The Entity Labeling Pattern for Modeling Operating Systems Access Control

Peter Amthor

Ilmenau University of Technology
PO Box 100565, 98684 Ilmenau, Germany
peter.amthor@tu-ilmenau.de

Abstract. To meet tightening security requirements, modern operating systems enforce mandatory access control based on formal security policies. To ensure the critical property of policy correctness, formal methods and models for both their specification and verification are used. The variety of these approaches reflects the diversity and heterogeneity of policy semantics, which makes policy engineering an intricate and error-prone process. Therefore, a common formal framework is needed that unifies both diverse access control systems on the one hand and diverse formal criteria of correctness on the other hand.

This paper presents a step towards this goal. We propose to leverage core-based model engineering, a uniform approach to policy formalization, and refine it by adding typical semantic abstractions of contemporary policy-controlled operating systems. This results in a simple, yet highly flexible framework for formalization, specification and analysis of operating system security policies. We substantiate this claim by applying our method to the SELinux system and demonstrating the practical usage of the resulting model.

Keywords: Security engineering, security policies, access control models, operating system security, SELinux.

1 Introduction

In the application domains of modern operating systems, such as server virtualization, mobile and ubiquitous computing or automotive computing, security requirements are more than ever of paramount importance. In order to meet these requirements, systems security engineering increasingly relies on formally specified security policies [35]. These policies define rules that, given their reliable enforcement, can be proven to achieve application-specific security goals such as confidentiality and integrity of a system and the information it processes. Consequently, their key role in a security engineering process yielded an increasing number policy-controlled operating systems over the past years [7, 9, 13, 18, 23, 29, 31, 34].

Given their critical nature, specification and verification of OS security policies has proven to require as much attention as their design, implementation

and enforcement. To this end, formal models have been developed for such policies based on two major objectives: (1.) Modeling a particular system and (2.) modeling a particular security property. While the first approach seeks to precisely specify the security-related semantics of an operating system, which are determined by its respective application domain (such as roles [24] or user relationships [12]), the second takes the opposite way: formalizing and analyzing a security property, which results from the security requirements of a particular application domain (such as right proliferation [11, 14] or information flows [15]). Both approaches yield models that are available to formal methods; however, models resulting from both approaches are often incompatible: when focusing on a formally analyzable property such as dynamic right proliferation, system-independent access control models based on state machines have proven to be valuable; when focusing on a formal framework for policy specification and communication on the other hand, system-specific models such as for the SELinux operating system have evolved, which may in turn sacrifice analyzability with respect to a whole family of security properties.

This problem has been addressed by the design paradigm of model-based security policy engineering [4, 5, 15, 17, 22]. Its goal is to derive a uniform pattern for designing security models, which flexibly fits (1.) diverse security policy semantics as well as (2.) diverse formal analysis goals. Such a uniform pattern would then serve as a fundamental prerequisite for both specifying and verifying security policies.

According to [4,22], this goal can be achieved through a flexible and extensible common model core based on a deterministic state machine (*core-based model engineering*). In practice, it requires to adapt domain-specific abstractions to a deliberately general formal framework. This yields a twofold result: On the one hand, core-based model engineering eliminates the need for a formal framework from scratch, whenever a given security policy is to be analyzed with one of the two objectives stated above. On the other hand, given the versatile and thus domain-independent semantics of the pattern, the actual engineering effort to create usable model instances is still significant in practice (as pointed out by [22, pp. 46 et seq.]).

This paper aims at further reducing this engineering effort—and thus the probability of errors—in the domain of policy-controlled operating systems by presenting a refinement of the core-based modeling pattern. Its idea is based on the general principle of entity labeling, which can be found in a large family of access control (AC) policies for contemporary operating systems. The resulting modeling pattern will then be applied to the SELinux operating system, which exhibits semantic features typical for OS security policies. Based on the resulting SELinux access control model, we will discuss the costs of model design and model instantiation.

Contributions and Paper Organization To introduce the context of this paper, we briefly discuss relevant related work (Sec. 2), followed by a summary of the fundamental concepts of one typical OS representative, SELinux (Sec. 3). Section 4 focuses on a formalization of the discussed concepts: First, the fundamentals of

core-based modeling are introduced (Sec. 4.1). We then present a novel, abstract policy modeling pattern based on *entity labeling* (Sec. 4.2), which enriches the core pattern by adequate access control semantics for the operating systems domain. It hence eases analysis and verification of contemporary operating system security policies with respect to an actual system's protection state (*dynamic analysis*) using existing formal methods and tools.

To substantiate this claim, we applied our pattern to SELinux. We create an entity labeling model of the SELinux access control system (Sec. 5) and discuss, how a real-world system's protection state and security policy can be transformed into an instance of this model (Sec. 6). This paves the way for subsequently applying tried and tested analysis methods for core-based models to SELinux, some of which we will discuss based on static and dynamic reachability properties. We conclude with Section 7.

2 Related Work

In the AC model community, considerable research has already been done to unify model semantics and formalisms. Notably, the access control meta-model by Barker [5], the Policy Machine [10], and core-based security models [17, 22] provide general formal frameworks for a precise specification of access control semantics and policies. While Barker's unifying meta-model and the Policy Machine are primarily designed for policy specification, core-based modeling aims at both specifying and analyzing/verifying a policy.

Another family of formal AC models is specifically tailored to OSs. Among numerous work in this area, most is tailored to specific operating systems such as SELinux [26,36,38] or special types of OS policies such as MLS [19]. While all of these approaches emphasize policy analysis with respect to a particular formal security property, they cannot be easily adapted to other OS AC semantics or other formal analysis goals. In our approach, we aim for both: streamlined adaption to versatile OS AC semantics that share only the abstract concept of labeling, and accessibility to a bandwidth of security properties and their appropriate formal analysis methods.

The basic idea of label-based AC modeling is far from being new. Dating back to the historical BLP model [6], which effectively introduced access permissions based on labels, a whole new class of attribute-based AC models (ABAC) evolved based on this principle [16,39]. However, they usually focus policy specification in the domain of service-oriented architectures [20, 28, 37] rather than system architectures. To this end, both the goal of formal policy analysis and the focus on the OS domain cannot be easily incorporated into existing ABAC models.

3 SELinux Access Control

Today's operating systems increasingly rely on mandatory access control (MAC) mechanisms governed by a security policy. In large parts, their authorization semantics are based on assigning policy-specific labels to entities, which are

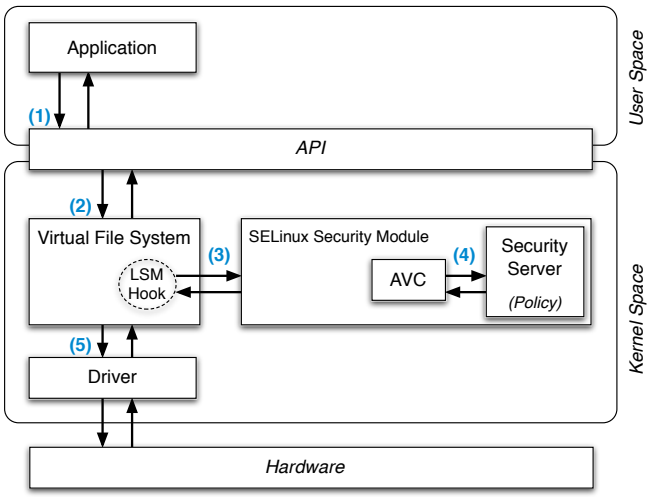


Fig. 1. Processing an access request in the SELinux security architecture.

divided into subjects (an activity abstraction such as process or thread) and objects (OS resources, described by abstractions such as files, handlers, sockets, etc.). The idea of label-based OS policies dates back to SELinux [18], one of the first policy-controlled OSs, and has been adopted by a wide range of later operating systems such as SEBSD [34], Oracle Solaris [9], Microsoft Windows [13], and Google's Android [29].

