A Formal Model of Access Control for Mobile Interactive Devices

Frédéric Besson, Guillaume Dufay, and Thomas Jensen *

IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France

Abstract. This paper presents an access control model for programming applications in which the access control to resources can employ user interaction to obtain the necessary permissions. This model is inspired by and improves on the Java MIDP security architecture used in Java-enabled mobile telephones. We consider access control permissions with multiplicities in order to allow to use a permission a certain number of times. An operational semantics of the model and a formal definition of what it means for an application to respect the security model is given. A static analysis which enforces the security model is defined and proved correct. A constraint solving algorithm implementing the analysis is presented.

1 Introduction

Access control to resources is classically described by a model in which an access control matrix specifies the actions that a subject (program, user, applet, ...) is allowed to perform on a particular object. Recent access control mechanisms have added a dynamic aspect to this model: applets can be granted permissions temporarily and the outcome of an access control depends on both the set of currently held permissions and the state of the machine. The most studied example of this phenomenon is the stack inspection of Java (and the stack walks of C$^\sharp$) together with the *privileged method calls* by which an applet grants all its permissions to its callers for the duration of the execution of a particular method call, see *e.g.* [1, 4, 6, 9]. Another example is the security architecture for embedded Java on mobile telephones [14] which uses interactive querying of the user to grant permissions on-the-fly to the applet executing on a mobile phone so that it can make internet connections, access files, send SMSs *etc.* An important feature of the MIDP model is the "one-shot" permissions that can be used once for accessing a resource. This quantitative aspect of permissions raises several questions of how such permissions should be modeled (*e.g.*, "do they accumulate?" or "which one to choose if several permissions apply?") and how to program with such permissions in a way that respects both usability and security principles such as Least Privilege [13] and the security property stated below. We review the MIDP model in Section 2.

* This work was partly funded by the IST-FET programme of the European Commission, under the IST-2005-015905 MOBIUS project.

In this paper, we present a formal model for studying such programming mechanisms with the purpose of developing a semantically well-founded and more general replacement for the Java MIDP model. We propose a semantics of the model's programming constructs and a logic for reasoning about the flow of permissions in programs using these constructs. This logic will notably allow to prove the basic security property that

> a program will never attempt to access a resource for which it does not have permission.

Notice that this is stronger than just ensuring that the program will never actually access the resource. Indeed, the latter property can be trivially achieved in systems with run-time checks—at the expense of accepting a security exception when an illegal access is detected. The basic security property is pertinent to systems with or without such dynamic controls. For systems without any run-time checks, it guarantees the absence of illegal accesses. For dynamically monitored systems, it guarantees that access control exceptions will never be raised.

The notion of permission is central to our model. Permissions have an internal structure (formalised in Section 3) that describes the actions that it enables and the set of objects to which it applies. The "one-shot" permissions alluded to above have motived a generalisation in which permissions also have multiplicities, stating how many times the given permission can be used. The security model we propose have two basic constructs for manipulating permissions:

- `grant` models the interactive querying of the user, asking whether he grants a particular permission with a certain multiplicity to the applet, and
- `consume` models the access to a method which requires (and hence consumes) permissions.

The choice of where to insert requests for user-granted permissions is important for the usability of an applet and has a clear impact on its security. We provide a static analysis that will verify automatically that a given choice will not violate the basic security property stated above.

The analysis is developed by integrating the `grant` and `consume` constructs into a program model based on control-flow graphs. The model and its operational semantics is presented in Section 4. In this section, we also formally define what it means for an execution trace (and hence for a program) to respect the basic security property. Section 5 defines a constraint-based static analysis for safely approximating the flow of permissions in a program with the aim of computing what permissions are available at each program point. Section 6 describes how to solve the constraints produced by the analysis. Section 7 describes related formal models and verification techniques for language-based access control and Section 8 concludes. Appendix A contains the formal correctness proof of the analysis with respect to the semantics.

2 The Java MIDP security model

The Java MIDP programming model for mobile telephones [14] proposes a thoroughly developed security architecture which is the starting point of our work. In the MIDP security model, several applications (called *midlets* in the MIDP jargon) are loaded into and co-execute in a virtual machine. At load time, the midlet is assigned a protection domain which determines how the midlet can access resources. It can be seen as a labelling function which classifies a resource access as either `allowed` or `user`.

- `allowed` means that the midlet is granted unrestricted access to the resource;
- `user` means prior to an access, an interaction with the user is initiated in order to ask for permission to perform the access and to determine how often this permission can be exercised. The MIDP model operates with three possibilities:
  - `blanket`: the permission is granted for as long as the midlet remains installed;
  - `session`: the permission is granted for as long as the MIDlet is running;
  - `oneshot`: the permission is granted for a single use.

The `oneshot` permissions correspond to dynamic security checks in which each access is protected by a user interaction. This clearly provides a secure access to resources but the potentially numerous user interactions are at the detriment of the usability and makes social engineering attacks easier. At the other end of the spectrum, the `allowed` mode which gets granted through signing provides a maximum of usability but leaves the user with absolutely no assurance on how resources are used as a signature is only a certificate of integrity and origin.

In the following we will propose a security model which extends the MIDP model by introducing permissions with multiplicities and by adding flexibility to the way in which permissions are granted by the user and used by applications. In this model, we can express:

- the `allowed` mode and `blanket` permissions as initial permissions with multiplicity ∞;
- the `session` permissions by prompting the user for an infinite number of a given permission once at application start-up;
- the `oneshot` permissions by prompting the user for a permission with a `grant` just before consuming it with a `consume`.

The added flexibility is obtained by allowing the programmer to insert user interactions for obtaining permissions at any point in the program (rather than only at the beginning and just before an access) and to ask for a batch of permissions in one interaction. The added flexibility can be used to improve the usability of access control in a midlet but will require formal methods to ensure that the midlet correctly asks for the necessary permissions.

3 The structure of permissions

In classical access control models, permissions held by a subject (user, program, ...) authorise certain *actions* to be performed on certain *resources*. Such permissions can be represented as a relation between actions and resources. To obtain a better fit with access control architectures such as that of Java MIDP we enrich this permission model with multiplicities and resource types. Concrete MIDP permissions are strings whose prefixes encode package names and whose suffixes encode a specific permission. For instance, one finds permissions `javax.microedition.io.Connector.http` and `javax.microedition.io.Connector.sms.send` which enable applets to make connections using the http protocol or to send a SMS, respectively. Thus, permissions are structured entities that for a given resource type define which actions can be applied to which resources of that type and how many times.

