Verifying Resource Access Control on Mobile Interactive Devices

Frédéric Besson[§] Guillaume Dufay[§◇] Thomas Jensen[‡]
David Pichardie[§*†]

[§] Inria Rennes, Campus de Beaulieu, 35042 Rennes Cedex, France

[‡] CNRS, Campus de Beaulieu, 35042 Rennes Cedex, France

[◇] Trusted Labs, 78000 Versailles, France

Abstract

A model of resource access control is presented 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 security architecture used in Java-enabled mobile telephones. We extend the Java model to include access control permissions with multiplicities in order to allow to use a permission a certain number of times. We define a program model based on control flow graphs together with its operational semantics and provide a formal definition of the basic security policy to enforce *viz* that an application will always ask for a permission before using it to access a resource. A static analysis which enforces the security policy is defined and proved correct. A constraint solving algorithm implementing the analysis is presented.

1 Introduction

Availability of services and resources has been identified as a fundamental property of information system security, together with confidentiality and integrity. One of the principal techniques that helps ensuring availability is dynamic monitoring of how processes consume their allocated resources, as *e.g.*, the resource allocation model for denial of service developed by Millen [15]. In this model, a trusted resource monitor maintains a resource allocation matrix which records for each process and each resource, how many units of the resource are currently allocated to the process. Prior to consuming a resource, the monitor verifies that the necessary resources have indeed been allocated to the process. This guarantees that no process will attempt to consume more resources than it has been allocated.

The purpose of this article is to transpose this process-oriented resource allocation model to a more programming language-oriented model in which the operations for resource manipulation becomes part of a high-level programming language. The goal is to provide a technique

*Corresponding author.

†This work was partly funded by the IST-FET programme of the European Commission, under the IST-2005-015905 MOBIUS project and by the ANR-SETI-06-010 grant.

for reasoning about such resource-aware programs and for proving *statically* that a particular program does not attempt to use more resources than it has been allocated. This is the major difference with respect to the dynamic monitoring approach cited above. More accurately, we will propose a model in which static verification is mixed with dynamic allocation and monitoring to offer the most flexible programming possible.

Access control to resources is traditionally 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. The language-based access control mechanisms proposed recently for languages such as Java and C# 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♯) 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.* [3, 7, 10, 13]. Another example is the security architecture for embedded Java on mobile telephones, defined in the Mobile Information Device Profile (MIDP) [23] for Java, 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 are 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 [20] and the security property stated below.

In this article, we present a formal model of resource access control 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 programming constructs used in the model and a logic for reasoning about the flow of permissions in programs using these constructs. The basic security property that we aim to prove for an application is that *a program will never access a resource for which it does not have permission.* This property can be established by inserting dynamic (run-time) checks that monitor the execution and examine the available permissions before each resource access. However, such dynamic monitoring has two drawbacks. First, it incurs an over-head in execution time—this is likely to be modest for the monitoring in question. Second, and more importantly, when an illegal access is detected a security exception is raised and must be dealt with, interrupting the normal execution of the application. While this does not jeopardize the security of the application as such, it is undesirable from a usability point of view. For these reasons, we propose to develop an access control model that allows to analyse and verify statically (*i.e.*, before execution) that an application will acquire all the permissions necessary for its execution.

The notion of permission is central to our model. Permissions have an internal structure (formalised in Section 3) that describes the actions that they enable and the set of objects to which they apply. The "one-shot" permissions from Java MIDP (presented in Section 2) have motivated a generalisation in which permissions now have *multiplicities*, stating how many times the given permission can be used. Multiplicities are important for controlling resource accesses that have a cost, such as sending of certain text messages and connections to certain network services on mobile telephones. In addition, adding a specific *zero* multiplicity allows to extend the MIDP model with revocation of permissions.

The security model we propose has two basic constructs for manipulating permissions:

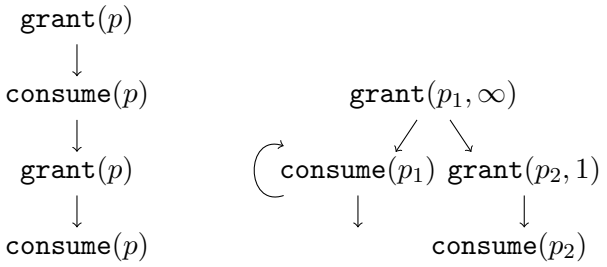


Figure 1: Left: current MIDP permissions. Right: new permission model

- `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.

An important feature of the new model we propose is that an application can request one or more permissions in advance instead of having to ask permission just before consuming it, as it currently happens in the MIDP access control model based on the "one-shot" permissions. For example, in our model it is possible for a user to grant an applet the permission to send 3 SMSs during a transaction. Figure 1 illustrates the difference between the two models with control flow graph example. To the left, a `grant` precedes directly `consume` whereas to the right we ask for a number of p_1 permissions in advance which are then consumed in the loop without any further user interaction

The choice of where to insert requests for user-granted permissions now becomes 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 of placement will ensure that an applet always has the permissions necessary for its further execution. 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. The correctness of this analysis is established in Section 6. Section 7 describes how to solve the constraints produced by the analysis. Section 8 describes related formal models and verification techniques for language-based access control and Section 9 concludes.

The present article is an extended and improved version of [6]. Apart from providing more explanations in general, the present article extends the program model to handle permission management inside loop structures and proposes an analysis that is more precise than the analysis proposed in *loc. cit.* in the way that exceptions are handled.

2 The Java MIDP security model

The Java MIDP programming model for mobile telephones [23] proposes a thoroughly developed security architecture which is the starting point of our work. In the MIDP security

model, applications (called *midlets* in the MIDP jargon) are downloaded and executed by a Java virtual machine. Midlets are made of a single archive (a jar file) containing complete programs. 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 that, 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. Within this protection domain, 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. For example, the user may be overwhelmed by innocuous requests, and then (when she is no longer taking them seriously) asked for permission on a costly resource like sending a message to a premium number. 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 at application start-up whether he grants the permission for the session and by assigning an infinite number of the given permission;
- 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 will not abuse permissions (security concern) and will be granted sufficient permissions by the programmer for a correct execution (usability concern). Another feature enabled in the extended model is the ability to *revoke* permissions which is not present in the current MIDP model. We return to this point at the end of Section 3.

