Towards a Uniform Framework for Dynamic Analysis of Access Control Models

Peter Amthor
Ilmenau University of Technology, Germany
peter.amthor@tu-ilmenau.de

Abstract. Security-critical system requirements are increasingly enforced through mandatory access control systems. These systems are controlled by security policies, highly sensitive system components, which emphasizes the paramount importance of formally verified security properties regarding policy correctness. For the class of safety-properties, addressing potential dynamic right proliferation, a number of known and tested formal analysis methods and tools already exist. Unfortunately, these methods need to be redesigned from scratch for each particular policy from a broad range of different application domains. In this paper, we seek to mitigate this problem by proposing a uniform formal framework, tailorable to a safety analysis algorithm for a specific application domain. We present a practical workflow, guided by model-based knowledge, that is capable of producing a meaningful formal safety definition along with an algorithm to heuristically analyze that safety. Our method is demonstrated based on security policies for the SELinux operating system.

Keywords: Security engineering, security policies, access control systems, access control models, safety, heuristic analysis, SELinux.

1 Introduction

In a wide range of modern applications, software systems engineers are challenged by tightening security requirements based on legal and economical requirements. This challenge has been met by the approach of policy-controlled systems, whose design and implementation relies on formally specified security policies \cite{35}. These policies define rules that, reliably enforced by a system’s implementation, can be proven to meet formalized security requirements. One of the most important families of security policies is described by access control (AC) models, which can be used to model and verify most security policies of today’s policy-controlled systems \cite{8,10,21,25,30,31,34}.

Because of the paramount importance of formal models, a plethora of methods has emerged for their specification, analysis and implementation. More specifically, models for access control policies can be subdivided based on two principal objectives: either, to precisely specify the semantics of a particular policy family, e.g. roles \cite{27} or user relationships \cite{9,14}, or to focus on a precise definition of a particular security property, which represents a formalized security requirement (such as confinement of access right proliferation \cite{12,16} or information
flows [19]. However, models resulting from both objectives have turned out incompatible due to different formalisms; moreover, any new model focused on either objective has always required a new formal calculus from scratch.

These problems have been addressed by the paradigm of model-based security engineering [4, 5, 19, 20, 22]. Its goal is a uniform pattern for designing and analyzing security models, which should encompass the full bandwidth of practically important analysis goals, including dynamic, time-variant security properties in the lineage of HRU safety [16]. Aiming at these requirements, a recent, promising formal framework is the core-based modeling pattern [20, 22].

To streamline model-based systems security engineering based on this pattern, we have introduced the entity labeling approach as a complementary pattern [2]. We have shown how access control policies of the SELinux operating system, as a typical representative of policy-controlled systems, can be modeled using standardized, uniform semantic abstraction based on the principle of attribution. The resulting SELinux access control model SELX was discussed with a focus on specification costs, nevertheless neglecting an in-depth review of the actual policy analysis merits and costs.

In this work, we argue that both the core-based and the entity labeling pattern pave the way towards a uniform model analysis framework. This claim is substantiated by demonstrating how to tailor a general, heuristic-based analysis method for dynamic system properties (commonly known as safety properties) to the SELX model by leveraging the additional semantic information. In the process, we will define fundamental abstract interfaces to safety analysis methods that can be easily reused to tailor the same approach to other policy domains.

Contributions of this work include

1. a generalization of the DepSearch heuristic for model safety analysis [3], with the goal to be adaptable to every core-based/EL access control model,
2. a partially automatable workflow of tailoring both a formal safety property and an appropriate heuristic to a specific model,
3. a demonstration of this process for SELX, resulting in a specialized safety analysis algorithm for SELinux security policies.

The rest of this paper is organized as follows: After a brief discussion of relevant related work (Sec. 2), both the core-based and the entity labeling pattern for model-based security engineering are described (Sec. 3). This provides a basis for reviewing a promising heuristic approach to dynamic model analysis, DepSearch, in Sec. 4 which is then rewritten in a generalized form (Sec. 5.1). Sec. 5.2 describes the novel workflow of tailoring this general heuristic to an actual access control model, which is finally applied to the practical example of SELinux in Sec. 6. We conclude with Section 7.

2 Related Work

This work is based on our previous work towards flexible modeling patterns [2, 4]. A significant body of research has covered similar goals of unifying security
models to reduce the costs of systems security engineering in practice. Notable work in this area includes meta models for abstract access control policies \[5, 6\], information flow policies \[19\], attribute-based access control policies \[18\], and diverse hybrid paradigms \[11, 17\]. All these approaches share the goal of unifying and thus streamlining policy specification and implementation; formal analysis of policies on the other hand is not regarded a focal goal in any of them. Consequently, a generalization of automaton-based model descriptions (\textit{core-based} modeling) was introduced to model vital system properties, such as PEP interfaces and protection state dynamics, in a uniform yet flexible calculus \[20, 22\]. Our approach to generalizing heuristics for safety analysis is based on this latter pattern.

Due to their paramount importance in practical attack scenario, security properties based on system dynamics have always received significant attention in terms of a broad body of formal analysis methods. Ultimately, however, the fundamental property of \textit{safety} (introduced by Harrison, Ruzzo and Ullman \[16\]) is proven to be undecidable – which led to massive efforts in reducing a model’s complexity and thus expressive power in a least-obstructive way, assuming a specific, well-defined application domain. In virtue of such expressive restrictions, dynamic model analysis approaches such as \[13, 21, 26, 29, 32\] share the goal of precisely deciding their particular descendant of \textit{safety}. Based on the goal of maximum flexibility, which naturally prohibits any restrictive assumptions about policy semantics, our approach takes the opposite path: instead of sacrificing parts of a model’s expressive power, heuristic safety analysis sacrifices decidability. This enables tractability of the original, merely semi-decidable safety property and naturally yields an approximate solution.  

