Command Dependencies in Heuristic Safety Analysis of Access Control Models

Peter Amthor and Martin Rabe

Technische Universität Ilmenau, P.O. Box 100565, 98684 Ilmenau, Germany
{peter.amthor,martin.rabe}@tu-ilmenau.de

Abstract. The principle merits of access control models lie in the ability to precisely reason about their security properties in lineage of the *safety* problem. It formalizes the question if future changes in a model's protection state may eventually violate a security requirement, thereby falsifying model correctness. One fundamental problem of safety analysis is that, as proven in the seminal HRU model calculus, this property is undecidable for the most expressive class of models. To tackle this problem in practical security engineering, a heuristic approach has proven useful that exploits the fact that model commands share dependencies, which are assumed to be (1) one-dimensional and (2) static. In complex models for modern application domains, such as type enforcement in operating systems, both assumptions cannot be made. This paper studies both problems and provides a heuristic solution approach for the problem of dynamic dependencies. Based on our heuristic, we demonstrate the practical impact of this analysis problem and discuss the general implications on model design and analysis strategies.

Keywords: Security policy · security model · access control · model safety · heuristic analysis · operating systems security · security engineering.

1 Introduction

Satisfying security properties of critical software systems relies on a correct security policy. A security policy precisely describes strategies that realize these properties, which makes it an extremely critical engineering artifact. To this end security models are used to guarantee policy correctness.

For decades, such guarantees are based on formal analyses of security models [8, 10, 14, 16, 15, 5, 13, 11] (model-based security engineering). Model analysis aims at confirming formal policy properties, some of which are tractable by static methods, while others require to reason about the dynamic evolution of a system. In the domain of access control (AC) models, such dynamic properties concern the authorization to execute security-critical operations (*commands* in model terms). Analyzing these properties has been termed *security analysis* [16], which can be subdivided in two classes of questions: Given some AC model at a given moment in time (in model terms, a *protection state* to analyze), is it possible (1) that some (desired) property will ever become false; or (2) that some (undesired) property will ever become true? While the first question mainly deals

with availability, the intention of the second question is to validate restrictions on authorized commands which are, for example, demanded by confidentiality or integrity goals of the security policy. For historical reasons, this second family of questions is called *safety* properties.

As has been proven in the seminal HRU model [8], safety properties are generally undecidable for models of unrestricted computational power. To nevertheless take advantage of the high expressiveness of such models, simulative analysis approaches are used, which leverage the semi-decidability of the problem [5, 6, 2]: The dynamic model behavior is implemented, simulated and every protection state change is tested for a possible safety violation. Once any such violation has been found, it can be demonstrated that the given model is *unsafe* for the given protection state. The practical value of such results for security engineers is to provide hints at possible errors in the security policy, by reproducing a sequence of legitimate operations in the underlying system (*command sequence* in the model) which ultimately leads to a safety violation.

In order to efficiently find such errors, simulative analysis is controlled by a heuristic algorithms. It makes assumptions about critical command sequences such that chances of a safety violation as an effect of one of these commands are maximal. For a promising class of heuristics used to date, called DepSearch [5, 2], these assumptions are quite restrictive: they demand a model to be specified in such a manner that only certain classes of causal dependencies between commands may occur. This makes the analysis approach inflexible and, as a consequence, impractical to use for more complex model semantics. Examples for such models are the SELX model for the SELinux operating system [1] or the RePM_G model for an online social network [11]: to cover the complexity of these AC policies, some amount of errors is introduced by rewriting them as a model that is heuristically analyzable. These errors must be balanced against those errors potentially eliminated through formal analysis. Even worse, the interpretation of simulative analysis results – a command sequence in the rewritten model – is hampered by semantical gaps, making it even more burdensome to identify the source of errors in the first place. Hence the goal of this paper is to study the idea of DepSearch heuristics for models in which possible dependencies between commands are less restricted. As a first step we will address the subproblem of dynamic dependencies. We study the costs introduced in terms of runtime overhead and demonstrate their practical impact.

The contributions of this work are (1) a formal generalization of command dependency, including a classification in static/dynamic and one-/multi-dimensional; (2) a formalism to represent dynamic dependencies for general access control models; (3) a heuristic strategy and the specification of an extended heuristic algorithm to analyze models with dynamic dependencies; (4) a study of heuristic runtime properties and generalizable model properties that follow from dynamic dependencies, based on a study of both a synthetic and a practical model.

Paper Organization After discussing related work in the next section, Sec. 3 introduces formalisms and presents the DepSearch heuristic. In Section 4, we theoretically define the generalization of command dependencies and adapt the idea of DepSearch to handle the class of dynamic dependencies. Sec. 5 then discusses the practical impact of this approach on analysis performance, includ-

ing implications on model design. We conclude with a summary of our findings and resulting goals of ongoing and future work in Section 6.

2 Related Work

This work is closely related to both model calculi for safety analysis as well as analysis approaches for tackling the potential undecidability of this problem. Both fields are tightly coupled, as is expressed in the literature: A large body of work related to safety analysis focuses on restricting the computational power of AC models in a non-harmful way w. r. t. some application domains, e. g. fixed-operations Take-Grant models [10], acyclic, ternary MTAM models [14], finite attribute domains in the $PreUCON_A^{finite}$ model [13], or trusted administration in the RePM_G model [11]. All these restrictions lead to models with a less-than-Turing-complete computational power, yet retaining the expressive power sufficient to being useful in their respective application domain.

A fundamentally different, approximative approach is followed by [15, 5, 6, 2]. Here, the goal is to reason about safety without restricting the computational power of a model, thus merely strengthening assumptions about the correctness of policies based on the absence of errors found, while accepting possible non-termination of analysis algorithms. In [7, 4–6, 2], the motivation behind this approach is to deliberately avoid restricting model semantics (and thus analysis strategies) to some application domain, but to provide a pattern for naturally expressing and analyzing arbitrary security policies – including different paradigms for authorization semantics (such as access control, information flow, and non-interference) as well as for model abstractions (such as identity-based, roles-based, attribute-based, or relationship-based access control). Since this motivation aims at a less application-dependent model engineering method, an according formal calculus is needed such as core-based modeling [9, 6, 12] or aspect-oriented modeling [1–3]. Consequently, this model calculus must not assume any restrictions in expressive nor computational power, and so must any safety analysis approach general enough to tackle such models.