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 combine this permission model with multiplicities and resource types, in a way inspired by the resource allocation matrices used by Millen in his resource allocation model [15]. However, we add more structure to the set of resources. 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:

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

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

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

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:*

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

relating a type of resources with the actions that can be performed on it[1]*. The element $\bot$ represents an invalid permission. By extension, we define the set of permissions Perm as the dependent product:*

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

[1]In the following, we use regular expressions to denote sets of resources (files, telephone numbers *etc*).

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 resource type, i.e., *$(\rho[rt \mapsto (p, m)])(rt) = (p, m)$.*

Example 1. *Given two resource types CheapSMS and ExpensiveSMS $\in ResType$, the permission $\rho \in Perm$ defined by*

$$[CheapSMS \mapsto ((+1800*, \{send\}), 2), ExpensiveSMS \mapsto ((0033*, \{send\}), 1)]$$

grants two accesses to a send action of the resource $+1800*$ (phone number starting with $+1800$) of type CheapSMS and one access to the send action of an expensive SMS on a number starting with 0033.

Notice that this way of modeling permissions imposes a restriction on how the granting of permissions can be expressed. In particular, it is not possible to express permission to send, say, two SMSs to number n_1 and three SMSs to number n_2 if n_1 and n_2 belong to the same resource type SMS, say. In this case, either permission can be granted to send five messages to the set of resources $\{n_1, n_2\}$ of type SMS or different resource types must be introduced to distinguish n_1 and n_2.

Definition 2. *The partial order $\sqsubseteq_p \subseteq Perm \times Perm$ on permissions is given by*

$$\rho_1 \sqsubseteq_p \rho_2 \stackrel{\triangle}{=} \forall rt \in ResType \quad \rho_1(rt) \sqsubseteq \rho_2(rt)$$

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. For example,

$$[ExpensiveSMS \mapsto ((0033*, \{send\}), 1)] \sqsubseteq [ExpensiveSMS \mapsto ((*, \{send\}), 2)]$$

The ordering induces a greatest lower bound operator $\sqcap : Perm \times Perm \to Perm$ 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:

- 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 $\rho \in Perm$, $rt \in ResType$, $p, p' \in Perm_{rt}$, $m \in Mul$ and assume that $\rho(rt) = (p, m)$. The operation consume : $Perm_{rt} \to Perm \to Perm$ 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. Attempting to consume a resource for which there is no permission ($p' \not\sqsubseteq_{rt} p$) is an error. Similarly, consuming a resource for which the multiplicity is zero will result in setting the multiplicity to $\perp_{Mul}$.

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 $grant : Perm_{rt} \times Mul \to Perm \to Perm$ 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 forbids that permissions be granted for performing one task and then used later on to accomplish another. The *grant* operation could also add the granted permission to the existing ones rather than replace the corresponding one. Besides cumulating the number of permissions for permissions sharing the same type and resource, this would allow different resources for the same resource type. However, the `consume` operation becomes much more complex, as a choice between the overlapping permissions may occur. Analysis would require handling multisets of permissions.

A consequence of the fact that permissions do not accumulate is that our model can accommodate the *revocation* of permissions, by asking the user to grant zero permissions of a given type. As pointed out in Section 2, this is a difference with respect to the Java MIDP model. In particular, this enables the programmer to impose scope on permissions by inserting a `grant` instruction with null multiplicity at the end of the permission scope.

4 Program model

In this section we present the formal definition of the permission usage of a program during its execution. We first present the control-flow graph model of programs, then explain how this kind of model can be generated from a real program and finally present the formal meaning of the term *safe execution* in such a model.

We model a program by a control-flow graph (CFG) that captures the manipulations of permissions (grant and consume), the handling of program loops, method calls and returns, and models the way that exceptions are thrown and handled in a language like Java. These operations are respectively represented by the instructions $\texttt{grant}(p, m)$, $\texttt{consume}(p)$, $\texttt{call}^i$, `return`, $\texttt{throw}(ex)$, with $i \in \mathbb{N}, ex \in EX, rt \in ResType, p \in Perm_{rt}$ and $m \in Mul$. This is an enriched version of the models used in previous work on modelling access control for Java [3, 7, 13]. The instruction $\texttt{call}^i$ combines the essential features of method calls and iteration into one idealized instruction. Its (non-deterministic) semantics is that it will call at a given point in the execution, a particular method k times with $1 \leq k \leq i$, unless an exception is raised. Ordinary method calls correspond to $\texttt{call}^i$ with $i = 1$. The $\texttt{throw}(ex)$ instruction will

throw the exception ex. This exception can be caught inside the method currently executing in which case an edge indicates the way to the relevant handler. Otherwise, it escapes the method and will be handled in the enclosing methods.

Definition 6. *A control-flow graph is a 7-tuple*

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

where:

- NO *is the set of nodes of the graph;*
- EX *is the set of exceptions;*
- $KD : NO \to \{\texttt{grant}(p, m), \texttt{consume}(p), \texttt{call}^i, \texttt{return}, \texttt{throw}(ex)\}$*, 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, which can capture dynamic method calls;*
- $EG \subseteq EX \times NO \times NO$ *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 $n \overset{TG}{\to} n'$ for $(n, n') \in TG$, $n \overset{CG}{\to} n'$ for $(n, n') \in CG$ and $n \overset{ex}{\to} n'$ for $(ex, n, n') \in EG$.

In the sequel, we only consider control flow graphs such that the entry-point of a method is unique. In terms of graphs, this translates to the fact that targets of intra-procedural edges are the roots of disjoint sub-graphs. Formally stated, we enforce the following property:

$$n \overset{CG}{\to} e \wedge n \overset{CG}{\to} e' \wedge e \overset{TG^\star}{\to} r \wedge e' \overset{TG^\star}{\to} r \Rightarrow e = e'$$

Figure 2 contains an example of the control flow graph of `grant` and `consume` operations from a fictitious 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. Once satisfied with information provided, he can either book the flight or pay the desired flight. In both cases, the identity of the user is required, hence the corresponding permission is asked from the outset.

In the case of payment, the application asks for permission to access information concerning bank detail such as credit card number. This could also have been asked from the start as part of p_{init}, but is instead obtained via a dynamic request that is being executed only if the payment branch of the application is actually chosen. In the example, the developer has chosen to delay asking for the permission of accessing credit card information concerning credit limit and card number 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 $file \mapsto (/wallet/*, 3)$ instead of $file \mapsto (/wallet/id, 1)$ which would grant the application with more permissions than strictly needed, going against the principle of least privilege.

