Formal Modeling and Reasoning about the Android Security Framework

Alessandro Armando[1,2], Gabriele Costa[2], and Alessio Merlo[3]

[1] Fondazione Bruno Kessler
armando@fbk.eu
[2] Università degli Studi di Genova
gabriele.costa@unige.it
[3] Università E-Campus
alessio.merlo@uniecampus.it

Abstract. Android OS is currently the most widespread mobile operating system and is very likely to remain so in the near future. The number of available Android applications will soon reach the staggering figure of 500,000, with an average of 20,000 applications being introduced in the Android Market over the last 6 months. Since many applications (e.g., home banking applications) deal with sensitive data, the security of Android is receiving a growing attention by the research community. However, most of the work assumes that Android meets some given high-level security goals (e.g. sandboxing of applications). Checking whether these security goals are met is therefore of paramount importance. Unfortunately this is also a very difficult task due to the lack of a detailed security model encompassing not only the interaction among applications but also the interplay between the applications and the functionalities offered by Android. To remedy this situation in this paper we propose a formal model of Android OS that allows one to formally state the high-level security goals as well as to check whether these goals are met or to identify potential security weaknesses.

1 Introduction

Modern smartphones not only act as cell phones, but also as handheld personal computers, where users manage their personal data, interact with online payment systems, and so on. As stated in [12], *"a central design point of the Android security architecture is that no application, by default, has permission to perform any operation that would adversely impact other applications, the operating system, or the user. This includes reading or writing the user's private data (such as contacts or e-mails), reading or writing another application's files, performing network access, keeping the device awake, etc."*. Android strives to achieve this security goal through a cross-layer security architecture, the Android Security Framework (ASF), leveraging the access control mechanisms offered by the underlying Linux kernel.

Recent work (e.g., [2,23,19,16]) unveiled a plethora of vulnerabilities occurring at different layers of the Android stack and a number of extensions to the Android native security policies (e.g., [17]) and to the framework itself (e.g., [13,8]) have been put forward. However, a systematic assessment of the ASF and of the proposed solutions is very difficult to achieve. Mainly, this is due to the lack of a detailed security model encompassing not only the interaction among applications but also the interplay between the applications and the functionalities offered by Android.

In this work we focus on modeling the Android OS in order to overcome the aforementioned aspects. The contribution of this paper is twofold. Firstly, we propose a formal model of Android that allows us to formally describe the security-relevant aspects of the ASF. Secondly, we present a type and effect system that we use for both producing the model of a platform and verifying whether it meets some expected security goals. For modeling, we adopt a process algebra-like formalism, namely *history expressions* [6], that can be exploited for different purposes, as we detail in the following.
Structure of the paper. In Section 2 we briefly introduce the architecture of Android and the principal interactions. In Section 3 we describe the ASF and its enforcement mechanisms. In Section 4 we present our formal model for the Android Security Framework. In Section 5 we present our type and effect system, we prove its key properties and we describe its possible exploitations. Finally, in Section 6 we draw some concluding remarks.

2 Android Architecture

The Android stack can be represented with 5 functional levels: Application, Application Framework, Application Runtime, Libraries and the underlying Linux kernel.

1. *Application Layer.* It includes both system (home,browser,email,..) and user-installed Java applications. Applications are made of *components* corresponding to independent execution modules, that interact with each others. There exist four kinds of components: 1) *Activity*, representing a single application screen with a user interface, 2) *Service*, which is kept running in background without interaction with the user, 3) *Content Provider*, that manages application data shared among components of (potentially) distinct applications, and 4) *Broadcast Receiver* which is able to respond to system-wide broadcast announcements coming both from other components and the system. Components are defined in *namespaces* that map components to a specific name which allow to identify components in the system.
2. *Application Framework.* It provides the main OS services by means of a set of APIs. This layer also includes services for managing the device and interacting with the underlying Linux drivers (e.g. *Telephony Manager* and *Location Manager*).
3. *Android Runtime.* This layer comprises the Dalvik virtual machine, the Android's runtime's core component which executes applications.
4. *Libraries.* It contains a set of C/C++ libraries providing useful tools to the upper layers and for accessing data stored on the device. Libraries are widely used by the Application Framework services.

5. *Linux kernel.* Android relies on a Linux kernel for core system services. Such services include process management and drivers for accessing physical resources and Inter-Component Communication (ICC).

2.1 Interactions in Android

In Android, interactions can be *horizontal* (i.e. application to application) or *vertical* (i.e. application to underlying levels). Horizontal interactions are used to exploit functionalities provided by other applications, while vertical ones are used to access system services and resources. Component services are invoked by means of a message passing paradigm, while resources are referred by a special formatted URI. Android URIs can also be used to address a content provider database.

Horizontal interactions are based on a message abstraction called *intent*. Intent messaging is a facility for dynamic binding between components in the same or different applications. An intent is a passive data structure holding an abstract description of an operation to be performed (called *action*) and optional data in URI format. Intents can be *explicit* or *implicit*. In the former case, the destination of the action is explicitly expressed in the intents (through the *name* of the receiving application/component), while in the latter case the system has to determine which is the target component accordingly to the action to be performed, the optional data value and the applications currently installed in the system.

Intent-based communications are granted by a kernel driver called Binder which offers a lightweight capability-based remote procedure call mechanism. Although intent messaging passes through the Binder driver, it is convenient to maintain intent's level of abstraction for modeling purpose. In fact, every Android application defines its entry points using *intent filters* which are lists of intent's actions that can be dispatched by the application itself. Furthermore, an intent can be used to start activities, communicate with a service or send broadcast messages.

Vertical interactions are used by applications to access system resources and functionalities which are exposed through a set of APIs. Although system calls can cause a cascade of invocations in the lower layers, possibly reaching the kernel, all of them are mediated by the application framework APIs. Hence, APIs mask internal platform details to the invoking applications. Internally, API calls are handled according to the following steps. When an application invokes a public API in the library, the invocation is redirected to a private interface, also in the library. The private interface is an RPC stub. Then, the RPC stub initiates an RPC request with the system process that asks a system service to perform the requested operation.

3 Android Security Framework

The Android Security Framework (ASF) consists of a set of decentralized security mechanisms spanning on all layers of the Android stack. The ASF enforces an informal and cross-layer security policy focused on the concept of *permission*.

3.1 Android Permissions

In Android, a permission is a string expressing the ability to perform a specific operation. Permissions can be system-defined or user-defined. Each application statically declares the set of permissions it requires to work properly. Such a set is generally a superset of the permissions effectively used at runtime. During installation of an application, the user must grant the whole set of required permissions, otherwise the installation is aborted. Once installed, an application cannot modify such permissions.

Each application package contains an XML file, called *Android Manifest*, containing two types of permissions:

– **declared permissions** are defined by the application itself and represent access rights that other applications must have for using its resources.
– **requested permissions** representing the permissions held by the application.

Since permissions specified in the manifest are static (i.e., they cannot possibly change at runtime), they are not suited to regulate access to resources that are dynamically defined by the application (e.g., shared data from a content provider). For this reason Android APIs include special methods for dynamically (i) *granting*, (ii) *revoking* and (iii) *checking* access permissions. These methods handle per-URI permissions thereby letting the application give temporary access privileges for some owned URI to other applications.