This paper describes a semi-decision approach for this which is based on the idea of trading precision for tractability. We build on previous work in the area of heuristic safety analysis by model simulation [5, 6, 2] which aims at falsifying some definition of safety for a given model of a security policy. More precisely, we scrutinize the DepSearch heuristic algorithm introduced there based on a more precise definition of "dependency" and point out its limitations w. r. t. certain properties of a security policy. Our notation for generic access control models is roughly based on similar, conventional notations from [8], [16] and [3].

3 Heuristic Safety Analysis

Before discussing dependency-based heuristic safety analysis, we agree upon a formalism for representing both AC models and their analysis questions in the first part of this section. After this, our existing approach for simulative analysis of HRU safety, called DepSearch, is sketched on a principal level.

Models and Queries To formally express AC policies, a set of formal model components is defined. These typically are either sets of atomic identifiers,

or mappings/relations[1] that associate them with each other in a meaningful way (e. g. to specify authorization rules). We will refer to such components as $A_1 \dots A_n$. An HRU model [8] e. g. contains the components $A_1 = S$ (subjects set), $A_2 = O$ (objects set), $A_3 = R$ (access rights set), and $A_4 = acm : S \times O \to 2^R$ (access control matrix). As a more complex example, a SELX model [1] contains components such as $A_1 = E$ (entities set), $A_2 = T$ (types set), $A_3 = cl$ (entity classification function), $A_4 = con$ (security context function), $A_5 = \hookrightarrow_r$ (role transition relation), or $A_6 = \hookrightarrow_t$ (type transition relation).

For reasoning about dynamic model properties, the definition of components is not sufficient. Instead, we consider these components as a specific view on model engineering, tailored to find the most appropriate and natural formal definitions for the semantics used in an AC policy. Another view, more tailored towards dynamic analysis, has been introduced by [9, 6, 12] and treats an AC system as a deterministic state machine, based on the original idea of [8].

Definition 1 (Dynamic AC Model). *A dynamic access control model is a state machine defined by a tuple* $\langle \Gamma, \Sigma, \Delta \rangle$*, where*

- *the* state space Γ *is a set of protection states;*
- *the* input set $\Sigma = \Sigma_C \times \Sigma_Y^*$ *defines possible inputs that may trigger state transitions, where* Σ_C *is a set of* command *identifiers used to represent operations a policy may authorize and* Σ_Y *is a set of* values *that may be used as actual parameters of commands;*[2]
- *the* state transition scheme (STS) $\Delta \subseteq \Sigma_C \times \Sigma_X^* \times \Phi \times \Phi$ *defines state transition pre- and post-conditions for any input of a command and formal parameters, where* Σ_X *denotes a set of* variables *to identify such parameters.*

We use Φ to represent the set of boolean expressions in first-order logic (without implying any specific language) and $\top$ as a shortcut for boolean *true*.

For defining each $\langle cmd, x, \phi, \phi' \rangle \in \Delta$, a notation borrowed from the classical HRU authorization scheme is used: $cmd(x) ::= \mathsf{PRE} : \phi\,; \mathsf{POST} : \phi'$. We call the boolean term ϕ the pre-condition (abbreviated $cmd.\mathsf{PRE}$) and ϕ' the post-condition (abbreviated $cmd.\mathsf{POST}$) of any state transition to be authorized via cmd. On a state machine level, this means that $cmd.\mathsf{PRE}$ restricts which states γ to legally transition from, while $cmd.\mathsf{POST}$ defines any differences between γ and the state γ' reachable by any input word $\langle cmd, x \rangle$. This matches the intention of the conditions part (indicated by the *if* keyword) and the body (indicated by *then*) of a command in the HRU authorization scheme. Since our goal is to reason about possible state transitions, we adopt the principle of only modeling commands cmd that modify γ, expressed by $cmd.\mathsf{POST} \neq \top$.

To distinguish between the value domains of individual variables in x, we use a refined definition of Σ_X to reflect distinct namespaces of variable identifiers for each model component. These are denoted by sets X_{A_i} so that $\bigcup_{1 \leq i \leq n} X_{A_i} = \Sigma_X$ for a model with n components. For example, an HRU model for an exemplary information system policy may be defined as follows: $\Gamma = 2^S \times 2^O \times ACM$, where $\gamma = \langle S_\gamma, O_\gamma, acm_\gamma \rangle \in \Gamma$ is a single protection state; $\Sigma_C = \{$ createRecord, delegateRead, $\dots \}$; $\Sigma_Y = S \cup O$; $\Sigma_X = X_S \cup X_O$; Δ is defined by a set of definitions as illustrated in Fig. 1a by the example of *delegateRead*.

[1] To formally comply with set algebra, we treat mappings equally to relations.

[2] We use the Kleene operator to indicate that multiple parameters may be passed.

▶ **delegateRead**$(s_{\text{caller}}, s_{\text{deleg}}, o_{\text{rec}})$ ::=
PRE: own $\in acm_\gamma(s_{\text{caller}}, o_{\text{rec}})$
$\wedge$ read $\in acm_\gamma(s_{\text{caller}}, o_{\text{rec}})$;
POST: $acm_{\gamma'} = acm_\gamma[\langle s_{\text{deleg}}, o_{\text{rec}}\rangle$
$\mapsto acm_\gamma(s_{\text{deleg}}, o_{\text{rec}}) \cup \{\text{read}\}]$

(a) *delegateRead* in HRU

▶ **relabel**(e, r', t') ::=
PRE: $e \in E_\gamma \wedge cl_\gamma(e) = \text{process}$
$\wedge\ con_\gamma(e) = \langle u, r, t\rangle$
$\wedge\ r \hookrightarrow_r r' \wedge t \hookrightarrow_t t'$;
POST: $con_{\gamma'} = con_\gamma[e \mapsto \langle u, r', t'\rangle]$

(b) *relabel* in SELX

Fig. 1: Exemplary command definitions.

To define our analysis goal, we express a safety question related to a dynamic AC model as a safety analysis query (or just query):

Definition 2 (Safety Analysis Query). *A safety analysis query q for a given dynamic AC model $\langle \Gamma, \Sigma, \Delta\rangle$ is a tuple $\langle \gamma_0, \tau\rangle$, where $\gamma_0 \in \Gamma$ is a model state and $\tau \in A_i$ is a value of some component A_i.*

The goal of heuristic analysis is to detect a leakage of τ. We define this as a necessary condition for reaching a state γ' that falsifies safety: Any input leading to a successful state transition from γ to γ' as defined by Δ is called *leaking* τ if τ appears in some model component in γ', where it not also appears in γ. In case such state transition exists, we say γ' contains a *leakage* of τ.