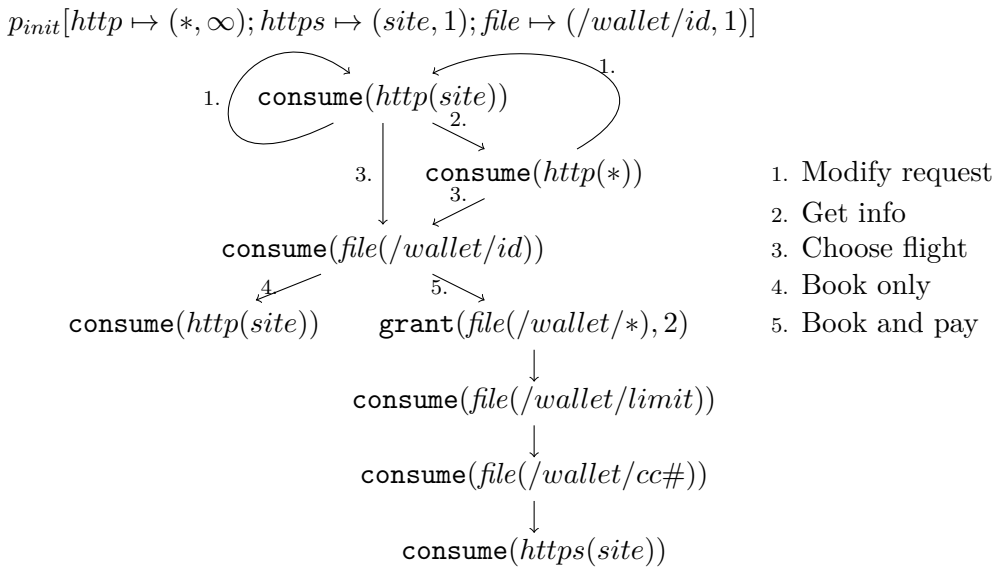


Figure 2: Example of grant/consume permissions patterns

Model generation

The control flow graph we consider here is relatively standard. Since we only focus on branching instructions, resource uses and method call/returns, this model can be seen as a compression of the general control flow graphs that can be generated by tools like Bandera [2], or Soot [21].

Starting from a standard CFG builder we need to perform several compressing transformation to fit in our model. A block without any method call or resource use can be compressed into one single node. For each loop containing a method call or a resource usage we try to statically bound the number of possible iterations of this loop in order to generate a suitable $\texttt{call}^i$ instruction. This non-trivial task is accomplished by a numerical analysis for bounding the number of times a program point (the head of the loop) is reached during the execution of a method. Such an information is easily instrumented by a ghost variable $LOOP(pc)$ increased before executing the instruction at program point pc. Standard numerical abstraction will then be able to bound $LOOP(pc)$ for standard "for" loop such as `for (int i=0; i<100; i++)` (here an interval on $LOOP(pc) - i$ is sufficient) but also some others statically bounded loops. A loop containing a resource usage (directly or indirectly via method call) which cannot be statically bounded by a constant is simply translated into a loop in the control flow graph.

Operational semantics

We define the small-step operational semantics of CFGs in Figure 3 which defines an interpretation of CFGs in terms of execution traces. The semantics is stack-based and mimics the behaviour of a standard programming language with methods and exceptions, e.g., as Java or C♯. 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 $\texttt{call}^i$ of Figure 3, 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 $s \in NO^*$. In order to take into account the iterative aspect of the $\texttt{call}^i$ instruction, we add a formal exponent i on the n component of the stack. This exponent is used to remember how many times the method can be called again and is updated in the rules for the method return instruction `return` as follows. The first rule for `return` deals with the states where the exponent i is strictly greater than 1. In this case the method can be executed again, in a context where the exponent now is decremented by one. The conditions $n \overset{CG}{\to} m$ and $m \overset{TG^*}{\to} r$ serve to ensure that the new iteration invokes the method from which execution returns (in case the calling node n is linked to several method entries). The second rule for `return` describes the case where iteration stops: if the exponent is greater than or equal to one then the iteration can stop and execution proceed at the node following the call node n.

Instructions may change the value of the permission along with the current state. *E.g.*, for the instruction `grant` of Figure 3, 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 in NO^*, e in $(EX \cup \{\epsilon\})$, ρ' in $Perm$, if there exists s' in NO^*, e' in $(EX \cup \{\epsilon\})$, ρ' in $Perm$ such that $s, e, \rho \twoheadrightarrow s', e', \rho'$, then for all ρ and ρ', the same transition holds.

For the instruction $\texttt{throw}(ex)$, we distinguish two cases depending on whether the exception is handled in the current method or not. In the first case ($n \overset{ex}{\to} h$), the execution continues in the same method following the corresponding exception edge. In the second case, the exception ex is recorded in the current state which becomes now an *exception* state (*i.e.* a state of the form (s, e, ρ) with $e \neq \epsilon$). The two last rules explain how to handle an exception state. In the first one, there is no exception edge in the current caller to catch the exception ex, the exception escapes the method and the control is transfered to the caller with an exception state. Otherwise, the caller catches the exception and receives the control at the node h targeted by the exception edge.

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

$$\frac{KD(n) = \texttt{call}^i \quad n \overset{CG}{\to} m}{n{:}s, \epsilon, \rho \twoheadrightarrow m{:}n^i{:}s, \epsilon, \rho}$$

$$\frac{KD(r) = \texttt{return} \quad n \overset{CG}{\to} m \quad m \overset{TG^*}{\to} r \quad i > 1}{r{:}n^i{:}s, \epsilon, \rho \twoheadrightarrow m{:}n^{i-1}{:}s, \epsilon, \rho} \qquad \frac{KD(r) = \texttt{return} \quad n \overset{TG}{\to} n' \quad i \geq 1}{r{:}n^i{:}s, \epsilon, \rho \twoheadrightarrow n'{:}s, \epsilon, \rho}$$

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

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

Figure 3: 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* $0 \leq i < k$ *there exists* $\rho, \rho' \in Perm$, $s, s' \in NO^*$ *such that* $n_i{:}s, e_i, \rho \twoheadrightarrow n_{i+1}{:}s', e_{i+1}, \rho'$.

For a program Pg represented by its control-flow graph G, we will denote by $[\![Pg]\!]$ *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.

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

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 $((\{r\}, Act), \infty)$ with that resource type.