3.2 Android Security Policy

The Android security policy defines restrictions on the interactions among applications and between each application and the system. The Android security policy is globally enforced by the ASF. Both the policy and the ASF strongly rely on permissions associated with the components. We detail here the security policy related to the architecture and the interactions explained in Sec. 2.

Horizontal interactions. Horizontal interactions between components are carried out through permissions associated with intents. Each application can declare a list of permissions for their incoming intents. When application A sends an intent I to application B, the platform delivers I only if A has the privileges (granted at installation time) requested by B. Otherwise, the intent does not reach B (still, it could be delivered to other recipients).

Vertical interactions. By default, an Android application can only access a limited range of system resources. These restrictions are implemented in different forms. Some capabilities are restricted by an intentional absence of APIs mediating sensitive accesses. For instance, there is no API that allows for the direct manipulation of the SIM card. In other cases, the sensitive APIs are reserved for trusted applications and are protected through permissions.

Each API in the library layer is associated with a certain permission. Once a component invokes an API, the privileges of the component are checked against the permission required for the API. If an application attempts to invoke an API without having the proper privileges, a security exception is thrown and the invocation fails.

Linux layer and IPC. The Android platform takes advantage of the Linux user-based access control model as a means to identify and isolate application resources. The Android system assigns a unique user ID (UID) to each Android application and runs it as that user in a separate Linux process. This sets up a kernel-level *Application Sandbox.* This approach uses the native Linux isolation for users to implement a security policy that avoids direct communications among Android applications by forcing all their interactions to rely upon the IPC system. However, such a policy does not prevent a Linux process (running an application) from communicating through one of the native UNIX mechanisms such as sockets or files. Notice that the Linux permissions apply on such channels.

The previous analysis shows the cross-layer nature of Android Security policy. The ASF is distributed and involves distinct security-relevant aspects (from UID and GID at Linux layer to human-readable and high level Android permissions). Assessing the effectiveness of all the security mechanisms and their interplay is difficult due to such heterogeneity and the lack of a detailed and comprehensive model of the security-relevant aspects of Android.

4 Android Model

In this section we describe how we model Android applications and components. In particular, we introduce a framework for defining an application in terms of its (*i*) components, (*ii*) manifest and (*iii*) name space. Moreover, we present a formal semantics for describing computations in our model. We believe that our framework can be used to accurately describe most of the security-relevant aspects of the Android OS. Indeed, even though Java-like languages have been proposed for the application of formal methods, e.g., see [7,15], here we aim at focussing on application-to-application and application-to-system interactions which do not depend on object orientation. To this purpose, we show, through examples, that our model covers a number of security flaws that have been recently reported.

4.1 Applications and Components

In Table 1 we report the syntax of the elements of our framework.

Intuitively, an application A is a triple consisting of a manifest M, a naming function $\varDelta$ and a finite list of components $\bar{C} = C_1 \ldots C_n$. A manifest M contains three parts: requested permissions $\varPi$, declared permissions P and an intents resolution function $\varLambda$. The permission request part $\varPi$ is a finite sequence of (*i*) intent permission requests $\rho\alpha$ and (*ii*) system permission requests $\rho\sigma$. Instead, the permission declaration P is a list of pairs $(\alpha, \bar{u})$ binding intent names α, α'

Table 1. Syntax of applications and components

$A ::= \langle M, \varDelta, \bar{C} \rangle$	Application
$M ::= \varPi; P; \varLambda$	Manifest
$\varPi ::= \varepsilon \mid \rho\alpha.\varPi \mid \rho\sigma.\varPi$	Permission requests
$P ::= \varepsilon \mid (\alpha, \bar{u}).P$	Exported permissions
$\varLambda ::= \varepsilon \mid (\alpha \mapsto \eta).\varLambda$	Intent binding
$\varDelta ::= \emptyset \mid \varDelta\{C/\eta\}$	Name space
$C ::= \texttt{skip} \mid \texttt{icast}\, E \mid \texttt{ecast}\, \eta\, E \mid \texttt{grant}_\sigma\, \eta\, E \mid$	Statements
$\texttt{revoke}_\sigma\, \eta\, E \mid \texttt{check}_\sigma\, \eta\, E \mid \texttt{new}\, x\, \texttt{in}\, C \mid \texttt{receive}_\alpha\, x \mapsto C \mid$	
$\texttt{apply}\, E\, \texttt{to}\, E' \mid \texttt{system}_\sigma\, E \mid \texttt{if}\, (E = E')\{C\}\, \texttt{else}\, \{C'\} \mid C; C'$	
$E ::= \texttt{null} \mid u \mid x \mid I_\alpha(E, E') \mid E.\texttt{d} \mid$	Expressions
$E.\texttt{e} \mid \texttt{proc}\, f(x)\{C\}$	

to lists of resources $\bar{u} = u_1, \ldots, u_k$. The function $\varLambda$ maps each intent name α to a set of (identifiers of) components $\{\eta_1, \ldots, \eta_n\}$ that can serve it, namely the available *receivers*. Finally, $\varDelta$ resolves components identifier η, η' into actual components C, C'.

Software components are obtained from the composition of *statements* and *expressions*. Expressions, ranged over by E, E', can be `null`, resources u, u', variables x, y, intents constructors $I_\alpha(E, E')$, data and extra field getters ($E.\texttt{d}$ and $E.\texttt{e}$, respectively) or procedure declarations $\texttt{proc}\, f(x)\{C\}$ (where f is bound in C).

Similarly, statements, denoted by C, C', can be a `skip` command, an implicit intent cast $\texttt{icast}\, E$, an explicit intent cast $\texttt{ecast}\, \eta\, E$, an access permission grant $\texttt{grant}_\sigma\, \eta\, E$, a permission revocation $\texttt{revoke}_\sigma \eta\, E$, a permission checking $\texttt{check}_\sigma\, \eta\, E$, a fresh resource creation $\texttt{new}\, x\, \texttt{in}\, C$, an intent receiver $\texttt{receive}_\alpha\, x \mapsto C$, an application of a procedure to a parameter $\texttt{apply}\, E\, \texttt{to}\, E'$, a system call $\texttt{system}_\sigma\, E$, a conditional branching $\texttt{if}\, (E = E')\{C\}\, \texttt{else}\, \{C'\}$ or a sequence $C; C'$.

4.2 Operational Semantics

The behaviour of programs follows the small step semantics rules given in Table 2. Computations are sequences of reductions steps from a source configuration to a target one. For expressions, a configuration only contains the element under evaluation E. The operational semantics reduces expressions E, E' to *values* v, v'. A value can be either the void element $\bot$, a resource u, a intent $I_\alpha(u, v)$, or a procedure $\texttt{proc}\, f(x)\{C\}$. If no reductions apply to a configuration E (where E is not a value), we write $E \not\rightarrow$ and we say it to be *stuck*.