In contrast to a mere leakage, our definition of safety adheres to the widespread interpretation that any state γ_τ reachable from γ_0 renders the former unsafe iff τ was entered into a set, matrix, relation etc., which did not already have this value in γ_0 (also called *simple-safety* [17, 2]). In particular, if an input sequence re-enters τ into the same component that already contained τ in γ_0, safety is not violated. The reason why we nevertheless aim at a *leakage* in heuristic analysis is that it can be easily detected (by comparing γ' with γ after any state transition), though we still need to subsequently falsify safety with respect to γ_0.

Heuristic Strategies The objective of a heuristic safety analysis strategy is to demonstrate the occurrence of a leakage. When the strategy finds an input sequence that eventually leaks τ in a state γ_τ, we may prove γ_0 to be unsafe with respect to τ; as long as no γ_τ is found, the search continues. Therefore, the chances of any single input to contribute to such a sequence must be maximized.

When simulating model behavior, test inputs are generated and the states reached via them are tested for a leakage of τ. For generating each input, a heuristic has to choose a command to execute and value assignments for its variables. In our previous work, we have identified command dependencies as a promising model property to maximize chances for a successful input sequence. This is the basis of the DepSearch strategy for HRU safety analysis [5].

DepSearch was developed based on the insight that in the hardest case, right leakages appear only after long state transition sequences where each command executed depends exactly on the execution of its predecessor. Essentially, the algorithm consists of two phases: In the first phase, a static analysis of the STS is performed. It yields a formal description of command dependencies, constituted by entering (as a part of POST) and requiring (part of PRE) the same right in two different commands. These dependencies are encoded in a *command dependency graph* (CDG) whose nodes are commands, and any edge from c to c' denotes that c' depends on the execution c:

Definition 3 (HRU CDG). *A command dependency graph (CDG) of an HRU model* $\langle \Gamma, \Sigma, \Delta \rangle$ *is an edge-weighted, directed multigraph* $\langle V, E \rangle, E \subseteq V \times V \times R$*, such that* $V \subset \Sigma_C$ *is the set of command identifiers and* $\langle c, c', r \rangle \in E$ *if a term in* $c.\mathsf{POST}$ *enters* r *in* $acm_{\gamma'}$ *and a term in* $c'.\mathsf{PRE}$ *requires* r *in* acm_{γ}*.*

The CDG is assembled such that all paths from nodes without incoming edges to nodes without outgoing edges indicate input sequences for reaching γ_τ from γ_0. To achieve this, two virtual commands c_0 and c_τ are generated: c_0 is the source of all paths in the CDG, since it mimics the state γ_0 to analyze, represented by a virtual command specification in Δ such that $c_0.\mathsf{POST}$ requires all subjects in S_{γ_0}, all objects in O_{γ_0}, and all rights in acm_{γ_0}. In a similar manner, c_τ is the destination of all paths, which represents all possible states γ_τ; $c_\tau.\mathsf{PRE}$ hence requires the presence of the target right τ in some matrix cell of acm_γ:

▶ $\mathbf{c_0}() ::=$	▶ $\mathbf{c}_\tau(s, o) ::=$
PRE: $\top$;	PRE: $\tau \in acm_\gamma(s, o)$;
POST: $S_{\gamma'} = S_{\gamma_0} \wedge O_{\gamma'} = O_{\gamma_0} \wedge acm_{\gamma'} = acm_{\gamma_0}$	POST: $\top$

In the second, simulative analysis phase, the CDG is used to generate input sequences. The commands in each sequence correspond to different paths from c_0 to c_τ, which we expect to leak τ once completely executed. In case a sequence fails because of an unsatisfiable PRE, another path is selected based on the last successfully reached state. This strategy is based on the assumption that even a partially executed command sequence contributes to pre-conditions of any next sequence generated. Each effected state transition is simulated by the algorithm, and once a CDG path is completed, the falsification of safety is checked.

4 Command Dependencies

To heuristically analyze more complex model semantics such as for the SELinux AC model SELX [1], rewriting the STS to HRU syntax is impractical, error-prone, and the results of the analysis may not be interpretable w. r. t. the underlying system. To this end, we generalize the definition of command dependency beyond entering and requiring access rights in an ACM cell. We then adapt the DepSearch idea to handle the more general class of dynamic dependencies.

4.1 Problem Analysis

HRU has two distinct properties that enable the static pre-analysis of STS commands in DepSearch: (1) dependencies are created solely by entering or requiring a right, (2) all rights are fixed by command definitions, i. e. static values during model simulation. Both properties enable DepSearch to create the CDG (Def. 3) as a representation of static command dependencies. Assume a model where dependencies between commands originate from multiple model components, represented in the STS by variables whose values are dynamically assigned during runtime. Since this model violates above properties, the static dependency analysis in DepSearch does no longer produce significant results.

To clarify these differences consider our exemplary HRU command *delegateRead* (Fig. 1a) and the command *relabel* of a SELX model (Fig. 1b): As becomes evident in *delegateRead*, PRE depends only on the presence of the right values "own" and "read" in some matrix cell, since a conjunction of such conditions is the only allowed PRE in HRU. In *relabel* however, a conjunction of more

heterogeneous conditions relate to different components of a SELX model: first, a set E_γ and a mapping cl_γ are checked for the presence of a process, second, a mapping con_γ is used to lookup security attributes, third, two relations $\hookrightarrow_r$ and $\hookrightarrow_t$ are checked to validate the requested relabeling.[3] The fact that values from a total of five different model components (entities, classes, three attributes) are checked violates property 1, the fact that all these values but one are represented by variables violates property 2.

Intuitively, applying a dependency analysis approach such as DepSearch to a model such as SELX raises both formal and semantical issues, notably related to how dependencies are formalized through a CDG. We will discuss these issues in the following based on an exemplary HRU STS:

$\blacktriangleright$ $\mathbf{c_1}(s_1, s_2, o)$ ::=
PRE: read $\in acm_\gamma(s_1, o)$;
POST: $acm_{\gamma'} = acm_\gamma[\langle s_2, o\rangle \mapsto acm_\gamma(s_2, o) \cup \{\text{read}\}]$

$\blacktriangleright$ $\mathbf{c_2}(s_1, s_2, o)$ ::=
PRE: read $\in acm_\gamma(s_1, o)$;
POST: $acm_{\gamma'} = acm_\gamma[\langle s_2, o\rangle \mapsto acm_\gamma(s_2, o) \cup \{\text{write}\}]$

If DepSearch would use this STS to create a CDG with the target right "write", the result would be a CDG as shown in Fig. 2a. In case the STS violates property 2, e. g. it contains only right variables, this would result in the graph in Fig. 2b. It is important to note that, despite showing dependencies in both cases, these graphs differ in edge semantics: the edges of the CDG in Fig. 2a represent actual dependencies, the edges in the graph in Fig. 2b represent potential dependencies. This explains why the graph in Fig. 2b contains more edges and thus more potential paths from c_0 to c_τ.