A trace is *safe* if none of its prefixes ends 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 static data 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. 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. We follow a standard approach for defining static analyses by providing rules for translating a program into a constraint system over a domain of abstract properties, the solution of which is the approximation sought [1]. In Section 7 we then provide the algorithms for solving these constraints.

The constraints are derived from the structure of the CFG following the rules in Figure 4. 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 defined below. This transfer function describes the effect of one execution of a method so to take into account the iteration in a $\texttt{call}^i$, we apply the R function i times to the set of permissions available at point of the method call.

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.

$$\frac{KD(n) = \texttt{grant}(p,m) \quad n \overset{TG}{\to} n'}{P_{n'} \sqsubseteq_p grant(p,m)(P_n)} \qquad \frac{KD(n) = \texttt{consume}(p) \quad n \overset{TG}{\to} n'}{P_{n'} \sqsubseteq_p consume(p)(P_n)}$$

$$\frac{KD(n) = \texttt{call}^i \quad n \overset{CG}{\to} m \quad n \overset{TG}{\to} n'}{P_{n'} \sqsubseteq_p \bigsqcap_{1 \le j \le i} (R^{\epsilon}_m)^j (P_n)} \qquad \frac{KD(n) = \texttt{call}^i \quad n \overset{CG}{\to} m}{P_m \sqsubseteq_p P_n}$$

$$\frac{KD(n) = \texttt{call}^i \quad n \overset{CG}{\to} m \quad n \overset{ex}{\to} h}{P_h \sqsubseteq_p (\bigsqcap_{0 \le j \le i-1} (R^{\epsilon}_m)^j ; R^{ex}_m)(P_n)} \qquad \frac{KD(n) = \texttt{throw}(ex) \quad n \overset{ex}{\to} m}{P_m \sqsubseteq_p P_n}$$

$$\frac{}{P_{n_0} \sqsubseteq_p p_{init}}$$

Figure 4: Constraints on minimal permissions

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;
- 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^{ex}_n 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 5 (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. Return nodes keep the permissions unchanged and are therefore modelled by the identity transfer functions. The rules for `call` nodes are best explained for $i = 1$. A method call $\texttt{call}^1$ at node n that does not cause an exception is modeled by R^{ϵ}_m (the effect of the called method m returning normally) followed by the effect of $R^{e}_{n'}$ (the effect of the continuation of the execution after the call). In case a node n calls a method m that raises an exception ex, the behaviour depends on whether the exception is

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

$$\frac{KD(n) = \texttt{return}}{R^e_n \sqsubseteq \lambda\rho.\rho} \qquad \frac{KD(n) = \texttt{call}^i \quad n \overset{CG}{\to} m \quad n \overset{TG}{\to} n'}{R^e_n \sqsubseteq \bigsqcap_{1\le j\le i}(R^\epsilon_m)^j; R^e_{n'}}$$

$$\frac{KD(n) = \texttt{call}^i \quad n \overset{CG}{\to} m \quad \forall n', n \overset{ex}{\not\to} n'}{R^{ex}_n \sqsubseteq \bigsqcap_{0\le j\le i-1}(R^\epsilon_m)^j; R^{ex}_m} \qquad \frac{KD(n) = \texttt{call}^i \quad n \overset{CG}{\to} m \quad n \overset{ex}{\to} h}{R^e_n \sqsubseteq \bigsqcap_{0\le j\le i-1}(R^\epsilon_m)^j; R^{ex}_m; R^e_h}$$

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

Figure 5: Summary functions of the effect of the execution on initial permission

caught or not. If the exception is uncaught (there is no exception edge $n \overset{ex}{\to} n'$), the effect R^{ex}_m is an upper bound for R^{ex}_n. If the exception is caught by a handler (there is an exception edge $n \overset{ex}{\to} h$), the effect on node n is the composition of effect R^{ex}_m followed by the effect R^e_h of the successor node. In order to generalise these rules for an arbitrary i, we use the fact that a $\texttt{call}^i$ amounts to calling a method, say m, at most i times ($1 \le i$). The effect of calling at most i times a method m that does not cause an exception is hence the intersection of the effects $(R^\epsilon_m)^j$ for any j less than i *i.e.,* $\bigsqcap_{1\le j\le i}(R^\epsilon_m)^j$. If an exception occurs during the execution of m, we take into account the j previous executions without exception, for $0 \le j < i$ with $\bigsqcap_{0\le j\le i-1}(R^\epsilon_m)^j$. The rules for `throw` are straightforward. If a node n throws an exception ex that is immediately caught at node h, the effect of n (R_n) is bounded by the effect of h (R_h). If the exception ex at node n is uncaught, the permissions reaching n escape with exception ex *i.e.,* R^{ex}_n is bounded by the identity function.

6 Correctness

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 (*cf.* Definition 4), then the program will behave safely. Formally,

Theorem 1 (Basic Security Property)**.** *Given a program Pg, let (P, R) be a solution to the constraints generated by Pg. If the functions $(R_n)_{n\in NO}$ are monotone, then*

$$\forall n, (\forall p, KD(n) = consume(p) \Rightarrow \neg Error(P_n)) \Rightarrow \forall tr \in [\![Pg]\!], Safe(tr)$$

The proof of this theorem uses a big-step operational semantics which is shown equivalent to the small-step semantics of Figure 3. This big-step semantics is easier to reason with (in particular for method invocation) and yields an accessibility relation Acc that also captures non-terminating methods. The result only holds for monotone R_n solutions to the summary function constraints. In Section 7 we show how to solve these over a domain of monotone functions.

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. To prove this, we first show (by induction over the definition of the big-step semantics) that summary functions R correctly model the effect of method calls on permissions. Then, we show a similar result for the permissions computed for each program point by the analysis. 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.