The semantic rules for commands are more tricky. Basically, we evaluate statements under a configuration $U, \varPhi, C$ where C is the program under computation, U is a resources ownership function (i.e., $U(\eta) = \mathcal{U}$ means that the component

Table 2. Semantics of expressions and statements (fragment)

$$(\texttt{E-NULL})\ \texttt{null} \to \bot \quad (\texttt{E-FLD})\ \frac{E \to E'}{E.f \to E'.f} \quad (\texttt{E-DATA})\ I_\alpha(u,v).\texttt{d} \to u$$

$$(\texttt{E-EXT})\ I_\alpha(u,v).\texttt{e} \to v \quad (\texttt{E-INT}_\texttt{L})\ \frac{E \to E''}{I_\alpha(E,E') \to I_\alpha(E'',E')} \quad (\texttt{E-INT}_\texttt{R})\ \frac{E \to E'}{I_\alpha(v,E) \to I_\alpha(v,E')}$$

$$(\texttt{S-SKIP})\ U,\Phi,\texttt{skip} \rightsquigarrow U,\Phi,\cdot \qquad (\texttt{S-GRNT})\ \frac{\texttt{self} = \eta' \quad u \in U(\eta') \quad \Phi' = \Phi \cup \{(\eta,\sigma,u)\}}{U,\Phi,\texttt{grant}_\sigma\, \eta\, u \rightsquigarrow U,\Phi',\cdot}$$

$$(\texttt{S-REVK})\ \frac{\texttt{self} = \eta' \quad u \in U(\eta') \quad \Phi' = \Phi \setminus \{(\eta,\sigma,u)\}}{U,\Phi,\texttt{revoke}_\sigma\, \eta\, u \rightsquigarrow U,\Phi',\cdot}$$

$$(\texttt{S-ICST})\ \frac{\texttt{self} = \eta \quad \eta' \in \Lambda(\alpha) \quad \eta,\alpha,u \models \Phi}{U,\Phi,\texttt{icast}\, I_\alpha(u,v) \overset{\alpha_\eta^{\eta'}(u,v)}{\rightsquigarrow}_\Lambda U,\Phi,\cdot} \quad (\texttt{S-ECST})\ \frac{\texttt{self} = \eta \quad \eta' \in \Lambda(\alpha) \quad \eta,\alpha,u \models \Phi}{U,\Phi,\texttt{ecast}\, \eta'\, I_\alpha(u,v) \overset{\alpha_\eta^{\eta'}(u,v)}{\rightsquigarrow}_\Lambda U,\Phi,\cdot}$$

$$(\texttt{S-CHK})\ \frac{\eta,\sigma,u \models \Phi}{U,\Phi,\texttt{check}_\sigma\, \eta\, u \rightsquigarrow U,\Phi,\cdot} \quad (\texttt{S-SYS})\ \frac{\texttt{self} = \eta \quad \eta,\sigma,u \models \Phi}{U,\Phi,\texttt{system}_\sigma\, u \overset{\sigma_\eta(u)}{\rightsquigarrow} U,\Phi,\cdot}$$

$$(\texttt{S-NEW})\ \frac{\texttt{self} = \eta \quad \text{fresh}\, u}{U,\Phi,\ \texttt{new}\, x\, \texttt{in}\, C \rightsquigarrow U \cup \{\eta/u\},\Phi,C[u/x]}$$

$$(\texttt{S-APP})\ U,\Phi,\texttt{apply proc}\, h(y)\{C\}\, \texttt{to}\, v \rightsquigarrow U,\Phi,C[v/y,\texttt{proc}\, h(y)\{C\}/h]$$

$$(\texttt{S-CND})\ U,\Phi,\ \texttt{if}\ (v = v')\{C_{tt}\}\, \texttt{else}\, \{C_{\mathit{ff}}\} \rightsquigarrow U,\Phi,C_{\mathcal{B}(v=v')}$$

$$(\texttt{S-SEQ})\ \frac{U,\Phi,C \overset{b}{\rightsquigarrow}, U',\Phi',C''}{U,\Phi,C;C' \overset{b}{\rightsquigarrow} U',\Phi',C'';C'} \qquad (\texttt{S-SEQ}^-)\ U,\Phi,\cdot;C \rightsquigarrow U,\Phi,C$$

(identified by) η owns the resources in $\mathcal{U}$) and Φ is the system policy (we write $\eta,\beta,u \models \Phi$ for $(\eta,\beta,u) \in \Phi$ with $\beta \in \{\alpha,\sigma\}$). Slightly abusing the notation, we also use $\cdot$ in the configuration to represent computation termination.

Computational steps consist of transitions from a source configuration to a target one. Transitions have the form $U,\Phi,C \overset{a}{\rightsquigarrow}_\Lambda U',\Phi',C'$ where a is an observable action, i.e., an intent or a system call, that the computation can perform and Λ is an intents destination table, i.e, $\Lambda(\alpha) = \{\eta_1,\ldots,\eta_k\}$ means that $\eta_1,\ldots,\eta_k$ are the candidates for handling an intent α. When not necessary, we feel free to omit a and Λ from the transitions.

According to the rules of table 2,[1] the commands behave as follows. The statement `skip` does not change the system state and terminates (`S-SKIP`). Both implicit (rule (`S-ICST`)) and explicit (rule (`S-ECST`)) casts produce an observable action $\alpha_\eta^{\eta'}(u,v)$ and reduce to $\cdot$. The only difference is that the receiver η' for an implicit cast can be any of the elements of the destination table Λ, while

[1] For brevity, table 2 only reports the rules which are more interesting our presentation. The full semantics can be found at `http://www.ai-lab.it/merlo/publications/AndroidModel.pdf`

an explicit cast declares the destination (which still must be a legal one, i.e., $\eta' \in \Lambda$). Note that, these reduction steps take place only if they are allowed by the current policy Φ. Permission granting and revocation (rules (S−GRNT) and (S−REVK)) are symmetrical. Indeed, granting a permission causes the current policy to be extended with a possibly new, allowed action, while revocation removes some existing privileges. Both the operations require u to be owned by the executing component, i.e., $u \in U(\eta')$ where $\texttt{self} = \eta'$. Then, a security check $\texttt{check}_\sigma\, \eta\, u$ interrupts the computation if η has no rights to access to u through σ (rule (S−CHK)). A system call $\texttt{system}_\sigma u$ is performed (rule (S−SYS)) if the current component is allowed to invoke it and generates a corresponding access action $\sigma_\eta(u)$ (where η is the source of the access σ). Resource creation (S−NEW) causes a statement C to be evaluated under a state in which a fresh resource u is associated to the variable x. As expected, the owner of the resource is the current component. Procedure application (rule (S−APP)) reduces to the computation of the procedure body C where the formal parameter y and the variable h are replaced with the actual parameter v and the procedure definition, respectively. A conditional statement (rule (S−CND)) reduces to one of its branches depending on the value of its guard (we write $\mathcal{B}(v = v')$ as an abbreviation of the two conditions $v = v'$ and $v \neq v'$ which evaluate to either *tt* or *ff*). Finally, a sequence of statements $C; C'$ behaves like C until it terminates and then reduces to the execution of C' (rules (S−SEQ) and (S−SEQ$^-$)).

