Model-based Safety Analysis of SELinux Security Policies

Peter Amthor, Winfried E. Kühnhauser, Anja Pölck
Ilmenau University of Technology, Germany
{peter.amthor, winfried.kuehnhauser, anja.poelck}@tu-ilmenau.de

Abstract—Since security has become an essential asset in numerous application areas, the integration of security policies has become a major issue in the design of security architectures, and many commodity operating systems have been furnished with abstractions to support policy protection and enforcement.

Given a security policy’s key position in defining and implementing a system’s security properties, quality attributes such as policy correctness, completeness, or consistency are essential objectives in policy engineering. On the other hand, considering the large amount of their responsibilities, security policies often are large and complex, rendering the analysis and proof of crucial quality attributes difficult.

This paper is a step towards tool-supported security policy analysis. It presents a model-based approach to analyze the dynamic proliferation of access rights in a policy-controlled SELinux access control system.

Index Terms—Security engineering, security architectures, security policies, security models, access control, HRU security models, Security Enhanced Linux.

I. INTRODUCTION

Quality properties such as security, safety, or robustness have long since evolved from elitist features of highly specialized systems to essential and commonplace requirements in many application areas. Whenever sophisticated assets such as analyzability, manageability, flexibility, scalability, or adaptability are required, such systems often are controlled by policies [1], [2], [3], [4], [5], [6].

In order to precisely describe security policies, formal security models such as [7], [8], [9], [10], [11], [12], [13], [14] are applied, allowing for formal analysis of security properties [15], [16], [17], [4], [18], [19] and serving as specifications from which policy implementations are engineered [20], [12], [22], [21].

In general, the goal of model-based policy analysis is the rigorous proof of security properties such as policy consistency, access right proliferation, or information flow leaks at an early design stage in a policy’s engineering process. This paper focuses on the analysis of access control policies for the Security Enhanced Linux (SELinux) operating system [22]. In order to support security policies, SELinux provides a kernel abstraction that allows to integrate security policies into the kernel’s security architecture [23], [24], thus supporting policy protection and policy enforcement.

This paper studies the dynamic proliferation of access rights in a policy-controlled SELinux access control system – the well-known general problem of model safety. Model safety analysis is the primary objective of HRU security models [25], and since 1975, a pool of methods, techniques, and tools for HRU safety analysis has matured [15], [26], [10], [27], [28]. The paper describes an approach to unlock this pool for the analysis of SELinux access control policies. To this end, SELinux policies are mapped to an isomorphic HRU model, which is then analyzed. Finally the results are mapped back to the original SELinux policy. The paper describes the mapping method and outlines its role as the foundation of an automatic policy transformation tool.

II. SELINUX PROTECTION SYSTEM

This section briefly introduces SELinux and its access control system. Also, some terms that are used throughout the paper will be clarified.

SELinux enforces mandatory access control (MAC) by integrating the Flask security architecture [23] into the Linux operating system [24]. This is done by reimplementing kernel-subsystems (such as process management and file systems) as object managers (OM) and integrating a security server (SS). A crucial property of Flask is implemented this way: Policy enforcement is cleanly separated from the policy logic.

While all security-relevant decisions are made by the SS, their actual enforcement is done by the object managers. Thus, Flask enables support for any arbitrary security policy by modifying or reimplementing the security server; however, this paper focuses on the default security server as shipped with the SELinux package due to its broad use in practice.

The policy logic represented by the SS is defined by its security policy, basically a set of text files created and edited by system administrators. These files have to be compiled into a binary format and installed into the SS to apply a policy.

In order to actually enforce the policy in the system, a security context is assigned to subjects and objects consisting of three security attributes: user, role, and type.1

An SELinux security policy is a set of rules controlling three separate access control (AC) types: Identity-based Access Control (IBAC), Role-based Access Control (RBAC), and Type Enforcement (TE). The most basic level of AC is implemented via TE, using sets of types defined by the policy and rules for accesses between these types depending

1 This paper does not consider MLS range, being a fourth attribute, which only occurs if the SELinux kernel was compiled with multi-level security support. However, the approach presented here can be easily adapted to consider the MLS attribute as well.
on certain permissions. For example, an access rule `allow system_t etc_t:file` could be used to grant each process of type `system_t` access to files of type `etc_t` using the permissions `getattr`, `read`, and `execute`. The type-attribute of a security context is not static; a set of special permissions controls type transitions (in SELinux terms). For example, an access rule for the transition permission is required between two types in order to allow a transition from one to the other.

RBAC provides a higher level of abstraction. The role of a subject is associated with a set of types that it may enter according to a role definition rule (objects, which do not have a role, are assigned a fixed role `object_r`). Unlike the access rules defined by TE, RBAC rules cannot be used to directly grant permissions; instead, they restrict the change of a subject’s type-attribute: A type transition of a subject in the given role is allowed only if the new type is defined as legal for that role (and, of course, if TE allows the transition). Since in SELinux a subject can only act in one role at a given time, there is another type of rule to explicitly allow a role transition (i.e. the change of a subject’s role-attribute).