The goal of this section is to take a closer look at the security architecture and policy semantics of SELinux as a typical representative of modern policy-controlled operating systems.

3.1 Security Architecture

The original goal of SELinux was to enforce MAC in the Linux operating system. To achieve this, the *Flask* security architecture [31] was implemented, which clearly distinguishes between policy enforcement points (PEP) and a singular policy decision point (PDP). The PDP logically encapsulates the whole security policy.

Today, SELinux is implemented as a dynamically loadable kernel module. Its architecture merges into the Linux kernel through the LSM interface (*Linux Security Modules*). It provides ready-made PEP hooks for all system call implementations, which are connected to the PDP (the *Security Server*) via the SELinux kernel module. In addition to the processing logic, that translates information about an OS resource access into the policy-related data structures that are used by the security server, this module also includes a caching mechanism for previously made decisions (the *Access Vector Cache*, AVC).

To illustrate how an access request by an application process (1) is handled in SELinux, we consider the following example based on Linux kernel 3.19 (cf.

Fig. 1): Once an according syscall is processed by the kernel, e.g. *read()* for accessing a file (2), the LSM hook (`security_file_permission()`) invokes the according interface of the SELinux Security Module (3). Here, the permissions needed for authorizing the specific request (here: `FILE__READ`) are checked against the AVC (calling `avc_has_perm()`) or, in case of a miss, the Security Server's `security_compute_av()`-interface (4). The decision is then returned through the LSM hook and enforced by the *read()*-implementation in `vfs_read()` (either invoking the respective file system interface to ultimately access the storage hardware (5), or returning to the caller with an "access denied" error).

Inside the Security Server logic, access decisions are based on the policy rules and SELinux *security contexts* associated to entities. The latter is a label consisting of four attributes, which is usually represented by a string

```
user : role : type [: range]
```

where `user` is the name of an SELinux user the process belongs to, `role` is the name of an SELinux role the process assumes, and `type` is the name of the domain (or type) in SELinux type enforcement (TE) in which the process currently runs. Finally, `range` is a collection of confidentiality classes and categories used by multi-level security (MLS) policy rules based on the BLP model. Since support for the MLS mechanism is neither required by the SELinux policy semantics nor by the security server, this fourth attribute is optional. We will discuss the semantics of these attributes in a security policy in the next section.

On implementation level, security contexts of processes are stored in their management data structures, represented as a part of the non-persistent `/proc` file system, while those of objects such as files or sockets are stored in extended attributes of the respective file system.

3.2 Policy Semantics

As already mentioned, the PDP logic in SELinux is configured by a security policy. At runtime, a binary representation of this policy resides in kernel address space; however, as for the rest of this paper, we will refer to its human-readable specification in SELinux policy language [30] as "the (security) policy".

An SELinux security policy consists of statements, which can be classified into different types of rules. Each rule basically supports one of three fundamental AC concepts supported by SELinux: type enforcement (TE), role-based access control (RBAC), and multi-level security (MLS). The most basic authorization mechanism is implemented through TE, using TE-`allow`-rules which basically associate a pair of types with a set of permissions. The rule

```
allow system_t etc_t : file {read execute}
```

for example will grant any process labeled with the `system_t` type the right to read and execute any file-class object labeled with `etc_t`. We call

$$\langle \texttt{system\_t}, \texttt{etc\_t}, \texttt{file} \rangle$$

the *key* of above TE-`allow`-rule. A second authorization mechanism, whose support by an SELinux kernel is still optional, is MLS. Its rules are based on defining a dominance relation over the attributes *confidentiality class* and *category*, which is then used to limit all read- or write access to particular objects.

Lastly, the RBAC mechanism is used for restricting permitted labels of a process. It was introduced to provide a policy administrator with an additional, user-centric layer of AC configuration. RBAC rules define compatible combinations of all three major attributes: The role declaration rule

```
role user_r types { user_t passwd_t }
```

is necessary for a process label to include both the `user_r` role and any of the types `user_t` and `passwd_t`. Similarly, any role can be tied to one or more users by a user declaration rule. For instance,

```
user peter roles { admin_r }
```

is necessary for a process label to include both user attribute `peter` and the `admin_r` role.

Both the type- and role-attribute of a security context may change during runtime (known as *transitions*). Accordingly, there are policy rules to control these changes: For role transitions, a role-`allow`-rule

```
allow user_r admin_r
```

is necessary to change the role-attribute `user_r` of a process to `admin_r`. Note that, despite of the same keyword, this rule is not related to access authorization through TE.

For type transitions on the other hand, a special set of SELinux permissions exists that must be assigned to types through the already discussed TE-`allow`-rules. Rules with these permissions can be used for fine-grained control over allowed, forbidden, or even mandatory type transitions; however, it should be noted that their semantics are entirely different to rules intended for object access:

- `allow init_t apache_t : process transition` is necessary for a process to change its type from `init_t` to `apache_t`.
- `allow apache_t apache_exec_t : file entrypoint` is necessary for a process to change its type to `apache_t` during execution of a program file of type `apache_exec_t` (which is therefore called an *entrypoint* type of `apache_t`).
- `allow init_t apache_exec_t : file execute_no_trans` is necessary for a process with type `init_t` to execute a program file of type `apache_exec_t` *without* a type transition.

Since type transitions are intended to exclusively happen on program execution, the regular access permission `execute` on `apache_exec_t : file` will also be necessary in each case. Note that both permissions `execute` and `execute_no_trans` used as indicated above are sufficient for a program execution without type transition, yet not forbidding it.

As a last rule type, SELinux policies support constraints, that may further restrict (i.e. override) any access decision based on the mechanisms discussed so far. Supported by a limited syntax for nested boolean expressions, policy constraints can be used to explicitly forbid an access based on the security contexts of both involved entities and the given logical expression.

4 Modeling Patterns

This section introduces the two basic formal approaches we will use to model the SELinux AC system: the core-based modeling pattern by Pölck, and the novel EL pattern which aims at simplifying a domain-specific model engineering for OS AC policies. Throughout the rest of this paper, we will use the following conventions for formal notation:

- $\models$ is a binary relation between variable assignments and formulas in second-order logic, where $I \models \phi$ iff I is an assignment of unbound variables to values that satisfies ϕ. In an unambiguous context, we will write $\langle x_0, \ldots, x_n \rangle \models \phi$ for any assignment of variables x_i in ϕ that satisfies ϕ.
- For any mapping f, $f[x \mapsto y]$ denotes the mapping which maps x to y and any other argument x' to $f(x')$.
- For any mapping $f : A \to B$, $f \restriction_{A'}$ denotes a restriction of f to $A' \subset A$ that maps any argument $x' \in A'$ to $f(x')$, whereas $f \restriction_{A'} (x)$ is undefined for any $x \in A \setminus A'$.
- For any set A, 2^A denotes the power set of A.
- $\mathbb{B} = \{\top, \bot\}$ is the set of Boolean values, where $\top$ (*true*) is interpreted as "allow", $\bot$ (*false*) as "deny".