In addition to the standard syntax, we define the following abbreviations which we adopt for the sake of presentation.

$$\texttt{if}\,(E \neq E')\{C\}\,\texttt{else}\,\{C'\} \triangleq \texttt{if}\,(E = E')\{C'\}\,\texttt{else}\,\{C\}$$

$$\texttt{if}\,(E_1 = E_1' \wedge E_2 = E_2')\{C\}\,\texttt{else}\,\{C'\} \triangleq \texttt{if}\,(E_1 = E_1')\{\,\texttt{if}\,(E_2 = E_2')\{C\}\,\texttt{else}\,\{C'\}\}\,\texttt{else}\,\{C'\}$$

$$\texttt{if}\,(E_1 = E_1' \vee E_2 = E_2')\{C\}\,\texttt{else}\,\{C'\} \triangleq \texttt{if}\,(E_1 \neq E_1')\{\,\texttt{if}\,(E_2 \neq E_2')\{C'\}\,\texttt{else}\,\{C\}\}\,\texttt{else}\,\{C\}$$

$$\texttt{if}\,(E \in \mathcal{U})\{C\}\,\texttt{else}\,\{C'\} \triangleq \texttt{if}\,(\bigvee_{u \in \mathcal{U}} E = u)\{C\}\,\texttt{else}\,\{C'\}$$

$$\texttt{while}\,(E \neq v)\,\texttt{do}\,\{C\} \triangleq \texttt{apply}\,\texttt{proc}\,w(x)\{\,\texttt{if}\,(E \neq x)\{C;\,\texttt{apply}\,w\,\texttt{to}\,E\}\,\texttt{else}\,\{\,\texttt{skip}\,\}\,\texttt{to}\,v$$

Finally, we say that a configuration is *stuck* (we write $U, \Phi, S \not\rightsquigarrow_\Lambda$) if $S \neq \cdot$ and the configuration admits no transitions. In real Android systems, this situation corresponds to program termination or exception raising, but this aspect does not impact on our framework. If a configuration reduces to a stuck one, we say it to *go wrong*.

Example 1. Consider the following statement.

$$C = \texttt{apply}\,\texttt{proc}\,f(x)\{\texttt{system}_\sigma\,x;\ \texttt{icast}\,I_\alpha(x, \texttt{null})\}\,\texttt{to}\,u$$

We simulate a computation under a configuration U, Φ, C where $\Phi = \{(\eta, \sigma, u)\}$ and $\texttt{self} = \eta$. The resulting computation follows.

$$U, \Phi,\ \texttt{apply}\,\texttt{proc}\,f(x)\{\texttt{system}_\sigma\,x;\ \texttt{icast}\,I_\alpha(x,\texttt{null})\}\,\texttt{to}\,u \rightsquigarrow_\Lambda U, \Phi, \texttt{system}_\sigma\,u;\ \texttt{icast}\,I_\alpha(u,\texttt{null})$$
$$\overset{\sigma_\eta(u)}{\rightsquigarrow_\Lambda} U, \Phi, \cdot;\ \texttt{icast}\,I_\alpha(u,\texttt{null}) \rightsquigarrow_\Lambda U, \Phi,\ \texttt{icast}\,I_\alpha(u,\texttt{null})$$

The first step consists of a procedure application to an argument u. This reduces the statement to the procedure body where the variable x is replaced by u. The next step is a system call σ. Since η is allowed to perform access, i.e., $\eta, \sigma, u \models \Phi$, the statement fires the corresponding action and reduces to an implicit cast statement. Then, as $\eta, \alpha, u \not\models \Phi$, the computation cannot proceed further and the configuration is stuck.

4.3 Execution Context

As described in Section 2, application manifests declare (i) *activities*, (ii) *receivers* and (iii) *content providers*. The information contained in the manifest contribute to defining how the components interact with each other and with the platform. We describe this mechanism by means of an *execution context* (and its semantics) which we define below.

Definition 1. *An execution context (context for short) is* $\mathbf{P} = U, \Phi, [C_1]_{\eta_1} \cdots [C_n]_{\eta_n}$. *The operational semantics of a context is defined by the rules*

$$\texttt{(CTX-S)} \quad \frac{U, \Phi, C_j \overset{b}{\rightsquigarrow}_{\Lambda} U', \Phi', C'}{U, \Phi, [C_1]_{\eta_1} \cdots [C_j]_{\eta_j} \cdots [C_n]_{\eta_n} \overset{b}{\Rightarrow}_{\Lambda} U', \Phi', [C_1]_{\eta_1} \cdots [C']_{\eta_j} \cdots [C_n]_{\eta_n}}$$

$$\texttt{(CTX-I)} \quad \frac{U, \Phi, C_i \overset{\alpha_{\eta_i}^{\eta_j}(u,v)}{\rightsquigarrow}_{\Lambda} U', \Phi', C'}{U, \Phi, \cdots [C_i]_{\eta_i} \cdots [\texttt{receive}\, x \mapsto C]_{\eta_j} \cdots \Rightarrow_{\Lambda} U', \Phi', \cdots [C']_{\eta_i} \cdots [C\{I_\alpha(u,v)/x\}]_{\eta_j} \cdots}$$

Intuitively, the state of a platform is entirely defined by its execution context, i.e., the configuration of the components running on it. Each component C is wrapped by a local context $[\cdot]_\eta$ labelled with its name. The execution context changes according to the computational steps performed by the components running on it and can see any action b (rule (CTX−S)). Also, the context provides the support for the intent-based communications (rule (CTX−I)). In practice, the context observes an action $\alpha_{\eta_i}^{\eta_j}(u, v)$ fired by a component η_i and delivers it to the right destination η_j.

When a platform is initialised, e.g., at system boot, a default, starting context is created. We now present the procedure that, given a system $S = A_1, \ldots, A_n$, returns the corresponding initial context. To do that, we introduce some preliminary notions.

Definition 2. *Given an application* $A = \langle M, \Delta, \bar{C} \rangle$ *such that* $M = \Pi; P; \Lambda$ *we define:*

– *the* permissions set *of A, in symbols* $\mathsf{Perm}(A) = \{\!|\, P \,|\!\}$*, where*

$$\{\!|\, \varepsilon \,|\!\} = \emptyset \qquad \{\!|\, (\alpha, \bar{u}).P' \,|\!\} = \bigcup_{u_i \in \bar{u}} \{(\alpha, u_i)\} \cup \{\!|\, P' \,|\!\}$$

– *the* privileges set *of A, in symbols* $\mathsf{Priv}_{\mathcal{P}}(A) = \bigcup_{\eta \in dom(\Delta)} \langle\!\langle \Pi \rangle\!\rangle^{\eta}_{\mathcal{P}}$*, where $\mathcal{P}$ is a permissions set and*

$$\langle\!\langle \varepsilon \rangle\!\rangle^{\eta}_{\mathcal{P}} = \emptyset \qquad \langle\!\langle \rho\sigma.\Pi \rangle\!\rangle^{\eta}_{\mathcal{P}} = \langle\!\langle \Pi \rangle\!\rangle^{\eta}_{\mathcal{P}} \cup \bigcup_{u} \{\sigma(u)\} \qquad \langle\!\langle \rho\alpha.\Pi' \rangle\!\rangle^{\eta}_{\mathcal{P}} = \langle\!\langle \Pi' \rangle\!\rangle^{\eta}_{\mathcal{P}} \cup \bigcup_{\tau} \{\alpha_{\eta}(u,\tau) \mid (\alpha,u) \in \mathcal{P}\})$$

Briefly, $\mathsf{Perm}(A)$ is the set of new permissions which A exposes in its manifest while $\mathsf{Priv}_{\mathcal{P}}(A)$ is the set of privileges it requests. We also write $\mathsf{Perm}(S)$ and $\mathsf{Priv}_{S}(A)$, where $S = A_1, \ldots, A_n$, as a shorthand for $\bigcup_i \mathsf{Perm}(A_i)$ and $\mathsf{Priv}_{\mathsf{Perm}(S)}(A)$, respectively. Even though Android does not check intents' extras, we annotate privileges with types τ, τ' (see Section 5). Intuitively, the expression $\bigcup_{\tau}\{\alpha_{\eta}(u,\tau) \mid \ldots\}$ denotes the set of intents α coming from η and carrying data u, no matter what extra (of type) τ they contain. We can now explain how we create an initial context.