To model this formally, we assume given a set $ResType$ of resource types. For each resource type rt there is a set of resources Res_{rt} of that type and a set of actions Act_{rt} applicable to resources of that type. We incorporate the notion of multiplicities by attaching to a set of actions a and a set of resources r a multiplicity m indicating how many times actions a can be performed on resources from r. Multiplicities are taken from the ordered set:

$$Mul \triangleq (\mathbb{N} \cup \{\perp_{Mul}, \infty\}, \leq).$$

The 0 multiplicity represents absence of a given permission and the ∞ multiplicity means that the permission is permanently granted. The $\perp_{Mul}$ multiplicity represents an error arising from trying to decrement the 0 multiplicity. We define the operation of decrementing a multiplicity as follows:

$$m - 1 = \begin{cases} \infty & \text{if } m = \infty \\ m - 1 & \text{if } m \in \mathbb{N}, m \neq 0 \\ \perp_{Mul} & \text{if } m = 0 \text{ or } m = \perp_{Mul} \end{cases}$$

Several implementations of permissions include an *implication ordering* on permissions. One permission implies another if the former allows to apply a particular action to more resources than the latter. However, the underlying object-oriented nature of permissions imposes that only permissions of the same resource type can be compared. We capture this in our model by organising permissions as a dependent product of permission sets for a given resource type.

Definition 1 (Permissions) *Given a set $ResType$ of resource types and $ResType$-indexed families of resources Res_{rt} and actions Act_{rt}, the set of atomic permissions $Perm_{rt}$ is defined as:*

$$Perm_{rt} \triangleq (\mathcal{P}(Res_{rt}) \times \mathcal{P}(Act_{rt})) \cup \{\perp\}$$

relating a type of resources with the actions that can be performed on it. The element $\perp$ represents an invalid permission. By extension, we define the set of *permissions Perm as the dependent product:*

$$Perm \triangleq \prod_{rt \in ResType} Perm_{rt} \times Mul$$

relating for all resource types an atomic permission and a multiplicity stating how many times it can be used.
For $\rho \in Perm$ and $rt \in ResType$, we use the notations $\rho(rt)$ to denote the pair of atomic permissions and multiplicities associated with rt in ρ. Similarly, $\mapsto$ is used to update the permission associated to a ressource type, i.e., $(\rho[rt \mapsto (p,m)])(rt) = (p,m)$.

Example 1 *Given a ressource type $SMS \in ResType$, the permission $\rho \in Perm$ satisfying $\rho(SMS) = ((+1800*, \{send\}), 2)$ gives the permission of twice sending an SMS to a +1800 number.*

Definition 2 *The ordering $\sqsubseteq_p \subseteq Perm \times Perm$ on permissions is given by*

$$\rho_1 \sqsubseteq_p \rho_2 \triangleq \forall rt \in ResType \quad \rho_1(rt) \sqsubseteq \rho_2(rt)$$

where $\sqsubseteq$ is the product of the subset ordering $\sqsubseteq_{rt}$ on $Perm_{rt}$ and the $\leq$ ordering on multiplicities.

Intuitively, being higher up in the ordering means having more permissions to access a larger set of resources. The ordering induces a greatest lower bound operator $\sqcap : Perm \times Perm \rightarrow Perm$ on permissions. For example, for $\rho \in Perm$

$$\rho[File \mapsto ((/tmp/*, \{read, write\}), 1)] \ \sqcap \ \rho[File \mapsto ((*/dupont/*, \{read\}), \infty)] = \rho[File \mapsto ((/tmp/*/dupont/*, \{read\}), 1)]$$

Operations on permissions

There are two operations on permissions that will be of essential use in the following:

- consumption (removal) of a specific permission from a collection of permissions;
- update of a collection of permissions with a newly granted permission.

Definition 3 *Let $\rho \in Perm$, $rt \in ResType$ and assume that $\rho(rt) = (p,m)$. The operation $consume : Perm_{rt} \rightarrow Perm \rightarrow Perm$ is defined by*

$$consume(p')(\rho) = \begin{cases} \rho[rt \mapsto (p, m-1)] & \text{if } p' \sqsubseteq_{rt} p \\ \rho[rt \mapsto (\bot, m-1)] & \text{otherwise} \end{cases}$$

There are two possible error situations when trying to consume a permission. Consuming a resource for which the number of multiplicities is zero will result in setting the multiplicity to $\bot_{Mul}$. Similarly, attempting to consume a resource for which there is no permission ($p' \not\sqsubseteq_{rt} p$) is an error.

Definition 4 *A permission $\rho \in Perm$ is an error, written $Error(\rho)$, if:*

$$\exists rt \in ResType, \exists (p, m) \in Perm_{rt} \times Mul, \rho(rt) = (p, m) \wedge (p = \perp \vee m = \perp_{Mul}).$$

Granting a number of accesses to a resource of a particular resource type is modeled by updating the component corresponding to that resource type.

Definition 5 *Let $\rho \in Perm$, $rt \in ResType$, the operation grant : $Perm_{rt} \times (\mathbb{N} \cup \{\infty\}) \to Perm \to Perm$ for granting a number of permissions to access a resource of a given type is defined by*

$$grant(p, m)(\rho) = \rho[rt \mapsto (p, m)]$$

Notice that granting such a permission erases all previously held permissions for that resource type, *i.e.*, permissions do not accumulate. This is a design choice: the model forbid that permissions be granted for performing one task and then used later on to accomplish another. Other models might want to allow the accumulation of permissions, possibly coupled with some time bound. The *grant* operation should then add the granted number of permissions to the existing amount rather than replace it.

4 Program model

We model a program by a control-flow graph (CFG) that captures the manipulations of permissions, the handling of method calls and returns, as well as exceptions. This model has been used in previous work on modelling access control for Java—see [1, 4, 9].