Our most promising approach to heuristic analysis, \textsc{DepSearch} \[3\], is therefore generalized in this paper to fit any core-based entity labeling model. The notion of safety used in our approach is based on Tripunitara’s and Li’s \[33\] precise and meaningful revisions of the original safety-definition. 

Our approach tries to merge the merits of two general information modeling paradigms: attributes (\textit{ABAC} in the security context) and ontologies. The latter is leveraged in our approach to introduce semantic knowledge into the security policy engineering as early as possible, just before formalization starts, based only on a security requirements analysis. The goal is to reduce the potential of human error in the later phases of formal analysis and specification of a security policy, which should be automated based on rules that can be derived from the richer semantical knowledge about the model.

3 Model Engineering

This section introduces the two basic formal approaches we will use to model and analyze AC systems: the core-based modeling pattern by Pööck \[20, 22\], and the entity labeling (EL) pattern \[1, 2\]. A security modeling pattern can be regarded

\footnote{An actually satisfied safety-requirement is not confirmable; however the critical case of unsatisfied safety, once confirmed, can be traced back to its cause.}
a specialized type of ontology, defined along with a particular formal calculus. Each modeling pattern reflects a slightly different view on an AC system, tailored to different policy specification and/or policy analysis goals.

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 $\phi$. In an unambiguous context, we will write $\langle x_0, \ldots, x_n \rangle \models \phi$ for any assignment of variables $x_i$ in $\phi$ that satisfies $\phi$. A logical formula assigned to a variable is delimited by $[ ]$. We will mark fixed values referenced from outside the formula by underlining, e.g. a fixed $x$ in an expression $\phi = [y = x]$. 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 $f : A \to B$, $f|_{A'}$ denotes a restriction of $f$ to $A' \subseteq A$ that maps any argument $x' \in A'$ to $f(x')$, whereas $f|_{A'}(x)$ is undefined for any $x \in A \setminus A'$. For any set $A$, $2^A$ denotes the power set of $A$. $\mathbb{B}$ is the set of Boolean values $\top$ (true) and $\bot$ (false).

### 3.1 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. A core-based access control model is defined as an extended state machine, called model core:

$$\langle Q, \Sigma, \delta, \lambda, q_0, \text{Ext} \rangle$$

where $Q$ is a (finite or infinite) set of protection states, $\Sigma$ 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 $\text{Ext}$ is an arbitrary tuple of static model extensions. The model core can be tailored to any domain-specific security policy in terms of dynamic state members and static model extensions. We formally do this by introducing two sets $DYN$ of model components that may change their state, and $STAT$ of such which may not. In practice, these sets are determined by the semantics of security policy rules: For a classical AC policy equivalent to an HRU model for example, we will define $DYN = \{ S, O, acm \}$ and $STAT = \{ R \}$ for the model components subjects set $S$, objects set $O$, access control matrix $acm : S \times O \to 2^R$, and rights set $R$.

Based on the abstract automaton and the definition of model components, 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 (based on $DYN$). (2.) Specializing $\text{Ext}$, i.e. defining static model components (based on $STAT$). (3.) Specializing $\delta$ and $\lambda$, 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 $\Sigma$ 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 $\delta$ 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 $\delta$ 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:

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

Because an access control system is fundamentally deterministic, POST requires that $q'$ equals $q$ where not redefined. In practice, PRE and POST are divided into commands that match particular interface calls of the modeled system: For any command $cmd \in \Sigma_C$, $\text{PRE}(cmd)$ denotes the partial, command-specific pre-condition of $cmd$ and $\text{POST}(cmd)$ its post-condition. The AC system’s interface is then modeled by $\Sigma = \Sigma_C \times \Sigma_X$, where $\Sigma_C$ is a set of command identifiers and $\Sigma_X$ contains sequences of possible values (command parameters) for variables in PRE and POST. This engineering-friendly notation dates back to the first access control models \cite{7, 16} and allows for a structured, system-specific notation of $\delta$ in terms of partial definitions depending on each input command. We will call the set of such partial definitions $\Delta = \{(cmd, x_{cmd}, \text{PRE}(cmd), \text{POST}(cmd)) | cmd \in \Sigma_C; x_{cmd} \in \Sigma_X; \text{PRE}(cmd), \text{POST}(cmd) \in \mathbb{B} \}$ a model’s state transition scheme, a specification of the behavior of $\delta$. The notation of a simple example command $\text{delegateRead}$ of an HRU model’s state transition scheme is shown in Fig. 1.

Finally, to describe authorization decisions of an AC system, the automaton features an output function $\lambda$. It enables the analysis of correct policy behavior and thus supports a formally verified specification. $\lambda$ defines a binary access decision $\lambda(q, \sigma) \iff \langle q, \sigma \rangle \models \text{PRE}$.

\begin{verbatim}
▷ delegateRead(s1, s2, o) ::= 
PRE: read_right \in acm_q(s1, o); 
POST: acm_q' = acm_q[\langle s2, o \rangle \mapsto acm_q(s2, o) \cup \{read_right\}]
\end{verbatim}

Fig. 1. Exemplary command definition for a core-based HRU model. We specify a command $\text{delegateRead}$ with parameters $\langle s_1, s_2, o \rangle$ that models delegation of $\text{read\_right}$ regarding $o$ by $s_1$ to $s_2$. $\text{PRE}(\text{delegateRead})$ expresses that $s_1$ needs to possess this right for a state transition by this command to occur, $\text{POST}(\text{delegateRead})$ expresses that $s_2$ needs to do so afterwards.

