A Formal Model and Correctness Proof for an Access Control Policy Framework

Chunhan Wu[1,2], Xingyuan Zhang[1], and Christian Urban[2]

[1] PLA University of Science and Technology, China
[2] King's College London, UK

Abstract. If an access control policy promises that a resource is protected in a system, how do we know it is really protected? To give an answer we formalise in this paper the Role-Compatibility Model—a framework, introduced by Ott, in which access control policies can be expressed. We also give a dynamic model determining which security related events can happen while a system is running. We prove that if a policy in this framework ensures a resource is protected, then there is really no sequence of events that would compromise the security of this resource. We also prove the opposite: if a policy does not prevent a security compromise of a resource, then there is a sequence of events that will compromise it. Consequently, a static policy check is sufficient (sound and complete) in order to guarantee or expose the security of resources before running the system. Our formal model and correctness proof are mechanised in the Isabelle/HOL theorem prover using Paulson's inductive method for reasoning about valid sequences of events. Our results apply to the Role-Compatibility Model, but can be readily adapted to other role-based access control models.

1 Introduction

Role-based access control models are used in many operating systems for enforcing security properties. The *Role-Compatibility Model* (RC-Model), introduced by Ott [5,6], is one such role-based access control model. It defines *roles*, which are associated with processes, and defines *types*, which are associated with system resources, such as files and directories. The RC-Model also includes types for interprocess communication, that is message queues, sockets and shared memory. A policy in the RC-Model gives every user a default role, and also specifies how roles can be changed. Moreover, it specifies which types of resources a role has permission to access, and also the *mode* with which the role can access the resources, for example read, write, send, receive and so on.

The RC-Model is built on top of a collection of system calls provided by the operating system, for instance system calls for reading and writing files, cloning and killing of processes, and sending and receiving messages. The purpose of the RC-Model is to restrict access to these system calls and thereby enforce security properties of the system. A problem with the RC-Model and role-based access control models in general is that a system administrator has to specify an appropriate access control policy. The difficulty with this is that *"what you specify is what you get but not necessarily what you want"* [4, Page 242]. To overcome this difficulty, a system administrator needs some kind of sanity check for whether an access control policy is really securing resources. Existing

works, for example [9,10], provide sanity checks for policies by specifying properties and using model checking techniques to ensure a policy at hand satisfies these properties. However, these checks only address the problem on the level of policies—they can only check "on the surface" whether the policy reflects the intentions of the system administrator—these checks are not justified by the actual behaviour of the operating system. The main problem this paper addresses is to check when a policy matches the intentions of a system administrator *and* given such a policy, the operating system actually enforces this policy.

Our work is related to the preliminary work by Archer et al [1] about the security model of SELinux. They also give a dynamic model of system calls on which the access controls are implemented. Their dynamic model is defined in terms of IO automata and mechanised in the PVS theorem prover. For specifying and reasoning about automata they use the TAME tool in PVS. Their work checks well-formedness properties of access policies by type-checking generated definitions in PVS. They can also ensure some "*simple properties*" (their terminology), for example whether a process with a particular PID is present in every reachable state from an initial state. They also consider "*deeper properties*", for example whether only a process with root-permissions or one of its descendents ever gets permission to write to kernel log files. They write that they can state such deeper properties about access policies, but about checking such properties they write that "*the feasibility of doing so is currently an open question*" [1, Page 167]. We improve upon their results by using our sound and complete static policy check to make this feasible.

The work we report is also closely related to the work on *grsecurity*, an access control system developed as a patch on top of Linux kernel [2]. It installs a reference monitor to restrict access to system resources. They model a dynamic semantics of the operating system with four rules dealing with executing a file, setting a role and setting an UID as well as GID. These rules are parametrerised by an arbitrary but fixed access policy. Although, there are only four rules, their state-space is in general infinite, like in our work. They therfore give an abstracted semantics, which gives them a finite state-space. For example the abstracted semantics dispenses with users and roles by introducing abstract users and abstract roles. They obtain a soundness result for their abstract semantics and under some weak assumptions also a completeness result. Comparing this to our work, we will have a much more fine-grained model of the underlying operating system. We will also obtain a soundness result, but more importantly obtain also a completeness result. But since we have a much more fine-grained model, it will depend on some stronger assumptions. The abstract semantics in [2] is used for model-checking policies according to whether, for example, information flow properties are ensured. Since their formalism consists of only a few rules, they can get away with "pencil-and-paper proofs", whereas reasoning about our more detailed model containing substantially more rules really necessitates the support of a theorem prover and completely formalised models.

Our formal models and correctness proofs are mechanised in the interactive theorem prover Isabelle/HOL. The mechanisation of the models is a prerequisite for any correctness proof about the RC-Model, since it includes a large number of interdependent concepts and very complex operations that determine roles and types. In our opinion it is

futile to attempt to reason about them by just using "pencil-and-paper". Following good experience in earlier mechanisation work [11], we use Paulson's inductive method for reasoning about sequences of events [8]. For example we model system calls as events and reason about an inductive definition of valid traces, that is lists of events. Central to this paper is a notion of a resource being *tainted*, which for example means it contains a virus or a back door. We use our model of system calls in order to characterise how such a tainted object can "spread" through the system. For a system administrator the important question is whether such a tainted file, possibly introduced by a user, can affect core system files and render the whole system insecure, or whether it can be contained by the access policy. Our results show that a corresponding check can be performed statically by analysing the initial state of the system and the access policy.