(a) static (b) dynamic

Fig. 2: Graphs for different types of dependencies.

Assume a STS that violates property 1 as follows: It contains a command that creates a certain subject in POST but does not enter any rights. Assume further that the only command that leaks our target right requires (aside from certain rights) the existence of the aforementioned subject in PRE. A CDG created by DepSearch would not contain the first command since DepSearch only checks dependencies regarding rights. The result would be that the command leaking our target right could never be executed since the command that would create the needed subject is not part of the CDG and thus also never part of any path generated.

4.2 Classes of Dependencies

We now generalize our observations regarding HRU made in the previous examples. On a more formal level, both properties mentioned there may be used in a

[3] For the sake of a more concise discussion, we ignore the SELinux-concept of *entrypoints*.

heuristic to recognize different types of dependency. This intention is reflected in the following definitions, which express *necessary* conditions for one command to require a previous execution of another command. We assume that any single value or variable used in a PRE or POST term is significant for its boolean value, as e. g. achieved by canonical CNF.

Definition 4 (Static Dependency). *Let $c_1, c_2 \in \Sigma_C$ be two commands in a dynamic AC model. If c_2 statically depends on c_1, then there is a model component A_i such that a value $y \in A_i$ occurs in both $c_1.\mathsf{POST}$ and $c_2.\mathsf{PRE}$.*

This type of dependency is attributed *static* since actual values must be matched to identify a dependency relationship. Likewise, we may also observe *dynamic dependency* based on variables that *potentially* match, depending on their dynamically assigned values:

Definition 5 (Dynamic Dependency). *Let $c_1, c_2 \in \Sigma_C$ be two commands in a dynamic AC model. If c_2 dynamically depends on c_1, then there is a model component A_i such that a variable $x_1 \in X_{A_i}$ occurs in $c_1.\mathsf{POST}$ and a variable $x_2 \in X_{A_i}$ occurs in $c_2.\mathsf{PRE}$.*

If more than one distinct model component A_i satisfies Def. 4 or Def. 5, we speak of *multi-dimensional* dependencies (*one-dimensional* otherwise).

These definitions basically yield four classes of possible models, each with different implications on a correct and efficient representation of dependencies as a heuristic criterion for safety analysis: such where (1) only static and one-dimensional, (2) only static but both one- and multi-dimensional, (3) static and dynamic, but only one-dimensional, and (4) all types of dependencies may occur.

It should be highlighted that HRU is already a model from the last class: the presence of a subject $s \in S_\gamma$ is a necessary condition for satisfying a PRE expression $r \in acm_\gamma(s, o)$. As a simple example, consider a special case of HRU whose STS features solely static values for subjects. In this case, executing any command that requires a right r assigned to a subject s depends both on any command that enters r and on any command that creates s. We hence observe multi-dimensional dependencies, which are – in case of general HRU models – also partially dynamic. However, DepSearch makes implicit assumptions about model semantics that allow the CDG to ignore both dynamic and multi-dimensional dependencies: PRE terms in HRU do not allow to directly check for the presence of subjects or objects, but only indirectly as having some rights assigned via *acm*. Based on this observation, DepSearch interprets the matrix as a more fine-grained rights set, which is therefore considered the only dependency-relevant model component. Only subject and object variables are allowed, while rights are always static in an HRU STS.

To this end, our existing heuristic algorithm is only capable to correctly handle static, one-dimensional command dependencies. To get a first understanding of the implications of these classes for models where they cannot be neglected, we focus on dynamic dependencies in the following, which are commonly found in todays AC policies (cf. Sec. 5.2).

4.3 Dealing with Dynamic Dependencies

To isolate the problem of dynamic dependencies, we start with a more concise model notation: we modify the original HRU syntax in a way that only dynamic,

one-dimensional dependencies can be expressed. Based on this model we address dependencies with a generalized CDG, termed pCDG, which is an overapproximation of any possible CDG. A CDG created from this graph, which we call a pCDG instance, may then be analyzed using DepSearch [7, 2].

As discussed in the last section, dynamic dependencies are observable in HRU. To make them more explicit however, we define a very simple calculus for dynamic AC models, which is used to isolate the relevant phenomena of one-dimensional, yet dynamic dependencies. The goal here is to have this type of dependency, as defined in Sec. 4.2, directly reflected in command definitions, while eliminating any syntax and semantics beyond it (e. g. the ACM).

Despite the discussion is based on HRU terminology, we are only interested in the impact of rights on establishing dynamic dependencies. This leads to the class of HRU* models defined as follows:

Definition 6 (HRU*). *An HRU* model is a dynamic AC model $\langle \Gamma, \Sigma, \Delta \rangle$ with a single component $A_1 = R$ and $\Gamma = 2^R$, $\Sigma_Y = R$, $\Sigma_X = X_R$.*

Note that HRU* may be expressed by an access control matrix consisting of a single cell, but not in an HRU model due to the difference in Σ_X. In the following we use HRU* models for different sets Σ_C. Consequently, for the sake of readability, we abbreviate the notation of Δ by only listing right values and right variables in PRE and POST, respectively:

Example 1. For an HRU* model with $R = \{\text{spam}, \text{ham}, \text{eggs}, \text{beans}\}$ and $\Sigma_C = \{c_1, c_2, c_3\}$, Δ is defined as

▶ $\mathbf{c_1}(r_1, r_2)$ ::=	▶ $\mathbf{c_2}(r_1, r_2)$ ::=	▶ $\mathbf{c_3}(r_1, r_2)$ ::=
PRE: r_1, r_2 ;	PRE: r_1 ;	PRE: r_1 ;
POST: spam	POST: r_1, r_2	POST: r_2

For globally unique variable identifies, we will write $c_1.r_1$ for variable r_1 in command c_1. We then write $c_1.X_{\mathsf{PRE}} = \{c_1.r_1, c_1.r_2\}$ for the set of all variables in pre-conditions of c_1 and likewise $c_1.X_{\mathsf{POST}} = \emptyset$ for its post-conditions.[4] The global set of right variable identifiers is denoted by $X_R = \{c_1.r_1, c_1.r_2, c_2.r_1, c_2.r_2, c_3.r_1, c_3.r_2\}$.