### 3.2 Entity Labeling

Entity Labeling (EL) is an abstract semantic modeling pattern for the formalization of contemporary access control policies. It was originally introduced with a focus on operating systems security policies \cite{1, 2} and is based on three observations regarding the semantics of such policies:

1. attributes that label entities are used for making access decisions
2. the protection state described is dynamic, i.e. the actual system’s configuration that policy rules refer to may change over time
3. time-invariant constraints restrict possible changes of the protection state; in practice, such constraints may be system-intrinsic (e.g. policy consistency criteria) as well as external (responding so a system’s environmental context).

These observations lead to six semantic categories that describe an access control system from a quite different angle than core-based modeling does: as an ontology describing attribution-based policy logic. In practice, core-model components in $DYN$ and $STAT$ also fall into one of these EL categories:

**Label Set (LS):** Set of legal label values.

**Relabeling Rule (RR):** Rules for legal label changes.

**Entity Set (ES):** Sets of entity identifiers.

**Label Assignment (LA):** Associations between a entity and its labels.

**Access Rule (AR):** Rules that describe, given two or more labels, operations that accordingly labeled entities are allowed to perform. AR components typically constitute an AC model’s access control function (ACF).

**Model Constraints (MC):** Constraints over the other components that must be satisfied in every model state.

For specializing these categories, their semantics have to be matched to policy logic – a task that should be performed based on a system’s security requirements, ideally during requirements analysis and system design. This highlights the importance of a rigorous requirements engineering, which lays the foundation of later model-based security analysis. At the same time, EL modeling helps to introduce and preserve such critical knowledge about an AC system’s intended semantics early in the whole engineering process.

On a formal level, the EL pattern is described by a tuple $\langle CAT, sem \rangle$ where $CAT = \{LS, RR, ES, LA, AR, MC\}$ is the set of EL category identifiers and $sem : CAT \rightarrow 2^{DYN \cup STAT}$ is their semantics association to model components as described above. For the sake of brevity, EL semantics of a model $M$ are denoted by $M|_{cat} = sem(cat)$ where $cat \in CAT$ is a semantic category identifier.

Note that the EL pattern complements core-based modeling by adding abstracted knowledge about a system’s requirements to the automaton-based calculus. Using both in combination, we can reason about meaningful analysis goals, their formal definition, and the interpretation of analysis results (e.g. the semantic origins of a right proliferation in a policy’s logic) as part of a generalizable security engineering approach. We argue for this claim by presenting a generalized analysis approach for dynamic model properties in the following sections.

### 4 Model Analysis

In this section, we discuss the basic idea of heuristic safety analysis. We will outline one of the most successful algorithmic approaches to this problem, DepSEARCH, presented for the HRU access control model in [3,4]. On this basis, we will later introduce a systematic generalization of DepSEARCH.
4.1 Heuristic Safety Analysis

The most general research questions targeted by the analysis of an access control system relate to the proliferation of access rights. As formalized in the seminal HRU access control model [16], the safety of a system fundamentally addresses such questions: Given a protection state of an HRU model, is it possible that some subject ever obtains a specific right with respect to some object? If this may happen, such a model state is considered unsafe with respect to that right. An intuitive yet practically meaningful interpretation of this question, (r)-simple-safety, has been defined (via its complement) by Tripunitara and Li [33]:

**Definition 1.** Given a core-based HRU model \( \langle Q, \Sigma, \delta, \lambda, q_0, (R) \rangle \), a state \( q = \langle S_q, O_q, acm_q \rangle \in Q \) is (r)-simple-unsafe with respect to a right \( r \in R \) iff \( \exists q' = \langle S_{q'}, O_{q'}, acm_{q'} \rangle \in \delta^*(q, a) \mid a \in \Sigma^* \):

\[
\exists s \in S_{q'}, \exists o \in O_{q'} : r \in acm_{q'}(s, o) \\
\land \ (s \notin S_q \lor o \notin O_q \lor r \notin acm_q(s, o)).
\]

where \( \delta^* : Q \times \Sigma^* \rightarrow Q \) is the transitive state transition function defined as \( \delta^*(q, \sigma \circ b) = \delta^*(\delta(q, \sigma), b) \) and \( \delta^*(q, \epsilon) = q \) for any \( \sigma \in \Sigma \cup \{ \epsilon \}, b \in \Sigma^* \).

Two facts are worth noting here: First, safety as per Def. 1 always relates to both a specific model state \( q \) to analyze (in practice, this is a momentary configuration of the system in question) and a specific access right \( r \) whose proliferation we are interested in. We call this \( r \), being the source of an HRU model’s authentication mechanics, a safety analysis target. Second, Def. 1 is least restrictive with respect to \( r \) since it allows any subject or object to violate safety. In practice, possible analysis questions may rather concentrate on a specific \( s \) or \( o \), leading to more specialized safety definitions such as \( (s,o,r)\)-simple-safety [33].

It is long since known that any variant of safety is not decidable, given an access control models with unrestricted expressive power and thus infinite state space [16]. In order to analyze such models, we must trade accuracy for tractability: using a heuristic algorithm, unsafe model states may be found (given such exist), while termination of the algorithm cannot be guaranteed. The idea of heuristic safety analysis thus leverages the semi-decidability of the problem. On the plus side, valuable hints on model correctness are obtained if unsafe states are found and policy engineers are pointed to input sequences that lead to such states.