Finally, IBAC rules may be used to define a set of legal roles for each user. In this way, the `user`-attribute in a subject’s security context is used to further restrict role transitions (as RBAC is used to restrict type transitions). Note that a `user` in SELinux AC is not the same as a Unix user – in SELinux, the association between a `user` and a (human) identity is generally much more strict and persistent.

### III. HRU Security Model Safety

This section summarizes HRU security models together with the HRU safety problem and discusses methods and tools for model safety analysis.

#### A. HRU Security Models

HRU security models [25] are the most powerful and general AC security models to date. In order to model dynamic behavior of AC systems, they combine access control matrices [29] with state machines. Each state of an HRU model is a snapshot of a system’s access control matrix (ACM); state transitions are triggered by application-specific operations that modify either the model’s subject set, object set, or cells of the ACM. Security properties such as right proliferation can now be analyzed by observing state transitions caused by input sequences; in particular, statements about proliferation of rights can be made by state reachability analysis.

Analysis of HRU models focuses on a fundamental family of questions: Given some model state (an ACM), is it possible that a subject ever obtains a specific right on some object? In terms of HRU, such a state is not considered `safe` with respect to that specific right. More precisely, a model state is called `safe` with respect to some right `r` if there is no sequence of inputs that writes `r` into a cell of the ACM that did not previously contain `r`. A general answer to this family of questions provides fundamental insights with respect to right proliferation in AC systems.

Formally, any HRU security model is a state machine `(Q, Σ, δ, q₀)` with a state set `Q`, an input set `Σ`, a state transition function `δ` and an initial state `q₀`. Each state `q ∈ Q` is a triple `(S_q, O_q, m_q)` where `S_q ⊆ S` is a state-specific subject set, `O_q ⊆ O` is a state-specific object set, and `m_q : S_q × O_q → 2^R` is a state-specific ACM with a finite right set `R`.

Model dynamics are defined by the transition function `δ : Q × Σ → Q` (often called `authorization scheme`), where the input set `Σ` includes all application-specific operations that modify the model state along with their parameters. For modeling AC of an operating system for example, operations in Σ may describe all system calls affecting its protection state, eventually involving users, rights, attributes, or files. This might include an operation to reclassify a document to a “confidential” level, where its parameters are some subject in `S` (typically the caller of the operation acting as a security administrator) and some object in `O` (the document to be reclassified). Precisely, Σ is the Cartesian product of (1) the set `C` of operations that may affect the model state and (2) the set `X` containing vectors of parameters (i.e. involved subjects and objects) of such operations. The transition function is now described as:

\[
\delta(q, (c, x)) \mapsto \begin{cases} q', & r_1 \in m_q(x_{s_1}, x_{o_1}) \land \ldots \land r_1 \in m_q(x_{s_1}, x_{o_1}) \\ q, & \text{otherwise} \end{cases}
\]

where `q ∈ Q` is the current policy state, `c ∈ C` is an input operation and `x = (x_1, \ldots, x_k) ∈ X` is an input vector for `c` (1 ≤ `s_i`, `o_i` ≤ `k, k ∈ N`). The new state `q'` is defined by a sequence of primitive operations specific for `c`. HRU defines six of these to reflect any possible alteration of the access matrix (`enter` or `delete` rights in matrix cells as well as `create` or `destroy` subjects or objects).

In practice, a more convenient notation for the above function is used, including both the conditions and primitive operations that modify the model state. As an example, Fig. 1 describes the state transition for an operation `delegateExecuteRight`, which allows a subject `s_1` to delegate an execute right to another subject `s_2` if and only if the delegating subject already owns that right.

\[
\delta(q, (\text{delegateExecuteRight}, s_1, s_2, o)) :=
\begin{cases}
\text{if } \text{execRight} \in m(s_1, o) \text{ then enter execRight into } m(s_2, o) \text{ end if.}
\end{cases}
\]

Fig. 1. State transition definition for `delegateExecuteRight`

Following this scheme, the state transition function `δ` is a set of partial definitions (such as for `delegateExecuteRight`), each one defining the parameters, conditions, and state-modifying effects for a single input operation `c`. In the example in Fig. 1, `q'` is defined by entering a right `execRight` into some matrix cell, effected by the primitive operation `enter`.

#### B. SELinux Policy Analysis

Although the HRU calculus was designed to study model safety properties, HRU safety basically turned out to be
undecidable for models written in the general HRU calculus [25], rendering it difficult to devise algorithms for automated analysis tools. As a consequence, several safety-decidable HRU fragments emerged [15], [26], [10] that bought safety decidability by limiting the expressive power of the calculus.