Contributions: We give a complete formalisation of the RC-Model in the interactive theorem prover Isabelle/HOL. We also give a dynamic model of the operating system by formalising all security related events that can happen while the system is running. As far as we are aware, we are the first ones who formally prove that if a policy in the RC-Model satisfies an access property, then there is no sequence of events (system calls) that can violate this access property. We also prove the opposite: if a policy does not meet an access property, then there is a sequence of events that will violate this property in our model of the operating system. With these two results in place we can show that a static policy check is sufficient in order to guarantee the access properties before running the system. Again as far as we know, no such check has been designed and proved correct before.

2 Preliminaries about the RC-Model

The Role-Compatibility Model (RC-Model) is a role-based access control model. It has been introduced by Ott [5] and is used in running systems for example to secure Apache servers. It provides a more fine-grained control over access permissions than simple Unix-style access control models. This more fine-grained control solves the problem of server processes running as root with too many access permissions in order to accomplish a task at hand. In the RC-Model, system administrators are able to restrict what the role of server is allowed to do and in doing so reduce the attack surface of a system.

Policies in the RC-Model talk about *users*, *roles*, *types* and *objects*. Objects are processes, files or IPCs (interprocess communication objects—such as message queues, sockets and shared memory). Objects are the resources of a system an RC-policy can restrict access to. In what follows we use the letter u to stand for users, r for roles, p for processes, f for files and i for IPCs. We also use obj as a generic variable for objects. The RC-Model has the following eight kinds of access modes to objects:

Read, Write, Execute, ChangeOwner, Create, Send, Receive and Delete

In the RC-Model, roles group users according to tasks they need to accomplish. Users have a default role specified by the policy, which is the role they start with whenever they log into the system. A process contains the information about its owner (a

user), its role and its type, whereby a type in the RC-Model allows system administrators to group resources according to a common criteria. Such detailed information is needed in the RC-Model, for example, in order to allow a process to change its ownership. For this the RC-Model checks the role of the process and its type: if the access control policy states that the role has *ChangeOwner* access mode for processes of that type, then the process is permitted to assume a new owner.

Files in the RC-Model contain the information about their types. A policy then specifies whether a process with a given role can access a file under a certain access mode. Files, however, also include in the RC-Model information about roles. This information is used when a process is permitted to execute a file. By doing so it might change its role. This is often used in the context of web-servers when a cgi-script is uploaded and then executed by the server. The resulting process should have much more restricted access permissions. This kind of behaviour when executing a file can be specified in an RC-policy in several ways: first, the role of the process does not change when executing a file; second, the process takes on the role specified with the file; or third, use the role of the owner, who currently owns this process. The RC-Model also makes assumptions on how types can change. For example for files and IPCs the type can never change once they are created. But processes can change their types according to the roles they have.

As can be seen, the information contained in a policy in the RC-Model can be rather complex: Roles and types, for example, are policy-dependent, meaning each policy needs to define a set of roles and a set of types. Apart from recording for each role the information which type of resource it can access and under which access-mode, it also needs to include a role compatibility set. This set specifies how one role can change into another role. Moreover it needs to include default information for cases when new processes or files are created. For example, when a process clones itself, the type of the new process is determined as follows: the policy might specify a default type whenever a process with a certain role is cloned, or the policy might specify that the cloned process inherits the type of the parent process.

Ott implemented the RC-Model on top of Linux, but only specified it as a set of informal rules, partially given as logic formulas, partially given as rules in "English". Unfortunately, some presentations about the RC-Model give conflicting definitions for some concepts—for example when defining the semantics of the special role "inherit parent". In [5] it means inherit the initial role of the parent directory, but in [7] it means inherit the role of the parent process. In our formalisation we mainly follow the version given in [5]. In the next section we give a mechanised model of the system calls on which the RC-Model is implemented.

3 Dynamic Model of System Calls

Central to the RC-Model are processes, since they initiate any action involving resources and access control. We use natural numbers to stand for process IDs, but do not model the fact that the number of processes in any practical system is limited. Similarly, IPCs and users are represented by natural numbers. The thirteen actions a process can perform are represented by the following datatype of *events*

$$\begin{array}{lllll} event ::= & CreateFile\ p\ f & \mid ReadFile\ p\ f & \mid Send\ p\ i & \mid Kill\ p\ p' \\ \mid & WriteFile\ p\ f & \mid Execute\ p\ f & \mid Recv\ p\ i & \\ \mid & DeleteFile\ p\ f & \mid Clone\ p\ p' & \mid CreateIPC\ p\ i & \\ \mid & ChangeOwner\ p\ u & \mid ChangeRole\ p\ r & \mid DeleteIPC\ p\ i & \end{array}$$

with the idea that for example in *Clone* a process p is cloned and the new process has the ID p'; with *Kill* the intention is that the process p kills another process with ID p'. We will later give the definition what the role r can stand for in the constructor *ChangeRole* (namely *normal roles* only). As is custom in Unix, there is no difference between a directory and a file. The files f in the definition above are simply lists of strings. For example, the file $/usr/bin/make$ is represented by the list $[make, bin, usr]$ and the *root*-directory is the *Nil*-list. Following the presentation in [5], our model of IPCs is rather simple-minded: we only have events for creation and deletion of IPCs, as well as sending and receiving messages.