The strategy behind heuristic safety analysis algorithms is to find an input sequence that, starting at \( q \), enters \( r \) into a matrix cell of some follow-up state \( q_{\text{target}} \). When this happens, \( q \) is proven to be unsafe with respect to \( r \); as long as no such target state is found, the search continues. Therefore, a successful algorithm must exploit model properties that maximize the probability of an input to contribute to a path from \( q \) to \( q_{\text{target}} \).

4.2 Dependency Search

We will now outline the DepSearch safety analysis heuristic for HRU models. We developed it based on the insight that in the most difficult case, right
leakages in a model are well hidden and appear only after long state transition sequences where each command executed depends exactly on the execution of its predecessor. Even for such hard analysis cases, DepSearch has turned out successful \[3\]. Essentially, DepSearch consists of two phases: static and dynamic analysis. We will discuss these phases on an informal basis, for an in-depth discussion and evaluation of the algorithm see \[3, 4\].

In the first phase, a static analysis of the HRU state transition scheme is performed. It yields a structured description of inter-command dependencies, constituted by entering (as a part of POST) and requiring (part of PRE) the same right in two different commands. The knowledge about these dependencies is encoded in a command dependency graph (CDG) whose vertices are commands, and an edge from command \(c_1\) to command \(c_2\) denotes that a post-condition of \(c_1\) matches at least one pre-condition of \(c_2\).

The CDG is assembled in a way that all paths from vertices without incoming edges to vertices without outgoing edges indicate input sequences for reaching \(q_{target}\) from \(q\). To achieve this, two virtual commands \(c_q\) and \(c_{target}\) are generated: \(c_q\) is the source of all paths in the CDG, since it represents the state \(q\) to analyze in terms of a command specification added to \(\Delta\). It is generated by encoding \(acm_q\) in POST\((c_q)\). In a similar manner, \(c_{target}\) is the sink of all paths in the CDG, which represents all possible states \(q_{target}\) by checking the presence of the target right in any matrix cell in PRE\((c_{target})\).

In the second, dynamic analysis phase, the CDG is used to guide dynamic state transitions by generating input sequences to the automaton. The commands involved in each sequence are chosen according to different paths from \(c_q\) and \(c_{target}\). Consequently, DepSearch successively generates input sequences by traversing the CDG on every possible path and in turn parameterizing the emerging sequence of commands with values that can be inferred from a constraint satisfaction problem (CSP) solver. Each effected state transition is simulated by the algorithm, and once a CDG path is completed, the validity of the unsafety-criteria (Def. \[1\]) is checked.

5 Generalized Framework

As introduced so far, the DepSearch algorithm is restricted to analyzing models of the classical HRU calculus. The family of dynamic model properties however spans the whole range of access control models, which requires an adaption of DepSearch for each and every such model. In \[4\], we have hinted at a uniform generalization of DepSearch, however, the previous work always resorts to HRU; problems related to tailoring a general core-based model to a specific notion of safety and interpreting this in terms of DepSearch have not yet been addressed in detail. In particular, there was no uniform workflow for heuristic-tailoring that avoids a re-design of each mode-dependent part of the heuristic.

With entity labeling we have introduced a plus of semantic information that can be used to tailor such a general safety analysis algorithm to a specific model and analysis goal in a consistent workflow. In this section, we will first present
the idea behind the generalization, followed by a description of the workflow how to specialize it. This workflow will then be demonstrated based on the practical case of an SELinux access control model in Sec. 6.

5.1 Generalizing DepSearch

In this section, we present fDepSearch as a variant of the original HRU algorithm including a number of abstract interfaces. These interfaces originate from a modular view on the heuristic, which is composed of both model-independent and mode-dependent modules. Fig. 2 illustrates this. Based on the model-dependent submodules, we consider three semantic abstractions: safety, dependency, and model dynamics. These abstractions and their related interfaces will now be discussed based on the fDepSearch algorithm specifications (Algs. 1 and 2).

The interfaces isLeaked (Alg. 1 (a)) and createCDGSink (Alg. 2 (b)) represent the analysis goal of the heuristics, expressed through a formal definition of safety (such as Def. 1 for HRU) and its target in terms of some model component (such as a right r). isLeaked is specified as a boolean function that implements a formal unsafety criteria, which may be satisfied by any model state $q'$ reached during heuristic execution. createCDGSink is part of the static model analysis and uses the knowledge about the safety target to generate a virtual command $c_{target}$ in the CDG as described in Sec. 4.2 which ensures that generated paths during the dynamic analysis always end in a command leaking the target.

We have stripped down the algorithm specification to the relevant parts for this discussion, leaving aside path generation strategies and graph traversal attributes, which are exclusively model-independent and thus need not to be tailored.
Algorithm 2: $f_{\text{DepSearch}}$: CDGAssembly

**In:** $\Delta \ldots$ model’s state transition scheme
- $q \ldots$ model’s state the safety of which is analyzed
- target ... leakage target

**Out:** $(V, E) \ldots$ command dependency graph
- $c_q \ldots$ starting point for the command sequence generation

**procedure** predecessors(in $v \in V$)

(c) $P \leftarrow \text{buildPredSet}(\Delta, v);$
    for $c \in P$ do
        if $c \notin V$ then
            $V \leftarrow V \cup \{c\};$
            predecessors($c$);
            $E \leftarrow E \cup \{(c, v)\};$
    end for

(e) $c_q \leftarrow \text{createCDGSource}(q);$
(b) $c_{\text{target}} \leftarrow \text{createCDGSink}(\text{target});$
    $\Delta \leftarrow \Delta \cup \{c_q\};$
    $V \leftarrow \{c_{\text{target}}\};$
    $E \leftarrow \emptyset;$
    predecessors($c_{\text{target}});$