4.1 Core-based Modeling

The goal of the core-based model engineering paradigm is to establish a uniform formal basis for specification, analysis and implementation of diverse security models. In this work, we build our modeling pattern on top of this paradigm to leverage its generality regarding formal analysis methods and its uniform yet flexible design.

A core-based access control model is described by an extended state machine

$$\langle Q, \Sigma, \delta, \lambda, q_0, \textsc{Ext} \rangle \tag{1}$$

where Q is a (finite or infinite) set of protection states, Σ is a (finite or infinite) set of inputs, $\delta : Q \times \Sigma \to Q$ is the state transition function, $\lambda : Q \times \Sigma \to \mathbb{B}$ is the output function, $q_0 \in Q$ is the initial protection state, and Ext is an arbitrary tuple of static model extensions. The state machine serves as a common basis for formalizing policy semantics, called *model core*, which can be tailored to any domain-specific security policy in terms of state members and model extensions.

Based on the abstract definition above, three steps are required to describe a particular AC system through a core-based model (cf. [22, pp. 25 et seq.]):

(1.) Specializing Q, i.e. explicitly defining the automaton's state space members (dynamic model components).
(2.) Specializing Ext, i.e. defining static model components which are not part of the automaton's state.
(3.) Specializing δ and λ, i.e. describing the dynamic behavior of the AC system.

Depending on step (1), the initial protection state q_0 has to be specified according to the particular analysis goal. Depending on both steps (1) and (2), the input alphabet Σ has to be specified according to the interface of the modeled access control system.

In step (3), protection state dynamics are described by the state transition function δ through pre- and post-conditions of every possible state transition. This is done by comparing each input with two formulas in second-order logic, PRE and POST. We then define δ by formally specifying the conditions that each pair of states q and q' has to satisfy w.r.t. an input $\sigma \in \Sigma$ for a state transition from q to q' to occur:

$$\delta(q, \sigma) = \begin{cases} q', \langle q, \sigma \rangle \models \mathsf{PRE} \wedge \langle q', \sigma \rangle \models \mathsf{POST} \\ q, \text{ otherwise.} \end{cases} \tag{2}$$

Since an access control system is usually deterministic, POST fundamentally requires that q' equals q where not redefined.

Finally, to describe authorization decisions at an AC system's interface, the automaton features an output function λ. It enables the analysis of correct policy behavior and thus supports a formally verified specification. λ defines a binary access decision based on PRE:

$$\lambda(q, \sigma) \Leftrightarrow \langle q, \sigma \rangle \models \mathsf{PRE}. \tag{3}$$

4.2 Entity Labeling

This section describes entity labeling (EL), an abstract semantic modeling pattern for the formalization of contemporary operating system security policies. Based on the observations on OS policy semantics discussed in Section 3, the design goal of an entity labeling model is to describe access control policies which

(1.) use attributes (labels) of system entities for access decisions,
(2.) have a dynamic protection state,
(3.) are governed by additional constraints, possibly subject to a dynamically changing context.

The domain-specific semantics of such models is directly derived from these goals: (1.) To support labeling, a basic set of possible label values is needed. Since our goal is a complete description of an access control system, a set of entity identifiers is needed as well as an association of these entities with one or more label values. Then, label-based access rules can be formalized as well. (2.) To model a dynamic protection state, these formal concepts can be mapped on

the model core as discussed in Section 4.1. On top of it, the rules for changing labels of existing entities (which are also part of a system's policy) have to be modeled. (3.) Lastly, model constraints express time-invariant side conditions for correct behavior of the AC system. Due to their static nature, such conditions are not part of the automaton's state; however, they can of course include variables referencing any system interface outside the AC system—which is of increasing importance in mobile systems (e.g. time of day, NFC device proximity, geographic location, etc.) [8, 27].

In summary, we define six abstract components of an EL model, each of which may be specialized to concrete formal components for describing a particular policy:

Label Set (LS): A set containing legal label values.
Relabeling Rule (RR): A requirement for legal label changes.
Entity Set (ES): A set containing identifiers of entities relevant to access control decisions.
Label Assignment (LA): An association between each entity and its label (or labels).
Access Rule (AR): Rule that describes, based on two or more labels, which operations entities with these labels are allowed to perform.
Model Constraints (MC): Constraints over the other components that must be satisfied in every model state.[1]

This design follows the basic idea of model component specialization, which has been adopted by the core-based modeling paradigm from object-oriented programming.

Note that the semantics of these components do neither dictate nor imply any specific formalism (other than the extended state machine required by the model core). In practice, any suitable formalism may be chosen based on the particular security property in question as well as the established methods and tools for its formal analysis. As an example, an SELinux policy analysis could target the security property of type- or role-reachability, possibly related to some security-critical labels such as `system_t` or `admin_r` (cf. Section 6.4). Both graph-based and inference-based formal approaches could be used for analysis, which would effectively lead to either a directed graph or a body of logical formulas that define a deductive database for expressing relabeling rules.

For specializing these abstract model components, their semantics have to be matched to policy abstractions of a real system. In order to support model dynamics, this also includes decisions about which specialized components are modifiable during policy runtime—these should be defined within the core-based model's state—and which are not. Again, EL does not impose any restrictions on this.

[1] To distinguish from SELinux "constraints" mentioned in Section 3.2, we will keep calling them *policy constraints*, while the term *model constraints* exclusively refers to the abstract EL component discussed here.

Note that EL spans a subfamily of core-based models by adding domain-specific semantic abstractions to the calculus, which are however orthogonal to those of the core paradigm. Models in this family can be further tailored to match contemporary OS security policies. In the next section, we will show an example of this based on the SELinux security policy.

5 SELinux Security Model

In this section, we will demonstrate the application of EL on the SELinux operating system. In Section 5.1, the concepts of the SELinux security policy as described in Section 3 will be formalized using the EL modeling scheme. In Section 5.2, a full core-based access control model will be defined from these components. At last, Section 5.3 proposes a specification for the SELinux-specific commands and their impact on protection state transitions and model output.

5.1 EL Components

The abstractions of system resources that are managed by the Linux operating system are completely covered by the SELinux security policy. Therefore, we could define a separate entity set for each of these abstractions (processes, files, message queues, sockets, ...). However, the policy semantics are written on a higher level of granulariy: instead of singular entities, object classes are used to distinguish between OS abstractions. Since these classes are assigned to each system entity on runtime much similar to its respective security context, we will uniformly model these information as labels. Consequently, we define the following **label sets**:

- C is the set of SELinux object classes
- U is the set of SELinux users as defined in the policy
- R is the set of all roles as defined in the policy
- T is the set of all types and domains as defined by the policy

Moreover, a single **entity set** E represents all processes and other system resources (such as files, sockets, etc.).