A different analysis approach is based on model simulation. Instead of restricting the HRU calculus, model simulation aims at a practical method for dealing with large real-world AC systems that require unrestricted models. Model simulation uses heuristic algorithms that exploit the fact that the HRU safety problem is semidecidable: once given two model states \( q \) and \( q' = \delta(q, a), a \in \Sigma^* \), and a right \( r \), it may be easily decided whether \( q' \) renders \( q \) unsafe wrt. \( r \). The core of these algorithms is a state search tree that, beginning with \( q \), searches \( q' \). The growth of the search tree is guided by a heuristic metric based on structural properties of a model’s authorization scheme.

A simple real-world scenario illustrates the analysis benefits of HRU model simulation: Assuming an organization runs a web server, hosting its public website as well as some text documents (e. g. software documentation) with restricted access. In order to protect these documents against, for instance, a hijacking attack on the web server process, it has to be assured that a subject called apache2 never gains the right to read the respective files. Since this is exactly the problem HRU models have been created for, it is self-evident to use them to model the web server’s AC system. However, restricting the expressive power of the calculus for the sake of decidable safety would inevitably neglect potential right proliferation, which could lead to an unsafe state. Simulating the behavior of the real AC system instead is a promising approach to answer the safety question for a specific scenario.

As indicated by the example, the attractiveness of HRU safety analysis is based on the fact that the calculus generally describes dynamic AC systems, which maintain a time-variant protection state. Obviously, this applies to SELinux AC as well, regarding the dynamics of processes, files etc. together with their respective set of security attributes. However, all security relevant decisions that may modify the protection state of an SELinux system are controlled by its security policy, which is a static body of rules in a defined syntax. This is why current research in SELinux security analysis primary focuses on that security policy and its rules: Besides practical contributions in this sector, such as the SETools software package for policy management and policy inspection by Tresys Technology (http://oss.tresys.com/projects/setools), a formal modeling of the SELinux policy description language has been developed by Zanin and Mancini [12] along with an analysis approach using the concept of access control spaces by Jaeger, Zhang and Edwards [17]. Similar model-related work, including a software implementation (Gokyo) but limited to the SELinux example policy, has been done by Jaeger, Sailer, and Zhang [30], [17]. Other recent work settles on information flow based policy analysis [31], [32], [4] and logical programming (querying security properties by mapping policies on logical programs) [33]. However, the aspect of time-variance, holding substantial information for analysis of an AC system, is not considered by any of these approaches. Thus, they cannot provide such analysis facilities as HRU models do, i. e. the analysis of dynamic right proliferation and safety.

In the example scenario described above, more specific questions arise if the web server is running on SELinux: May subject apache2 ever gain a permission read on an object of type privatedoc_f? Or, even more specific: May subject apache2 ever gain the read-permission on object private-file.txt? Using an HRU model created from the SELinux AC system of the server, safety analysis with respect to the read-permission are insightful in order to answer these questions.

Consequently, our effort is to unlock those promising HRU-based analysis methods and tools for SELinux AC systems as subject of analysis. As shown by [22], [12], SELinux security policies are model-based; they are specified by a precisely described formal language. Adding the aspect of a dynamic AC system now, a mapping onto the HRU calculus becomes possible. We propose a transformation method for this mapping which may eventually provide a strong foundation for model-based safety analysis of SELinux AC systems.

IV. FORMALIZATION OF AN SELINUX-SECURITY POLICY

As mentioned earlier, our general objective is to analyze a given SELinux security policy with respect to the proliferation of access rights or, to be more specific, the security impact of integrating new software into an existing system. The latter problem arises from the amendment of the original system policy by a new (or updated) policy fragment coming with the new software.

The approach of exploiting existing, elaborated analysis methods and tools for HRU models requires an isomorphic mapping of SELinux security policies to the HRU calculus. Once such a mapping is found, tools like the security engineering software IGraphoscope [28] can be applied, and analysis results can be mapped back onto the policy. This procedure imposes two vital requirements, that have to be met by a method for transforming an SELinux security policy into an HRU model:

1) **Equivalence of behavior:** After each possible system call, the configuration of the SELinux AC system must exactly meet the semantics of the HRU model state after an input modeling the respective call.

2) **Reversibility:** In order to allow for the application of model analysis results to the underlying security policy, all information describing the state and behavior of the SELinux AC system must be preserved in the corresponding HRU model state.

At this point, the meaning of the term “security policy” has to be circumscribed clearly. On the one hand, there is that static body of rules that SELinux calls its security policy. On the other hand, besides these rules, a time-variant system configuration state is needed in order to exploit the aspect of state variability expressed in HRU models (in fact, it is exactly that detail we are expecting analysis benefits from). So from
now on, the term “security policy” is used with not only some rules in mind, but also a particular, dynamic protection state.