**return** $(V, E), \hat{d}, \hat{\hat{d}}, c_q;$

---

**Fig. 2.** Modules in $f_{\text{DepSearch}}$. Generic interfaces are printed italic.
Implementing the safety abstractions also enables an implementation of the model’s notion of **dependency** through the interfaces `buildPredSet` (Alg. 2 (c)) and `assignParams` (Alg. 1 (d)). The former steers the generation of the CDG during static model analysis, where a formal definition of dependency (based on the previously defined safety and target) is used to add predecessor-edges between a command and other commands that establish necessary conditions for it to be executed. In an HRU model for example, this means for any given command `cmd` in the state transition scheme, a command `cmd'` is a predecessor of `cmd` iff entering rights in `POST(cmd')` has an impact on the value of `PRE(cmd)`.

The latter interface, `assignParams`, is actually another heuristic in itself. Since the static phase of DepSearch analyzes dependencies based on `PRE` and `POST` formulas only, yet disregarding the actual assignment of parameters that determine their values, parameter selection becomes a problem on its own. Again, the goal here is to maximize satisfiability and thus execution of commands on a CDG path that may eventually leak the safety target. Therefore, an implementation of `assignParams` must also take into account dependencies between commands, which may now established by their parameters relating to common model components. We have developed an approach to express the parameter selection problem as a constraint satisfaction problem by introducing a second graph, the parameter constraint network, that is input to a heuristically modified version of Poole’s and Mackworth’s arc consistency algorithm [23]. An evaluation of this parameter selection heuristic is still ongoing work. Formally, `assignParams` is an interface that evaluates both a base state `q` for path execution and the definition of commands in the path, to return a mapping of these commands to assignments for their individual parameters.

As a last abstract interface, `createCDGSource` (Alg. 2 (e)) generates the virtual command `c_q` in the CDG. Its implementation is independent from the other abstractions, since it only takes into account knowledge of **model dynamics** to construct `POST(c_q)` based on the model state the safety of which is to be analyzed.

### 5.2 Tailoring fDepSearch

In practice, manual implementation of these interfaces from scratch is still an error-prone process. To support this step and thus streamline the model-independent use of fDepSearch, EL categories are used to guide a three-phases workflow:

1. **Model Specification** Goal of the first phase is to model an access control policy in a formal calculus suitable for dynamic analysis.
(Step 1.1) As soon as an AC policy’s semantics are designed, EL categories should be used to identify and classify formal structures such as sets, relations, and mapping according to Sec. 3.2. Result is an EL model $\langle\text{CAT}, \text{sem}\rangle$.

(Step 1.2) In [22, pp. 25–27], the basic steps of specializing a core-based model for an access control policy are outlined (we summarized them in Sec. 3.1). In order to enable the modeling of dynamic behavior, these steps should now be applied to our EL model. We therefore classify model components into $DYN$ and $STAT$ and derive a state transition scheme $\Delta$ from policy rules. Result is a second, core-based model $M = \langle Q, \Sigma, \delta, \lambda, q_0, \text{Ext}\rangle$ covering the same policy.

2. Safety Specification In the second phase, the analysis goal has to be formalized in terms of a safety-property.

(Step 2.1) The access control function describes the impact of model components on the dynamic behavior of $\lambda$. Therefore, it formally defines the necessary conditions for any kind of safety-violating state transition (i.e., leakage). In practice, the ACF of a system may be obtained from two different sources: On the one hand, requirements engineering on a rigorously low level of abstraction may have already stipulated access restrictions in a formal way. On the other hand, especially in case of rich and complex policies that steer diverse security mechanisms a condensed and precise ACF may not be achievable right from the beginning of system engineering. In this latter case however, EL model semantics can be used to bridge the gap between informal requirements and a formal ACF definition as follows.

Formally, an ACF is a function $\text{acf} : \Sigma \rightarrow \mathbb{B}$ with $\text{acf}(\sigma) \Leftrightarrow \phi_{\text{acf}}$. Since the occurrence of model components from $AR$ in $\text{PRE}(c)$ is a necessary condition for a command $c$ to make an authorization decision (cf. EL semantics), a disjunction over such conditions determines the ACF as follows:

$$\phi_{\text{acf}} := [\bigvee_{\phi_i \in \Phi_{AR}} \phi_i]$$

where

$$\Phi_{AR} := \{\text{PRE}(c), c \in \Sigma_C | \exists x \in M|_{AR} : \text{PRE}(c) \text{ depends on the value of } x\}.$$  (1)

(Step 2.2) Based on the ACF, we will now extract such clauses involving model components in the set $DYN$ from the core-based pattern: this yields a set of dynamically changing leakage conditions. Each possible effect of a state transition influencing one of these clauses can be considered a leak, which leads to a maximal number of safety definitions based on these leaks. These definitions together with the target model components can be inferred automatically, as demonstrated in Sec. 6.2.

(Step 2.3) For a complex model featuring multiple components responsible for authorization, it becomes obvious that a number of alternate safety definitions is formally possible. Which one of them however is meaningful in practice

---

3 A detailed example on how the latter can be obtained in practice is given in [2].
4 In today’s operating systems for example, a hybrid of attribute-based access control and information flow policies is common.
must be decided in the next step and requires human knowledge, possibly once again by comparison against security requirements. Model components from the EL category RR are responsible for authorization changes in a protection state and therefore give a first indicator for filtering irrelevant safety definitions. More details on a practical case will be presented in Sec. 6.2.