Definition 6 *A control-flow graph is a 7-uple*

$$G = (NO, EX, KD, TG, CG, EG, n_0)$$

where:

- *NO is the set of nodes of the graph;*
- *EX is the set of exceptions;*
- *$KD : NO \to \{\texttt{grant}(p, m), \texttt{consume}(p), \texttt{call}, \texttt{return}, \texttt{throw}(ex)\}$, with $ex \in EX, rt \in ResType, p \in Perm_{rt}$ and $m \in Mul$ associates a kind to each node, indicating which instruction the node represents;*
- *$TG \subseteq NO \times NO$ is the set of intra-procedural edges;*
- *$CG \subseteq NO \times NO$ is the set of inter-procedural edges;*
- *$EG \subseteq EX \times NO \times NO$ is the set of intra-procedural exception edges that will be followed if an exception is raised at that node;*
- *n_0 is the entry point of the graph.*

In the following, given $n, n' \in NO$ and $ex \in EX$, we will use the notations $n \stackrel{TG}{\to} n'$ for $(n, n') \in TG$, $n \stackrel{CG}{\to} n'$ for $(n, n') \in CG$ and $n \stackrel{ex}{\to} n'$ for $(ex, n, n') \in EG$.

Example 2 *Figure 1 contains the control-flow graph of* `grant` *and* `consume` *operations during a flight-booking transaction (for simplicity, actions related to permissions, such as* $\{connect\}$ *or* $\{read\}$*, are omitted). In this transaction, the user first transmits his request to a travel agency, site. He can then modify his request or get additional information. Finally he can either book or pay the desired flight. Corresponding permissions are summarised in the initial permission* p_{init}*, but they could also be granted using the* `grant` *operation. In the example,*

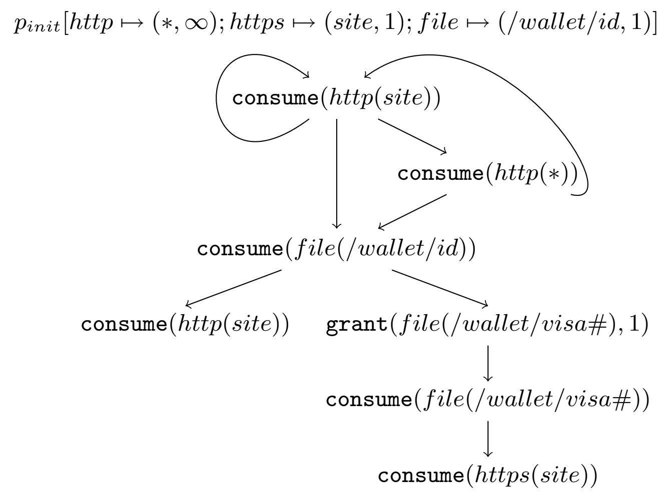

Fig. 1. Example of grant/consume permissions patterns

the developer has chosen to delay asking for the permission of accessing credit card information until it is certain that this permission is indeed needed. Another design choice would be to grant this permission from the outset. This would minimise user interaction because it allows to remove the querying `grant` *operation. However, the initial permission* p_{init} *would then contain* $file \mapsto (/wallet/*, 2)$ *instead of* $file \mapsto (/wallet/id, 1)$ *which violates the Principle of Least Privilege.*

Operational semantics

Given a control-flow graph, we define the small-step operational semantics of CFGs in Figure 2. The semantics is stack-based and follows the behaviour of a standard programming language with exceptions such as Java or C$^\sharp$.

The operational semantics operates on a state consisting of a standard control-flow stack of nodes, enriched with the permissions held at that point in the execution. Thus, the small-step semantics is given by a relation $\twoheadrightarrow$ between elements of $(NO^* \times (EX \cup \{\epsilon\}) \times Perm)$, where NO^* is a sequence of nodes. For example, for the instruction `call` of Figure 2, if the current node n leads through an inter-procedural step to a node m, then the node m is added to the top of the stack $n{:}s$, with $s \in NO^*$. Instantiating this model to such languages consists of identifying in the code the desired `grant` and `consume` operations and describing the action of the other instructions on the stack.

Instructions may change the value of the permission along with the current state. *E.g.*, for the instruction `grant` of Figure 2, the current permission ρ of the state will be updated with the new granted permissions. The current node of the stack n will also be updated, at least to change the program counter, depending on the desired implementation of `grant`. Note that the instrumentation is *non-intrusive*, *i.e.* a transition will not be blocked due to the absence of a permission. Thus, for s, s' in NO^*, e, e' in $(EX \cup \{\epsilon\})$, if there exists ρ and ρ' in $Perm$ such that $n, e, \rho \twoheadrightarrow n', e', \rho'$ then for all ρ and ρ', the same transition holds.

$$\frac{KD(n) = \texttt{grant}(p,m) \quad n \stackrel{TG}{\to} n'}{n{:}s, \epsilon, \rho \twoheadrightarrow n'{:}s, \epsilon, grant(p,m)(\rho)} \quad \frac{KD(n) = \texttt{consume}(p) \quad n \stackrel{TG}{\to} n'}{n{:}s, \epsilon, \rho \twoheadrightarrow n'{:}s, \epsilon, consume(p)(\rho)}$$

$$\frac{KD(n) = \texttt{call} \quad n \stackrel{CG}{\to} m}{n{:}s, \epsilon, \rho \twoheadrightarrow m{:}n{:}s, \epsilon, \rho} \qquad \frac{KD(r) = \texttt{return} \quad n \stackrel{TG}{\to} n'}{r{:}n{:}s, \epsilon, \rho \twoheadrightarrow n'{:}s, \epsilon, \rho}$$

$$\frac{KD(n) = \texttt{throw}(ex) \quad n \stackrel{ex}{\to} h}{n{:}s, \epsilon, \rho \twoheadrightarrow h{:}s, \epsilon, \rho} \quad \frac{KD(n) = \texttt{throw}(ex) \quad \forall h, n \stackrel{ex}{\not\to} h}{n{:}s, \epsilon, \rho \twoheadrightarrow n{:}s, ex, \rho}$$

$$\frac{\forall h, n \stackrel{ex}{\not\to} h}{t{:}n{:}s, ex, \rho \twoheadrightarrow n{:}s, ex, \rho} \qquad \frac{n \stackrel{ex}{\to} h}{t{:}n{:}s, ex, \rho \twoheadrightarrow h{:}s, \epsilon, \rho}$$

Fig. 2. Small-step operational semantics

This operational semantics will be the basis for the notion of program execution traces, on which global results on the execution of a program will be expressed.