To allow label changes, an SELinux policy uses special permissions such as `transition` or `entrypoint`, whose semantics drastically differ from those of other permissions used in TE-`allow`-rules (cf. Section 3). To this end, we refrain from modeling these elements of the policy language as actual access rules. Instead, type- and role-transitions are modeled by two **relabeling rules** as follows:

- $\hookrightarrow_r \subseteq R^2$ is a binary relation defined as $r \hookrightarrow_r r'$ iff a role transition from r to r' is allowed according to the policy's role-`allow`-rules
- $\hookrightarrow_t \subseteq T^3$ is a ternary relation defined as $t \overset{et}{\hookrightarrow}_t t'$ iff a type transition from t to t' via an entrypoint type et is allowed according to the policy's TE-`allow`-rules

User transitions can never be allowed by an SELinux policy and are therefore not modeled. The above notation serves as a shorthand here; for model checking purposes, both relations can be interpreted as edges (weighted in case of $\hookrightarrow_t$) of directed graphs.

Consequently, the remaining portion of TE-`allow`-rules in a policy is modeled by the following **access rule**. The mapping $allow : T \times T \times C \to 2^P$ represents the combined semantics of all TE-`allow`-rules:

$$allow(t_1, t_2, c) = \{p \mid \text{a TE-}\texttt{allow}\text{-rule for } p \text{ with key } \langle t_1, t_2, c\rangle \text{ exists in the policy}\}$$

where P is the set of SELinux permissions.

As already mentioned, SELinux stores label assignments as part of its protection state rather than in the policy. Nevertheless, we need to model the following **label assignments** for a meaningful analysis of the model's dynamic protection state:

- $cl : E \to C$ is the class assignment, which labels each entity with its SELinux object class.
- $con : E \to SC$ is the context assignment, which labels each entity with its SELinux security context. Here, the set of security contexts $SC = U \times R \times T$ represents all possible security contexts (labels) for entities under the given policy.

Concluding, two further restrictions on type- and role transitions have to be taken into account: those imposed by user and role declarations. For both, we use the following **model constraints**:

- $UR \subseteq U \times R$ associates users with roles they are allowed to assume according to the security policy's user declaration statements
- $RT \subseteq R \times T$ associates roles with types they are allowed to assume according to the security policy's role declaration statements
- $\tau_{UR} ::= \forall e \in E : con(e) = \langle u, r, t\rangle \Rightarrow \langle u, r\rangle \in UR$ ensures that no role is assumed a user is not authorized for
- $\tau_{RT} ::= \forall e \in E : con(e) = \langle u, r, t\rangle \Rightarrow \langle r, t\rangle \in RT$ ensures that no type is assumed a role is not authorized for

5.2 Core-based Model

The formal EL components defined above will be put into context of a core-based model now. Therefore, it has to be decided which component is part of the automaton's state (thus dynamic) and which is part of the extension vector (thus static). In case of SELinux, these components directly reflect the semantics of a policy that configures the security server, which again is static during runtime—except for E, cl and con. This results in the classification shown in Table 1.[2]

[2] For a minimal example, we did not include MLS and policy constraints in this model. To do this, additional label sets and label assignments for "classification" and "category", an authorization rule for the MLS dominance relation and another set of model constraints for expressing policy constraints is needed.

Table 1. Classification of SELX model components in EL and core-based modeling patterns.

EL Component	Q Members	Ext Members
LS	—	C, U, R, T
RR	—	$\hookrightarrow_r, \hookrightarrow_t$
ES	E	—
LA	cl, con	—
AR	—	$allow, P$
MC	—	τ_{UR}, τ_{RT}, UR, RT

According to the basic definition of the model core (1), we define an EL model for SELinux as a tuple

$$\mathsf{SELX} = \langle Q, \Sigma, \delta, \lambda, q_0, \textsc{Ext} \rangle$$

where

$$\begin{aligned} Q &= 2^E \times CL \times CON \\ \Sigma &= \Sigma_C \times \Sigma_X \\ \textsc{Ext} &= \langle C, U, R, T, \hookrightarrow_r, \hookrightarrow_t, allow, P, \tau_{UR}, \tau_{RT}, UR, RT \rangle \end{aligned}$$

Each state $q \in Q$ of the model is a triple $\langle E_q, cl_q, con_q \rangle$ with the semantics defined above, where we use the sets $E_q \subseteq E$ of all entities in state q, $CL = \{cl_q | cl_q : E_q \to C\}$ of all state-specific class assignments, and $CON = \{con_q | con_q : E_q \to SC\}$ of all state-specific context assignments. The input set Σ is defined by a set of commands Σ_C (that may be SELinux system calls, but also operations on application level for different implementations) and a set of arbitrary parameter sequences $\Sigma_X = (E \cup C \cup P \cup U \cup R \cup T)^*$. δ and λ are defined as in definitions (2) and (3). The extensions in Ext are defined as given in Sec. 5.1.

Both δ and λ are controlled by the conditions PRE and POST, which are partially defined using the following scheme. For each element of a model-specific set of commands $cmd \in \Sigma_C$ along with its parameters vector $x_{cmd} \in \Sigma_X$, we write:

▶ $cmd(x_{cmd})$ ::=
PRE: $\phi_0 \wedge \cdots \wedge \phi_n$;
POST: $\psi_E \wedge \psi_{CL} \wedge \psi_{CON} \wedge \psi_{\mathrm{MC}}$

where ϕ_i and ψ_j are expressions that q, q' and x_{cmd} should satisfy. While the above notation is used to define $\mathsf{PRE}(cmd)$ and $\mathsf{POST}(cmd)$ of each command, these conditions constitute the global terms:

$$\mathsf{PRE} = \bigvee_{cmd \in \Sigma_C} \Big(\sigma = \langle cmd, x_{cmd} \rangle \wedge \mathsf{PRE}(cmd) \Big)$$

$$\mathsf{POST} = \bigvee_{cmd \in \Sigma_C} \Big(\sigma = \langle cmd, x_{cmd} \rangle \wedge \mathsf{POST}(cmd) \Big)$$

While any number of arbitrary pre-condition clauses can be used in this scheme, post-conditions require a stricter pattern due to the fact that each command definition should yield exactly one possible follow-up state: Since post-conditions describe the modifications that q' should undergo with respect to q, the first three boolean clauses ensure an unambiguous definition of the entity set (ψ_E), class assignment (ψ_{CL}), and context assignment (ψ_{CON}) of q'. This requirement has to be considered for each specific EL model based on its particular state members. The last clause ψ_{MC} is mandatory, since it ensures that model constraints are satisfied in each follow-up state. In case of SELX, it is defined as

$$\psi_{\mathrm{MC}} ::= q' \models \tau_{UR} \wedge q' \models \tau_{RT}$$

while $q_0 \models \tau_{UR} \wedge q_0 \models \tau_{RT}$ must also hold for every correct SELX model instance.

For brevity, we will omit any of these clauses when writing command definitions iff the respective state component in q and q' is equal. ψ_{MC} will be generally considered implicit due to its mandatory nature.