3. Heuristic Specification In the last phase, the abstract interfaces of fDepSearch can be implemented based on the formal model and the definitions of safety and analysis target. As outlined in Sec. 5.1, the safety and target definitions guide the implementations of isLeaked and createCDGSink directly and, indirectly via the indirection of dependency abstraction drawn from the safety property, also the implementation of buildPredSet and assignParams. The most straight-forward implementation is that of createCDGSource, based on the model dynamics declared through DYN. Again, Sec. 6.2 will illustrate this process by implementing some of these interfaces.

6 Application to SELinux

The goal of this section is to demonstrate the tailoring of fDepSearch based on a real-world case study. To this end, we specify a safety analysis algorithm for the widespread policy-controlled operating system SELinux. For the sake of clarity, we will restrict our study to the analysis goal of safety analysis and cover the EL modeling of our policy only to that extent; more specifically, we will omit the model constraints (MC) components which are intended to support the verification of static model consistency properties. As used in SELinux, these model components could be used for proving interactive policy properties similar to BLP security [7].

In the following sections, we will successively address the three major phases in tailoring our analysis framework: model specification, safety specification, and heuristic specification as described in Sec. 5.2.

6.1 SELinux Model

To set the stage for safety analysis, we will briefly revisit the core-based/EL model for the SELinux access control system (SELX) from [2]. The model components that formally represent the SELinux access control semantics are

- $C$: the set of object classes
- $U$: the set of SELinux users
- $R$: the set of roles
- $T$: the set of types
- $E$: the set of all system resources, including processes
- $\rightarrow_r \subseteq R^2$: a relation modeling allowed role transitions
- $\rightarrow_t \subseteq T^3$: a relation modeling entrypoints and allowed type transitions
- $P$: the set of permissions
Table 1. Classification of SELX model components in EL and core-based modeling patterns.

| SELX   | Q Members | Ext Members |
|--------|-----------|-------------|
| LS     | —         | C, U, R, T  |
| RR     | —         | → 𝑟, → 𝑡   |
| ES     | E         | —           |
| LA     | cl, con   | —           |
| AR     | —         | allow, P    |

- allow : T × T × C → 2^{P}: a mapping of keys for TE-allow-rules to their associated permissions
- el : E → C: a mapping of resources to their object classes
- con : E → U × R × T: a mapping of resources to their security contexts

These model components are classified according to both patterns as shown in Tab. 1.

6.2 SELinux Safety

In [2], it was argued that a two-step approach to specifying dynamic model behavior (δ and λ of a core-based model) reduces analysis complexity and streamlines a generic approach to dynamic analyses. We will now substantiate this claim by using basic commands for deriving a safety definition that allows to tailor fDEPSEARCH to this model. The definitions of SELX basic commands to create and remove system resources, relabel processes and authorize access requests are given in Fig. 3.

(Step 2.1) To determine the SELX ACF, only conditions checking AR model components need to be taken into account. For example, \( \text{PRE}(\text{remove}) = \{ e \in E \} \) in Fig. 3 checks for the presence of a given entity e, which is independent of AR model components and therefore constitutes a mere consistency-check rather than an authorization decision. In case of our SELX basic commands, only access is influenced by AR components (which hints at the significance of a multi-level commands specification). Since \( \text{acf} \) solely depends on \( \text{PRE}(\text{access}) \), \( \phi_{\text{acf}} \) from Equation 1 expands to

\[
\text{acf}(e, e', p) \iff \exists t, t', c' : \\
(2.1) \{ e, e' \} \subseteq E_q \\
(2.2) \land cl_q(e) = \text{process} \\
(2.3) \land cl_q(e') = c' \\
(2.4) \land con_q(e) = \langle \_ , t \rangle \\
(2.5) \land con_q(e') = \langle \_ , t' \rangle \\
(2.6) \land p \in \text{allow}(t, t', c')
\] (2)

(Step 2.2) In the next step, we identify those clauses in \( \phi_{\text{acf}} \) that relate to common dynamic model components in the core-based sense. In SELX, three such cases can be found:
Definition 4. Given a SELX model $\langle Q, \Sigma, \delta, \lambda, q_0, \text{EXT} \rangle$, a state $q \in Q$ is (t)-unsafe with respect to a type $t \in T$ iff $\exists q' \in \{\delta^*(q, a) \mid a \in \Sigma^*\}$:

$$\exists e \in E_q \cap E_q : \text{con}_q(e) = \langle u, r, t \rangle \land \text{con}_q(e) = \langle \_ \_ t_q \rangle \land t \neq t_q .$$
Two conclusions can be drawn from the three definitions above: first, irrelevant safety definitions can be filtered to some extent by leveraging the information provided by core- and EL-semantics. In case of Def. 3, it becomes clear by studying POST expressions in the state transition scheme (Fig. 3) that entity classification is only performed during entity creation. Alternatively, the same fact can be inferred from the absence of any reference to the label set \( C \) within SELX|RR.

Second, these safety definitions are deliberately demarcated with respect to their targets, which means they can be combined in a straightforward way. As an example, an alternate definition of \( (t)\)-safety could also cover any new entity not present in \( q \). This results in a definition of \( (t)\)-simple-safety aligned with Def. 4.

**Definition 5.** Given a SELX model \( \langle Q, \Sigma, \delta, \lambda, q_0, \text{Ext} \rangle \), a state \( q \in Q \) is \( (t)\)-simple-unsafe with respect to a type \( t \in T \) iff \( \exists q' \in \{ \delta^*(q, a) \mid a \in \Sigma^* \} : q \) is \( (t)\)-unsafe with respect to \( t \) or \( \exists e \in E_{q'} : \text{con}_{q'}(e) = (\_ - t) \land q \) is \( (e)\)-unsafe with respect to \( e \).