Definition 7 (Trace of a CFG) *A partial trace* $tr \in (NO, (EX \cup \{\epsilon\}))^*$ *of a CFG is a sequence of nodes* $(n_0, \epsilon) :: (n_1, e_1) :: \ldots :: (n_k, e_k)$ *such that for all* $0 \le i < k$ *there exists* $\rho, \rho' \in Perm$, $s, s' \in NO^*$ *and verifying* $n_i{:}s, e_i, \rho \twoheadrightarrow n_{i+1}{:}s', e_{i+1}, \rho'$.

For a program P *represented by its control-flow graph* G*, we will denote by* $[\![P]\!]$ *the set of all partial traces of* G.

To state and verify the safety of a program that acquires and consumes permissions, we first define what it means for an execution trace to be safe. We define the permission set available at the end of a trace by induction over its length.

$$
\begin{array}{ll}
PermsOf(nil) & \triangleq p_{init} \\
PermsOf(tr :: (\texttt{consume}(p), e)) & \triangleq consume(p, PermsOf(tr)) \\
PermsOf(tr :: (\texttt{grant}(p, m), e)) & \triangleq grant((p, m), PermsOf(tr)) \\
PermsOf(tr :: (n, e)) & \triangleq PermsOf(tr) \qquad \text{otherwise}
\end{array}
$$

p_{init} is the initial permission of the program, for the state n_0. By default, if no permission is granted at the beginning of the execution, it will contain $((\emptyset, \emptyset), 0)$ for each resource type. The `allowed` mode and `blanket` permissions for a resource r of a given resource type can be modeled by associating the permission (r, ∞) with that resource type.

A trace is *safe* if none of its prefixes end in an error situation due to the access of resources for which the necessary permissions have not been obtained.

Definition 8 (Safe trace) *A partial trace $tr \in (NO, (EX \cup \{\epsilon\}))^*$ is* safe*, written $Safe(tr)$, if for all prefixes $tr' \in prefix(tr)$, $\neg Error(PermsOf(tr'))$.*

5 Static analysis of permission usage

We now define a constraint-based static flow analysis for computing a safe approximation, denoted P_n, of the permissions that are guaranteed to be available at each program point n in a CFG when execution reaches that point. Thus, safe means that P_n underestimates the set of permissions that will be held at n during the execution. The approximation will be defined as a solution to a system of constraints over P_n, derived from the CFG following the rules in Figure 3. The rules for P_n are straightforward data flow rules: *e.g.*, for `grant` and `consume` we use the corresponding semantic operations *grant* and *consume* applied to the start state P_n to get an upper bound on the permissions that can be held at end state $P_{n'}$. Notice that the set $P_{n'}$ can be further constrained if there is another flow into n'. The effect of a method call on the set of permissions will be modeled by a transfer function R that will be defined below. Finally, throwing an exception at node n that will be caught at node m means that the set of permissions at n will be transferred to m and hence form an upper bound on the set of available permissions at this point.

Our CFG program model includes procedure calls which means that the analysis must be inter-procedural. We deal with procedures by computing *summary functions* for each procedure. These functions summarise how a given procedure consumes resources from the entry of the procedure to the exit, which can happen either normally by reaching a `return` node, or by raising an exception which is not handled in the procedure. More precisely, for a given CFG we compute the quantity $R : (EX \cup \{\epsilon\}) \to NO \to (Perm \to Perm)$ with the following meaning:

- the partial application of R to ϵ is the effect on a given initial permission of the execution from a node until return;

$$\frac{}{P_{n_0} \sqsubseteq_p p_{init}} \qquad \frac{KD(n) = \mathtt{grant}(p,m) \quad n \stackrel{TG}{\rightarrow} n'}{P_{n'} \sqsubseteq_p grant(p,m)(P_n)}$$

$$\frac{KD(n) = \mathtt{consume}(p) \quad n \stackrel{TG}{\rightarrow} n'}{P_{n'} \sqsubseteq_p consume(p)(P_n)} \qquad \frac{KD(n) = \mathtt{call} \quad n \stackrel{CG}{\rightarrow} m \quad n \stackrel{TG}{\rightarrow} n'}{P_{n'} \sqsubseteq_p R_m(P_n)}$$

$$\frac{KD(n) = \mathtt{call} \quad n \stackrel{CG}{\rightarrow} m}{P_m \sqsubseteq_p P_n} \qquad \frac{KD(n) = \mathtt{call} \quad n \stackrel{CG}{\rightarrow} m \quad n \stackrel{ex}{\rightarrow} h}{P_h \sqsubseteq_p R_m^{ex}(P_n)}$$

$$\frac{KD(n) = \mathtt{call} \quad n \stackrel{CG}{\rightarrow} m \quad \forall h, n \stackrel{ex}{\nrightarrow} h}{P_n \sqsubseteq_p R_m^{ex}(P_n)} \qquad \frac{KD(n) = \mathtt{throw}(ex) \quad n \stackrel{ex}{\rightarrow} m}{P_m \sqsubseteq_p P_n}$$

Fig. 3. Constraints on minimal permissions

– the partial application of R to $ex \in EX$ is the effect on a given initial permission of the execution from a node until reaching a node which throws an exception ex that is not caught in the same method.

Given nodes $n, n' \in NO$, we will use the notation R_n and R_n^{ex} for the partial applications of $R\ \epsilon\ n$ and $R\ ex\ n$. The rules are written using diagrammatic function composition ; such that $F; F'(\rho) = F'(F(\rho))$. We define an order $\sqsubseteq$ on functions $F, F' : Perm \to Perm$ by extensionality such that $F \sqsubseteq F'$ if $\forall \rho \in Perm, F(\rho) \sqsubseteq_p F'(\rho)$.