Definition 3. *Given a system $S = A_1, \ldots, A_n$, such that $A_i = \langle M_i, \Delta_i, C^i_1 \ldots C^i_{k_i}\rangle$*[2] *and $M_i = \Pi_i; P_i; \Lambda_i$, we define*

– $\mathbf{U}_S = \lambda\eta.\emptyset$*;*
– $\mathbf{\Phi}_S = \bigcup_{A_i \in S} \{(\eta, \sigma, u) \mid \sigma_{\eta}(u) \in \mathsf{Priv}_S(A_i)\} \cup \bigcup_{A_i \in S} \{(\eta, \alpha, u) \mid \alpha_{\eta}(u,\tau) \in \mathsf{Priv}_S(A_i)\}$*;*
– $\mathbf{\Lambda}_S = \lambda\alpha. \bigcup_i \Lambda_i(\alpha)$*;*

Then, the default context for S is $\mathbf{U}_S, \mathbf{\Phi}_S, [C^1_1]_{\eta^1_1} \cdots [C^n_{k_n}]_{\eta^n_{k_n}}$ where $C^i_j = \Delta_i(\eta^i_j)$. The computation is then driven by $\Rightarrow_{\mathbf{\Lambda}_S}$.

In words, when a platform is initialised, all the components are loaded in the execution context. Also, the applications contribute to create the ownership function $\mathbf{U}_S$, the policy $\mathbf{\Phi}_S$ and the destinations table $\mathbf{\Lambda}_S$. Initially, we assume no resources to be owned by the applications, i.e., $\mathbf{U}_S = \lambda\eta.\emptyset$. Note that still resources can exist and we call them *static* or *system* resources. Instead, the system policy $\mathbf{\Phi}_S$ is obtained from the union of all the privileges requested by the applications (according to the existing permissions). In particular, we combine the privileges for the system calls, i.e., (η, σ, u) and those for intents, i.e., (η, α, u). The destination table $\mathbf{\Lambda}_S$ is straightforward: for each intent α it returns the set of all the declared receivers. Finally, all the components are labelled with the unique name[3] that is declared in the name space function of their application.

Example 2. We propose the following implementation of the Denial of Service (DoS) attack reported in [2]. The *zygote* socket is a system resource of the Android platform. Briefly, upon receiving a request (intent `fork`) from an application, the system connects to the zygote (system call `zygote`) for creating and starting a new process. For balancing the computational load, the system service grants that only certain processes can be allocated (we assume a finite set $T = \{t_1, \ldots, t_k\}$). We model the corresponding component as

$$C_Z = \mathtt{receive}_{\mathtt{fork}} w \mapsto \mathtt{if}\,(w.\mathsf{d} \in T)\{\mathtt{system}_{\mathtt{zygote}} w.\mathsf{d}\}\,\mathtt{else}\,\{\,\mathtt{skip}\,\}$$

[2] We also assume that $\forall i, j. dom(\Delta_i) \cap dom(\Delta_j) = \emptyset$.
[3] Recall that we assumed $\forall i, j. i \neq j \Rightarrow dom(\Delta_i) \cap dom(\Delta_j) = \emptyset$.

and then the service application is $A_Z = \langle M_Z; \Delta_Z; C_Z \rangle$ with $M_Z = \rho\,\texttt{zygote}.\varepsilon$; $(\texttt{fork}, T).\varepsilon$;$(\texttt{fork} \mapsto \eta_Z).\varepsilon$ and $\Delta_Z(\eta_Z) = C_Z$.

Due to a wrong implementation of the access permissions, any application having the network privileges can communicate with the zygote socket. Hence, the application $A = \langle M, \Delta, C \rangle$ where $M = \rho\,\texttt{zygote}.\varepsilon;\varepsilon;(\texttt{start} \mapsto \eta).\varepsilon$ and $\Delta(\eta) = C$ with $C = \texttt{new}\, x\, \texttt{in}\, \texttt{system}_{\texttt{zygote}} x$.

The elements of the initial context for $S = A, A_Z$ (see definition 3) are

– $\mathbf{U}_S = \lambda\eta.\emptyset$ and $\mathbf{\Phi}_S = \{(\eta_Z, \texttt{zygote}, \_), (\eta, \texttt{zygote}, \_)\}$ (where _ means "any value");
– $\mathbf{\Lambda}_S$ such that $\mathbf{\Lambda}_S(\texttt{fork}) = \{\eta_Z\}$.

Hence, the initial context is $\mathbf{U}_S, \mathbf{\Phi}_S, [C]_\eta [C_Z]_{\eta_Z}$. A possible reduction for it is (CTX−S)

$$\mathbf{U}_S, \mathbf{\Phi}_S, [\texttt{new}\, x\, \texttt{in}\, \texttt{system}_{\texttt{zygote}} x]_\eta [C_Z]_{\eta_Z} \Rightarrow_{\mathbf{\Lambda}_S} \mathbf{U}_S \cup \{\eta/u\}, \mathbf{\Phi}_S, [\texttt{system}_{\texttt{zygote}} u]_\eta [C_Z]_{\eta_Z}$$

where u is a fresh resource. A further step is again (CTX−S)

$$\mathbf{U}_S \cup \{\eta/u\}, \mathbf{\Phi}_S, [\texttt{system}_{\texttt{zygote}} u]_\eta [C_Z]_{\eta_Z} \Rightarrow_{\mathbf{\Lambda}_S} \mathbf{U}_S \cup \{\eta/u\}, \mathbf{\Phi}_S, [\cdot]_\eta [C_Z]_{\eta_Z}$$

This last reduction is legal for the platform since $(\eta, \texttt{zygote}, u) \in \mathbf{\Phi}_S$. However, as $u \notin T$, this operation corresponds to a violation of the requirement described above.

5 Type and Effect

In this section we present our type and effect system for the language introduced in Section 4. Also, we conclude this section with a brief dissertation about the advantages and the possible applications of history expressions for the analysis and verification of security properties which we plan to investigate in future work.

5.1 History Expressions

The type and effect system assigns types to expressions and *history expressions* to statements. Intuitively, a history expression represents the security-relevant, side effects produced by computations. History expressions are defined through the following syntax.

Definition 4. *(Syntax of history expressions)*

$$\begin{aligned} H, H' ::= \; & \varepsilon \mid h \mid \alpha_\eta(u, \tau) \mid \bar{\alpha}_\eta h.H \mid \sigma_\eta(u) \mid \upharpoonleft^\eta_{\sigma,u} \mid \downharpoonleft^\eta_{\sigma,u} \mid ?^\eta_{\sigma,u} \mid \\ & \nu u.H \mid H \cdot H' \mid H + H' \mid H \parallel H' \mid \mu h.H \mid H\backslash_L \end{aligned}$$

Briefly, they can be empty ε, variables h, h', parametric actions α_η, input prefixed expressions $\bar{\alpha}_\eta h.H$, system actions σ_η, permission granting $\upharpoonleft^\eta_{\sigma,u}$, permission revocations $\downharpoonleft^\eta_{\sigma,u}$, permission checks $?^\eta_{\sigma,u}$, resource creation $\nu u.H$, sequences