We define the big-step semantics of CFGs used to prove Theorem 1 in Figure 6. This semantics is a step away from the execution but is still equivalent to the small-step semantics defined in Figure 3. The structure of the proof is standard and similar to that in [5, Section 2.3] and to a similar proof for a small-step and a big-step operational semantics of the JVM [18] that has been machine-checked with the Coq proof assistant. 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 6, the relation $\overset{ex}{\triangleright}$ denotes that an exception ex has been thrown and not yet caught. In the semantics, there are three groups of rules: intra-

$$\textbf{Grant}\frac{KD(n) = \texttt{grant}(p,m) \quad n \overset{TG}{\rightarrow} n'}{n, \rho \triangleright n', grant(p,m)(\rho)} \qquad \textbf{Consume}\frac{KD(n) = \texttt{consume}(p) \quad n \overset{TG}{\rightarrow} n'}{n, \rho \triangleright n', consume(p)(\rho)}$$

$$\textbf{ThrowCatch}\frac{KD(n) = \texttt{throw}(ex) \quad n \overset{ex}{\rightarrow} h}{n, \rho \triangleright h, \rho} \qquad \textbf{ThrowEscape}\frac{KD(n) = \texttt{throw}(ex) \quad \forall h, n \overset{ex}{\not\rightarrow} h}{n, \rho \overset{ex}{\triangleright} n, \rho}$$

$$\textbf{CallReturn}\frac{\begin{array}{c} KD(n) = \texttt{call}^i \quad KD(r) = \texttt{return} \\ n \overset{CG}{\rightarrow} m \quad n \overset{TG}{\rightarrow} n' \\ 1 \leq k \leq i \quad \forall j \in 1, \ldots, k,\ m, \rho_j \triangleright r, \rho_{j-1} \end{array}}{n, \rho_k \triangleright n', \rho_0}$$

$$\textbf{CallEscape}\frac{KD(n) = \texttt{call}^i \quad n \overset{CG}{\rightarrow} m \quad \forall h, n \overset{ex}{\not\rightarrow} h \quad m, \rho \overset{ex}{\triangleright} t, \rho'}{n, \rho \overset{ex}{\triangleright} n, \rho'} \qquad \textbf{CallCatch}\frac{KD(n) = \texttt{call}^i \quad n \overset{CG}{\rightarrow} m \quad n \overset{ex}{\rightarrow} h \quad m, \rho \overset{ex}{\triangleright} t, \rho'}{n, \rho \triangleright h, \rho'}$$

$$\textbf{Refl}\frac{}{n, \rho \triangleright n, \rho} \qquad \textbf{Trans}\frac{n, \rho \triangleright n_1, \rho_1 \quad n_1, \rho_1 \triangleright n', \rho'}{n, \rho \triangleright n', \rho'} \qquad \textbf{TransExc}\frac{n, \rho \triangleright n_1, \rho_1 \quad n_1, \rho_1 \overset{ex}{\triangleright} n', \rho'}{n, \rho \overset{ex}{\triangleright} n', \rho'}$$

Figure 6: Big-step operational semantics

procedural, inter-procedural and closure rules. The rules **Grant**, **Consume**, **ThrowCatch** and **ThrowEscape** are intra-procedural. Their direct counterparts in the small-steps semantics only update the current node and permissions. The rules **CallReturn**, **CallEscape** and **CallCatch** are inter-procedural. The rule **CallReturn** is used to match $\texttt{call}^i$/`return` pairs. It is the purpose of this semantics to model a call as a single *big* step. Exceptions are propagated from callee to caller by the rules **CallEscape** and **CallCatch**. If there is no handler, the exception continues to escape (rule **CallEscape**); if there a handler, the excep-

tion is caught (rule **CallCatch**). The last three rules are closure rules. Rule **Refl** and **Trans** state that the transition relation is reflexive and transitive. The last rule state a weak form of transitivity which applies to exceptions.

Using the big-step instrumented semantics, we define for a CFG G, the set $Acc(G)$ of accessible nodes and permissions from the initial node n_0 as follows:

$$\frac{}{(n_0, p_{init}) \in Acc(G)} \qquad \frac{(n,\rho) \in Acc(G) \quad n \overset{CG}{\to} m}{(m,\rho) \in Acc(G)} \qquad \frac{(n,\rho) \in Acc(G) \quad n, \rho \triangleright n', \rho'}{(n',\rho') \in Acc(G)}$$

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 $\texttt{call}^i$ node, a `return` node must be reached (sixth inference rule of Figure 6).

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 if 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. *Given a graph G and a solution (P, R) to the constraints generated by G, if the functions $(R_n)_{n \in NO}$ are monotone, then*

$$\forall n, (\forall c, KD(n) = consume(c) \Rightarrow \neg Error(P_n)) \Rightarrow \forall (n,\rho) \in Acc(G), \neg Error(\rho)$$

Proof. We need three intermediary results:

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

$$\begin{array}{l} \forall n\ n', \forall \rho\ \rho', n, \rho \triangleright n', \rho' \Rightarrow \forall e,\ R^e_n(\rho) \sqsubseteq_p R^e_{n'}(\rho') \\ \wedge \\ \forall n\ n', \forall \rho\ \rho', \forall ex,\ n, \rho \overset{ex}{\triangleright} n', \rho' \Rightarrow R^{ex}_n(\rho) \sqsubseteq_p \rho' \end{array}$$

  This proof is done by mutual induction over the definition of $\triangleright$ and $\overset{ex}{\triangleright}$. For example, in the case of a method call we have as hypothesis that $n \overset{CG}{\to} m, n \overset{TG}{\to} n'$, $1 \leq k \leq i$ and that for all $j = 1, \ldots, k$, $m, \rho_j \triangleright r, \rho_{j-1}$ and we will have to prove $R^e_n(\rho_k) \sqsubseteq_p R^e_{n'}(\rho_0)$. By the induction hypothesis used on the transitions $m, \rho_j \triangleright r, \rho_{j-1}$ we get that $R^e_m(\rho_j) \sqsubseteq_p R^e_r(\rho_{j-1})$ for $j = 1, \ldots, k$. Using the constraint $R^e_r \sqsubseteq \lambda\rho.\rho$ on returns this can be simplified to $R^e_m(\rho_j) \sqsubseteq_p \rho_{j-1}$ and hence, by monotony of R^e_m, $(R^e_m)^k(\rho_k) \sqsubseteq_p \rho_0$. From the constraint on method calls we have that $R^e_n \sqsubseteq (\bigsqcap_{1 \leq j \leq i}(R^\epsilon_m)^j); R^e_{n'} \sqsubseteq (R^\epsilon_m)^k; R^e_{n'}$ and then we conclude that $R^e_n(\rho_k) \sqsubseteq_p R^e_{n'}(\rho_0)$.