5.3 Specifying SELX Commands

As an input to the state machine that triggers state transitions and output (access decisions), commands are the interface between a formalized security policy and a formalized analysis goal. As in most complex security architectures, security-relevant commands in an SELinux system may be modeled on at least two different levels of abstraction (see Fig. 2): (I) at PEP level, i.e. based on the access handling logic in the SELinux security module; (II) at API level, thus covering the rich and complex semantics of all API calls.[3]

Both semantical levels may be used depending on a particular analysis scenario: If a security engineer has the goal to verify a given policy based on the behavior of the security server, she will opt for Level I (in practice, this may be relevant e.g. if an attacker model includes control flow or code manipulation in a user process' address space). On the other hand, if the focus is on OS behavior from a user space perspective—considering kernel implementation as a black box—Level II commands have to be specified, accepting the more comprehensive and detailed degree of security-relevant interaction that is capsuled in an SELinux API call. In practice, given the huge flexibility of the Linux kernel with respect to differing library wrappers, kernel features and architectures, API implementations may vary in any case—thus yielding different command specifications in the model.

Moreover, complexity of the state transition function that results from command specification is another important point in question. As previous work on

[3] In practice, there is another choice to make here: either modeling library wrapper functions only, or including the syscall interface of the Linux kernel. Again, the decision depends on whether our respective analysis scenario includes applications that directly use syscalls. We will not further go into detail on when to prefer which degree of detail, and assume in the following that both are modeled.

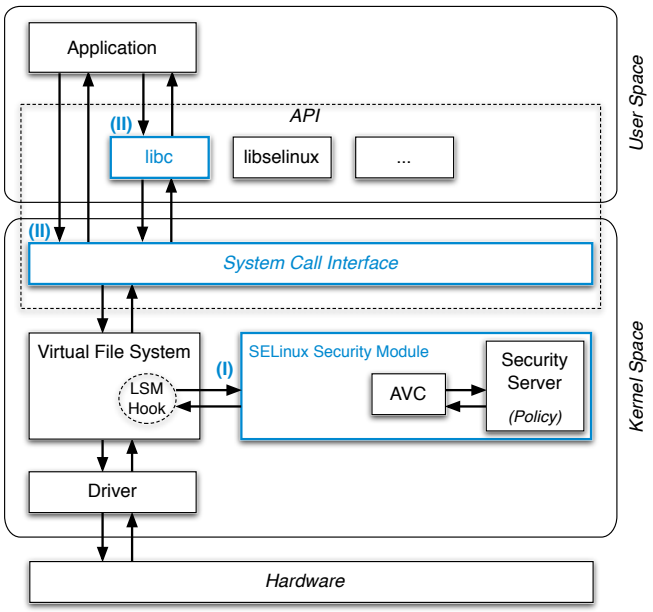


Fig. 2. Semantical levels for modeling commands.

model analysis has shown [3, 14, 25, 32], most approaches stand or fall with a certain degree of complexity. Thus, a clean separation of Level I and Level II commands serves two goals:

I. Keep command specifications as *small* and *uniform* as possible, even across different SELinux implementations, to support dynamic model analysis.
II. Enable flexible specification of tailored, implementation-specific model dynamics that expose a high-level interface for security analyses on application level.

Since Level II commands have to partially include the semantics of Level I commands, we follow a two-step approach for modeling dynamics in SELX: We first specify a small number of commands on Level I (Sec. 5.3), that are general enough to be used for every SELinux implementation. We then define a pattern for specifying commands on Level II (Sec. 5.3), that leverages the previous specifications and may thus be disassembled into Level I commands.

Basic Commands Level I commands, which we call basic commands, are *access*, *create*, *remove*, and *relabel*. They are defined as follows.

access specifies the semantics of any access decision. It does not model any state transitions and thus impacts only the automaton's output (λ). Any access by a process e to an entity e' that requires permission p is defined as

▶ **access**(e, e', p) ::=
PRE: $\{e, e'\} \subseteq E_q$
$\wedge\ cl_q(e) = \text{process}$
$\wedge\ cl_q(e') = c'$
$\wedge\ con_q(e) = \langle u, r, t\rangle$
$\wedge\ con_q(e') = \langle u', r', t'\rangle$
$\wedge\ p \in allow(t, t', c')$;
POST: $\top$

create specifies how a new entity is created in the protection system. In SELX, this entity may represent a resource such as a file, directory, or a socket, but also a process. Any creation of an entity e' of class c' with parent entity[4] e is defined as

▶ **create**(e, e', c') ::=
PRE: $e \in E_q$
$\wedge\ e' \in E \setminus E_q$
$\wedge\ c' \in C$
$\wedge\ con_q(e) = \langle u, r, t\rangle$;
POST: $E_{q'} = E_q \cup \{e'\}$
$\wedge\ cl_{q'} = cl_q[e' \mapsto c']$
$\wedge\ con_{q'} = con_q[e' \mapsto \langle u, r, t\rangle]$

Corresponding to *create*, *remove* specifies removing an entity e from the system:

▶ **remove**(e) ::=
PRE: $e \in E_q$;
POST: $E_{q'} = E_q \setminus \{e\}$
$\wedge\ cl_{q'} = cl_q \restriction_{E_{q'}}$
$\wedge\ con_{q'} = con_q \restriction_{E_{q'}}$

In an EL model, assigning new permissions to entities is done through labels. For SELX, a last basic command is needed that describes relabeling processes with a new security context. In SELinux, such process transitions occur on the execution of an "entrypoint" program. Changing the security context of a process e to a role r' and a type t' via an entrypoint program file f is defined as

▶ **relabel**(e, f, r', t') ::=
PRE: $e \in E_q$
$\wedge\ cl_q(e) = \text{process}$
$\wedge\ con_q(e) = \langle u, r, t\rangle$
$\wedge\ con_q(f) = \langle uf, rf, tf\rangle$
$\wedge\ r \hookrightarrow_r r'$
$\wedge\ t \overset{tf}{\hookrightarrow}_t t'$;
POST: $con_{q'} = con_q[e \mapsto \langle u, r', t'\rangle]$

[4] SELinux uses the term "parent entity" to generalize the concept of label inheritance: whenever a process is created, e is its parent process; whenever a file or directory is created, it is the respective parent directory.

Note that, from an abstract view, this collection of basic commands expresses operations fundamental to every EL model—even though their particular PRE and POST terms have been tailored to SELinux policies. This is another example for our basic assumption towards the generality of the EL model family, and how it can be leveraged to enhance the model-based engineering idea in the OS domain.