A. SELinux Security Policies

SELinux security policies are formalized in the policy language described in [22]. However, some components determining the behavior of the AC system are part of the particular system configuration or the SELinux kernel (the security server) itself. The transformation target – the HRU security model – on the other hand is described using mathematic standard concepts as sets, matrices, and functions. Thus, a similar, uniform notation for all of the SELinux security policy components has to be found in order to eventually transform them into an HRU model. These components originate from three different sources: Subjects and objects of the system configuration and their associated SELinux object classes (processes, files, sockets, etc.), AC types of the security server and their security attributes (users, roles, types), and policy rules (transition rules, access rules). The following sections take a closer look at the individual components of such a security policy and describe them formally.

B. Subjects and Objects

In the following, security relevant subjects (processes) and objects (files, sockets, etc.) in a system may be called entities. They are formally described by an infinite set \( E \), spanning a system’s whole lifetime (which is theoretically infinite as well). Whenever it is necessary to consider a particular point in time, we will call this a system’s state \( i \), having its specific entity set \( E_i \subseteq E \).

The SELinux security server classifies all entities according to which permissions apply for accessing them. For example, entities of class file only provide access via open, read, write, append, etc.; such of class filesystem only via mount, remount, unmount, and so on. \( P \) denotes the total set of all permissions.

The classification of entities is defined by a classification function \( cf : E \rightarrow C \) on the basic set of all defined classes \( C \). For example, a running DHCP server (a process called dhcpd) would impose \( cf(dhcpd) \mapsto \text{process} \) with \( \text{process} \in C \) and \( dhcpd \in E \). The elements of each entity set \( E_i \) originate from the \( i \)-th state of the system configuration (counting from one well-defined initial state with \( i = 0 \)), whereas \( C \) and \( cf \) merely represent properties of the security server.

C. Access Control Concepts

Each AC concept in SELinux (IBAC, RBAC, and TE) makes up one attribute of an entity’s security context, implying basic sets of identities (users), roles, and types. Formally, \( U \) describes the set of all users, \( RO \) the set of all roles and \( T \) the set of all types. The elements of all three sets originate from the actual security policy.

The respective security context of an entity is defined by three functions: A user function \( af : E \rightarrow U \) represents the user-attribute, a role function \( rf : E \rightarrow RO \) represents the role-attribute, and a typing function \( tf : E \rightarrow T \) represents the type-attribute of the security context. Their actual values and the entities, whose security context they describe (see above), are defined by the initial state of the system configuration. For instance, the dhcpd-process introduced before may have the SELinux security context \( \text{system}_u: \text{system}_r: \text{dhcpd}_t \) leading to the mappings \( af(dhcpd) \mapsto \text{system}_u \), \( rf(dhcpd) \mapsto \text{system}_r \), and \( tf(dhcpd) \mapsto \text{dhcpd}_t \) where \( \text{system}_u \in U \), \( \text{system}_r \in RO \), and \( \text{dhcpd}_t \in T \).

D. Policy Rules

In addition to users, roles, and types, an SELinux security policy defines rules that control both accesses between entities as well as attribute changes (cf. Section II). For instance, a policy could contain one AC rule \( \text{allow dhcpd}_t \text{device}_t: \text{dir} \{ \text{getattr} \text{ search} \text{ open} \} \). This rule lists a set of permissions \( \{ \text{getattr, search, open} \} \) denoting all legal access modes for entities of type \( \text{dhcpd}_t \) on entities of type \( \text{device}_t \). Here, the second entity (target of the access) is restricted to a class \( \text{dir} \), which indicates the application scope of the rule (so in the above case, it only applies for directories). The former entity (the accessing one) is always of class process.

Formally, all allow-rules of a security policy can be described by an ACM \( am : T \times (T \times C) \rightarrow 2^P \). Given the policy rule above, it is then represented by \( \{ \text{getattr, search, open} \} \subseteq am(\text{dhcpd}_t, (\text{device}_t, \text{dir})) \). Note that the type of \( am \) exactly matches the structure of the corresponding allow-rule.

Three more functions are defined analogous to \( am \) in order to formalize the whole policy-specific body of rules: \( ur, r_{\text{def}} \) and \( r_{\text{trans}} \). As with \( am \), each of these functions represents one particular type of policy rule; they are discussed in detail in [34]. For a full review of all policy language elements and their semantics see [24], [22].

Combining all of the above elements, a formalized security policy of an SELinux system is a 14-tuple defined as follows:

\[
(E, C, U, RO, T, P, ur, am, r_{\text{def}}, r_{\text{trans}}, cf, uf, rf, tf).
\]

This precise notation provides a foundation to build an HRU model including both dynamic and static policy components (as pointed out in Section IV-A). The transformation itself is described in the following.