Fixing independent variables such as \( e \) in Def. 4 opens up even more variants of safety, similar to the specializations of \( (r)\)-simple-safety presented in [33]. For example, a specialized definition of \( (e,t)\)-safety could target a combination of both an entity leak and a type assignment leak. This task of refining safety definitions is, as above, subject to human interaction that provides additional semantic information about relevant analysis goals.

### 6.3 fDepSearch for SELX

We now tailor fDepSearch to \( (t)\)-safety as an example. The safety definition is the foundation for determining heuristics behavior, which is defined through the model-specific modules introduced in Sec. 5.1. The tailoring of these modules is implemented in the algorithms 3–6, given in Appendix A. For the scope of this paper, we have opted for omitting a detailed discussion of assignParams, which is still subject to ongoing work.

Firstly, the function SELX::tUnsafe::isLeaked (Alg. 3) is a direct implementation of the unsafety property given in Def. 4. The actual dependency analysis is performed by SELX::tUnsafe::buildPredSet (Alg. 4). Based on some CDG vertex \( c = \langle \text{succ}, x_{\text{succ}}, \text{PRE}(\text{succ}), \text{POST}(\text{succ}) \rangle \), the algorithm builds a set

\[
\{ c_{\text{pred}} = \langle \text{pred}, x_{\text{pred}}, \text{PRE}(\text{pred}), \text{POST}(\text{pred}) \rangle \mid \exists t \in T : \text{PRE}(\text{succ}) \text{ and } \text{POST}(\text{pred}) \text{ depend on the value of } t \}
\]

of commands whose prior execution \( c \) depends on. In the actual implementation, label assignment model components (SELX|LA) in pre- and post-conditions are checked to identify dependencies. An auxiliary set \( TDep \) is then used to store and compare types these dependencies relate to.
createCDGSource, implemented as SELX::createCDGSource (Alg. 5), can be inferred easily if multi-level command specification is used to define a distinguished create command: in that case, POST(cmd) in Alg. 5 matches POST(create) in the state transition scheme.

At last, the implementation of createCDGSink as SELX::tUnsafe::createCDGSink (Alg. 6) must be a direct consequence from the implementation of isLeaked.

7 Conclusions

This paper aimed at a uniform formal framework, tailorable for dynamic analysis of a wide range of access control models. The high-level goal behind this work is to demonstrate the practical benefits of combining semantically enhanced modeling patterns, such as EL and core-based modeling, when it comes to model analysis in a holistic security engineering process.

To achieve this, we have adapted the DepSearch heuristic safety analysis algorithm to an access control model for the SELinux operating system. Model-independent interfaces have been introduced into the heuristic, which are then tailored to the SELinux model in a workflow that is guided by semantic knowledge drawn from the modeling patterns. Based on such guiding rules, this workflow is to a significant degree automatable. Because the main portion of human intelligence is required in the early systems design phase of requirements engineering, we argue that this approach reduces later design and verification errors introduced through manually deriving formal methods for these steps.

Subject to future work is a thoroughly formalized, rule-based framework for heuristic tailoring, which enables a prototypical implementation of a model engineering toolkit to automate the process as far as possible. We have also started to evaluate the fDepSearch heuristic for SELX in terms of efficiency and success, which is subject to further improvements that aim at analyzing real-world systems in a productive setting.
A Appendix

Algorithm 3: SELX::tUnsafe::isLeaked

In: \( q = \langle E, cl, con \rangle \) ... model state the safety of which is to be analyzed
\( q' = \langle E', cl', con' \rangle \) ... model state to check for a leak
\( t_{target} \) ... leakage target type
Out: \( \top \) iff \( q \) is \((t)\)-simple-unsafe w.r.t. \( t_{target} \)

for \( e \in E' \cap E \) do
\( \langle \_ \_ \_ t \rangle \leftarrow con(e); \)
\( \langle \_ \_ \_ t' \rangle \leftarrow con'(e); \)
if \( t' = t_{target} \land t \neq t_{target} \) then return \( \top \);
return \( \bot \);

Algorithm 4: SELX::tUnsafe::buildPredSet

In: \( \Delta \) ... model's state transition scheme
\( c = \langle succ, x_{succ}, PRE(succ), POST(succ) \rangle \) ... command whose predecessors are to be found
Out: \( P \subseteq \Delta \) ... set of predecessor commands to \( c \)

\( P \leftarrow \emptyset; \)
\( TDep \leftarrow \emptyset; \)
for \( \phi \in \{ \phi_1 ... \phi_n \mid PRE(succ) = [\phi_1 \land ... \land \phi_n] \} \) do
if \( \phi = \{ con_q(\_ ) = (\_ \_ t) \} \) then \( TDep \leftarrow TDep \cup \{t\}; \)
for \( c_{pred} = \langle pred, x_{pred}, PRE(pred), POST(pred) \rangle \in \Delta \) do
for \( \psi \in \{ \psi_1 ... \psi_n \mid POST(pred) = [\psi_1 \land ... \land \psi_n] \} \) do
if \( \psi = \{ con_{q'} = con_q[\_ \_ \_ (\_ \_ t)] \land t \in TDep \} \) then
\( P \leftarrow P \cup \{c_{pred}\}; \)
return \( P \);

Algorithm 5: SELX::createCDGSource

In: \( q = \langle E, cl, con \rangle \) ... model state the safety of which is to be analyzed
Out: \( cmd \) ... identifier for the virtual CDG source command
\( x_{cmd} \) ... sequence of formal parameters for \( cmd \)
\( PRE(cmd) \) ... pre-condition of \( cmd \)
\( POST(cmd) \) ... post-condition of \( cmd \)