- Then, we have to show a correctness result on the definition of P, stated as:

$$\forall n\ n', \forall \rho\ \rho',\ n, \rho \triangleright n', \rho' \wedge P_n \sqsubseteq_p \rho \Rightarrow P_{n'} \sqsubseteq_p \rho'$$

  We prove this result by induction over $\triangleright$. For the same example of a method call as before, and with the same hypotheses as above, we will have to prove $P_{n'} \sqsubseteq_p \rho_0$ under the hypothesis $P_n \sqsubseteq_p \rho_k$. From the P constraints, we have that $P_{n'} \sqsubseteq_p \bigsqcap_{1 \leq j \leq i}(R^\epsilon_m)^j(P_n) \sqsubseteq_p (R^\epsilon_m)^k(P_n) \sqsubseteq_p (R^\epsilon_m)^k(\rho_k)$. From the result above and R constraints on return, we know that for all $j = 1, \ldots, k$, we have $R^\epsilon_m(\rho_j) \sqsubseteq_p \rho_{j-1}$ so therefore also $(R^\epsilon_m)^k(\rho_k) \sqsubseteq_p \rho_0$. We can hence deduce that $P_{n'} \sqsubseteq_p \rho_0$ and this concludes the case for method call.

- Then, we have to relate the notion of accessibility and the definition of P_n:

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

  We prove this result by induction over Acc. The two first cases directly match with the corresponding rule on P_n and the third case corresponds to the previous intermediary result.

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

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 graph G:*

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

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

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

where $\twoheadrightarrow^*$ is the reflexive-transitive closure of $\twoheadrightarrow$. 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. The structure of the proof is standard [5, 18]. Given this, 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.

7 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 5) are functions from $Perm$ to $Perm$ that have infinite domains. It makes the current constraint system difficult to solve with standard, iterative techniques [1]. 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 4) are solved by standard fixpoint iteration.

The rest of this section is devoted to the resolution of the R-constraints. The resolution technique consists in applying solution-preserving transformations to the constraints until they can be solved either symbolically or iteratively.

7.1 On simplifying R-constraints

In our model, resources are partitioned depending on their resource type. At the semantic level, *grant* and *consume* operations ensure that permissions of different types do not interfere *i.e.*, that it is impossible to use a resource of a given type with a permission of a different type. We exploit this property to derive from the original system of constraints a family of independent *ResType*-indexed constraint systems. A system modelling a given resource type, say rt, is a copy of the original system except that *grant* and *consume* are indexed by rt and are specialized accordingly:

$$grant_{rt}(p'_{rt'}, m') = \begin{cases} \lambda(p,m).(p',m') & \text{if } rt = rt' \\ \lambda(p,m).(p,m) & \text{otherwise} \end{cases}$$

$$consume_{rt}(p'_{rt'}) = \begin{cases} \lambda(p,m).(\text{if } p' \sqsubseteq_{rt'} p \text{ then } p \text{ else } \bot, m-1) & \text{if } rt = rt' \\ \lambda(p,m).(p,m) & \text{otherwise} \end{cases}$$

Further inspection of these operators shows that multiplicities and atomic permissions also behave in an independent manner. As a result, each *ResType* indexed system can be split into a pair of systems: one modelling the evolution of atomic permissions; the other modelling the evolution of multiplicities. Hence, solving the R-constraints 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.

7.2 Constraints on multiplicity transformers

Before presenting our encoding of multiplicity transformers, we identify the structure of the constraints we have to solve. Multiplicity constraints are terms of the form $x \dot{\leq} e$ where $x : Mul \to Mul$ is a variable over multiplicity transformers, $\dot{\leq}$ is the point-wise ordering of multiplicity transformers induced by $\leq$ and e is an expression built over the terms

$$e ::= v \mid grant_{Mul}(m) \mid consume_{Mul}(m) \mid id \mid e \sqcap e \mid e; e$$

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 $\lambda x.x$;
- $f \sqcap g$ is the lower bound;
- 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. Subtraction is lifted to multiplicities. It generalises the decrementing of multiplicities presented in Section 3 and is defined as follows.

$$\begin{array}{lcll} \infty - x & = & \infty & \\ x - \infty & = & \bot_{Mul} & \text{if } x \neq \infty \\ x - \bot_{Mul} & = & \infty & \\ \bot_{Mul} - n & = & \bot_{Mul} & \text{if } n \in \mathbb{N} \\ n - n' & = & n - n' & \text{if } n' \leq n \text{ and } n, n' \in \mathbb{N} \\ n - n' & = & \bot_{Mul} & \text{if } n' > n \text{ and } n, n' \in \mathbb{N} \end{array}$$

Elements of *MulF* encode constant functions, decrementing functions and are closed under function composition as shown by the following equalities:

$$\begin{array}{rcl} grant_{Mul}(m) & = & \lambda x.\min(m, x - \bot_{Mul}) \\ consume_{Mul}(m) & = & \lambda x.\min(\infty, x - m) \\ \lambda x.\min(c, x-d) \sqcap \lambda x.\min(c', x-d') & = & \lambda x.\min(\min(c, c'), x - \max(d, d')) \\ \lambda x.\min(c, x-d); \lambda x.\min(c', x-d') & = & \lambda x.\min(\min(c - d', c'), x - (d' + d)) \end{array}$$

We represent a function $\lambda x.\min(c, x-d) \in MulF$ by the pair (c,d) of multiplicities. Constraint solving over *MulF* can therefore be recast into constraint solving over the domain $MulF^\sharp = Mul \times Mul$ equipped with the interpretation $[\![(c,d)]\!] \stackrel{\triangle}{=} \lambda x.\min(c, x-d)$ and the ordering $\sqsubseteq^\sharp$ defined as $(c,d) \sqsubseteq^\sharp (c',d') \stackrel{\triangle}{=} c \leq c' \wedge d' \leq d$.

7.3 Solving multiplicity constraints

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

Example 2. *$C = \{Y \sqsubseteq^\sharp (c,d), Y' \sqsubseteq^\sharp X, X \sqsubseteq^\sharp Y ;^\sharp Y'\}$ is transformed into $C' = C_1 \cup C_2$ with $C_1 = \{Y_1 \leq c, Y_1' \leq X_1, X_1 \leq \min(Y_1 - Y_2', Y_1')\}$ and $C_2 = \{Y_2 \geq d, Y_2' \geq X_2, X_2 \geq Y_2' + Y_2\}$.*

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 $(Mul, \leq, max, \bot_{Mul})$. 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, the resolution consists in applying symbolic transformations of the equations that preserve the least solution. The purpose of the following Proposition 1 is to break the remaining direct, recursive dependencies between equations.