Table 3. History expressions semantics

$$\alpha_\eta(u,\tau) \xrightarrow{\alpha_\eta(u,\tau)} \varepsilon \qquad \sigma_\eta(u) \xrightarrow{\sigma_\eta(u)} \varepsilon \qquad \upharpoonleft^\eta_{\sigma,u} \xrightarrow{\upharpoonleft^\eta_{\sigma,u}} \varepsilon \qquad \downharpoonleft^\eta_{\sigma,u} \xrightarrow{\downharpoonleft^\eta_{\sigma,u}} \varepsilon \qquad ?^\eta_{\sigma,u} \xrightarrow{?^\eta_{\sigma,u}} \varepsilon \qquad H \dot{\rightarrow} H$$

$$\nu u.H \dot{\rightarrow} H \qquad \frac{H \xrightarrow{a} H''}{H \parallel H' \xrightarrow{a} H'' \parallel H'} \qquad \frac{H' \xrightarrow{a} H''}{H \parallel H' \xrightarrow{a} H' \parallel H''} \qquad \frac{H \xrightarrow{\alpha_{\eta'}(u,\tau)} H''}{H \parallel \bar{\alpha}_\eta h.H' \dot{\rightarrow} H'' \parallel H'\{\alpha_\eta(u,\tau)/h\}}$$

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

$$[\![H]\!] = \{a_1 \dots a_n \mid \exists H'.H \xrightarrow{a_1} \cdots \xrightarrow{a_n} H'\}$$

$H \cdot H'$, non deterministic choices $H + H'$, concurrent compositions $H \parallel H'$, recursions $\mu h.H$ or action restrictions $H\backslash_L$.

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

As expected, most of the transitions of Table 3 are common to many process algebrae semantics. In particular, history expressions $\alpha_\eta(u,\tau)$, $\sigma_\eta(u)$, $\upharpoonleft^\eta_{\sigma,u}$, $\downharpoonleft^\eta_{\sigma,u}$ and $?^\eta_{\sigma,u}$ simply fire the corresponding actions and reduce to ε. A sequence $H \cdot H'$ behaves like H until $H = \varepsilon$ (in which case we force $\varepsilon \cdot H' = H'$), while a resource creation $\nu u.H$ reduces to H producing no visible effects. Instead, a restriction $H\backslash_L$ makes the same transitions as H, provided they are allowed by L, i.e., $a \in L$. Two concurrent history expressions $H \parallel H'$ admit different reductions: either one of the two sub-expressions independently performs one step or both of them synchronise on a certain action. In order to perform a synchronisation, one of the two must be a *receiver* for an action emitted by the other, i.e., $\bar{\alpha}_{\eta'} h.H$. Note that received actions are relabelled with the identity of the receiver. Instead, Non conditional choice $H + H'$ can behave like H or H', respectively. Finally, a recursive history expression $\mu h.H$ can reduce to H where the instances of the variable h have been replaced by the recursive expression.

Denotational semantics function $[\![\cdot]\!]$ maps each history expression H into a set of finite execution traces which H can generate.

5.2 Type and Effect System

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

Definition 5. *(Types and type environment)*

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

Table 4. Typing rules

$$(\mathtt{T_E{-}NULL})\ \Gamma \vdash \mathtt{null} : \mathbf{1} \qquad (\mathtt{T_E{-}RES})\ \Gamma \vdash u : \{u\} \qquad (\mathtt{T_E{-}VAR})\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad (\mathtt{T_E{-}PROC})\ \frac{\Gamma\{\tau/y, \tau \xrightarrow{H} \mathbf{1}/h\} \rhd^{\eta}_{O} C : H}{\Gamma \vdash \mathtt{proc}\, h(y)\{C\} : \tau \xrightarrow{H} \mathbf{1}}$$

$$(\mathtt{T_E{-}INT})\ \frac{\Gamma \vdash E : \mathcal{U} \quad \Gamma \vdash E' : \tau}{\Gamma \vdash I_\alpha(E, E') : \mathcal{I}_\alpha(\mathcal{U}, \tau)} \qquad (\mathtt{T_E{-}DATA})\ \frac{\Gamma \vdash E : \mathcal{I}_\alpha(\mathcal{U}, \tau)}{\Gamma \vdash E.\mathtt{d} : \mathcal{U}} \qquad (\mathtt{T_E{-}EXT})\ \frac{\Gamma \vdash E : \mathcal{I}_\alpha(\mathcal{U}, \tau)}{\Gamma \vdash E.\mathtt{e} : \tau}$$

$$(\mathtt{T_S{-}SKIP})\ \Gamma \rhd^{\eta}_{O} \mathtt{skip} : \varepsilon \qquad (\mathtt{T_S{-}ICST})\ \frac{\Gamma \vdash E : \mathcal{I}_\alpha(\mathcal{U}, \tau)}{\Gamma \rhd^{\eta}_{O} \mathtt{icast}\, E : \sum_{u \in \mathcal{U}} \alpha_\eta(u, \tau)} \qquad (\mathtt{T_S{-}ECST})\ \frac{\Gamma \vdash E : \mathcal{I}_\alpha(\mathcal{U}, \tau)}{\Gamma \rhd^{\eta}_{O} \mathtt{ecast}\, \eta'\, E : \sum_{u \in \mathcal{U}} \alpha_\eta(u, \tau)}$$

$$(\mathtt{T_S{-}SYS})\ \frac{\Gamma \vdash E : \mathcal{U}}{\Gamma \rhd^{\eta}_{O} \mathtt{system}_\sigma\, E : \sum_{u \in \mathcal{U}} \sigma_\eta(u)} \qquad (\mathtt{T_S{-}CHK})\ \frac{\Gamma \vdash E : \mathcal{U}}{\Gamma \rhd^{\eta'}_{O} \mathtt{check}_\sigma\, \eta\, E : \sum_{u \in \mathcal{U}} ?^{\eta}_{\sigma,u}}$$

$$(\mathtt{T_S{-}GRNT})\ \frac{\Gamma \vdash E : \mathcal{U}}{\Gamma \rhd^{\eta'}_{O} \mathtt{grant}_\sigma\, \eta\, E : \sum_{u \in \mathcal{U} \cap O} \upharpoonleft^{\eta}_{\sigma,u}} \qquad (\mathtt{T_S{-}REVK})\ \frac{\Gamma \vdash E : \mathcal{U}}{\Gamma \rhd^{\eta'}_{O} \mathtt{revoke}_\sigma\, \eta\, E : \sum_{u \in \mathcal{U} \cap O} \downharpoonleft^{\eta}_{\sigma,u}}$$

$$(\mathtt{T_S{-}APP})\ \frac{\Gamma \vdash E : \tau \xrightarrow{H} \mathbf{1} \quad \Gamma \vdash E' : \tau}{\Gamma \rhd^{\eta}_{O} \mathtt{apply}\, E\, \mathtt{to}\, E' : H} \qquad (\mathtt{T_S{-}SEQ})\ \frac{\Gamma \rhd^{\eta}_{O} C : H \quad \Gamma \rhd^{\eta}_{O} C' : H'}{\Gamma \rhd^{\eta}_{O} C; C' : H \cdot H'}$$

$$(\mathtt{T_S{-}NEW})\ \frac{\Gamma\{\{u\}/x\} \rhd^{\eta}_{O \cup \{u\}} C : H \quad \text{fresh } u}{\Gamma \rhd^{\eta}_{O} \mathtt{new}\, x\, \mathtt{in}\, C : \nu u.H} \qquad (\mathtt{T_S{-}RECV})\ \frac{\Gamma\{\mathcal{I}_\alpha(\mathcal{U}, \tau)/x\} \rhd^{\eta}_{O} C : H}{\Gamma \rhd^{\eta}_{O} \mathtt{receive}_\alpha\, x \mapsto C : \bar{\alpha}_\eta h.H}$$

$$(\mathtt{T_S{-}CND})\ \frac{\Gamma \rhd^{\eta}_{O} C : H \quad \Gamma \rhd^{\eta}_{O} C' : H}{\Gamma \rhd^{\eta}_{O} \mathtt{if}\, (E = E')\{C\}\, \mathtt{else}\, \{C'\} : H} \qquad (\mathtt{T_S{-}WKN})\ \frac{\Gamma \rhd^{\eta}_{O} C : H' \quad H' \sqsubseteq H}{\Gamma \rhd^{\eta}_{O} C : H}$$