Events essentially transform one state of the system into another. The system starts with an initial state determining which processes, files and IPCs are active at the start of the system. We assume the users of the system are fixed in the initial state; we also assume that the policy does not change while the system is running. We have three sets, namely *init_procs*, *init_files* and *init_ipcs* specifying the processes, files and IPCs present in the initial state. We will often use the abbreviation

$$obj \in init \stackrel{def}{=} obj \in init\_files \vee obj \in init\_procs \vee obj \in init\_ipcs$$

There are some assumptions we make about the files present in the initial state: we always require that the *root*-directory $[]$ is part of the initial state and for every file in the initial state (excluding $[]$) we require that its parent is also part of the initial state. A state is determined by a list of events, called the *trace*. The empty trace, or empty list, stands for the initial state. Given a trace s, we prepend an event to s to stand for the state in which the event just happened. We need to define functions that allow us to make some observations about traces. One such function is called *current_procs* and calculates the set of "alive" processes in a state:

$$\begin{array}{lcl} current\_procs\ [] & \stackrel{def}{=} & init\_procs \\ current\_procs\ (Clone\ p\ p'::s) & \stackrel{def}{=} & \{p'\} \cup current\_procs\ s \\ current\_procs\ (Kill\ p\ p'::s) & \stackrel{def}{=} & current\_procs\ s - \{p'\} \\ current\_procs\ (\_::s) & \stackrel{def}{=} & current\_procs\ s \end{array}$$

The first clause states that in the empty trace the processes are given by *init_processes*. The events for cloning a process, respectively killing a process, update this set of processes appropriately. Otherwise the set of live processes is unchanged. We have similar functions for alive files and IPCs, called *current_files* and *current_ipcs*.

We can use these functions in order to formally model which events are *admissible* by the operating system in each state. We show just three rules that give the gist of this definition. First the rule for changing an owner of a process:

$$\frac{p \in current\_procs\ s \quad u \in init\_users}{admissible\ s\ (ChangeOwner\ p\ u)}$$

We require that the process p is alive in the state s (first premise) and that the new owner is a user that existed in the initial state (second premise). Next the rule for creating a new file:

$$\frac{p \in \mathit{current\_procs}\ s \quad f \notin \mathit{current\_files}\ s \quad \mathit{is\_parent}\ f\ \mathit{pf} \quad \mathit{pf} \in \mathit{current\_files}\ s}{\mathit{admissible}\ s\ (\mathit{CreateFile}\ p\ f)}$$

It states that a file f can be created by a process p being alive in the state s, the new file does not exist already in this state and there exists a parent file pf for the new file. The parent file is just the tail of the list representing f. Finally, the rule for cloning a process:

$$\frac{p \in \mathit{current\_procs}\ s \quad p' \notin \mathit{current\_procs}\ s}{\mathit{admissible}\ s\ (\mathit{Clone}\ p\ p')}$$

Clearly the operating system should only allow to clone a process p if the process is currently alive. The cloned process will get the process ID generated by the operating system, but this process ID should not already exist. The admissibility rules for the other events impose similar conditions.

However, the admissibility check by the operating system is only one "side" of the constraints the RC-Model imposes. We also need to model the constraints of the access policy. For this we introduce separate *granted*-rules involving the sets *permissions* and *compatible* r: the former contains triples describing access control rules; the latter specifies for each role r which roles are compatible with r. These sets are used in the RC-Model when a process having a role r takes on a new role r'. For example, a login-process might belong to root; once the user logs in, however, the role of the process should change to the user's default role. The corresponding *granted*-rule is as follows

$$\frac{\mathit{is\_current\_role}\ s\ p\ r \quad r' \in \mathit{compatible}\ r}{\mathit{granted}\ s\ (\mathit{ChangeRole}\ p\ r')}$$

where we check whether the process p has currently role r and whether the RC-policy states that r' is in the role compatibility set of r.

The complication in the RC-Model arises from the way the current role of a process in a state s is calculated—represented by the predicate *is_current_role* in our formalisation. For defining this predicate we need to trace the role of a process from the initial state to the current state. In the initial state all processes have the role given by the function *init_current_role*. If a *Clone* event happens then the new process will inherit the role from the parent process. Similarly, if a *ChangeRole* event happens, then as seen in the rule above we just change the role accordingly. More interesting is an *Execute* event in the RC-Model. For this event we have to check the role attached to the file to be executed. There are a number of cases: If the role of the file is a *normal* role, then the process will just take on this role when executing the file (this is like the setuid mechanism in Unix). But there are also four *special* roles in the RC-Model: *InheritProcessRole*, *InheritUserRole*, *InheritParentRole* and *InheritUpMixed*. For example, if a file to be executed has *InheritProcessRole* attached to it, then the process that executes this file keeps its role regardless of the information attached to the file. In this way programs can be can quarantined; *InheritUserRole* can be used for login shells to make

sure they run with the user's default role. The purpose of the other special roles is to determine the role of a process according to the directory in which the files are stored.