As for the entities P_n, the function R is defined as solutions to a system of constraints. The rules for generating these constraints are given in Figure 4 (with $e \in EX \cup \{\epsilon\}$). The rules all have the same structure: compose the effect of the current node n on the permission set with the function describing the effect of the computation starting at n's successors in the control flow. This provides an upper bound on the effect on permissions when starting from n. As with the constraints for P, we use the functions *grant* and *consume* to model the effect of `grant` and `consume` nodes, respectively. A method call at node n is modeled by the R function itself applied to the start node of the called method m. The combined effect is the composition $R_m; R_{n'}^e$ of the effect of the method call followed by the effect of the computation starting at the successor node n' of call node n.

The correctness of our analysis is stated on execution traces. For a given program, if a solution of the constraints computed during the analysis does not contain errors in permissions, then the program will behave safely. Formally,

Theorem 1 (Basic Security Property) *Given a program P:*

$$(\forall n \in NO, \neg Error(P_n)) \Rightarrow \forall tr \in [\![P]\!], Safe(tr)$$

The proof of this theorem is given in Appendix A.

$$\frac{KD(n) = \texttt{grant}(p, m) \quad n \stackrel{TG}{\to} n'}{R^e_n \sqsubseteq grant(p, m); R^e_{n'}} \qquad \frac{KD(n) = \texttt{consume}(p) \quad n \stackrel{TG}{\to} n'}{R^e_n \sqsubseteq consume(p); R^e_{n'}}$$

$$\frac{KD(n) = \texttt{return}}{R_n \sqsubseteq \lambda\rho.\rho} \qquad \frac{KD(n) = \texttt{call} \quad n \stackrel{CG}{\to} m \quad n \stackrel{TG}{\to} n'}{R^e_n \sqsubseteq R_m; R^e_{n'}}$$

$$\frac{KD(n) = \texttt{call} \quad n \stackrel{CG}{\to} m \quad \forall n', n \stackrel{ex}{\not\to} n'}{R^{ex}_n \sqsubseteq R^{ex}_m} \qquad \frac{KD(n) = \texttt{call} \quad n \stackrel{CG}{\to} m \quad n \stackrel{ex}{\to} h}{R_n \sqsubseteq R^{ex}_m; R_h}$$

$$\frac{KD(n) = \texttt{throw}(ex) \quad n \stackrel{ex}{\to} h}{R^e_n \sqsubseteq R^e_h} \qquad \frac{KD(n) = \texttt{throw}(ex) \quad \forall n', n \stackrel{ex}{\not\to} n'}{R^{ex}_n \sqsubseteq \lambda\rho.\rho}$$

Fig. 4. Summary functions of the effect of the execution on initial permission

6 Constraint solving

Computing a solution to the constraints generated by the analysis in Section 5 is complicated by the fact that solutions to the R-constraints (see Figure 4) are functions from $Perm$ to $Perm$ that have infinite domains and hence cannot be represented by a naive tabulation [15]. To solve this problem, we identify a class of functions that are sufficient to encode solutions to the constraints while restricted enough to allow effective computations. Given a solution to the R-constraints, the P-constraints (see Figure 3) are solved by standard fix-point iteration starting from the most permissive permission MPP defined by $MPP(rt) = ((Res_{rt}, Act_{rt}), \infty)$ for any $rt \in ResType$.

The rest of this section is devoted to the resolution of the R-constraints. The resolution technique consists in applying solution-preserving transformations to our initial set of constraints. The theoretical background is provided by Propositions 1 and 2. They consider a variable X of type D subject to a finite set of constraints $C = \bigcup_{i\in I}\{X \sqsubseteq f_i(X)\}$ and show how to build abstract systems $C^\sharp$ such that a solution $\hat{X}^\sharp$ to $C^\sharp$ yields a solution $\hat{X}$ to C. Proposition 1 shows that constraints can be solved over an abstract domain. Proposition 2 shows how to split product types into a product of variables.

Proposition 1 *Let the domains $(D, \sqsubseteq)$ and $(D^\sharp, \sqsubseteq^\sharp)$ be linked by a monotone function $\gamma : D^\sharp \to D$ and let $C^\sharp = \bigcup_{i\in I}\{X^\sharp \sqsubseteq f^\sharp_i(X^\sharp)\}$ be a constraint system satisfying that $\forall i \in I, x^\sharp \in D^\sharp, \gamma \circ f^\sharp_i(x^\sharp) \sqsubseteq f(\gamma(x^\sharp))$. If $\hat{X}^\sharp$ is a solution of $C^\sharp$ then $\gamma(\hat{X}^\sharp)$ is a solution of C.*

Proposition 2 *Let the product domain $D = \Pi_{i\in[1,n]}D_i$ be ordered component-wise and let $C^\sharp = \bigcup_{i\in I, j\in[1,n]}\{X_j \sqsubseteq_j proj_j \circ f_i((X_1, \ldots, X_n))\}$ where $proj_j$ is the projection onto the j^{th} component. A mapping $[X_1 \mapsto \hat{X}_1, \ldots, X_n \mapsto \hat{X}_n]$ is a solution of $C^\sharp$ if and only if the product $\hat{X} = (\hat{X}_1, \ldots, \hat{X}_n)$ is a solution of C.*

In the following, Propositions 1 and 2 are used to simplify the constraints until obtaining constraints that can be solved either symbolically or iteratively.

6.1 On simplifying *R*-constraints

A solution to the R-constraints is a mapping R_n^e which for each node n and exception e returns a function of type $Perm \to Perm$. However, by inspection of the constraints, one observes that *consume* and *grant* only modify a single resource type of a permission $Perm$. Using Proposition 1, we abstract $Perm \to Perm$ by the product $\prod_{rt \in ResType}(Perm_{rt} \times Mul \to Perm_{rt} \times Mul)$. Proposition 2 then allows to deal with each resource type separately. Moreover, operations on multiplicities and atomic permissions also act separately. Hence, using Proposition 1, for each resource type rt, we transform $Perm_{rt} \times Mul \to Perm_{rt} \times Mul$ to the product $(Perm_{rt} \to Perm_{rt}) \times (Mul \to Mul)$.

Solving the R-constraints now amounts to computing for each exception e, node n and resource type rt a pair of mappings: an atomic permission transformer $(Perm_{rt} \to Perm_{rt})$ and a multiplicity transformer $(Mul \to Mul)$. In the next sections, we define syntactic representations of these multiplicity transformers that are amenable to symbolic computations.

6.2 Syntactic functions for multiplicities

Before presenting our encoding of multiplicities transformers, we formally define the kind of constraints that we consider.