\( cmd \leftarrow "virtualSourceCmd"; \)
\( x_{cmd} \leftarrow \epsilon; \)
\( PRE(cmd) \leftarrow [\top]; \)
\( POST(cmd) \leftarrow [\top]; \)
for \( e \in E \) do
\( \psi_e \leftarrow [E_{q'} = E_q \cup \{e\}] \)
\( \land cl_{q'} = cl_q[e \mapsto cl(e)]; \)
\( \land con_{q'} = con_q[e \mapsto con(e)]; \)
\( POST(cmd) \leftarrow [POST(cmd) \land \psi_e]; \)
return \( cmd, x_{cmd}, PRE_{cmd}, POST_{cmd} \);
Algorithm 6: SELX::tUnsafe::createCDGSink

In: $t_{\text{target}}$ . . . leakage target type
Out: $cmd$ . . . identifier for the virtual CDG sink command

$x_{cmd} \in \Sigma X$ . . . sequence of formal parameters for $cmd$

$\text{PRE}(cmd)$ . . . pre-condition of $cmd$

$\text{POST}(cmd)$ . . . post-condition of $cmd$

$cmd \leftarrow \text{"virtualSinkCmd"};$

$x_{cmd} \leftarrow \epsilon;$

$\text{PRE}(cmd) \leftarrow \left[ \exists e \in E_q : \text{con}_q(e) = \left( - - t_{\text{target}} \right) \right];$

$\text{POST}(cmd) \leftarrow \left[ \top \right];$

return $cmd, x_{cmd}, \text{PRE}_{cmd}, \text{POST}_{cmd};$

References

1. Amthor, P.: A Uniform Modeling Pattern for Operating Systems Access Control Policies with an Application to SELinux. In: Proceedings of the 12th International Conference on Security and Cryptography. pp. 88–99. SECRIPT 2015 (2015), http://dx.doi.org/10.5220/0005551000880099

2. Amthor, P.: E-Business and Telecommunications: 12th International Joint Conference, ICETE 2015, Colmar, France, July 20–22, 2015, Revised Selected Papers, chap. The Entity Labeling Pattern for Modeling Operating Systems Access Control, pp. 270–292. Springer International Publishing, Cham (2016), http://dx.doi.org/10.1007/978-3-319-30222-5_13

3. Amthor, P., Kühnhauser, W.E., Pöck, 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., Pöck, 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. Barker, S.: Personalizing Access Control by Generalizing Access Control. In: Proceedings of the 15th ACM Symposium on Access Control Models and Technologies. pp. 149–158. SACMAT ’10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1809842.1809866

7. Bell, D., LaPadula, L.: Secure Computer System: Unified Exposition and Multics Interpretation. Technical Report AD-A023 588, MITRE (Mar 1976)

8. 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)

9. Crampton, J., Sellwood, J.: Path Conditions and Principal Matching: A New Approach to Access Control. In: Proceedings of the 19th ACM Symposium on Access Control Models and Technologies. pp. 187–198. SACMAT ’14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2613087.2613094
10. 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

11. 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)

12. 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

13. Ferrara, A.L., Madhusudan, P., Parlato, G.: Policy Analysis for Self-administrated Role-Based Access Control. In: Piterman, N., Smolka, S.A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 7795, pp. 432–447. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-36742-7_30

14. 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

15. Grimes, R.A., Johansson, J.M.: Windows Vista Security: Securing Vista Against Malicious Attacks. John Wiley & Sons, Inc., New York, NY, USA (2007)

16. 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

17. Huang, J., Nicol, D.M., Bobba, R., Huh, J.H.: A Framework Integrating Attribute-based Policies into Role-based Access Control. In: Proceedings of the 17th ACM Symposium on Access Control Models and Technologies. pp. 187–196. SACMAT ’12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2295136.2295170

18. Jin, X., Krishnan, R., Sandhu, R.: A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC. In: Cuppens-Boulahia, N., Cuppens, F., Garcia-Alfaro, J. (eds.) Data and Applications Security and Privacy XXVI, Lecture Notes in Computer Science, vol. 7371, pp. 41–55. Springer Berlin Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-31540-4_4

19. Kapuru, 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

20. Künnhauser, W.E., Pöck, 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

21. 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)

22. Pöck, A.: Small TCBs of Policy-controlled Operating Systems. Universitätsverlag Ilmenau (May 2014)

23. Poole, D.L., Mackworth, A.K.: Artificial Intelligence. Cambridge University Press, New York (2010)
24. Ranise, S., Truong, A., Armando, A.: Scalable and precise automated analysis of administrative temporal role-based access control. In: Proceedings of the 19th ACM Symposium on Access Control Models and Technologies. pp. 103–114. SACMAT ’14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2613087.2613102

25. 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

26. Sandhu, R.: The Schematic Protection Model: Its Definition and Analysis for Acyclic Attenuating Schemes. Journal of the ACM 35(2), 404–432 (1988)

27. 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

28. 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

29. Shahen, J., Niu, J., Tripunitara, M.: Mohawk+T: Efficient Analysis of Administrative Temporal Role-Based Access Control (ATRBAC) Policies. In: Proceedings of the 20th ACM symposium on Access Control Models and Technologies. pp. 15–26. SACMAT ’15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2752952.2752966

30. Smalley, S., Craig, R.: Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In: 20th Annual Network & Distributed System Security Symposium (NDSS) (Feb 2013)

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