Having the notion of current role in place, we can define the granted rule for the *Execute*-event: Suppose a process p wants to execute a file f. The RC-Model first fetches the role r of this process (in the current state s) and the type t of the file. It then checks if the tuple $(r,\ t,\ Execute)$ is part of the policy, that is in our formalisation being an element in the set *permissions*. The corresponding rule is as follows

$$\frac{is\_current\_role\ s\ p\ r \quad is\_file\_type\ s\ f\ t \quad (r,\ t, Execute) \in permissions}{granted\ s\ (Execute\ p\ f)}$$

The next *granted*-rule concerns the *CreateFile* event. If this event occurs, then we have two rules in our RC-Model depending on how the type of the created file is derived. If the type is inherited from the parent directory *pf*, then the *granted*-rule is as follows:

$$\frac{\begin{array}{l} is\_parent\ f\ pf \quad is\_file\_type\ s\ pf\ t \quad is\_current\_role\ s\ p\ r \\ default\_type\ r = InheritPatentType \quad (r,\ t, Write) \in permissions \end{array}}{granted\ s\ (CreateFile\ p\ f)}$$

We check whether *pf* is the parent file (directory) of f and check whether the type of *pf* is t. We also need to fetch the role r of the process that seeks to get permission for creating the file. If the default type of this role r states that the type of the newly created file will be inherited from the parent file type, then we only need to check that the policy states that r has permission to write into the directory *pf*.

The situation is different if the default type of role r is some *normal* type, like text-file or executable. In such cases we want that the process creates some predetermined type of files. Therefore in the rule we have to check whether the role is allowed to create a file of that type, and also check whether the role is allowed to write any new file into the parent file (directory). The corresponding rule is as follows.

$$\frac{\begin{array}{l} is\_parent\ f\ pf \\ is\_file\_type\ s\ pf\ t \quad is\_current\_role\ s\ p\ r \quad default\_type\ r = NormalFileType\ t' \\ (r,\ t, Write) \in permissions \quad (r,\ t', Create) \in permissions \end{array}}{granted\ s\ (CreateFile\ p\ f)}$$

Interestingly, the type-information in the RC-model is also used for processes, for example when they need to change their owner. For this we have the rule

$$\frac{is\_current\_role\ s\ p\ r \quad is\_process\_type\ s\ p\ t \quad (r,\ t, ChangeOwner) \in permissions}{granted\ s\ (ChangeOwner\ p\ u)}$$

whereby we have to obtain both the role and type of the process p, and then check whether the policy allows a *ChangeOwner*-event for that role and type.

Overall we have 13 rules for the admissibility check by the operating system and 14 rules for the granted check by the RC-Model. They are used to characterise when an event e is *valid* to occur in a state s. This can be inductively defined as the set of valid states.

$$\frac{}{valid\ []} \qquad \frac{valid\ s \quad admissible\ s\ e \quad granted\ s\ e}{valid\ (e::s)}$$

4 The Tainted Relation

The novel notion we introduce in this paper is the *tainted* relation. It characterises how a system can become infected when a file in the system contains, for example, a virus. We assume that the initial state contains some tainted objects (we call them *seeds*). Therefore in the initial state [] an object is tainted, if it is an element in *seeds*.

$$\frac{obj \in seeds}{obj \in tainted\ []}$$

Let us first assume such a tainted object is a file f. If a process reads or executes a tainted file, then this process becomes tainted (in the state where the corresponding event occurs).

$$\frac{f \in tainted\ s \quad valid\ (Execute\ p\ f::s)}{p \in tainted\ (Execute\ p\ f::s)} \qquad \frac{f \in tainted\ s \quad valid\ (ReadFile\ p\ f::s)}{p \in tainted\ (ReadFile\ p\ f::s)}$$

We have a similar rule for a tainted IPC, namely

$$\frac{i \in tainted\ s \quad valid\ (Recv\ p\ i::s)}{p \in tainted\ (Recv\ p\ i::s)}$$

which means if we receive anything from a tainted IPC, then the process becomes tainted. A process is also tainted when it is a produced by a *Clone*-event.

$$\frac{p \in tainted\ s \quad valid\ (Clone\ p\ p'::s)}{p' \in tainted\ (Clone\ p\ p'::s)}$$

However, the tainting relationship must also work in the "other" direction, meaning if a process is tainted, then every file that is written or created will be tainted. This is captured by the four rules:

$$\frac{p \in tainted\ s \quad valid\ (CreateFile\ p\ f::s)}{f \in tainted\ (CreateFile\ p\ f::s)} \qquad \frac{p \in tainted\ s \quad valid\ (WriteFile\ p\ f::s)}{f \in tainted\ (WriteFile\ p\ f::s)}$$

$$\frac{p \in tainted\ s \quad valid\ (CreateIPC\ p\ i::s)}{i \in tainted\ (CreateIPC\ p\ i::s)} \qquad \frac{p \in tainted\ s \quad valid\ (Send\ p\ i::s)}{i \in tainted\ (Send\ p\ i::s)}$$

Finally, we have three rules that state whenever an object is tainted in a state s, then it will be still tainted in the next state $e::s$, provided the object is still *alive* in that state. We have such a rule for each kind of objects, for example for files the rule is:

$$\frac{f \in tainted\ s \quad valid\ (e::s) \quad f \in current\_files\ (e::s)}{f \in tainted\ (e::s)}$$