Based on the DepSearch idea, dynamic dependencies are encoded in a graph. For this we introduce the abstraction of dependency variables:

Definition 7 (pCDG). *A potential command dependency graph (pCDG) of a dynamic AC model $\langle \Gamma, \Sigma, \Delta \rangle$ is an edge-weighted, directed multigraph $\langle V_p, E_p \rangle$, $E_p \subseteq V_p \times V_p \times VAR$, such that $V_p = \Sigma_C$ is the set of command identifiers and VAR is a set of dependency variables based on Σ_X.*

A pCDG is constructed just as an HRU CDG, with the only difference that the original dependency condition (cf. Def. 3) is modified according to Def. 5: Any edge $\langle u, v, x_{uv} = \langle x_u, x_v \rangle \rangle$ now means that assigning some value to variable $x_u \in u.X_{\mathsf{POST}}$ implies that command v potentially satisfies its pre-condition based on some value assigned to variable $x_v \in v.X_{\mathsf{PRE}}$. This means that for any CDG containing this edge, both values must be the same. We call the tuple x_{uv} a *dependency variable* (an alias for two variable identifiers, i. e. $VAR \subseteq X_R \times X_R$).

Due to the abstraction of *potential* dependency represented in a pCDG, static dependencies according to Def. 4 cannot be captured by its edges. However, our approach is to first model potential dependency in the pCDG, which

[4] Note that "spam" is a right value, not a variable.

are mapped to actual dependencies in a CDG in the next step. To this end, static dependencies have to be considered in pCDG creation as if it were potential dependencies: By substituting each value used in command definitions by a unique synthetic variable, such as r_{spam} for "spam" in the example. When later assigning values to variables, we require that those synthetically introduced may only be assigned one fixed value (we henceforth call them *fixed-value variables*). This approach enables us to deal with an STS containing mixed dynamic and static dependencies. For the above Example 1, this leads to $VAR = \{x^1_{11} = \langle c_1.r_{\text{spam}},\, c_1.r_1\rangle, x^2_{11} = \langle c_1.r_{\text{spam}}, c_1.r_2\rangle\,, x^1_{12} = \langle c_1.r_{\text{spam}}, c_2.r_1\rangle\,, x^1_{13} = \langle c_1.r_{\text{spam}},\, c_3.r_1\rangle, x^1_{21} = \langle c_2.r_1, c_1.r_1\rangle\,, x^2_{21} = \langle c_2.r_2, c_1.r_1\rangle\,, \dots\}$.

The resulting graph is designed to contain every possible edge of any CDG that may be built from the STS, hence the attribute "potential". This implies that a pCDG is always a complete graph, additionally including all possible self-loops. Fig. 3 shows the pCDG resulting from the STS in Example 1.

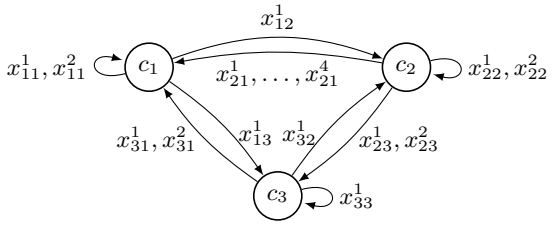


Fig. 3: pCDG for Example 1. Multi-edges are denoted by the union of their dependency variables.

By Def. 7, pCDG creation is independent from any analysis query $q = \langle \gamma_0, \tau\rangle$. However, for answering such queries, we first need to break down its semantics to solely represent *actual* dependencies which we are able to heuristically analyze. This step is called *instantiation*.

The goal of pCDG instantiation is to enable heuristic search strategies to infer command sequences (paths in this graph) just as with a regular CDG. As a consequence, the resulting graph needs to satisfy necessary properties of a CDG. From these requirements, we derive the approach of instantiation: first, nodes for c_0 and c_τ are added and connected to ensure the presence of paths significant for q; second, a variable assignment function is found that substitutes dependency variables by specific values; third, validity rules for a CDG are applied to restrict E_p based on this assignment. The result is called an *instance* of a pCDG:

Definition 8 (pCDG Instance). *A* pCDG *instance is an edge-weighted, directed multigraph* $\langle V, E\rangle\,, E \subseteq V \times V \times R$ *created from a* pCDG $\langle V_p, E_p\rangle$ *via a query* $\langle \gamma_0, \tau\rangle$ *and an assignment function* $\mathcal{I} : VAR \to R$*, where* $V = V_p \cup \{c_0, c_\tau\}$.

To meet the base assumption of our heuristics, that traversing certain paths in a CDG subsequently establishes necessary conditions for a right leakage, we define the nodes c_0 and c_τ as source and destination of any such path. This is done based on γ_0 and τ, in a similar manner as with an HRU CDG (cf. Def. 3).

Example 2. Given the STS in Example 1, a state $\gamma_0 = \{\text{ham}, \text{beans}\}$ and a target $\tau = \text{eggs}$. We then need to add two commands c_0 and c_τ defined as:

▶ $\mathbf{c_0}()$ ::= PRE: $\top$; POST: ham, beans ▶ $\mathbf{c}_\tau()$ ::= PRE: eggs ; POST: $\top$

Note that, before performing the variable assignment, fixed-value variables ($c_0.r_{\text{ham}}, c_0.r_{\text{beans}}, c_\tau.r_{\text{eggs}}$ in the example) have to be introduced as already done for those commands in Σ_C. Because of its significance in terminating any CDG

path, we will refer to the single variable in $c_\tau.X_{\mathsf{PRE}}$ as r_τ. We eventually connect both additional nodes with those in V_p using the same dependency condition as during pCDG construction: By introducing edges and dependency variables for $c_0.X_{\mathsf{POST}}$ and any $c.X_{\mathsf{PRE}}$, and any $c.X_{\mathsf{POST}}$ and $c_\tau.X_{\mathsf{PRE}}$, respectively.