Composed Commands Based on the specifications of basic commands, we can now give a design pattern for such commands that model a specific system's API. For this purpose, we compose such Level II commands by using the *composition operator* $\circ : \Sigma \times \Sigma \cup \{\epsilon\} \rightarrow \Sigma$, which is defined as follows:

$$\langle c_1, x_{c_1} \rangle \circ \epsilon ::= \langle c_1, x_{c_1} \rangle$$
$$\langle c_1, x_{c_1} \rangle \circ \langle c_2, x_{c_2} \rangle ::= \langle c_{12}, x_{c_1} x_{c_2} \rangle$$

where $x_{c_1} x_{c_2} \in \Sigma_X$ is a concatenated parameter sequence and $c_{12} \in \Sigma_C$ is a composed command defined as

▶ $c_{12}(x_{c_1} x_{c_2})$ ::=
  PRE: $\mathsf{PRE}(c_1) \wedge \mathsf{PRE}(c_2)$;
POST: $\mathsf{POST}(c_1) \wedge \mathsf{POST}(c_2)$

We can then model any interface to the SELinux security policy by the resulting *composed commands*. As an example, *fork()* and *execve()* may be composed as follows:

▷ fork(*caller, child*) ::=
  access(*caller, caller,* fork)
∘ create(*caller, child,* process)

▷ execve(*caller, exec_file, post_r, post_t*) ::=
  access(*caller, exec_file,* execute)
∘ access(*caller, exec_file,* getattr)
∘ relabel(*caller, exec_file, post_r, post_t*)

where *caller* $\in E_q$, *child* $\in E_q$ are processes, *exec_file* $\in E_q$ is the program file to execute, *post_r* is the role that should be assumed by *caller* after execution, and *post_t* is the type that should be assumed by *caller* after execution.

Note that using composed commands, access control semantics of different granularity can be modeled: since basic commands cover all relevant behavior of the security policy, they can be composed on API level (outlined above), but as well on bare syscall level or even on level of a particular middleware interface.

6 Model Instantiation

The goal of this section is to demonstrate how a core-based EL model can be used in practice. Based on our SELinux model discussed in Section 5, we will focus on

the problem of extracting model components from a real-world SELinux system. Note that this is only one of two possible model analysis use cases: the other one focuses on designing an SELinux-based AC system from scratch, including API design and the policy itself. The practical process however can be considered symmetrical to the one outlined in the following. We will conclude this section with a summary of model extraction results and ongoing work regarding model analysis.

As discussed in Sec. 4.1, there are generally three specialized definitions required to tailor a core-based model to a particular AC system: the automaton's state space (Q), model extensions (Ext), and model dynamics (δ and λ). In the following, we present our methods to perform each of these three steps in practice. We used a Linux 3.19 kernel in a Debian distribution with SELinux enabled; for most of the following steps, tools of our model engineering workbench *WorSE* [4] have been used.

6.1 State Space

A protection state in SELX consists of an entity set and label assignments. Entities in SELinux are processes, whose labels are stored in the `attr` namespace of the `/proc` file system, and files representing OS objects, whose labels are stored in extended file system attributes.

Consequently, a protection state can be extracted from an SELinux system by parsing the whole file system. In practice, we build on our previous work described in [2] and [4, p. 49]: a file system crawler, originally intended for extracting ACLs from inodes, was slightly modified to recursively scan through a file system and extract each inode number i along with its associated file type ft and the associated SELinux security context sec using `stat`. These information are then compiled to form the initial state of the model, where $i \in E_{q_0}, cl_{q_0}(i) = ft, con_{q_0}(i) = sec$.[5] For processes, the directories `/proc/`pid`/attr` are scanned with a similar result.

Snapshot consistency, being a major problem in this step, could be ensured by different approaches: Disabling preemption for all other user processes while running the crawler would prevent runtime changes to the protection state, but requires critical tampering with the kernel. To this end, in our approach a frozen snapshot of a virtual machine is used instead. More information about this are provided in the aforementioned papers.

6.2 Model Extensions

The static model extensions in SELX consist of authorization and relabeling rules, which are equivalent to particular rule types in the SELinux security policy, and label sets these rules are based on. Model constraints regarding user-/role-/type-compatibility correspond to another type of policy rules.

[5] Technically, there is another, isomorphic mapping of file types to object classes that yields $cl_{q_0}(i)$ based on ft.

To extract model extensions, we have modified the policy compiler *sepol2hru* from [2]. It parses policy source files in plain syntax, i.e. after expanding auxiliary `m4`-macros, and produces an XML-based specification for the components of Ext. For evaluation purposes, we have applied it on a basic, non-MLS configuration of the reference policy by Tresys Technology [21].

The modified compiler is designed to isomorphically map statements in the SELinux policy language to definitions of the Ext components as follows:

Elements of C, P, U, R and T are explicitly declared through the statements `class`, `common`, `user`, `role`, and `type`.

allow is defined by assembling all TE-`allow`-statements as described in Sec. 5.1. We do not take into account the `neverallow` rule of the policy language, since it acts similar to an assertion tested by the policy compiler, but not reflected in any way in the resulting binary policy that steers the security server.

UR and *RT* are defined by assembling all `user`- and `role`-statements as described in Sec. 5.1.

$\hookrightarrow_r$ is defined by assembling all role-`allow`-statements. For each number of parsed rules $i \geq 0$ of the form `allow` $\{\ r_0 \dots r_n\ \}\ \{\ r'_0 \dots r'_m\ \}$, $\hookrightarrow_r$ is extended iteratively as follows:

- $\hookrightarrow_r^0 = \emptyset$
- $\hookrightarrow_r^{i+1} = \hookrightarrow_r^i\ \cup \{r_0 \dots r_n\} \times \{r'_0 \dots r'_m\}$

The result is $\hookrightarrow_r = \hookrightarrow_r^n$, where n denotes the total number of parsed role transition rules.

$\hookrightarrow_t$ is defined by assembling all TE-`allow`-statements for one of the three permissions `transition`, `entrypoint`, and `execute_no_trans` (we investigated their respective semantics in Sec. 3.2). Depending on which permission p is assigned to a key $\langle t_1, t_2, c\rangle$ by the i-th parsed rule ($i \geq 0$), $\hookrightarrow_t$ is extended iteratively using the *transition graph union* operator $\sqcup$ as follows:

- $\hookrightarrow_t^0 = \emptyset$
- $p = \texttt{transition} \wedge c = \texttt{process} \Rightarrow \hookrightarrow_t^{i+1} = \hookrightarrow_t^i\ \sqcup\ \langle t_1, \epsilon, t_2\rangle$
- $p = \texttt{entrypoint} \wedge c = \texttt{file} \Rightarrow \hookrightarrow_t^{i+1} = \hookrightarrow_t^i\ \sqcup\ \langle \epsilon, t_2, t_1\rangle$
- $p = \texttt{execute\_no\_trans} \wedge c = \texttt{file} \Rightarrow \hookrightarrow_t^{i+1} = \hookrightarrow_t^i\ \sqcup\ \langle t_1, t_2, t_1\rangle$

where $\sqcup : 2^{T^3} \times T^3 \rightarrow 2^{T^3}$ is defined as

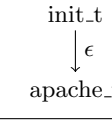
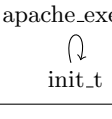
$$A \sqcup \langle a_1, a_2, a_3\rangle = \begin{cases} A \cup \{\langle a_1, a'_2, a_3\rangle \mid \langle \epsilon, a'_2, a_3\rangle \in A\} \cup \{\langle a_1, \epsilon, a_3\rangle\}, & a_2 = \epsilon \\ A \cup \{\langle a'_1, a_2, a_3\rangle \mid \langle a'_1, \epsilon, a_3\rangle \in A\} \cup \{\langle \epsilon, a_2, a_3\rangle\}, & a_1 = \epsilon \\ A \cup \{\langle a_1, a_2, a_3\rangle\} & \epsilon \notin \{a_1, a_2, a_3\} \end{cases}$$

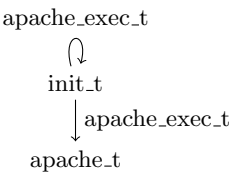
The result is $\hookrightarrow_t = \hookrightarrow_t^m$, where m denotes the total number of parsed type transition rules. Table 2 depicts a graphical representation of $\hookrightarrow_t$ as resulting from a sample set of policy rules.