Definition 9 *A multiplicity constraint is a term $x \dot{\leq} e$ where*

- *x is variable of type $Mul \to Mul$;*
- *$\dot{\leq}$ is the point-wise ordering of multiplicity transformers induced by $\leq$;*
- *e is an expression built over the terms*

$$e ::= v|grant_{Mul}(m)|consume_{Mul}(m)|id|e;e$$

where v is a variable; $grant_{Mul}(m)$ is the constant function $\lambda x.m$; $consume_{Mul}(m)$ is the decrementing function $\lambda x.x - m$; id is the identity function $\lambda x.x$ and $f;g$ is function composition ($f;g = g \circ f$)

We define $MulF = \{\lambda x.min(c, x - d)|(c,d) \in Mul \times Mul\}$ as a restricted class of multiplicity transformers that is sufficiently expressive to represent the solution to the constraints. Elements of $MulF$ encode constant functions, decrementing functions and are closed under function composition as shown by the following equalities:

$$grant_{Mul}(m) = \lambda x.min(m, x - \bot_{Mul})$$
$$consume_{Mul}(m) = \lambda x.min(\infty, x - m)$$
$$\lambda x.min(c, x - d'); \lambda x.min(c', x - d') = \lambda x.min(min(c - d', c'), x - (d' + d))$$

We represent a function $\lambda x.min(c, x - d) \in MulF$ by the pair (c, d) of multiplicities. Using Proposition 1, we abstract our constraints over the domain $MulF^\sharp = Mul \times Mul$ equipped with the concretization $\gamma(c,d) \stackrel{\triangle}{=} \lambda x.min(c, x - d)$ and the abstract ordering $\sqsubseteq^\sharp$ defined as $(c,d) \sqsubseteq^\sharp (c',d') \stackrel{\triangle}{=} c \leq c' \wedge d' \leq d$.

Note however that because $MulF^\sharp$ is isomorphic to $MulF$, this abstraction does not incur any loss of precision.

6.3 Solving multiplicity constraints

The domain $MulF^\sharp$ does not satisfy the so-called *descending chain condition*. This means that iterative solving of the constraints might not terminate. Instead, we use an elimination-based algorithm. First, we use Proposition 2 to split our constraint system over $MulF^\sharp = Mul \times Mul$ into two constraint systems over Mul. Example 3 shows this transformation for a representative set of constraints.

Example 3 $C = \{Y \sqsubseteq^\sharp (c, d), Y' \sqsubseteq^\sharp X, X \sqsubseteq^\sharp Y;^\sharp Y'\}$ *is transformed into* $C' = C_1 \cup C_2$ *with* $C_1 = \{Y_1 \le c, Y_1' \le X_1, X_1 \le min(Y_1 - Y_2', Y_1')\}$ *and* $C_2 = \{Y_2 \ge d, Y_2' \ge X_2, X_2 \ge Y_2' + Y_2\}$.

Notice that C_1 depends on C_2 but C_2 is independent from C_1. This result holds generally and, as a consequence, these sets of constraints can be solved in sequence: C_2 first, then C_1.

To be solved, C_2 is converted into an equivalent system of fixpoint equations defined over the complete lattice $(Mul, \le, max, \perp_{Mul})$. The equations have the general form $x = e$ where $e ::= var \mid max(e, e) \mid e + e$. The elimination-based algorithm unfolds equations until a direct recursion is found. After a normalisation step, recursions are eliminated using a generalisation of Proposition 3 for an arbitrary number of occurences of the x variable.

Proposition 3 $x = max(x + e_1, e_2)$ *is equivalent to* $x = max(e_2 + \infty \times e_1, e_2)$.

Given a solution for C_2, the solution of C_1 can be computed by standard fixpoint iteration as the domain $(Mul, \le, min, \infty)$ does not have infinite descending chains. This provides multiplicity transformer solutions of the R-constraints. A similar methodology allows to compute permission transformers.

7 Related work

To the best of our knowledge, there is no formal model of the Java MIDP access control mechanism. A number of articles deal with access control in Java and $C^\sharp$ but they have focused on the stack inspection mechanism and the notion of granting permissions to code through privileged method calls. Earlier work by some of the present authors [4, 9] proposed a semantic model for stack inspection but was otherwise mostly concerned with proving behavioural properties of programs using these mechanisms. Closer in aim with the present work is that of Pottier *et al.* [12] on verifying that stack inspecting programs do not raise security exceptions because of missing permissions. Bartoletti *et al.* [1] also aims at proving that stack inspecting applets will not cause security exceptions and proposes the first proper modelling of exception handling. Both these works prove properties that allow to execute the program without dynamic permission checks. In this respect, they establish the same kind of property as we do in this paper. However, the works cited above do not deal with multiplicities of permissions and do not deal with the aspect of permissions granted on the fly through user interaction. The analysis of multiplicities leads to systems of numerical constraints which do not appear in the stack inspecting analyses.

Language-based access control has been studied for various idealised program models. Igarashi and Kobayashi [8] propose a static analysis for verifying that resources are accessed according to access control policies specified *e.g.* by finite-state automata, but do not study specific language primitives for implementing such an access control. Closer to the work presented in this article is that of Bartoletti *et al.* [2] who propose with $\lambda^{[]}$ a less general resource access control framework than Igarashi and Kobayashi, and without explicit notions of resources, but are able to ensure through a static analysis that no security violations will occur at run-time. They rely for that purpose on a type and effect system on $\lambda^{[]}$ from which they extract history expressions further model-checked. In the context of mobile agent, Hennessy and Riely [7] have developed a type system for the π-calculus with the aim of ensuring that a resource is accessed only if the program has been granted the appropriate permission (capability) previously. In this model, resources are represented by locations in a π-calculus term and are accessed via channels. Permissions are now capabilities of executing operations (*e.g.* read, transmit) on a channel. Types are used to restrict the access of a term to a resource and there is a notion of sub-typing akin to our order relation on permissions. The notion of multiplicities is not dealt with but could probably be accommodated by switching to types that are multi-sets of capabilities.