Similarly for alive processes and IPCs (then respectively with premises $p \in \mathit{current\_procs}\ (e::s)$ and $i \in \mathit{current\_ipcs}\ (e::s)$). When an object present in the initial state can be tainted in *some* state (system run), we say it is *taintable*:

$$\mathit{taintable\ obj} \stackrel{\mathit{def}}{=} \mathit{obj} \in \mathit{init} \wedge \exists\, s.\ \mathit{obj} \in \mathit{tainted}\ s$$

Before we can describe our static check deciding when a file is taintable, we need to describe the notions *deleted* and *undeletable* for objects. The former characterises whether there is an event that deletes these objects (files, processes or IPCs). For this we have the following four rules:

$$\frac{}{\mathit{deleted}\ \ p'\ (\mathit{Kill}\ p\ p'::s)} \qquad \frac{\mathit{deleted\ obj}\ s}{\mathit{deleted\ obj}\ (e::s)}$$

$$\frac{}{\mathit{deleted}\ \ f\ (\mathit{DeleteFile}\ p\ f::s)}$$

$$\frac{}{\mathit{deleted}\ \ i\ (\mathit{DeleteIPC}\ p\ i::s)}$$

Note that an object cannot be deleted in the initial state []. An object is then said to be *undeletable* provided it did exist in the initial state and there does not exists a valid state in which the object is deleted:

$$\mathit{undeletable\ obj} \stackrel{\mathit{def}}{=} \mathit{obj} \in \mathit{init} \wedge \neg\, (\exists\, s.\ \mathit{valid}\ s \wedge \mathit{deleted\ obj}\ s)$$

The point of this definition is that our static taintable check will only be complete for undeletable objects. But these are the ones system administrators are typically interested in (for example system files).

It should be clear that we cannot hope for a meaningful check by just trying out all possible valid states in our dynamic model. The reason is that there are potentially infinitely many of them and therefore the search space would be infinite. For example starting from an initial state containing a process p and a file pf, we can create files f_1, f_2, ... via *CreateFile*-events. This can be pictured roughly as follows:

Initial state:

$$\{p, pf\} \quad \Longrightarrow \quad \{p, pf, f_1::pf\} \quad \Longrightarrow \quad \{p, pf, f_1::pf, f_2::f_1::pf\}\ \ldots$$

$$\mathit{CreateFile}\ p\ (f_1::pf) \qquad \mathit{CreateFile}\ p\ (f_2::f_1::pf)$$

Instead, the idea of our static check is to use the policies of the RC-model for generating an answer, since they provide always a finite "description of the system". As we will see in the next section, this needs some care, however.

5 Our Static Check

Assume there is a tainted file in the system and suppose we face the problem of finding out whether this file can affect other files, IPCs or processes? One idea is to work on the level of policies only, and check which operations are permitted by the role

and type of this file. Then one builds the "transitive closure" of this information and checks for example whether the role *root* has become affected, in which case the whole system is compromised. This is indeed the solution investigated in [3] in the context of information flow and SELinux.

Unfortunately, restricting the calculations to only use policies is too simplistic for obtaining a check that is sound and complete—it over-approximates the dynamic tainted relation defined in the previous section. To see the problem consider the case where the tainted file has, say, the type *bin*. If the RC-policy contains a role *r* that can both read and write *bin*-files, we would conclude that all *bin*-files can potentially be tainted. That is indeed the case, *if* there is a process having this role *r* running in the system. But if there is *not*, then the tainted file cannot "spread". A similar problem arises in case there are two processes having the same role *r*, and this role is restricted to read files only. Now if one of the processes is tainted, then the simple check involving only policies would incorrectly infer that all processes involving that role are tainted. But since the policy for *r* is restricted to be read-only, there is in fact no danger that both processes can become tainted.

The main idea of our sound and complete check is to find a "middle" ground between the potentially infinite dynamic model and the too coarse information contained in the RC-policies. Our solution is to define a "static" version of the tainted relation, called $tainted^s$, that records relatively precisely the information about the initial state of the system (the one in which an object might be a *seed* and therefore tainted). However, we are less precise about the objects created in every subsequent state. The result is that we can avoid the potential infinity of the dynamic model. For the $tainted^s$-relation we will consider the following three kinds of *items* recording the information we need about processes, files and IPCs, respectively:

	Recorded information:
Processes:	$P(r, dr, t, u)^{po}$
Files:	$F(t, a)^{fo}$
IPCs:	$I(t)^{io}$

For a process we record its role *r*, its default role *dr* (used to determine the role when executing a file or changing the owner of a process), its type *t* and its owner *u*. For a file we record just the type *t* and its *anchor a* (we define this notion shortly). For IPCs we only record its type *t*. Note the superscripts *po*, *fo* and *io* in each item. They are optional arguments and depend on whether the corresponding object is present in the initial state or not. If it *is*, then for processes and IPCs we will record *Some id*, where *id* is the natural number that uniquely identifies a process or IPC; for files we just record their path *Some f*. If the object is *not* present in the initial state, that is newly created, then we just have *None* as superscript. Let us illustrate the different superscripts with the following example where the initial state contains a process *p* and a file (directory) *pf*. Then this process creates a file via a *CreateFile*-event and after that reads the created file via a *Read*-event:

Initial state:

$$\{p, pf\} \xrightarrow[\textit{CreateFile } p\ (f::pf)]{} \{p, pf, f::pf\} \xrightarrow[\textit{ReadFile } p\ (f::pf)]{} \{p, pf, f::pf\}$$