A type can be a *unit* $\mathbf{1}$, a finite set of resources $\mathcal{U} = \{u_1, \ldots, u_n\}$, an intent $\mathcal{I}_\alpha(\mathcal{U}, \tau)$ or an annotated arrow $\tau \xrightarrow{H} \mathbf{1}$. We use annotated types in the style of [4,22] (to which we refer the reader for more details) for denoting the latent effect that a procedure can generate when applied to a target input. A type environment Γ maps variable names into types and can be either empty $\emptyset$ or a new binding in an existing environment $\Gamma\{\tau/x\}$.

Type judgements assign types to expressions and history expressions to statements. For expressions, the syntax is $\Gamma \vdash E : \tau$ and shall be read "expression E has type τ under environment Γ". Similarly, for statements we have $\Gamma \rhd^{\eta}_{O} C : H$ with the meaning that, under environment Γ, statement C (which is part of package η) generates effect H. Also, we use O to denote the set of resources owned by the package η. The rules of the type and effect system are reported in Table 4.

In words, the expression `null` has type $\mathbf{1}$ and a resource u has type $\{u\}$ (rules ($\mathtt{T_E{-}NULL}$) and ($\mathtt{T_E{-}RES}$)). Instead, the type of a variable x is provided by the environment Γ (rule ($\mathtt{T_E{-}VAR}$)). Procedures require more attention (rule ($\mathtt{T_E{-}PROC}$)). Indeed, we say that a procedure $\mathtt{proc}\, f(x)\{C\}$, has arrow type $\tau \xrightarrow{H} \mathbf{1}$ where τ is the type of its input and H is the latent effect obtained by typing C (see rules for statements). Also, typing C requires to recursively keep trace of the type of x and of the procedure f. Typing intents (rule ($\mathtt{T_E{-}INT}$)) is quite intuitive: an intent $I_\alpha(E, E')$ has type $\mathcal{I}_\alpha(\mathcal{U}, \tau)$ where $\mathcal{U}$ and τ are the types of the sub-expressions E and E'. Conversely, the type of the data and extra fields (rules ($\mathtt{T_E{-}DATA}$) and ($\mathtt{T_E{-}EXT}$)) of an intent of type $\mathcal{I}_\alpha(\mathcal{U}, \tau)$ have type $\mathcal{U}$ and τ, respectively.

Typing rules for statements are also straightforward. A `skip` command (rule ($\mathtt{T_S{-}SKIP}$)) generates the void effect ε, while casting an intent (both implicitly or explicitly, rule ($\mathtt{T_S{-}ICST}$) and ($\mathtt{T_S{-}ECST}$)) inside a component η, can generate an action $\alpha_\eta(u, \tau)$ for each possible instance of u compatible with the intent type (we use $\sum H_i$ as a shorthand for the finite summation $H_1 + H_2 + \ldots$). Similarly, system calls, permission granting, revocation and checks produce corresponding, observable actions (rules ($\mathtt{T_S{-}SYS}$), ($\mathtt{T_S{-}GRNT}$), ($\mathtt{T_S{-}REVK}$) and ($\mathtt{T_S{-}CHK}$), respectively). In particular, a command $\mathtt{system}_\sigma(E)$ is typed to the sum of all the possible accesses σ to the resources denoted by E. Instead, permission granting (revocation) evaluates to the special action $\upharpoonleft^\eta_{\sigma,u}$ ($\downharpoonleft^\eta_{\sigma,u}$). Then, permission checks produce the special actions $?^\eta_{\sigma,u}$. Applying a procedure to a parameter (rule ($\mathtt{T_S{-}APP}$)) results in its latent effect to be carried out. The sequence of statements ($\mathtt{T_S{-}SEQ}$) is typed to the sequence of their effects, the resource creation command ($\mathtt{T_S{-}NEW}$) results in the history expression $\nu u.H$, a receiver (rule ($\mathtt{T_S{-}RECV}$)) has effect $\bar{\alpha}_\eta h.H$ and a conditional branching (rule ($\mathtt{T_S{-}CND}$)) has effect equal to those of its two branches. Finally, we include a rule, called *weakening* ($\mathtt{T_S{-}WKN}$), for extending the effect of statements (where $H' \sqsubseteq H$ iff $[\![H']\!] \subseteq [\![H]\!]$).

Example 3. Consider the following two statements:

$$\begin{aligned} C &= \mathtt{apply}\,(\,\mathtt{proc}\,f(y)\{\,\mathtt{receive}\,x \mapsto \mathtt{system}_\sigma\,(x.\mathtt{d});\ \mathtt{apply}\,f\,\mathtt{to}\,y\})\,\mathtt{to}\,\mathtt{null} \\ C' &= \mathtt{icast}\,I_\alpha(u, \mathtt{null}) \end{aligned}$$

We type them as follows:

$$\begin{aligned} &\emptyset \rhd^\eta_\emptyset \mathtt{apply}\,(\,\mathtt{proc}\,f(y)\{\,\mathtt{receive}\,x \mapsto \mathtt{system}_\sigma\,(x.\mathtt{d});\ \mathtt{apply}\,f\,\mathtt{to}\,y\})\,\mathtt{to}\,\mathtt{null} : \mu h.\bar{\alpha}_\eta h'.\sigma_\eta(u) \cdot h \\ &\emptyset \rhd^{\eta'}_\emptyset \ \mathtt{icast}\,I_\alpha(u, \mathtt{null}) : \alpha_{\eta'}(u, \mathbf{1}) \end{aligned}$$

The complete derivations are reported at `http://www.ai-lab.it/merlo/publications/AndroidModel.pdf`.

A fundamental property of our type system is that it generates history expressions which *correctly* represent the behaviour of the statements they are extracted from. This is granted by the following lemma.

Lemma 1. *For each C such that $\emptyset \rhd^\eta_O C : H$ and for each Φ, Λ and U such that $U(\eta) = O$, for all arbitrary long sequences of actions performed by U, Φ, C there exists a trace in $[\![H]\!]$ denoting it.*

As far as the overall behaviour of a system depends on several components and their permissions and privileges, typing each single component is not sufficient to create a model of an entire platform. Hence, we define a compositional operator, based on our typing rules, which, given a system generates a corresponding model.