V. Transformation

This section briefly presents the transformation method for SELinux security policies – taking into account both equivalence of behavior and reversibility, as defined before. A detailed description of each transformation step including the authorization scheme can be found in [34].

In the previous section, a formalization for SELinux security policies consisting of heterogeneous elements was presented. However, since HRU models encode their state as a single matrix, this formalization is not yet a suitable starting point for modeling. Thus, rewriting these elements as a single composed matrix will be the first major step. To ensure compatibility with HRU semantics, this matrix has to feature the design of an ACM according to Lampson [29].
Afterwards, the actual HRU model will be defined, first considering the state set \( Q = 2^S \times 2^O \times M \), where \( M = \{ m \mid m : S \times O \rightarrow 2^R \} \) is the set of all possible ACMs. For this purpose, the composed matrix defined before will serve as a scheme for each \( m \in M \). Finally, an authorization scheme described by the state transition function \( \delta \) will be defined, in order to fulfill the requirement of equivalence of behavior. This last step involves some extensions unfamiliar to the classical HRU models, whose correctness and compliance with the original calculus is discussed in [34].

A. Matrix Representation

The encoding of a composed matrix follows a consistent scheme. Here, a unary function (\( cf \), \( uf \), \( rf \), \( tf \), \ldots) is encoded by one vector of function values, a binary function (such as \( am \)) is encoded by one matrix containing the function values within its cells, and sets \((E, C, U, \ldots)\) are used to index vectors and matrices. Moreover, \( r_{\text{def}} \) and \( r_{\text{trans}} \) may be efficiently encoded within a joint matrix \( rm \).

All elements are assembled in one composed matrix, diagonally appending independent partial matrices such as \( am \). Five fixed identifiers \( \text{classes} \), \( \text{users} \), \( \text{roles} \), \( \text{types} \), and \( ur \) indicate the rows of the appropriate vectors of function values.

So far, only a mapping between types (or, to be exact, types and type-class pairs) and sets of permissions has been defined. However, to meet the required layout of an ACM, a direct mapping of entities onto sets of permissions is necessary. So at first, a proper distinction between “active” and “passive” entities has to be made. SELinux does this by entity classification: The \( \text{process} \) class denotes active entities, while all other classes denote passive entities. Based on this, \( E_P := \{ e \in E \mid cf(e) = \text{process} \} \) describes the set of all processes, \( E_R := E \setminus E_P \) the set of all system resources. It is now possible to define an ACM \( lm : E_P \times E \rightarrow 2^P \). Since processes may interact with system resources as well as other processes, \( E_P \cup E_R = E \) denotes the set of all column indexes, while rows are only indexed by elements of \( E_P \).

Considering the cell contents of \( lm \), every subject-object pair can be associated with a subset of \( P \) via \( tf(s) \), \( tf(o) \), and \( cf(o) \). It holds \( \forall s \in E_P, \forall o \in E : lm(s, o) \rightarrow am(tf(s), (tf(o), cf(o))) \). Obviously, \( lm \) contains only such information already included in \( am \). However, \( lm \) associates these permission sets with actual entities in the system rather than their attributes, thus representing the dynamic nature of the protection state: Without this, an HRU model state could not consider any of the system’s entities.

Combining the matrix of all SELinux security policy components and \( lm \), the composed matrix \( cm \) is built as shown in Fig. 2. It is now capable of encoding the full protection state of the SELinux AC system as an HRU model state.

B. HRU Model Components

An HRU model is a quadruple \((Q, \Sigma, \delta, q_0)\) as described in Section III-A. Now the matrix notation of a formalized SELinux security policy \( cm \) will serve as a scheme to define the state set \( Q = 2^S \times 2^O \times M \). In order to be able to reference each particular system state \( i \), the composed matrix may be considered state-specific \((cm_i)\) like the entity set before (cf. Section IV-B).

For a formalized SELinux security policy with state-specific entity sets \( E_i = E_{P_i} \cup E_{R_i} \), the sets \( S \), \( O \), and \( M \) are defined as follows:

\[
S := \{ E_{P_i} \cup RO \cup T \cup \{ \text{classes}, \text{users}, \text{roles}, \text{types}, ur \} \mid E_{P_i} \subseteq E_P \},
\]

\[
O := \{ E_i \cup RO \cup (T \times C) \mid E_i \subseteq E \},
\]

\[
M := \{ cm_i \mid i \geq 0 \}.
\]

Note that by convention, the ACM of a particular state of an HRU model is called \( m_i \in M \) (being nothing else than \( cm_i \) in the context of this paper). The right set \( R \) is then defined as

\[
R := P \cup C \cup U \cup RO \cup T \cup \{ \text{roletrans} \}.
\]

It is now possible to encode an arbitrary initial state \( q_0 \) for the HRU model using the scheme of \( cm_i \).