For the two objects in the initial state our static check records the information $P(r, dr, t, u)^{Some\ p}$ and $F(t', a)^{Some\ pf}$ (assuming r, t and so on are the corresponding roles, types etc). In both cases we have the superscript $Some(...)$ since they are objects present in the initial state. For the file f::pf created by the *CreateFile*-event, we record $F(t', a')^{None}$, since it is a newly created file. The *ReadFile*-event does not change the set of objects, therefore no new information needs to be recorded. The problem we are avoiding with this setup of recording the precise information for the initial state is where two processes have the same role and type information, but only one is tainted in the initial state, but the other is not. The recorded unique process ID allows us to distinguish between both processes. For all newly created objects, on the other hand, we do not care. This is crucial, because otherwise exploring all possible "reachable" objects can lead to the potential infinity like in the dynamic model.

An *anchor* for a file is the "nearest" directory that is present in the initial state and has not been deleted in a state s. Its definition is the recursive function

$$\begin{array}{lcl} anchor\ s\ [] & \stackrel{def}{=} & if\ \neg\ deleted\ []\ s\ then\ Some\ []\ else\ None \\ anchor\ s\ (f::pf) & \stackrel{def}{=} & if\ f::pf \in init\_files \wedge \neg\ deleted\ \ (f::pf)\ s \\ & & then\ Some\ (f::pf)\ else\ anchor\ s\ pf \end{array}$$

generating an optional value. The first clause states that the *root*-directory is always its own anchor unless it has been deleted. If a file is present in the initial state and not deleted in s, then it is also its own anchor, otherwise the anchor will be the anchor of the parent directory. For example if we have a directory *pf* in the initial state, then its anchor is *Some pf* (assuming it is not deleted). If we create a new file in this directory, say f::*pf*, then its anchor will also be *Some pf*. The purpose of *anchor* is to determine the role information when a file is executed, because the role of the corresponding process, according to the RC-model, is determined by the role information of the anchor of the file to be executed.

There is one last problem we have to solve before we can give the rules of our $tainted^s$-check. Suppose an RC-policy includes the rule $(r, foo, Write) \in permissions$, that is a process of role r is allowed to write files of type *foo*. If there is a tainted process with this role, we would conclude that also every file of that type can potentially become tainted. However, that is not the case if the initial state does not contain any file with type *foo* and the RC-policy does not allow the creation of such files, that is does not contain an access rule $(r, foo, Create) \in permissions$. In a sense the original $(r, foo, Write)$ is "useless" and should not contribute to the relation characterising the objects that are tainted. To exclude such "useless" access rules, we define a relation $reachable^s$ restricting our search space to only configurations that correspond to states in our dynamic model. We first have a rule for reachable items of the form $F(t, f)^{Some\ f}$ where the file f with type t is present in the initial state.

$$\frac{f \in init\_files \quad is\_file\_type\ []\ f\ t}{F(t, f)^{Some\ f} \in reachable^s}$$

We have similar reachability rules for processes and IPCs that are part of the initial state. Next is the reachability rule in case a file is created

$$\frac{\begin{array}{l} F(t, a)^{fo} \in reachable^s \\ P(r, dr, pt, u)^{po} \in reachable^s \quad default\_type\ r = NormalFileType\ t' \\ (r,\ t, Write) \in permissions \quad (r,\ t', Create) \in permissions \end{array}}{F(t', a)^{None} \in reachable^s}$$

where we require that we have a reachable parent directory, recorded as $F(t, a)^{fo}$, and also a process that can create the file, recorded as $P(r, dr, pt, u)^{po}$. As can be seen, we also require that we have both $(r, t, Write)$ and $(r, t', Create)$ in the *permissions* set for this rule to apply. If we did *not* impose this requirement about the RC-policy, then there would be no way to create a file with *NormalFileType* t' according to our "dynamic" model. However in case we want to create a file of type *InheritPatentType*, then we only need the access-rule $(r, t, Write)$:

$$\frac{\begin{array}{l} F(t, a)^{fo} \in reachable^s \quad P(r, dr, pt, u)^{po} \in reachable^s \\ default\_type\ r = InheritPatentType \quad (r,\ t, Write) \in permissions \end{array}}{F(t, a)^{None} \in reachable^s}$$

We also have reachability rules for processes executing files, and for changing their roles and owners, for example

$$\frac{P(r, dr, t, u)^{po} \in reachable^s \quad r' \in compatible\ r}{P(r', dr, t, u)^{po} \in reachable^s}$$

which states that when we have a process with role r, and the role r' is in the corresponding role-compatibility set, then also a process with role r' is reachable.

The crucial difference between between the "dynamic" notion of validity and the "static" notion of $reachable^s$ is that there can be infinitely many valid states, but assuming the initial state contains only finitely many objects, then also $reachable^s$ will be finite. To see the difference, consider the infinite "chain" of events just cloning a process p_0:

$$\begin{array}{l} \text{Initial state:} \\ \{p_0\} \quad \Longrightarrow \quad \{p_0, p_1\} \quad \Longrightarrow \quad \{p_0, p_1, p_2\} \ \ldots \\ \qquad\quad Clone\ p_0\ p_1 \qquad\quad Clone\ p_0\ p_2 \end{array}$$

The corresponding reachable objects are

$$\{P(r, dr, t, u)^{Some\ (p_0)}\} \implies \{P(r, dr, t, u)^{Some\ (p_0)}, P(r, dr, t, u)^{None}\}$$

where no further progress can be made because the information recorded about p_2, p_3 and so on is just the same as for p_1, namely $P(r, dr, t, u)^{None}$. Indeed we can prove the lemma:

Lemma 1. *If finite init, then finite* $reachable^s$.