Definition 6. *Given a system $S = A_1, \ldots, A_n$ such that $A_i = \langle M_i, \Delta_i, C^i_1 \ldots C^i_{k_i} \rangle$ we define*

$$\textsc{he}(S) = (H^1_1 \backslash_{L_{A_1}}) \parallel \ldots \parallel (H^1_{k_1} \backslash_{L_{A_1}}) \parallel \ldots \parallel (H^n_1 \backslash_{L_{A_n}}) \parallel \ldots \parallel (H^n_{k_n} \backslash_{L_{A_n}}) \qquad \text{where}$$

- $\emptyset \rhd^{\eta}_{\emptyset} C^i_j : H^i_j$ *(with $\Delta_i(\eta) = C^i_j$);*
- $L_{A_i} = \{\alpha_\eta(u, \tau) \mid \alpha_\eta(u, \tau) \in \mathsf{Priv}_S(A_i)\}$.

The operator $\textsc{he}(S)$ generates a history expression which correctly models S as stated by the following theorem.

Theorem 1. *For each $S = A_1, \ldots, A_n$ such that $A_i = \langle M_i, \Delta_i, C^i_1 \ldots C^i_{k_i} \rangle$ for any arbitrary long computation performed by $\mathbf{U}_S, \mathbf{\Phi}_S, [C^1_1]_{\eta^1_1} \cdots [C^n_{k_n}]_{\eta^n_{k_n}}$ there exists a trace in $[\![\textsc{he}(S)]\!]$ denoting it.*

Such property guarantees that any possible behaviour that a platform has at runtime is contained in its model which we can analyse statically.

5.3 Future Directions

We showed that type and effect systems can be used to compute an over-approximation of the behaviours of programs called the history expressions. History expressions can be exploited for different kinds of analysis, e.g., validation against security policies [5] or deployment of extra security checks [22]. We plan to investigate the existing techniques which rely on history expressions for verifying whether they apply, as we believe, to our model.

Another possibility is to exploit type systems as proof systems. Let H be a history expression and C be a statement. Typing $\emptyset \rhd^{O}_{\eta} C : H$ corresponds to proving that the behaviour of C is bounded by H. This means that, if we specify security policies through history expressions, then we can verify a program by typing it to that particular history expression. Also, we can obtain similar results by typing a statement and checking whether the obtained history expression is a subtype (relation $\sqsubseteq$) of the policy ones. A convenient way to do that can be via simulation-based techniques, which can be applied here since $\sqsubseteq$ is indeed a simulation relation (see `http://www.ai-lab.it/merlo/publications/AndroidModel.pdf`).

6 Conclusion and Related Work

In this work we presented an approach which aims at modeling the Android Application Framework. Furthermore, such model is automatically inferred by means of a type and effect system. The type and effect system can either generate or verify history expressions from the Android applications (components and manifests). The resulting model is safe in the sense that it correctly represents all the possible runtime computations of the applications. Moreover, the history expressions representing (the components of) 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. [5], have been successfully applied to the security analysis of Java applications [3] and web services [4], and we plan to apply similar approaches to Android.

Related work. Only recently researchers focussed on the formal modeling and analysis of the Android platform and its security aspects. In [21] the authors formalise the permission scheme of Android. Briefly, their formalisation 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 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 [10] 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].

Most of the literature on Android security contains proposals for i) extending the native security policy, ii) enhancing the ASF with new tools for specific security-related checks, and iii) detecting vulnerabilities and security threats. Regarding the first category, in [20] Android security policy is analysed 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 [24] 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, [13] 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 [8] and Crowdroid [9]) and application certification (e.g. Scandroid [14] and Comdroid [11]).

Some works have 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 [2], covert channels [19], web attacks [16] and privilege escalation (see [8]).

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 common and comprehensive reference model for the security of the Android platform.

References

1. Abadi, M., Fournet, C.: Access control based on execution history. In: Proceedings of the 10th Annual Network and Distributed System Security Symposium, pp. 107–121 (2003)
2. Armando, A., Merlo, A., Migliardi, M., Verderame, L.: Would you mind forking this process? A denial of service attack on Android (and some countermeasures). In: Gritzalis, D., Furnell, S., Theoharidou, M. (eds.) SEC 2012. IFIP AICT, vol. 376, pp. 13–24. Springer, Heidelberg (2012)
3. Bartoletti, M., Costa, G., Degano, P., Martinelli, F., Zunino, R.: Securing Java with Local Policies. Journal of Object Technology 8(4), 5–32 (2009)
4. Bartoletti, M., Degano, P., Ferrari, G.L.: Planning and verifying service composition. Journal of Computer Security (JCS) 17(5), 799–837 (2009)
5. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Types and effects for resource usage analysis. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 32–47. Springer, Heidelberg (2007)
6. Bartoletti, M., Degano, P., Ferrari, G.-L., Zunino, R.: Local policies for resource usage analysis. ACM Transactions on Programming Languages and Systems 31(6), 1–43 (2009)
7. Bierman, G.M., Parkinson, M.J., Pitts, A.M.: MJ: An imperative core calculus for Java and Java with effects. Technical report, University of Cambridge (2003)
8. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R.: Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Univ. Darmstadt (April 2011)
9. Burguera, I., Zurutuza, U., Nadjm-Therani, S.: Crowdroid: behavior-based malware detection system for android. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM 2011 (2011)
10. Chaudhuri, A.: Language-based security on Android. In: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, PLAS 2009, pp. 1–7. ACM, New York (2009)
11. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, MobiSys 2011, pp. 239–252. ACM, New York (2011)
12. Android Developers. Security and permissions, `http://developer.android.com/guide/topics/security/security.html`
13. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, pp. 627–638 (2011)
14. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: Automated security certification of android applications
15. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A Minimal Core Calculus for Java and GJ. ACM Transactions on Programming Languages and Systems, 132–146 (1999)

16. Luo, T., Hao, H., Du, W., Wang, Y., Yin, H.: Attacks on webview in the android system. In: Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC 2011, pp. 343–352. ACM, New York (2011)
17. Nauman, M., Khan, S., Zhang, X.: Apex: extending android permission model and enforcement with user-defined runtime constraints. In: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2010, pp. 328–332. ACM, New York (2010)
18. Ongtang, M., Mclaughlin, S., Enck, W., Mcdaniel, P.: Semantically rich application-centric security in android. In: ACSAC 2009: Annual Computer Security Applications Conference (2009)
19. Schlegel, R., Zhang, K., Zhou, X., Intwala, M., Kapadia, A., Wang, X.: Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In: Proceedings of the 18th Annual Network & Distributed System Security Symposium (2011)
20. Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S., Glezer, C.: Google android: A comprehensive security assessment. IEEE Security Privacy 8(2), 35–44 (2010)
21. Shin, W., Kiyomoto, S., Fukushima, K., Tanaka, T.: A Formal Model to Analyze the Permission Authorization and Enforcement in the Android Framework. In: Proceedings of the 2010 IEEE Second International Conference on Social Computing, SOCIALCOM 2010, pp. 944–951. IEEE Computer Society, Washington, DC (2010)
22. Skalka, C., Smith, S.: History effects and verification. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 107–128. Springer, Heidelberg (2004)
23. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of the Second ACM Conference on Data and Application Security and Privacy, CODASPY 2012, pp. 317–326. ACM, New York (2012)
24. Zhou, Y., Zhang, X., Jiang, X., Freeh, V.W.: Taming information-stealing smartphone applications (on android). In: McCune, J.M., Balacheff, B., Perrig, A., Sadeghi, A.-R., Sasse, A., Beres, Y. (eds.) TRUST 2011. LNCS, vol. 6740, pp. 93–107. Springer, Heidelberg (2011)