Then, for assigning values to dependency variables, an $\mathcal{I}$ is defined such that any variable $x_{uv} = \langle x_u, x_v \rangle$ mapped to some right value $\mathcal{I}(x_{uv}) = y$ implies that both x_u and x_v are assigned y. This leads to an edges set E annotated with rights instead of variables. Since for any pCDG, a multitude of assignment functions may produce different instances, care must be taken in handling fixed-value variables. Since their assignment is constant over all $\mathcal{I}$, we model it through an auxiliary alias function: Any $\mathcal{I}$ is from a function space; any fixed-value variable is from an identifier set Σ_X^{syn}. We require for all $\mathcal{I}$ and $x^{syn} \in \Sigma_X^{syn}$ that $\mathcal{I}(\langle x_u, x^{syn} \rangle) = \mathcal{I}(\langle x^{syn}, x_v \rangle) = \mathcal{C}(x^{syn})$ where $\mathcal{C} : \Sigma_X^{syn} \to R$ is the alias function, independent from any $\mathcal{I}$, which assigns constant values to fixed-value variable identifiers. To ensure variable assigment validity, $\mathcal{I}$ must satisfy three validation rules:

$$\langle u, c_\tau, \mathcal{I}(x_{u\tau}) \rangle \in E \Rightarrow \mathcal{I}(x_{u\tau}) = \tau \tag{1}$$

$$\langle c_0, v, \mathcal{I}(\langle x_0, x_v \rangle) \rangle \in E \Rightarrow \mathcal{I}(\langle x_0, x_v \rangle) = \mathcal{C}(x_0) \tag{2}$$

$$\begin{aligned} &\langle u, v, \mathcal{I}(x_{uv}) \rangle, \langle u', v', \mathcal{I}(x'_{uv}) \rangle \in E, \\ &\langle u, v, x_{uv} \rangle \approx_n \langle u', v', x'_{uv} \rangle \Rightarrow \mathcal{I}(x_{uv}) = \mathcal{I}(x'_{uv}) \end{aligned} \tag{3}$$

where $\approx_n \subseteq E_p \times E_p$ is the edge neighborship relation. Two pCDG edges are neighbors iff they represent the same variable of an incident node:

$$\begin{aligned} \langle u, v, \langle x_u, x_v \rangle \rangle \approx_n \langle u', v', \langle x'_u, x'_v \rangle \rangle \Leftrightarrow (u = u' \wedge x_u = x'_u) \vee (u = v' \wedge x_u = x'_v) \\ \vee (v = u' \wedge x_v = x'_u) \vee (v = v' \wedge x_v = x'_v). \end{aligned}$$

Validation rule 1 ensures that the target value is leaked after executing a path from c_0 to c_τ; rule 2 ensures that pre-conditions of commands on any such path can be satisfied by values already present in γ_0. Note that by design of the instantiation approach – through the definitions of c_0, c_τ and $\mathcal{C}$ – we have already satisfied both rules. Rule 3 ensures that any two different dependency variables representing the same variable in the STS are assigned the same value.

Example 3. In the pCDG in Fig. 3, the dependency variable x^1_{12} is an alias for $c_1.r_{\text{spam}}$ and $c_2.r_1$. Likewise, x^1_{23} is an alias for $c_2.r_1$ and $c_3.r_1$. Let $\mathcal{I}$ be an assignment function that instantiates this pCDG, then $\mathcal{I}(x^1_{12}) = \mathcal{C}(c_1.r_{\text{spam}}) =$ spam since $c_1.r_{\text{spam}}$ is a synthetic variable for the static right "spam". Assume that $\mathcal{I}(x^1_{23}) =$ beans: from rule 3, we now infer that not both edges $\langle c_1, c_2, x^1_{12} \rangle$ and $\langle c_2, c_3, x^1_{23} \rangle$ must be present in E since otherwise, $\mathcal{I}$ is contradictory w. r. t. the value assigned to $c_2.r_1$.

Any edge that violates assignment validity must be removed from E. In case there are multiple candidates for removal, such as in Example 3, this offers room for heuristically optimizing CDG properties that are more prospective for efficiently producing a leak; this problem is subject to ongoing work.

After removing invalid edges, the resulting graph may become partitioned – in the worst case with c_0 and c_τ in separate partitions. To exclude such cases from the analysis, any pCDG instance must be validated against a connectivity

rule in order to be analyzable in the same manner as a traditional CDG: A pCDG instance $\langle V, E\rangle$ is a CDG iff

$$v \in V \setminus \{c_0, c_\tau\} \Rightarrow v \text{ is on a path from } c_0 \text{ to } c_\tau. \tag{4}$$

After the CDG is validated, any further pre-processing may be performed before running DepSearch. Especially, efficiency optimizations through statically analyzable leakage properties as we have described in [2] may be applied after this step.

4.4 Path Search

The original idea of DepSearch is to simulate a sequence of commands where each command creates a necessary condition for the execution of the next, to ultimately cause a right leakage. This sequence is created by searching a path from c_0 to c_τ in the CDG.

The pCDG introduced in Def. 7 represents potential dependencies. The approach described above is to instantiate it in a manner that results in a CDG, which may then be analyzed using DepSearch. This approach has the drawback that paths used for the analysis always stem from the same CDG, which can be mitigated by creating different CDGs and switching between them during simulation. As already mentioned, a criterion for comparing CDGs to decide which is more prospective for leaking the target right is subject to ongoing work.

We therefore decided to perform a path search directly in the pCDG. For this, c_0 and c_τ are added to the graph (the result is referred to as pCDG$^+$). Then all dependency variables of this path are assigned in such a way that assignment validity is satisfied.[5] Afterwards the corresponding command sequence can be used to analyze the model for a right leakage. This approach allows to evaluate the path in terms of its prospects of success during the runtime of the analysis. Fig. 4a shows a possible path of the pCDG$^+$ created from the pCDG in Fig. 3.

(a) path from c_0 to c_τ (b) instance of the path from c_0 to c_τ

Fig. 4: Path instantiation examples.

The dependency variables of this path are now assigned in such a way that the assignment validity is satisfied, e. g. $\mathcal{I}(x^1_{01}) = \text{ham}, \mathcal{I}(x^1_{13}) = \text{spam}$ and $\mathcal{I}(x^1_{3\tau}) = \text{eggs}$. With this assignment, the path shown in Fig. 4a can be instantiated which results in the path shown in Fig. 4b. This path can then be used by existing analysis tools [6]. If no right leakage is found using this path, we may either change the assignment function or search a new path in the pCDG$^+$.

5 Impact on Safety Analysis

In this section we study the implications of dynamic dependencies on the efficiency of safety analyses. To this end, in Sec. 5.1, we discuss both properties of the heuristic algorithm and representative model properties of HRU*. In Sec. 5.2 we then show that the latter are indeed properties of practical models, which supports their significance. This will be done in the context of SELX, a model designed to enable safety analyses of SELinux operation systems.

[5] All right variables not assigned in this step may be randomly assigned with values.