Proposition 1. *Let e_1 and e_2 be arbitrary expressions and x be a variable. The least solution of the equation $x = max(x + e_1, e_2)$ is given by*

$$x = max(e_2 + \infty \times e_1, e_2)$$

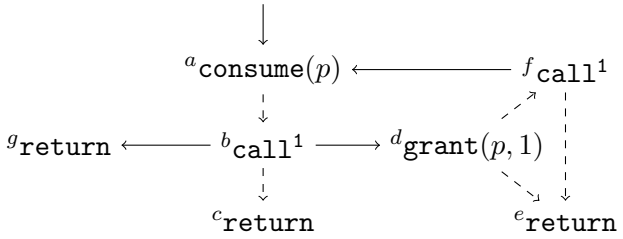


Figure 7: A CFG graph

It should be noted that the use of Proposition 1 might compromise termination if x occurs in e_2, in which case e_2 gets duplicated and hence also the number of occurrences of x. For this more complicated case, we refer to the more complex techniques proposed by Su and Wagner [22] and Leroux and Sutre [14] that mix iteration and symbolic unfolding.

Given a solution for C_2, the solution of C_1 can be computed by standard fixpoint iteration as the domain $(\mathit{Mul}, \leq, \min, \infty)$ does not have infinite descending chains. This provides multiplicity transformer solutions of the R-constraints.

Example 3. *Consider the CFG graph of Figure 7. Dotted arrows represent intra-procedural ($\overset{TG}{\rightarrow}$) edges and bold arrows represent inter-procedural ($\overset{CG}{\rightarrow}$) edges. Nodes are identified by a lower case letter. The R constraints, obtained according to Figure 5, are:*

$$
\begin{array}{ll}
R_a \sqsubseteq \mathit{consume}(p); R_b & \\
R_b \sqsubseteq R_d; R_c & R_b \sqsubseteq R_g; R_c \\
R_c \sqsubseteq \lambda x.x & \\
R_d \sqsubseteq \mathit{grant}(p,1); R_e & R_d \sqsubseteq \mathit{grant}(p,1); R_f \\
R_e \sqsubseteq \lambda x.x & \\
R_f \sqsubseteq R_a; R_e & \\
R_g \sqsubseteq \lambda x.x &
\end{array}
$$

These constraints give rise to an equivalent system of equations (see Nielson et al. *[17] for a formal justification) obtained by taking the lower bound of expressions constraining the same variable. Here, the lower bound is the point-wise* min *function that we shall write $\sqcap$. The fixpoint equations are therefore as follows:*

$$
\begin{array}{lcl}
R_a & = & \mathit{consume}(p); R_b \\
R_b & = & (R_d; R_c) \sqcap (R_g; R_c) \\
R_c & = & \lambda x.x \\
R_d & = & (\mathit{grant}(p,1); R_e) \sqcap (\mathit{grant}(p,1); R_f) \\
R_e & = & \lambda x.x \\
R_f & = & R_a; R_e \\
R_g & = & \lambda x.x
\end{array}
$$

After unfolding, we obtain a recursive equation in R_d,

$$R_d = \mathit{grant}(p,1) \sqcap (\mathit{grant}(p,1); \mathit{consume}(p); (R_d \sqcap \lambda x.x))$$

which, after symbolic simplifications, becomes

$$R_d = (\lambda x.1) \sqcap ((\lambda x.0); (R_d \sqcap \lambda x.x))$$

This latter equation can be rewritten as $R_d(x) = \min(1, \min(R_d(0), 0)) = \min(0, R_d(0))$ *In this particular case, the* greatest *solution can be computed by standard fixpoint iteration over the MulF domain. The iteration starts from the greatest function* $R_d^0 = \lambda x.\infty$. *We then obtain* $R_d^1 = \lambda x.0$ *because* $R_d^1(x) = \min(0, R_d^0(0)) = \min(0, \infty) = 0$. *As we have* $R_d^2(x) = \min(0, \min(R_d^1(0)) = 0 = R_d^1(x)$, *this is actually the greatest solution.*

Given R_d, *the other transfer functions can be easily deduced.*

$$\begin{array}{lcl} R_a = R_f & = & \lambda x.\, \min(0, x-1) \\ R_b & = & \lambda x.\, \min(0, x) \\ R_c = R_e = R_g & = & \lambda x.x \\ R_d & = & \lambda x.0 \end{array}$$

The transfer function R_a *which under-approximates the permissions at the method exit (*i.e., *node c) is such that* $R_a(n) = 0$ *for all* $n \geq 1$. *As a result, this piece of program is only permission-safe if it is called with at least one permission of type p. Moreover, in the worst case, all these permissions will be either consumed or overridden by a grant and the number of permission at node c will be zero. Note that this result is tight, as shown by the execution trace* $a{:}s$, $b{:}s$, $d{:}b{:}s$, $f{:}b{:}s$, $a{:}f{:}b{:}s$, $b{:}f{:}b{:}s$, $g{:}b{:}f{:}b{:}s$, $c{:}f{:}b{:}s$, $e{:}b{:}s$, $c{:}s$ *which reaches the return node labeled c with zero permissions left.*

8 Related work

To the best of our knowledge, this article presents the first formal model of the Java MIDP access control mechanism. A number of articles deal with access control in Java and C♯ 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 [7, 13] proposed a semantic model for stack inspection but was otherwise mostly concerned with proving behavioural properties of programs that use these mechanisms. They differ from the present work because the access control check in MIDP does not involve a stack walk. In addition, the multiplicative aspect of permissions is absent in the work stack inspection, meaning that the core analysis is over a different kind of domain. The closest of our own work is perhaps the technique for computing interfaces for stack inspection [5], in which we compute the set of permissions required for a method to execute without raising a security exception due to a failed stack inspection.

Closer in aim with the present work is that of Pottier *et al.* [19] on verifying that stack inspecting programs do not raise security exceptions because of missing permissions, but the program model (the lambda calculus) is different and the abstract domain is sets of permissions. Bartoletti *et al.* [3] also aim at proving that stack inspecting applets will not cause security exceptions and propose 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 [12] 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.* [4] 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 [11] 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 of only having one kind of permission for each resource type has a counterpart in their restriction to at most one capability for each channel. Like in our work, there is a proof of a "subject reduction" result stating that well-typed programs do not go wrong.

Our analysis is closely related to the work by Chander *et al.* [8] on combining static analysis with dynamic (run-time) checks for enforcing bounds on the resource consumption. The setting and the basic safety property considered in *loc. cit.* is similar to ours: both are concerned with programming the dynamic allocation and consumption of resources by inserting specific operations that allocate sufficient resources to accomplish a computation and both ensure statically that a program always acquires resources before consuming them.