Including all examples given in Section IV, a hypothetical HRU state \( q_j = (S_j, O_j, m_j) \) could be as follows:

\[
S_j := \{ \text{init, dhcpd, system_r, object_r, dhcpd_t, device_t, init_t, getty_t, \ldots, classes, users, roles, types, ur} \},
\]

\[
O_j := \{ \text{init, dhcpd, /usr/sbin/dhcpd, system_r, object_r, (dhcpd_t, process), (dhcpd_t, dir), ( dhcpd_t, file), \ldots, (device_t, process), (device_t, dir), \ldots} \},
\]
\( \delta(q_i, (\text{call}, x_1, \ldots, x_k))) := \)
\[
\text{if } \text{perm}_i \in m_i(x_{p_1}, x_{c_1}) \\
\quad \ldots \\
\quad \text{perm}_h \in m_i(x_{p_h}, x_{c_h}) \\
\quad \text{perm}_{i+1} \in m_i(x_{t_{i+1}}, (x_{t_{i+2}}, x_{c_{i+1}})) \\
\quad \ldots \\
\quad \text{perm}_{i+j} \in m_i(x_{t_{i+j}}, (x_{t_{i+j+1}}, x_{c_{i+j}})) \\
\quad \ldots \\
\quad \text{perm}_{i+j+1} \in m_i(x_{t_{i+j+1}}, (x_{t_{i+j+2}}, x_{c_{i+j+1}})) \\
\quad \ldots \\
\quad \text{perm}_{i+j+r} \in m_i(x_{t_{i+j+r}}, (x_{t_{i+j+r+1}}, x_{c_{i+j+r+1}})) \\
\quad \ldots \\
\quad \text{uf}_{x_{c_{i+j+r+1}}} \in m_i(x_{c'_{i+j+r+1}}, x_{r_{i+j+r+1}}) \\
\text{then } \text{op}_{e_i}; \\
\quad \ldots \\
\quad \text{op}_{e_h}; \\
\end{cases}
\]
\( \text{end if.} \)

Fig. 4. Pattern of a state transition via \( \delta \)

\[ R := \{ \text{open, read, search, execute, \ldots, process, dir, file, } \]
\[ \ldots, \text{system_u, unconfined_u, \ldots, system_r, \ldots, } \]
\[ \text{dhcpd_t, device_t, init_t, getty_t, \ldots, roletrans} \]

with \( m_j \) as shown in extracts in Fig. 3.

So far the state set and its components were covered as well as the layout of an initial state. The missing HRU components – authorization scheme and input set – will be discussed now.

According to the first main requirement of this transformation, equivalence of behavior, the state transition function \( \delta \) shall reproduce the behavior of the modeled SELinux AC system. Thus, all security-relevant system calls (i.e., those resulting in a modification of the model state) must be considered as inputs. For this purpose, we define a set \( SC := \{ \text{call} \} \) call is identifier of an SELinux system call, which creates or destroys entities or modifies \( rf \) or \( tf \). The corresponding state transition for each \( \text{call} \in SC \) can be modeled now: For all inputs \( (\text{call}, (x_1, \ldots, x_k)) \in \Sigma \) in model state \( q_i \in Q \), the state transition function \( \delta \) is defined according to the pattern in Fig. 4.

The variables here represent permissions \( \text{perm}_{i,j} \), \( \ldots \), \( \text{perm}_{h,j} \in P \), \( \text{process entities } x_{p_1}, \ldots, x_{p_h} \in E_p \), \( \text{entities } x_{c_1}, \ldots, x_{c_{h+j}} \in E_i \), \( \text{types } x_{t_{i+1}}, x_{t_{i+2}}, \ldots, x_{t_{i+j}}, x_{t_{i+j+1}}, \ldots, x_{t_{i+j+r}} \in T \), \( \text{roles } x_{r_{i+1}}, x_{r_{i+2}}, \ldots, x_{r_{i+j}}, x_{r_{i+j+1}}, \ldots, x_{r_{i+j+r+1}} \in RO \), \( \text{classes } x_{c_{i+1}}, \ldots, x_{c_{j}} \in C \), and parameter indexes \( 1 \leq p_i, \ldots, p_h, c_1, \ldots, c_{h+j}, t_{i+1}, t_{i+2}, \ldots, t_{i+j}, t_{i+j+1}, t_{i+j+r}, c_1, \ldots, c_{j}, r_{11}, r_{12}, \ldots, r_{i+j}, r_{i+j+1}, r_{i+j+r}, \ldots, r_{i+j+r+1} \leq k \).