This fact of $reachable^s$ being finite enables us to design a decidable tainted-check. For this we introduce inductive rules defining the set $tainted^s$. Like in the "dynamic" version of tainted, if an object is element of *seeds*, then it is $tainted^s$.

$$\frac{obj \in seeds}{[\![obj]\!] \in tainted^s}$$

The function $[\![\_]\!]$ extracts the static information from an object. For example for a process it extracts the role, default role, type and user; for a file the type and the anchor. If a process is tainted and creates a file with a normal type t' then also the created file is tainted. The corresponding rule is

$$\frac{\begin{array}{l} P(r, dr, pt, u)^{po} \in tainted^s \\ F(t, a)^{fo} \in reachable^s \quad default\_type\ r = NormalFileType\ t' \\ (r,\ t, Write) \in permissions \quad (r,\ t', Create) \in permissions \end{array}}{F(t', a)^{None} \in tainted^s}$$

If a tainted process creates a file that inherits the type of the directory, then the file will also be tainted:

$$\frac{\begin{array}{l} P(r, dr, pt, u)^{po} \in tainted^s \quad F(t, a)^{fo} \in reachable^s \\ default\_type\ r = InheritPatentType \quad (r,\ t, Write) \in permissions \end{array}}{F(t, a)^{None} \in tainted^s}$$

If a tainted process changes its role, then also with this changed role it will be tainted:

$$\frac{P(r, dr, t, u)^{po} \in tainted^s \quad r' \in compatible\ r}{P(r', dr, t, u)^{po} \in tainted^s}$$

Similarly when a process changes its owner. If a file is tainted, and a process has read-permission to that type of files, then the process becomes tainted. The corresponding rule is

$$\frac{F(t, a)^{fo} \in tainted^s \quad P(r, dr, pt, u)^{po} \in reachable^s \quad (r,\ t, Read) \in permissions}{P(r, dr, pt, u)^{po} \in tainted^s}$$

If a process is tainted and it has write-permission for files of type t, then these files will be tainted:

$$\frac{P(r, dr, pt, u)^{po} \in tainted^s \quad F(t, a)^{fo} \in reachable^s \quad (r,\ t, Write) \in permissions}{F(t, a)^{fo} \in tainted^s}$$

We omit the remaining rules for executing a file, cloning a process and rules involving IPCs, which are similar. A simple consequence of our definitions is that every tainted object is also reachable:

Lemma 2. $tainted^s \subseteq reachable^s$

which in turn means that the set of $tainted^s$ items is finite by Lemma 1.

Returning to our original question about whether tainted objects can spread in the system. To answer this question, we take these tainted objects as seeds and calculate the

set of items that are $tainted^s$. We proved this set is finite and can be enumerated using the rules for $tainted^s$. However, this set is about items, not about whether objects are tainted or not. Assuming an item in $tainted^s$ arises from an object present in the initial state, we have recorded enough information to translate items back into objects via the function $|\_|$:

$$\begin{array}{ll} |P(r, dr, t, u)^{po}| & \stackrel{def}{=} po \\ |F(t, a)^{fo}| & \stackrel{def}{=} fo \\ |I(t)^{io}| & \stackrel{def}{=} io \end{array}$$

Using this function, we can define when an object is $taintable^s$ in terms of an item being $tainted^s$, namely

$$taintable^s\ obj \stackrel{def}{=} \exists\, item.\ item \in tainted^s \wedge |item| = Some\ obj$$

Note that $taintable^s$ is only about objects that are present in the initial state, because for all other items $|\_|$ returns *None*.

With these definitions in place, we can state our theorem about the soundness of our static $taintable^s$-check for objects.

Theorem 1 (Soundness). *If $taintable^s$ obj then taintable obj.*

The proof of this theorem generates for every object that is "flagged" as $taintable^s$ by our check, a sequence of events which shows how the object can become tainted in the dynamic model. We can also state a completeness theorem for our $taintable^s$-check.

Theorem 2 (Completeness). *If undeletable obj and taintable obj then $taintable^s$ obj.*

This completeness theorem however needs to be restricted to undeletebale objects. The reason is that a tainted process can be killed by another process, and after that can be "recreated" by a cloning event from an untainted process—remember we have no control over which process ID a process will be assigned with. Clearly, in this case the cloned process should be considered untainted, and indeed our dynamic tainted relation is defined in this way. The problem is that a static test cannot know about a process being killed and then recreated. Therefore the static test will not be able to "detect" the difference. Therefore we solve this problem by considering only objects that are present in the initial state and cannot be deleted. By the latter we mean that the RC-policy stipulates an object cannot be deleted (for example it has been created by *root* in single-user mode, but in the everyday running of the system the RC-policy forbids to delete an object belonging to *root*). Like $taintable^s$, we also have a static check for when a file is undeletable according to an RC-policy.