6.3 Model Dynamics

The dynamic behavior of the SELinux AC system is based on the implementation of both the SELinux security module and library wrappers of API calls. While

Table 2. Examples for extracting $\hookrightarrow_t$ as a type transition graph.

Policy Rules Parsed	$\hookrightarrow_t$
`allow init_t apache_t : process transition;`	
`allow init_t apache_exec_t : file execute_no_trans;`	

`allow init_t apache_exec_t : file execute_no_trans;`
`allow init_t apache_t : process transition;`
`allow apache_t apache_exec_t : file entrypoint;`
	

the combination of both leads to the definition of composed commands, basic commands solely depend on the PDP logic and thus stick to their fundamental semantics, independent from an actual AC interface. As already discussed in Sec. 5.3, we consider this one of their essential merits.

In contrast to the other model components, extracting the definitions of composed commands is a task that cannot be automated. It requires insight into the implementation behind the desired interface, in our case both of the kernel and any wrapper functions. We have restricted to a subset of common syscalls in this study, such as *fork()*, *execve()*, *read()* etc. Once LSM hooks involved in a syscall have been identified, such as `security_file_permission()` in the example of *read()* in Sec. 3.1, specifying a composed command usually boils down to tracking subsequent calls of the `avc_has_perm()`-function in the SELinux security module. These give information about which parameters for the *access* basic command are needed. Moreover, protection-state-changing system calls such as *fork()* or *execve()* include more logic such as for relabeling or entity creation, and thus require the corresponding basic commands. An example of this was shown in Sec. 5.3.

Note that, when specifying composed commands, we are not interested in mere information retrieval concerning entity names and contexts, default type transitions and the like (which is why we did not consider the latter in the *execve* composed command). Instead, our goal is to model AC-related logic as precisely as possible, while any additional management logic for protection state data is deliberately excluded. This supports a clean separation of security model and analysis scenario, that may provide any of this information through the model's formal interface (i.e. via command parameters).

6.4 Results

Using the techniques described in this section, our method yields a machine-readable specification of a formal SELX model in ELM, an XML-based EL model format, which can be parsed by model analysis and verification tools such as *WorSE* (cf. [4, Sec. 4]).

To understand model complexity and scalability in real-world scenarios, we have conducted studies on different SELinux-based setups whose evaluation with respect to different analysis goals is subject to ongoing work. As a quantitative example, a real policy of one of our group's web servers included 2,847 types, 22 roles, 18 users, 4,330 relabeling rules, and 130,912 authorization rules. The corresponding protection state consists of approx. 390,000 entities, each with their associated security context labels.

Model Analysis Based on the SELX model extracted from the mentioned web server, we are currently applying and evaluating formal analysis approaches for core-based access control models. We here focus on two fundamental classes of analysis goals: static and dynamic reachability properties. While the first class requires analysis of a finite set of possible model states (thus generally related to static model components), the second one deals with properties that can affect an unknown, potentially infinite number of states (generally related to dynamic model components).

In the first class, we are adapting a graph-based method originally intended for information flow analyses [1] to analyze type- and role-reachability with respect to certain base types and roles (based on a type- and role transition graphs such as depicted in Table 2). A possible live monitoring framework for an SELinux system using this method is subject to future work.

In the second class, we are currently working on a heuristic method for model simulation, which may lead to permission proliferation with respect to particular entities in a system (e.g. a web server process). We have developed the *DepSearch* algorithm [3] for (r)-simple-safety analysis of HRU models (cf. Tripunitara and Li [33]), which we currently modify for SELX safety analysis.

7 Conclusions

In this paper, we addressed the problem of creating a uniform yet flexible access control model for operating system security policies. We aimed at complementing the core-based model engineering approach with an abstract, label-based modeling pattern that helps in tailoring the automaton's components to an OS AC system.

After discussing essential properties of a typical policy for the SELinux OS, we have presented the design of a formal policy model for this OS based on the core-based entity labeling pattern. We have substantiated its feasibility by demonstrating a model instantiation method for a real-world system and discussing tool-based model analysis methods.

Regarding the costs of model engineering in this case study, we made two major observations: First, tailoring the core-based pattern to the SELinux AC system was streamlined considerably by using the ready-made abstract categories of the EL pattern. We thus argue that, based on these categories, other OS security policies can be formalized in a very similar manner. Second, instantiating the model for a real-world system essentially required manual effort for formalizing commands. Here, a two-stage pattern consisting of basic commands and composed commands helps to reduce modeling complexity; again, we argue for this approach to be adaptable to other policy-controlled OSs.

Ongoing work focuses on the problem of generalizing and evaluating both the modeling pattern and analysis approaches based on it with respect to a larger family of OS policies. Besides, we believe that further specialization of the analysis methods discussed in Section 6.4 on SELX will support SELinux administration tasks. We therefore push complementary work on policy checking and runtime monitoring tools, that can help improve correctness and quality of an SELinux security policy in practice. Future work focuses on the more formal problem of proving transformation correctness for core-based EL models. In this area, we pursue the idea of a pattern-based formal proof of model-to-policy isomorphism, that is based on fundamental automaton properties and specialized by the semantics of the EL components used for the respective OS.

References

1. Amthor, P., Kühnhauser, W.: An Information Flow View on Privacy in Social Networks. ACM Trans. Internet Technol. 0(0), 0:1–0:17 (Dec 2015), (under review)
2. Amthor, P., Kühnhauser, W.E., Pölck, A.: Model-based Safety Analysis of SELinux Security Policies. In: Samarati, P., Foresti, S., Hu, J., Livraga, G. (eds.) In Proc. of 5th Int. Conference on Network and System Security. pp. 208–215. IEEE (2011)
3. Amthor, P., Kühnhauser, W.E., Pölck, A.: Heuristic Safety Analysis of Access Control Models. In: Proceedings of the 18th ACM symposium on Access control models and technologies. pp. 137–148. SACMAT '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2462410.2462413
4. Amthor, P., Kühnhauser, W.E., Pölck, A.: WorSE: A Workbench for Model-based Security Engineering. Computers & Security 42(0), 40–55 (2014), http://www.sciencedirect.com/science/article/pii/S0167404814000066
5. Barker, S.: The Next 700 Access Control Models or a Unifying Meta-Model? In: Proceedings of the 14th ACM Symposium on Access Control Models and Technologies. pp. 187–196. SACMAT '09, ACM, New York, NY, USA (2009)
6. Bell, D., LaPadula, L.: Secure Computer System: Unified Exposition and Multics Interpretation. Technical Report AD-A023 588, MITRE (Mar 1976)
7. Bugiel, S., Heuser, S., Sadeghi, A.R.: Flexible and Fine-Grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In: 22nd USENIX Security Symposium (USENIX Security '13). USENIX (Aug 2013)
8. Conti, M., Crispo, B., Fernandes, E., Zhauniarovich, Y.: Crêpe: A system for enforcing fine-grained context-related policies on android. Information Forensics and Security, IEEE Transactions on 7(5), 1426–1438 (Oct 2012)