A few remarks should be made concerning the semantics of this pattern. Regarding the conditions, four parts can be distinguished. (1) Conditions in lines 1 to \( h \) deal with the most immediate AC checks: Such involving two currently existing entities (e.g., a process trying to access a file via the \( \text{open} \) permission). These permission checks may be performed indirectly using \( am \) since it encodes all \( \text{allow-rules} \) of the policy. However, with a model-conform ACM \( ln_i \) providing direct association between entity pairs and permission sets, we may opt for the latter. (2) Permission checks involving one or two types that are not associated with existing entities may be required as well (e.g., the validity of type for a new process has to be checked before it is actually created). These checks (if required) are performed in the following \( j \) lines, now involving \( am \). (3) In case \( \text{call} \) includes a type- or role-transition, the third part (lines \( h + j + 1 \) to \( h + j + r \)) performs checks in \( rm \). For instance, the legality of a particular type acting in a particular role (function \( r_{def} \)) or transition rules for roles (function \( r_{trans} \)) may be checked here. (4) The last part consisting of the last \( s \) lines works the same way. It checks for the association of roles to user identities and is thus only needed for authorizing role transitions. Note that, due to the pattern-like definition of state transitions, any of the four parts may be omitted when modeling a particular system call. The subsequent primitive operations \( op_{e_1}, \ldots, op_{e_h} \) of \( \text{call} \) originate from a set of all six HRU primitive operations extended by eight more here: \( \text{for all subjects, for all objects, set cf(e), set uf(e), set rf(e), set tf(e), clear subject, and clear object} \). This general pattern can be applied to every system call, only depending on their particular Linux implementation. Fig. 5 shows the modeling of a sample implementation of the \( \text{execve} \) system call.

Now, that the possible contents of a parameter vector \( x_1, \ldots, x_k \) is precisely known, the input set may be finally defined as \( \Sigma := SC \times \{ E \cup \{ C, RO \cup T \} \}^k \) where \( k \in \mathbb{N} \).

As revealed by the example in Fig. 5, some concepts of the authorization scheme presented here do not comply with the original HRU calculus. In particular, the calculus does neither support referencing other matrix cells in conditions (what is done by using \( cf(e), uf(e), \text{etc.} \)) nor any of the new primitive operations. However, based on Turing completeness of the original HRU calculus [8], we argue for the correctness of these generalizations. A comprehensive review of this issue can be found in [34].

VI. PRACTICAL APPLICATION

As apparent by Sections IV and V, the transformation method presented here is comprehensive and, regarding an actual real-world scenario to analyze, bulky if performed manually. However, this characteristic is shared with another kind of transformation that has a well-developed automation: For any algorithm to be processed by a computer system, there is a specific formalization (e.g., a high-level programming language) that has to be compiled into another kind of notation (assembler language, machine code, etc.) while keeping its semantics. Thinking of additional debugging information (such as a symbol table), this transformation may be even isomorphic, thus reversible. Hence, a compiler software similar to those for programming languages can be built, to generate an HRU model out of an SELinux security policy.

We developed a compiler prototype (called sepol2hru) capable of performing this model transformation [34]. Based on an SELinux security policy in its original syntax and information about the actual system configuration to analyze, it generates an HRU model of the layout described in Section V-B. This
\[ \delta(q_i, (\text{execve}, \text{current}, \text{parent}, \text{exec}, \text{exec_path}, fdesc_0, \ldots, fdesc_{\text{max}}, \text{new_role}, \text{new_type})) := \]
\[
\text{if } \text{search} \in \text{lm}_i(\text{current}, \text{exec_path}) \land \text{execute} \in \text{lm}_i(\text{current}, \text{exec}) \land \text{transition} \in \text{am}(f(\text{current}), (\text{new_type}, \text{process})) \land \text{entrypoint} \in \text{am}(\text{new_type}, (f(\text{exec}), f(\text{exec}))) \land \text{execute} \in \text{am}(\text{new_type}, (f(\text{exec}), f(\text{exec}))) \land \text{ptrace} \in \text{am}(f(\text{parent}), (\text{new_type}, \text{process})) \land \text{inherit} \in \text{am}(\text{new_type}, (f(fdesc_0), f(fdesc_0))) \land \ldots \land \text{inherit} \in \text{am}(\text{new_type}, (f(fdesc_{\text{max}}), f(fdesc_{\text{max}}))) \land \text{new_type} \in r_{\text{def}}(\text{new_role}) \land \text{roletrans} \in r_{\text{trans}}(rf(\text{current}), \text{new_role}) \land uf(\text{current}) \in ur(\text{new_role}) \Rightarrow \text{clear object } \text{current}; \]
\[
\text{for all subjects } s:\]
\[
\text{enter } am(f(s), (\text{new_type}, \text{process})) \land \text{into } \text{lm}_i(s, \text{current}); \]
\[
\text{clear subject } \text{current}; \]
\[
\text{for all objects } o:\]
\[
\text{enter } am(\text{new_type}, (f(o), f(o))) \land \text{into } \text{lm}_i(\text{current}, o); \]
\[
\text{set } cf(\text{current}) \text{ to } \text{process}; \]
\[
\text{set } uf(\text{current}) \text{ to } uf(\text{parent}); \]
\[
\text{set } rf(\text{current}) \text{ to } \text{new_role}; \]
\[
\text{set } tf(\text{current}) \text{ to } \text{new_type}\]
\[
\text{end if.} \]