The model of resources considered by Chander *et al.* is slightly simpler than ours as resources are just identified by name. In our setting this would correspond to having a resource type *e.g.*, Memory with one action 'allocate' that can be called a number of times according to how much memory is granted. However, nothing prevents an extension to the resource structure considered here. In addition, their resource allocation accumulates resources whereas we have opted for a non-accumulating allocation of permissions. For traditional resources such as memory, the accumulative allocation is natural. For permissions related to access control, we have argued that the non-accumulative allocation of permissions adheres more closely to the principle of giving the minimum amount of permissions needed for accomplishing a task.

The main technical difference between the two works lies in the expressiveness and the degree of automation of the underlying program analyses. The approach of Chander *et al.* relies on the programmer to provide loop invariants and pre- and post-conditions for methods in order to link program variables to the amount of resources available. These relational invariants allow powerful transformations such as hoisting resource allocations out of loops with variable bounds but in the current approach they must be provided by an external prover. In contrast, we have defined an analysis which is tailored to the type of applets that are downloaded on mobile devices but which can only handle restricted forms of loops. The benefits of this restriction is that the underlying resource analysis is inter-procedural and hence infers all necessary invariants and pre-/post-conditions fully automatically.

9 Conclusions

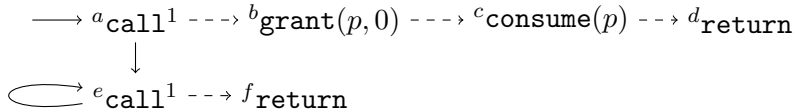
We have proposed an access control model for programs which dynamically acquire permissions to access resources. The model extends the current access control model of the Java MIDP profile for mobile telephones by introducing multiplicities of permissions together with explicit instructions for granting 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, e.g., the number of times a permission-consuming loop is executed. Allowing a `grant` to take a variable as multiplicity parameter combined with a relational analysis based on polyhedra would allow to verify a program as the following.

```
grant(sendSMS(*),addr_book.length) ;
// 0 ≤ addr_book.length = #SMSpermissions
for (i = 0 , i <  addr_book.length, i++)
// 0 < addr_book.length − i ≤ #SMSpermissions
     if (*) consume(SMS(addr_book[i].no),send);
// 0  ≤  #SMSpermissions
```

Here, the number of requested permissions depends on the size of the address book data structure, and the verification needs to establish a relation between several program entities in order to prove that the number of permissions is always non-negative.

The permission analysis presented here is not complete in the sense that there are certain graphs whose traces are all safe but that are not deemed secure by the analysis. For instance, consider the following graph:



The analysis computes that at node c there are no permission of type p. As a consequence, it concludes that the following `consume` operation might be responsible for a security violation. Actually, because the call from node a to node e never returns, the `consume` of node c never gets executed and no security violation is ever raised. In general, inaccessible nodes in the CFG may be flagged by the analysis as responsible for security violations despite the fact that they will never be executed. The analysis can be strengthened by first computing reachable nodes and only launch the permission analysis on these. For reachable nodes, we conjecture

that the analysis is computing tight bounds for permission multiplicities *i.e,* for each node n, there is a trace of graph reaching n such that the permission multiplicities are those computed by the analysis. If this conjecture holds, as reachability can be computed exactly for our model of programs, the permission analysis would then be complete. Completeness is useful to understand and eliminate certain potential sources of false alarms when our analysis is combined with other analyses. Indeed, only the CFG construction can be blamed for a program being wrongly flagged as violating the permission policy. To eventually prove program safety, the CFG construction would have to be refined in order to rule out the false alarms.

This work is intended for serving as the basis for a Proof Carrying Code (PCC) [16] architecture aiming at ensuring that a program will not use more resources than what have been declared. In the context of mobile devices, where inappropriate use of 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.

As pointed out in Section 4, the CFGs on which the verification is done can be constructed automatically using well-known control flow analyses. As a consequence, the entire verification can be made fully automatic. However, in order to make the current work practical in the MIDP platform, there are a several issues to be solved, including a number of static analyses such as analysis of strings and aliases. The major challenge specific to this application domain is that this has to be integrated into a larger model of interactive midlets that use object-oriented graphical user interfaces to present security screens to the user. In more technical terms, the CFGs here should be grafted into a larger structure that describes where control flows depending on which screen is presented to the user and which (virtual) button the user presses. Such structures are sometimes called navigation graphs and automatic analyses for constructing such graphs are only starting to emerge [9].

References

[1] Alexander Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2):79–111, 1999.

[2] Bandera. `http://bandera.projects.cis.ksu.edu`.

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

[4] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. History-based access control with local policies. In *Proceedings of the 8th International Conference on Foundations of Software Science and Computational Structures (FOSSACS 2005)*, volume 3441 of *Lecture Notes in Computer Science*, pages 316–332. Springer-Verlag, 2005.

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

[6] Frédéric Besson, Guillaume Dufay, and Thomas Jensen. A formal model of access control for mobile interactive devices. In *Proceedings of the 11th European Symposium On Research In Computer Security (ESORICS'06)*, volume 4189 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.

[7] 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.

[8] 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.

[9] Pierre Crégut. Extracting control from data: user interfaces of MIDP applications. In *Trustworthy Global Computing 2007 (TGC'07)*, volume 4912 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

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

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

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

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

[14] Jérôme Leroux and Grégoire Sutre. Accelerated data-flow analysis. In *Proceedings of the 14th International Symposium on Static Analysis (SAS'2007)*, volume 4634 of *Lecture Notes in Computer Science*, pages 184–199. Springer-Verlag, 2007.

[15] Jon K. Millen. A resource allocation model for denial of service. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 137–147. IEEE Computer Society Press, 1992.

[16] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119. ACM Press, 1997.

[17] Flemming. Nielson, Hanne Riis. Nielson, and Chris. Hankin. *Principles of Program Analysis*. Springer, 1999.

[18] D. Pichardie. Bicolano – Byte Code Language in Coq. `http://mobius.inria.fr/bicolano`., 2006.

[19] 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.

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

[21] Soot: a Java Optimization Framework. `http://www.sable.mcgill.ca/soot/`.

[22] Zhendong Su and David Wagner. A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computer Science*, 345(1):122–138, 2005.

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