9. Faden, G.: Multilevel Filesystems in Solaris Trusted Extensions. In: Proceedings of the 12th ACM Symposium on Access Control Models and Technologies. pp. 121–126. SACMAT '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1266840.1266859
10. Ferraiolo, D., Atluri, V., Gavrila, S.: The Policy Machine: A Novel Architecture and Framework for Access Control Policy Specification and Enforcement. Journal of Systems Architecture: the EUROMICRO Journal 57(4), 412–424 (Apr 2011)
11. Ferrara, A.L., Madhusudan, P., Parlato, G.: Policy Analysis for Self-administrated Role-based Access Control. In: Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 432–447. TACAS'13, Springer-Verlag, Berlin, Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-36742-7_30
12. Fong, P.W., Siahaan, I.: Relationship-based Access Control Policies and Their Policy Languages. In: Proceedings of the 16th ACM Symposium on Access Control Models and Technologies. pp. 51–60. SACMAT '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1998441.1998450
13. Grimes, R.A., Johansson, J.M.: Windows Vista Security: Securing Vista Against Malicious Attacks. John Wiley & Sons, Inc., New York, NY, USA (2007)
14. Harrison, M.A., Ruzzo, W.L., Ullman, J.D.: Protection in Operating Systems. Communications of the ACM 19(8), 461–471 (Aug 1976), http://doi.acm.org/10.1145/360303.360333
15. Kafura, D., Gracanin, D.: An Information Flow Control Meta-model. In: Proceedings of the 18th ACM Symposium on Access Control Models and Technologies. pp. 101–112. SACMAT '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2462410.2462414
16. Kuhn, D., Coyne, E., Weil, T.: Adding Attributes to Role-Based Access Control. IEEE Computer 43(6), 79–81 (June 2010)
17. Kühnhauser, W.E., Pölck, A.: Towards Access Control Model Engineering. In: Proc. 7th Int. Conf. on Information Systems Security. pp. 379–382. ICISS'11, Springer-Verlag, Berlin, Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-25560-1_27
18. Loscocco, P.A., Smalley, S.D.: Integrating Flexible Support for Security Policies into the Linux Operating System. In: Cole, C. (ed.) 2001 USENIX Annual Technical Conference. pp. 29–42 (2001)
19. Naldurg, P., Raghavendra, K.: SEAL: A Logic Programming Framework for Specifying and Verifying Access Control Models. In: Proceedings of the 16th ACM Symposium on Access Control Models and Technologies. pp. 83–92. SACMAT '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1998441.1998454
20. Park, S.M., Chung, S.M.: Privacy-preserving Attribute-based Access Control for Grid Computing. Int. J. Grid Util. Comput. 5(4), 286–296 (Oct 2014), http://dx.doi.org/10.1504/IJGUC.2014.065372
21. PeBenito, C.J., Mayer, F., MacMillan, K.: Reference Policy for Security Enhanced Linux. In: Proceedings of the 3rd Annual SELinux Symposium (2006)
22. Pölck, A.: Small TCBs of Policy-controlled Operating Systems. Universitätsverlag Ilmenau (May 2014)
23. Russello, G., Conti, M., Crispo, B., Fernandes, E.: MOSES: Supporting Operation Modes on Smartphones. In: Proceedings of the 17th ACM symposium on Access Control Models and Technologies. pp. 3–12. SACMAT '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2295136.2295140

24. Sandhu, R., Ferraiolo, D., Kuhn, R.: The NIST Model for Role-Based Access Control: Towards a Unified Standard. In: Proc. 5th ACM Workshop on Role-Based Access Control. pp. 47–63. ACM, New York, NY, USA (2000), iSBN 1-58113-259-X
25. Sandhu, R.S.: The Typed Access Matrix Model. In: Proceedings of the 1992 IEEE Symposium on Security and Privacy. pp. 122–136. SP '92, IEEE Computer Society, Washington, DC, USA (1992), http://dl.acm.org/citation.cfm?id=882488.884182
26. Sarna-Starosta, B., Stoller, S.D.: Policy Analysis for Security-Enhanced Linux. In: Proceedings of the 2004 Workshop on Issues in the Theory of Security (WITS) (2004)
27. Shebaro, B., Oluwatimi, O., Bertino, E.: Context-based Access Control Systems for Mobile Devices. IEEE Transactions on Dependable and Secure Computing PP(99), 1–1 (2014)
28. Shen, H.: A Semantic-Aware Attribute-Based Access Control Model for Web Services. In: Proceedings of the 9th International Conference on Algorithms and Architectures for Parallel Processing. pp. 693–703. ICA3PP '09, Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-03095-6_65
29. Smalley, S., Craig, R.: Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In: 20th Annual Network & Distributed System Security Symposium (NDSS) (Feb 2013)
30. Smalley, S.D.: Configuring the SELinux Policy. Tech. Rep. 02-007, NAI Labs (Feb 2005)
31. Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D., Lepreau, J.: The Flask Security Architecture: System Support for Diverse Security Policies. In: Proc. 8th USENIX Security Symposium (1999)
32. Stoller, S.D., Yang, P., Gofman, M., Ramakrishnan, C.R.: Symbolic Reachability Analysis for Parameterized Administrative Role Based Access Control. Computers & Security 30(2-3), 148–164 (2011)
33. Tripunitara, M.V., Li, N.: The Foundational Work of Harrison-Ruzzo-Ullman Revisited. IEEE Trans. Dependable Secur. Comput. 10(1), 28–39 (Jan 2013), http://dx.doi.org/10.1109/TDSC.2012.77
34. Watson, R., Vance, C.: Security-Enhanced BSD. Tech. rep., Network Associates Laboratories, Rockville, MD, USA (Jul 2003)
35. Watson, R.N.M.: A Decade of OS Access-control Extensibility. ACM Queue 11(1), 20:20–20:41 (Jan 2013), http://doi.acm.org/10.1145/2428616.2430732
36. Xu, W., Shehab, M., Ahn, G.J.: Visualization-based policy analysis for SELinux: framework and user study. International Journal of Information Security 12(3), 155–171 (2013), http://dx.doi.org/10.1007/s10207-012-0180-7
37. Yuan, E., Tong, J.: Attributed Based Access Control (ABAC) for Web Services. In: ICWS '05: Proceedings of the IEEE International Conference on Web Services. pp. 561–569. IEEE Press, Washington, DC, USA (2005)
38. Zanin, G., Mancini, L.V.: Towards a Formal Model for Security Policies Specification and Validation in the SELinux System. In: Proc. of the 9th ACM Symposium on Access Control Models and Technologies. pp. 136–145. ACM (2004)
39. Zhang, X., Li, Y., Nalla, D.: An Attribute-based Access Matrix Model. In: Proc. 2005 ACM Symposium on Applied Computing. pp. 359–363. SAC '05, ACM, New York, NY, USA (2005)