This restriction to undeletable objects might be seen as a great weakness of our result, but in practice this seems to cover the interesting scenarios encountered by system administrators. They want to know whether a virus-infected file introduced by a user can affect the core system files. Our test allows the system administrator to find this out provided the RC-policy makes the core system files undeletable. We assume that this proviso is already part of best practice rule for running a system.

We envisage our test to be useful in two kind of situations: First, if there was a break-in into a system, then, clearly, the system administrator can find out whether the existing access policy was strong enough to contain the break-in, or whether core system files could have been affected. In the first case, the system administrator can just plug the hole and forget about the break-in; in the other case the system administrator is wise to completely reinstall the system. Second, the system administrator can proactively check whether an RC-policy is strong enough to withstand serious break-ins. To do so one has to identify the set of "core" system files that the policy should protect and mark every possible entry point for an attacker as tainted (they are the seeds of the $tainted^s$ relation). Then the test will reveal whether the policy is strong enough or needs to be redesigned. For this redesign, the sequence of events our check generates should be informative.

6 Conclusion and Related Works

We have presented the first completely formalised dynamic model of the Role-Compatibility Model. This is a framework, introduced by Ott [5], in which role-based access control policies can be formulated and is used in practice, for example, for securing Apache servers. Previously, the RC-Model was presented as a collection of rules partly given in "English", partly given as formulas. During the formalisation we uncovered an inconsistency in the semantics of the special role *InheritParentRole* in the existing works about the RC-Model [5,7]. By proving the soundness and completeness of our static $taintable^s$-check, we have formally related the dynamic behaviour of the operating system implementing access control and the static behaviour of the access policies of the RC-Model. The crucial idea in our static check is to record precisely the information available about the initial state (in which some resources might be tainted), but be less precise about the subsequent states. The former fact essentially gives us the soundness of our check, while the latter results in a finite search space.

The two most closely related works are by Archer et al and by Guttman et al [1,3]. The first describes a formalisation of the dynamic behaviour of SELinux carried out in the theorem prover PVS. However, they cannot use their formalisation in order to prove any "deep" properties about access control rules [1, Page 167]. The second analyses access control policies in the context of information flow. Since this work is completely on the level of policies, it does not lead to a sound and complete check for files being taintable (a dynamic notion defined in terms of operations performed by the operating system). While our results concern the RC-Model, we expect that they equally apply to the access control model of SELinux. In fact, we expect that the formalisation is simpler for SELinux, since its rules governing roles are much simpler than in the RC-Model. The definition of our admissibility rules can be copied verbatim for SELinux; we would need to modify our granted rules and slightly adapt our static check. We leave this as future work. Another direction of future work could be to reason formally about confidentiality in access control models. This would, of course, need the explicit assumption about the absence of any covert channels in systems.

Our formalisation is carried out in the Isabelle/HOL theorem prover. It uses Paulson's inductive method for reasoning about sequences of events [8]. We have approx-

imately 1000 lines of code for definitions and 6000 lines of code for proofs. Our formalisation is available from the Mercurial repository at http://www.dcs.kcl.ac.uk/staff/urbanc/cgi-bin/repos.cgi/rc/.

References

1. M. Archer, E. I. Leonard, and M. Pradella. Analyzing Security-Enhanced Linux Policy Specifications. In *Proc. of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 158–169, 2003.
2. M. Bugliesi, S. Calzavara, R. Focardi, and M. Squarcina. Gran: Model Checking Grsecurity RBAC Policies. In *Proc. of the 25th IEEE Computer Security Foundations Symposium (CSF)*, pages 126–138, 2012.
3. J. D. Guttman, A. L. Herzog, J. D. Ramsdell, and C. W. Skorupka. Verifying Information Flow Goals in Security-Enhanced Linux. *Journal of Computer Security*, 13(1):115–134, 2005.
4. S. Jha, N. Li, M. V. Tripunitara, Q. Wang, and W. H. Winsborough. Towards Formal Verification of Role-Based Access Control Policies. *IEEE Transactions Dependable and Secure Computing*, 5(4):242–255, 2008.
5. A. Ott. The Role Compatibility Security Model. In *Proc. of the 7th Nordic Workshop on Secure IT Systems (NordSec)*, 2002.
6. A. Ott. *Mandatory Rule Set Based Access Control in Linux: A Multi-Policy Security Framework and Role Model Solution for Access Control in Networked Linux Systems*. PhD thesis, University of Hamburg, 2007.
7. A. Ott and S. Fischer-Hübner. A Role-Compatibility Model for Secure System Administration. http://www.rsbac.org/doc/media/rc-paper.php.
8. L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.
9. B. Sarna-Starosta and S. D. Stoller. Policy Analysis for Security-Enhanced Linux. In *Proc. of the 2004 Workshop on Issues in the Theory of Security (WITS)*, pages 1–12, 2004.
10. E. Uzun, V. Atluri, S. Sural, J. Vaidya, G. Parlato, A. L. Ferrara, and P. Madhusudan. Analyzing Temporal Role Based Access Control Models. In *Proc. of the 17th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 177–186, 2012.
11. X. Zhang, C. Urban, and C. Wu. Priority Inheritance Protocol Proved Correct. In *Proc. of the 3rd Conference on Interactive Theorem Proving (ITP)*, volume 7406 of *LNCS*, pages 217–232, 2012.