by proposing modification of the Android architecture and related security mechanisms. In particular, literature 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 [21] Android security policy is analyzed in terms of efficacy and some extensions are proposed. Besides, in [17] authors propose an extension to the basic Android permission systems and corresponding new policies. Moreover, in [26] new privacy-related security policies are proposed for addressing security problems related to users' personal data. Related to ASF, many proposal have been made to extend native security mechanisms. For instance, [12] and [18] are focused on permissions: the first proposes a monitoring tool for assessing the actual privileges of Android applications while the latter describes SAINT, a modification to Android stack that allows to manage install-time permissions assignment. Other tools are mainly focused on malware detection (e.g. XManDroid [9] and Crowdroid [10]) and application certification (e.g. Scandroid [13]). Some works have also been carried out to detect vulnerabilities which are often independent from the Android version. Many of them show that the Android platform may suffer from DoS attacks [4], covert channels [20], and web attacks [16]. All the analysed approaches are unrelated and may work independently on the same Android stack. However, since different approaches often share common security features they should integrate one another. Such result is currently unachievable, due to the lack of a formal and comprehensive reference model for the security of the Android platform. To this aim, only recently researchers focused on the formal modeling and analysis of the Android platform and its security aspects. In [22] 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 [11] proposes a language-based approach to infer security properties from Android applications. Moreover, this work propose a type system that guarantees that well-typed programs respects user data access permissions. The type and effect system that we presented here extends the proposal of [11] as it also infers/verifies history expressions. History expressions can denote complex interactions and behaviours and which allow for the verification and enforcement of a rich class of security policies [1].

7 Conclusion and Related Work

In this work we presented a framework for modeling the behaviour 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 virtual organisations, 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 our prototype.

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 (NDSS'03), San Diego, California, USA*, pages 107–121, February 2003.

[2] H. R. Andersen. Partial model checking (extended abstract). In *Proc. of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS'95), Sand Diego, California, USA*, pages 398–407. IEEE, June 1995.

[3] A. Armando, G. Costa, and A. Merlo. Formal modeling and reasoning about the android security framework. In *Proc. of the 7th International Symposium on Trustworthy Global Computing(TGC'12), Newcastle upon Tyne, UK*, pages 1–20, September 2012.

[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 TC 11 Information Security and Privacy Conference (SEC'12), Heraklion, Crete, Greece*, pages 13–24, June 2012.

[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, June 2009.

[6] M. Bartoletti, P. Degano, and G. L. Ferrari. History-based access control with local policies. In *Proc of the 8th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'05), Edinburgh, UK*, pages 316–332, April 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 (FOSSACS'07), Braga, Portugal*, pages 32–47, March-Aprill 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, 2011.

[10] I. Burguera, U. Zurutuza., and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proc. of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM'11), NewYork, USA*, pages 15–26. ACM, October 2011.

[11] A. Chaudhuri. Language-based security on android. In *Proc. of the ACM SIGPLAN 4th Workshop on Programming Languages and Analysis for Security (PLAS '09), New York, USA*, pages 1–7. ACM, June 2009.

[12] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proc. of the 18th ACM conference on Computer and Communications Security (ACM CCS'11), Chicago, Illinois, USA*, pages 627–638. ACM, October 2011.

[13] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated security certification of android applications.

[14] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In *Proc. of the 7th Colloquium Noordwijkerhout, the Netherlands, LNCS*, volume 85, pages 299–309, July 1980.

[15] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: A minimal core calculus for java and gj. *Journal of ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, May 2001.

[16] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on webview in the android system. In *Proc. of the 27th Annual Computer Security Applications Conference (ACSAC'11), NewYork, USA*, pages 343–352. ACM, December 2011.

[17] 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 (ASIA CCS'10), Beijing, China*, pages 328–332. ACM, April 2010.

[18] M. Ongtang, S. Mclaughlin, W. Enck, and P. Mcdaniel. Semantically rich application-centric security in android. In *Proc. of the 25th Annual Computer Security Applications Conference (ACSAC'09), Honolulu, Hawaii, USA*, pages 340–349. ACM, December 2009.

[19] C. Pettey. Gartner says sales of mobile devices grew 5.6 percent in third quarter of 2011; smartphone sales increased 42 percent, November 2011. `http://www.gartner.com/it/page.jsp?id=1848514`.

[20] 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, San Diego, California, USA*, pages 17—-33, February 2011.

[21] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google Android: A comprehensive security assessment. *IEEE Security & Privacy*, 8(2):35–44, 2010.

[22] 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 2nd International Conference on Social Computing (SocialCom'10), Washington, DC, USA*, pages 944–951, August 2010.

[23] C. Skalka and S. Smith. History effects and verification. In *Proc. of the 2nd ASIAN Symposium on Programming Languages and Systems (APLAS '04), Taipei, Taiwan*, pages 107–128, November 2004.

[24] C. Skalka, S. F. Smith, and D. V. Horn. A type and effect system for flexible abstract interpretation of java. *Electronic Notes in Theorical Computer Science*, 131(1):111–124, may 2005.

[25] D. A. Willis. Bring your own device: New opportunities, new challenges. Gartner, August 2012. `http://www.gartner.com/id=2125515`.

[26] 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), Pittsburgh, PA, USA*, pages 93–107, June 2011.

[27] Z. Zorz. Pentagon officials allowed to use android. HELP NET SECURITY, December 2011. `http://www.net-security.org/secworld.php?id=12141`.

Alessandro Armando is associate professor at the University of Genova, where he received his Laurea degree in Electronic Engineering in 1988 and his Ph.D in Electronic and Computer Engineering in 1994. His appointments include a postdoctoral research position at the University of Edinburgh (1994-1995) and one as visiting researcher at INRIA-Lorraine in Nancy (1998-1999). He is co-founder and leader (since 2003) of the Artificial Intelligence Laboratory (AI-Lab) at DIBRIS. He is also head of the Security and Trust Research Unit at the Center for Information Technologies of Bruno Kessler Foundation in Trento. He has contributed to the discovery of a serious vulnerability on the SAML-based Single Sign-On for Google Apps and to the discovery and fixing of a vulnerability that leads to a Denial of Service attack on all Android devices. His current focus is on developing cutting-edge automated reasoning techniques and on using them to build a new generation of push-button software verification and debugging tools supporting the development of complex, large-scale, distributed IT applications.

Gabriele Costa is a researcher of the Artificial Intelligence Laboratory (AI-Lab) of the Informatics, Bioengineering, Robotics and System Engineering Department (DIBRIS) at the University of Genova. He received his PhD in Computer Science from the University of Pisa in 2011. From 2008 to 2011 he worked as researcher for the IIT intitute of the National Research Council (CNR) in Pisa. His main research activities and interests include language-based security, formal methods for security analysis, inforation flow analysis, distributed systems security, security verification and enforcement. He activelly participated in European projects Aniketos, CONNECT, NESSoS and SPaCIoS.

Alessio Merlo received his Ph.D. in Computer Science from University of Genova (Italy) where he worked on performance and access control issues related to Grid Computing. He is currently serving as a Researcher at eCampus University, Department of Engineering. His research interests are focused on performance and security issues related to Web and distributed systems (Grid, Cloud). He is currently working on security issues related to Android platform.

Luca Verderame is a graduated student, currently working at the Artificial Intelligence Laboratory, DIBRIS, University of Genova as a research fellow. He obtained his master degree in Computer Engineering, University of Genova, on march 2012. His research interests mainly cover information security applied, in particular, to mobile devices. Recently his work led to discover a previously unknown vulnerability in Android OS.