5.1 HRU*

This section determines model properties relevant for analysis efficiency. Based on the runtime complexity of the algorithms and the influence of command definitions a worst-case STS is specified, which we then use to demonstrate the actual impact on analysis runtime.

Runtime Complexity The generation of the pCDG$^+$ has a runtime complexity of $\mathcal{O}(n^2)$, where $n = |\Sigma_C|$.[6] The path search runs in $\mathcal{O}(m)$, where m is the number of edges in the pCDG$^+$. Since m depends on $|\Sigma_C|^2$, the runtime complexity is $\mathcal{O}(n^2)$. The edge neighborship check has a runtime complexity of $\mathcal{O}(m^2) = \mathcal{O}(n^4)$ and the creation of the assignment function has a runtime complexity of $\mathcal{O}(m) = \mathcal{O}(n^2)$. This confirms that the number of commands $|\Sigma_C|$ has the greatest impact on analysis runtime.

STS Specification We now discuss how the definition of commands affects analysis efficiency. To this end we examine relevant types of commands in HRU*. First of all, consider an STS without static dependencies. From this STS, commands that only have $\top$ in PRE can be ignored, since they always lead to a right leakage and can be found statically. Thus, for the evaluation model, only commands with both PRE and POST other than $\top$ are of interest. Such commands can be divided into two classes: (1) every right variable that occurs in POST is also checked in PRE, and (2) at least one right variable in POST does not occur in PRE. Examples for both classes are the following c (class 1) and c' (class 2):

► $\mathbf{c}(r)$::= PRE: r ; POST: r ► $\mathbf{c'}(r_1, r_2)$::= PRE: r_1 ; POST: r_2

Commands of class 1 cannot cause a right leakage, since they always enter the same rights in POST that were checked in PRE. Thus, if the variable r is assigned with τ, the target right must already exist in the current state for this command to be executable. This in turn means that another command must already have entered τ and the analysis is thus already terminated.[7] This is the reason why these commands can be ignored for this evaluation. Commands of class 2 however may immediately lead to a right leakage, since all variables are freely assignable: a variable in POST can be assigned with τ and all variables in PRE can be assigned with rights that exist in the current state. In this respect, commands of class 2 resemble commands that have only $\top$ in PRE. This illustrates a property of models which have a command of class 2 that makes it possible to circumvent the simulative analysis, since these commands can be found with a static preanalysis.

Because of this we use a model with both static and dynamic dependencies for the runtime measurements:

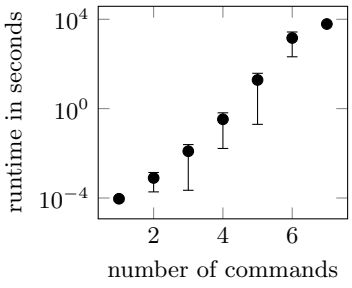
► $\mathbf{c_1}()$::=	► $\mathbf{c_2}()$::=	► $\mathbf{c_3}()$::=		► $\mathbf{c_8}(r)$::=
PRE: 1 ;	PRE: 2 ;	PRE: 3 ;	...	PRE: 8 ;
POST: 2	POST: 3	POST: 4		POST: r_2

[6] The runtime is also influenced by the number of right variables ($|X_{\mathsf{PRE}} \cup X_{\mathsf{POST}}|$). However for a static set R this means that: $\forall c \in \Sigma_c : |c.X_{\mathsf{PRE}}| \leq |R|$ (analogous for $c.X_{\mathsf{POST}}$). Therefore this impact can be assumed as constant.

[7] Note that this implication of the command classification holds for HRU* only.

$\lvert\Sigma_C\rvert$	Runtime	Class
1	0,0001 s	A
2	0,0007 s	
3	0,0124 s	
4	0,3335 s	
5	19,198 s	B
6	1451,4652 s ($\approx$ 24 min)	
7	6133,83 s ($\approx$ 102 min)	C

(a)



(b)

Fig. 5: Evaluation results (error bars: standard deviation).

To perform the simulative analysis the definition of the component R and the query $\langle \gamma_0, \tau \rangle$ are required: $R = \{1, 2, 3, 4, 5, 6, 7, 8, 42\}, \gamma_0 = \{1\}$ and $\tau = 42$.

Model Simulation This model was analyzed for right leakages. In order to be able to represent the runtime behavior with increasing number of commands, we performed analyses of models with one through eight commands. These models fulfill the same assumptions as the one specified above, i. e. they have both static and dynamic dependencies. Fig. 5a shows the runtime measurements of these analyses. For models with $|\Sigma_C| \leq 5$ the specified values are averaged from 100 runs, for $|\Sigma_C| = 6$ from 10 runs, and for $|\Sigma_C| = 7$ one run was performed. The models were divided into runtime classes (class A for < 1 s, class B for < 1 h and class C for ≥ 1 h). Fig. 5b illustrates these results. The runtime was measured from the generation of the pCDG$^+$ to the successful finding of a right leakage.[8]

Discussion of Results As becomes evident from Fig. 5, analyzing models with both dynamic and static dependencies drastically increases the runtime. While for classes A and B a sensible runtime may be achieved, the time required for class C exceeds one hour, already at $|\Sigma_C| = 7$. This demands an alternative approach for such models e. g. via the static classification introduced above.

5.2 SELX

In this section, a practical system will be used to demonstrate that the model property discussed above, i. e. the existence of commands with no static dependencies and different variables in PRE and POST, is in fact a property of practical models. To this end we specify a model SELX* which mimics the behavior of a type transition in a SELX model. In SELX the safety property is not focused on whether a right is leaked, but whether a type is leaked (*(t)-simple-unsafety* [3, Def. 5.5]). This is due to the fact that e. g. processes have no rights, but types directly assigned while access rights are assigned to these types. Processes are represented in the entities set E and the types in the types set T. The association of entities to types is the so called security context, which is modeled via the function $con : E \to T$. We define the SELX* model as follows:

Definition 9 (SELX*). *A SELX* model is a dynamic AC model $\langle \Gamma, \Sigma, \Delta \rangle$ with the components $A_1 = E$, $A_2 = T$ and $A_3 = acm : E \times E \to T$. It is defined as follows: $\Gamma = 2^E \times ACM$, where for any $\gamma = \langle E_\gamma, acm_\gamma \rangle \in \Gamma$, $acm_\gamma : E_\gamma \times E_\gamma \to T$, $\Sigma_Y = E \cup T$ and $\Sigma_X = X_E \cup X_T$.*

[8] A machine with an Intel i5 CPU at 2.90GHz and 8GB RAM was used.

	sh	usr_t	admin_t
sh	usr_t		
usr_t			$\top$
admin_t		$\top$	

(a) acm_γ

▶ **relabel**(p_1, t_1, t_2) ::=
PRE: $t_1 \in acm_\gamma(p_1, p_1)$
$\wedge\ \top \in acm_\gamma(t_1, t_2)$;
POST: $acm_{\gamma'} = acm_\gamma[\langle p_1, p_1\rangle \mapsto \{t_2\}]$

(b) *relabel* basic command

Fig. 6: SELX* model specifications.