Our permission model adds a resource aspect to permissions which means that the analysis shares some goals with the analysis by Chandar *et al.* [5] on dynamic checks for verifying resource consumption. The safety property to verify here is that a program allocates memory cells before they are consumed. In our setting this would correspond to having a resource type Memory with one action 'allocate' that can be called a number of times according to how much memory is granted.

8 Conclusions

We have proposed an access control model for programs which dynamically acquire permissions to access resources. The model extends the current access control model of the Java MIDP profile for mobile telephones by introducing multiplicities of permissions together with explicit instructions for granting and consuming permissions. These instructions allow to improve the usability of an application by fine-tuning the number and placement of user interactions that ask for permissions. In addition, programs written in our access control model can be formally and statically verified to satisfy the fundamental property that a program does not attempt to access a resource for which it does not have the appropriate permission. The formalisation is based on a model of permissions which extends the standard object $\times$ action model with multiplicities. We have given a formal semantics for the access control model, defined a constraint-based analysis for computing the permissions available at each point of a program, and shown how the resulting constraint systems can be solved. To the best of our knowledge, it is the first time that a formal treatment of the Java MIDP model has been proposed.

The present model and analysis has been developed in terms of control-flow graphs and has ignored the treatment of data such as integers *etc.* By combining our analysis with standard data flow analysis we can obtain a better approximation of integer variables and hence the number of times a permission-consuming loop is executed. In the present model, we either have to require that there is a `grant` executed for each `consume` inside the loop or that the relevant permission has been granted with multiplicity ∞ before entering the loop. Allowing a `grant` to take a variable as multiplicity parameter combined with a relational analysis (the *octagon* analysis by Miné [10]) is a straightforward extension that would allow to program and verify a larger class of programs.

This work is intended for serving as the basis for a Proof Carrying Code (PCC) [11] architecture aiming at ensuring that a program will not use more resources that what have been declared. In the context of mobiles devices where such resources could have an economic (via premium-rated SMS for instance) or privacy (via address-book access) impact, this would provide improved confidence in programs without resorting to third-party signature. The PCC certificate would consist of the precomputed P_n and R_n^e. The host device would then check that the transmitted certificate is indeed a solution. Note that no information is needed for intra-procedural instructions other than `grant` and `consume`—this drastically reduces the size of the certificate.

References

[1] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. Static analysis for stack inspection. *Electronic Notes in Computer Science*, 54, 2001.

[2] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. History-based access control with local policies. In *Proceedings of FOSSACS 2005*, volume 3441 of *Lecture Notes in Computer Science*, pages 316–332. Springer-Verlag, 2005.

[3] Frédéric Besson, Thomas de Grenier de Latour, and Thomas Jensen. Interfaces for stack inspection. *Journal of Functional Programming*, 15(2):179–217, 2005.

[4] Frédéric Besson, Thomas Jensen, Daniel Le Métayer, and Tommy Thorn. Model ckecking security properties of control flow graphs. *Journal of Computer Security*, 9:217–250, 2001.

[5] Ajay Chander, David Espinosa, Nayeem Islam, Peter Lee, and George C. Necula. Enforcing resource bounds via static verification of dynamic checks. In *Proceedings of the 14th European Symposium on Programming, ESOP 2005*, volume 3444 of *Lecture Notes in Computer Science*, pages 311–325. Springer-Verlag, 2005.

[6] Cedric Fournet and Andy Gordon. Stack inspection: theory and variants. In *Proceedings of the 29th ACM Symp. on Principles of Programming Languages (POPL'02).* ACM Press, 2002.

[7] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173(1):82–120, 2002.

[8] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *Proceedings of the 29th ACM Symp. on Principles of Programming Languages (POPL'02)*, pages 331–342, 2002.

[9] Thomas Jensen, Daniel Le Métayer, and Tommy Thorn. Verification of control flow based security properties. In *Proceedings of the 20th IEEE Symp. on Security and Privacy*, pages 89–103. New York: IEEE Computer Society, 1999.

[10] Antoine Miné. The octogon abstract domain. In *Proceedings of the 8th Working Conference On Reverse Engineering (WCRE 01)*, pages 310–320. IEEE, 2001.

[11] George C. Necula. Proof-carrying code. In Neil D. Jones, editor, *Proceedings of the 24th ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, January 1997. ACM Press.

[12] François Pottier, Christian Skalka, and Scott F. Smith. A systematic approach to static access control. In *Proceedings of the 10th European Symposium on Programming, ESOP 2001*, volume 2028 of *Lecture Notes in Computer Science*, pages 30–45. Springer-Verlag, 2001.

[13] Jerry H. Saltzer and Mike D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63:1278–1308, 1975.

[14] Sun Microsystems, Inc., Palo Alto/CA, USA. *Mobile Information Device Profile (MIDP) Specification for Java 2 Micro Edition, Version 2.0*, 2002.

[15] Hisao Tamaki and Taisuke Sato. OLD resolution with tabulation. In *Proceedings on Third International Conference on Logic Programming*, volume 225 of *Lecture Notes in Computer Science*, pages 84–98. Springer-Verlag, 1986.

A Correctness

In this section, we prove the correctness of our analysis, as stated in Theorem 1. This proof relies on a big-step semantics on CFGs, defined in Figure 5. This semantics is further from the small-step semantics defined in Figure 2 but is easier to reason with and forms an important part of the correctness proof of the analysis.

The big-step semantics is formally defined by a relation $\triangleright$ between elements of $(NO \times Perm)$. Note that in the inference rules of Figure 5, the relation $\overset{ex}{\triangleright}$ denotes that an exception ex has been thrown and not yet caught.