Fig. 5. Sample authorization scheme for execve

The proof of concept is our practical groundwork towards a more significant evaluation of the approach.

sepol2hru is a command-line tool implemented in C, using standard compiler-building tools such as Flex and GNU Bison. It takes input data from three different sources: (1) a security policy description (text files) containing all rules, (2) lists of classes and permissions (depending on the security server actually used for analysis), and (3) entity lists which encode an initial system configuration to analyze. The latter item is optional, since the initial model state does not need to contain any entities. However, in order to model a real-world SELinux system, this information may be provided as well.

Fig. 6 shows the process of an exemplary analysis. While all necessary policy data is available as text files, the entity lists for construction of the time-variant model state have to be generated first. Note that in practice, this is not a trivial prerequisite, as these lists are both extensive in size as well as difficult to obtain. In theory, a real-world system configuration has to be “frozen” with respect to every possible change of the protection state: Effectively, every single system call modeled in \( \delta \) must be globally blocked while “crawling” over the existing entities. One possible way to manage this is the Systrace-tool by Niels Provos (http://www.citi.umich.edu/~u/provos/systrace/). That way the security label of every single file, every running process, socket, file descriptor, etc. can be recorded in order to generate a consistent initial model state.

For the time being, our work focuses on offline-inspection of the file system and subsequent definition of a well-known boot state of the system. For this purpose, a file system crawler for the extraction of entity lists (file names associated with their security contexts) has been developed as an auxiliary tool.

The output of sepol2hru is a single text file containing the HRU model description in an XML-based format. This format is readable by the security engineering software IGraphoscope [28] for further analysis, featuring (amongst others) plugin-based tools for HRU model simulation.

The compiler does not yet take into account the authorization scheme. However, since this is the same for every SELinux security policy – provided one and the same implementation of the operating system, or rather, of its AC system – it is not subject of the compilation process. It may be either entered manually into the resulting HRU model description or, if specified as an extra input, be just passed through verbatim by the compiler.

sepol2hru has been tested using some common SELinux policy modules. While a small policy (dhcp.te) of about 1,400 rules in policy description language took less than 9 seconds to compile on our testing hardware (AMD Turion 64, 2.00 GHz), a middle-sized policy (apache.te) of about 5,800 rules took almost 118 seconds. The apache policy was tested on a system configuration consisting of 36 running processes and 42,922 files (for testing purposes, only file-objects were taken into account). Classes and permissions were extracted from the SELinux reference policy (available at http://oss.tresys.com/projects/refpolicy). Under these conditions, an initial matrix with 216 rows and 52,593 columns was generated.

The strong impact of input length on the compiler’s runtime performance is a direct consequence of its straight-forward implementation resulting in a quadratic upper bound of runtime. However, this is a mere implementation issue; since the development of the prototype did focus on practical application of the described method rather than runtime performance, significant improvements are possible here.

sepol2hru demonstrates the feasibility of the presented
analysis approach. So far, by applying the compiler and our file system crawler tool on a dynamic, real-world SELinux system configuration, we have pioneered its further analysis, which we also provide tools and methods for. Hence, a family of questions like those asked in section III-B may be answered by studying safety properties of the generated HRU model. However, this next step is still subject to ongoing research.

VII. CONCLUSIONS

We described the foundations of a method for analyzing dynamic access right proliferation in policy-controlled SELinux access control systems. The approach maps an SELinux access control system to an isomorphic HRU security model whose safety properties can then be analyzed by applying methods and tools available for the analysis of HRU model safety.

Mapping an SELinux access control system to an isomorphic HRU model is done in three steps. First, all items of the system’s security policy (such as users, roles, types, permission and transition rules) and all items representing a discrete system state (such as processes and files) are represented by sets and functions. Second, these sets and functions are encoded and composed into a single matrix syntactically equivalent to an HRU ACM. Third, the model’s authorization scheme is inferred from the semantics of the SELinux security server and system calls (such as fork, exec, or open).

To put this analysis method into practice, tool support has been implemented for processing both the security policy as well as the time-variant system state. The transformation of an SELinux security policy into an HRU model is complex and time-consuming when performed manually. Consequently, an HRU model generator similar to a programming language compiler has been built for its automation. Together with an auxiliary software for constructing the model’s initial state, it provides practical foundations for conducting SELinux policy analysis by the approach presented in this paper.

REFERENCES