E and T are encoded in acm_γ as follows: $p_1 \in E_\gamma, t_1 \in T : acm_\gamma(p_1, p_1) = \{t_1\} \Leftrightarrow con_\gamma(p_1) = t_1$ specifies which type is the active type of a process and $t_1, t_2 \in T : acm_\gamma(t_1, t_2) = \{\top\} \Leftrightarrow t_1 \hookrightarrow_t t_2$ specifies which type transitions are allowed. Fig. 6a shows an example matrix containing the entity sh $\in E_\gamma$ with usr_t $\in T$ assigned as the active type. There is another type admin_t $\in T$ and a transition is allowed from usr_t to admin_t, but not from each type to itself.

There are fixed basic commands in the SELX model for describing protection state changes. When adapted to SELX*, only the basic command *relabel* (Fig. 6b) may fulfill (t)-simple-unsafety, since it allows a process to transition to a new type. As becomes evident, this command has a right variable in POST that does not occur in PRE, so it has the same properties as commands of class 2 in HRU*. As established in the last section, such a definition always leads to a right leakage, or in this context to the model being (t)-simple-unsafe and can be found statically. The conclusion is that practical models may indeed have commands that have the same properties as class 2 commands of HRU*.

6 Conclusions and Future Work

We studied command dependencies in AC models, focused on the class of dynamic dependencies, for which we presented a safety analysis algorithm. The definition of dependency was generalized and formalized, resulting in a graph abstraction which enables a heuristic search to steer a simulation.

This heuristic serves as a basis to study the practical impact of dynamic dependencies. Our findings show that (1) the runtime complexity mainly depends on the number of commands in the STS, (2) the generalized dependency definition increases runtime complexity of the simulation to $\mathcal{O}(n^4)$ compare to $\mathcal{O}(n)$ in DepSearch. The severe increase demonstrates that dynamic dependencies are a serious obstacle to heuristic safety analysis. To mitigate this conclusion in practice, we have identified model properties that circumvent the need for simulation, since they can be found statically. While this result in principle strengthens the semi-decision approach, it might as well indicate that traditional safety properties to describe an undesired model state are too unspecific. As a consequence, general queries like "leaking of a certain right" might not be significant in models featuring dynamic dependencies. In models where dependency based analysis turns out to be fundamentally inefficient due to dynamic dependencies, we may now at least identify such cases based on a formal criterion.

Our immediate ongoing work extends this study to multi-dimensional dependencies, as well as to explore more fine-grained safety queries and additional heuristic information that allow for efficient analyses of well-defined cases of command dependencies.

References

1. Amthor, P.: The Entity Labeling Pattern for Modeling Operating Systems Access Control. In: Obaidat, S.M., Lorenz, P. (eds.) E-Business and Telecommunications:

12th International Joint Conference, ICETE 2015, Colmar, France, July 20–22, 2015, Revised Selected Papers, pp. 270–292. Springer International Publishing, Cham (2016), http://dx.doi.org/10.1007/978-3-319-30222-5_13
2. Amthor, P.: Efficient Heuristic Safety Analysis of Core-based Security Policies. In: Proc. 14th International Conference on Security and Cryptography. pp. 384–392. SECRYPT 2017 (2017), http://dx.doi.org/10.5220/0006477103840392
3. Amthor, P.: Aspect-oriented Security Engineering. Cuvillier Verlag, Göttingen, Germany (2019), iSBN 978-3-7369-9980-0
4. Amthor, P., Kühnhauser, W.E., Pölck, A.: Model-based Safety Analysis of SELinux Security Policies. In: Samarati, P., Foresti, S., Hu, J., Livraga, G. (eds.) In Proc. of 5th Int. Conference on Network and System Security. pp. 208–215. IEEE (2011)
5. Amthor, P., Kühnhauser, W.E., Pölck, A.: Heuristic Safety Analysis of Access Control Models. In: Proc. 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
6. Amthor, P., Kühnhauser, W.E., Pölck, A.: WorSE: A Workbench for Model-based Security Engineering. Computers & Security **42**(0), 40–55 (2014). https://doi.org/http://dx.doi.org/10.1016/j.cose.2014.01.002, http://www.sciencedirect.com/science/article/pii/S0167404814000066
7. Fischer, A., Kühnhauser, W.E.: Efficient Algorithmic Safety Analysis of HRU Security Models. In: Katsikas, S., Samarati, P. (eds.) Proc. International Conference on Security and Cryptography (SECRYPT 2010). pp. 49–58. SciTePress (2010)
8. 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
9. Kühnhauser, W.E., Pölck, A.: Towards Access Control Model Engineering. In: Proc. 7th Int. Conf. on Information Systems Security. pp. 379–382. ICISS'11, Springer-Verlag, Berlin, Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-25560-1_27
10. Lipton, R.J., Snyder, L.: A Linear Time Algorithm for Deciding Subject Security. Journal of the ACM **24**(3), 455–464 (1977)
11. Masoumzadeh, A.: Security Analysis of Relationship-Based Access Control Policies. In: Proc. 8th ACM Conference on Data and Application Security and Privacy. pp. 186–195. CODASPY '18, ACM, New York, NY, USA (2018), http://doi.acm.org/10.1145/3176258.3176323
12. Pölck, A.: Small TCBs of Policy-controlled Operating Systems. Universitätsverlag Ilmenau (May 2014)
13. Rajkumar, P.V., Sandhu, R.: Safety Decidability for Pre-Authorization Usage Control with Finite Attribute Domains. IEEE Transactions on Dependable and Secure Computing **13**(5), 582–590 (Sept 2016). https://doi.org/10.1109/TDSC.2015.2427834
14. Sandhu, R.S.: The Typed Access Matrix Model. In: Proc. 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
15. 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)
16. Tripunitara, M.V., Li, N.: A theory for comparing the expressive power of access control models. J. Comput. Secur. **15**(2), 231–272 (Apr 2007), http://dl.acm.org/citation.cfm?id=1370659.1370662
17. 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