$$\frac{KD(n) = \mathtt{grant}(p, m) \quad n \overset{TG}{\to} n'}{n, \rho \triangleright n', grant(p, m)(\rho)} \qquad \frac{KD(n) = \mathtt{consume}(p) \quad n \overset{TG}{\to} n'}{n, \rho \triangleright n', consume(p)(\rho)} \qquad \frac{KD(n) = \mathtt{throw}(ex) \quad n \overset{ex}{\to} h}{n, \rho \triangleright h, \rho}$$

$$\frac{}{n, \rho \triangleright n, \rho} \qquad \frac{n, \rho \triangleright n_1, \rho_1 \quad n_1, \rho_1 \triangleright n', \rho'}{n, \rho \triangleright n', \rho'} \qquad \frac{KD(n) = \mathtt{call} \quad KD(r) = \mathtt{return} \quad n \overset{CG}{\to} m \quad n \overset{TG}{\to} n' \quad m, \rho \triangleright r, \rho'}{n, \rho \triangleright n', \rho'}$$

$$\frac{KD(n) = \mathtt{throw}(ex) \quad \forall h, n \overset{ex}{\not\to} h}{n, \rho \overset{ex}{\triangleright} n, \rho} \qquad \frac{n, \rho \triangleright n_1, \rho_1 \quad n_1, \rho_1 \overset{ex}{\triangleright} n', \rho'}{n, \rho \overset{ex}{\triangleright} n', \rho'}$$

$$\frac{KD(n) = \mathtt{call} \quad n \overset{CG}{\to} m \quad \forall h, n \overset{ex}{\not\to} h \quad m, \rho \overset{ex}{\triangleright} t, \rho'}{n, \rho \overset{ex}{\triangleright} n, \rho'} \qquad \frac{KD(n) = \mathtt{call} \quad n \overset{CG}{\to} m \quad n \overset{ex}{\to} h \quad m, \rho \overset{ex}{\triangleright} t, \rho'}{n, \rho \triangleright h, \rho'}$$

Fig. 5. Big-step operational semantics

Using the big-step instrumented semantics, we define the set Acc of accessible nodes and permissions from the initial node n_0 as follows:

$$\frac{}{(n, p_{init}) \in Acc} \qquad \frac{(n, \rho) \in Acc \quad n \overset{CG}{\to} m}{(m, \rho) \in Acc} \qquad \frac{(n, \rho) \in Acc \quad n, \rho \triangleright n', \rho'}{(n', \rho') \in Acc}$$

It captures all nodes and permissions reachable through the $\triangleright$ relation from the initial node and permission plus those for methods that do not return (second inference rule). Indeed, in the big-step semantics, in order to relate a node and permission with $\triangleright$ to a `call` node, a `return` node must be reached (sixth inference rule of Figure 5).

This definition of accessibility allows to structure the correctness proof into two parts. The first part of the proof of Theorem 1 amounts to showing that if the analysis declares that no abstract state indicates an access without the proper permission then this is indeed the case for all the accessible states in program.

Lemma 1

$$(\forall n \in NO, \neg Error(P_n)) \Rightarrow \forall (n, \rho) \in Acc, \neg Error(\rho)$$

Proof. We need two intermediary results:

- First, we have to show a correctness result on the definition of R (which is used in the definition of P_n), stated as:

$$\forall n\ n' \in NO, \forall \rho\ \rho' \in Perm,\ n, \rho \triangleright n', \rho' \wedge KD(n') = \mathtt{return} \Rightarrow R_n(\rho) \sqsubseteq_p \rho'$$

  This proof is done by induction over the definition of $\triangleright$. For example, in the case of a method call ($n \overset{CG}{\to} m \quad n \overset{TG}{\to} n' \quad m, \rho \triangleright r, \rho'$), we will have to prove $R_n(\rho) \sqsubseteq_p \rho'$, which is done using transitivity. The step $R_n(\rho) \sqsubseteq_p R_{n'}(R_m(\rho))$ is obtained from the constraint on R_n for method calls. The step $R_{n'}(R_m(\rho)) \sqsubseteq_p R_m(\rho)$ is obtained from the constraint on $R_{n'}$ for returns. The last transitivity step, $R_m(\rho) \sqsubseteq_p \rho'$ is given by the induction hypothesis.
- Then, we have to relate the notion of accessibility and the definition of P_n:

$$\forall n \in NO, \forall \rho \in Perm,\ (n, \rho) \in Acc \Rightarrow P_n \sqsubseteq_p \rho$$

  We prove this result first by induction over *Acc* (the two first cases directly match with the corresponding rule on P_n) then by induction over $\triangleright$ for the third rule of *Acc*. For the same example of a method call as before, we will have to prove $P_{n'} \sqsubseteq_p \rho'$. We split this goal using transitivity into $P_{n'} \sqsubseteq_p R_m(P_n)$ (deduced from constraints on $P_{n'}$ for method calls), $R_m(P_n) \sqsubseteq_p R_m(\rho)$ (R_m is proved to be monotone and $(P_n) \sqsubseteq_p \rho$ by induction hypothesis) and $R_m(\rho) \sqsubseteq_p \rho'$ (from the first intermediary result above, since $m, \rho \triangleright r, \rho'$ with $KD(r) = \mathtt{return}$).

The lemma is a consequence of this last result, using proof by contradiction. We suppose $(n, \rho) \in Acc$ with $Error(\rho)$, then we get $P_n \sqsubseteq_p \rho$, which contradicts $\neg Error(P_n)$ given $Error(\rho)$. □

The second part links the trace semantics with the big-step instrumented semantics by proving that if no accessible state in the instrumented semantics has a tag indicating an access control error then the program is safe with respect to the definition of safety of execution traces. This part amounts to showing that the instrumented semantics is a monitor for the $Safe$ predicate.

Lemma 2 *Given a program P:*

$$\forall (n, \rho) \in Acc, \neg Error(\rho) \Rightarrow \forall tr \in [\![P]\!], Safe(tr)$$

Proof. First, we relate the small-step to the big-step operational semantics:

$$\forall n \in NO, \forall s \in NO^*, \forall \rho \in Perm,\ n_0, \epsilon, p_{init} \twoheadrightarrow^* n{:}s, \epsilon, \rho \Rightarrow (n, \rho) \in Acc$$

where $\twoheadrightarrow^*$ is the reflexive-transitive closure of $\twoheadrightarrow$. The sketch of the proof is similar to [3, Section 2.3]. It amounts to first restraining the result to a fixed stack that can not be popped by transition relations (to relate to intra-procedural big step transitions) and then to include call, return and exception steps. Finally, we prove the lemma by contradiction, assuming that for a node n in the trace is such that the associated permission is an error. By definition of the trace, this node is accessible from n_0, p_{init} with $\twoheadrightarrow^*$, then we have $(n, \rho) \in Acc$ with $Error(\rho)$ that contradicts the hypothesis of our lemma. □

The proof of Theorem 1 is a direct consequence of Lemmas 1 and 2.