Efficient Heuristic Safety Analysis of Core-based Security Policies

Peter Amthor
Ilmenau University of Technology, Germany

Keywords: Security Engineering, Security Policies, Access Control Systems, Access Control Models, Safety, Heuristic Analysis, SELinux.

Abstract: Being of paramount importance for the correctness of a security policy, the property of *safety* has received decades of attention in the field of model-based security engineering. To analyze the safety of a security model, heuristic approaches are used to avoid restrictions of the model calculus while accepting semi-decidability of this property. Within this field, this paper addresses three open problems concerning the DEPSEARCH heuristic safety analysis framework: Inefficient state-space exploration, static verification of unsafety-unsatisfiability, and parameter dependency analysis. We describe these problems on a formal basis, specify solution proposals, and implement these in the current, model-independent *f*DS framework. A practical evaluation based on SELinux is performed to study effectiveness and future optimization of the framework.

1 INTRODUCTION

While security requirements of IT systems have become of paramount importance, the tasks of rigorous specification, verification, and implementation of security properties have received increasing attention (Sandhu, 1988; Kleiner and Newcomb, 2007; Stoller et al., 2011; Ranise et al., 2014; Shahen et al., 2015). The problems of systems security specification and implementation have found a solution in policy-controlled systems: Here, a security policy is a precise and unambiguous definition of rules, e. g. related to isolation, access control, or information flow control mechanisms, which forms the backbone for reliable and correct enforcement of these rules. Consequently, solving the problem of verification inevitably requires to formally analyze precisely that backbone (a security model) for properties such as policy consistency, information flow leaks, or potential proliferation of privileges.

Considering the importance of such security properties, the fundamental *safety* undecidability result has been relevant since its first presentation in (Harrison et al., 1976). The seminal HRU model for dynamic access control systems is introduced there, along with the *safety* property describing potential proliferation (also known as *leakage*) of privileges. Informally, a snapshot of an access control system is considered safe with respect so some privilege if and only if no future modification to that snapshot is able to allow an access using the privilege in question, which was not allowed in the initial snapshot.

Given its critical nature, the proof of undecidability of this problem in (Harrison et al., 1976) led to massive efforts in reducing a model's complexity and thus expressive power in a least-obstructive way (Sandhu, 1988; Sandhu, 1992; Stoller et al., 2011; Ferrara et al., 2013; Ranise et al., 2014; Shahen et al., 2015). Most related work in this area features models decidable with respect to a restricted flavor of HRU safety which, being constructed with a specific application domain in mind, completely satisfies the semantic needs of that domain's policies. However, since all these models significantly differ in their semantics, analysis algorithms derived for them are usually not applicable to other policy domains, which in turn need to be built from scratch. Because of the huge semantic gap between an informal policy and the formalized model, this is an error-prone process.

Our current work aims at overcoming this problem. Instead of restricting a model's expressive power to achieve a decidable flavor of safety, our approach accepts the impossibility of confirming a *safe* model. We instead try to confirm the critical case of safety *violation* in unrestricted HRU models (which is possible due to the actual semi-decidable nature of the problem). This is done in an approximative, heuristics-based approach resulting in the DEPSEARCH (DS) algorithm (Amthor et al., 2013). To apply this approach to a broader range of models,

Amthor, P.
Efficient Heuristic Safety Analysis of Core-based Security Policies.
DOI: 10.5220/0006477103840392
In *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications (ICETE 2017) - Volume 4: SECRYPT*, pages 384-392
ISBN: 978-989-758-259-2

we have merged it with two generalized *modeling patterns* (core-based modeling (Kühnhauser and Pölck, 2011; Pölck, 2014) and entity labeling (Amthor, 2015; Amthor, 2016)). The resulting *f*DS framework allows to tailor heuristic safety analysis to a multitude of policy semantics beyond traditional HRU (Amthor, 2017).

However, despite promising first results, our safety analysis algorithms still expose weaknesses with respect to both heuristic effectivity and efficiency. Based on our previous work and some critical observations made there, the goal of this paper is to take the next step towards practice by presenting *f*DS++, an optimized framework for safety analysis based on DEPSEARCH. In details, we have identified and solved the following weaknesses:

1. Inefficient heuristic for state-space exploration, leading to a significant portion of ineffective heuristic steps;
2. Negligence of static model properties influencing *unsafety-satisfiability*, which effectively allows to identify known-safe policies and to avoid heuristic analysis where unnecessary;
3. Lack of support for parameter assignment algorithms (which contribute to an efficient state space traversal) based on static dependency analysis.

We have specified these improvements and implemented the resulting algorithms tailored to SELX, an access control model for the SELinux operating system (Amthor, 2015) (Section. 4). As a preliminary proof-of-concept validation that complements ongoing experimentation, we outline an exemplary analysis of an excerpt of the SELinux reference policy (Section 5). Our results hint at a feasibility of the generalized safety analysis approach in practice and indicate the direction of ongoing and future work in that field.

2 DYNAMIC SECURITY MODELS

The goal of this section is to introduce the core-based modeling pattern by Pölck (Kühnhauser and Pölck, 2011; Pölck, 2014) as a formal environment to describe our safety analysis approach. Here and for the rest of this paper, we will use the following conventions for formal notation:

$\models$ is a binary relation between variable assignments and formulas in second-order logic, where $I \models \phi$ iff I is an assignment of unbound variables to values that satisfies ϕ. In an unambiguous context, we will write $\langle x_0, \ldots, x_n \rangle \models \phi$ for any assignment of variables x_i in ϕ that satisfies ϕ. 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 $\phi = [\![y = \underline{x}]\!]$. $\mathbb{B}$ is the set of Boolean values $\top$ (*true*) and $\bot$ (*false*). In a directed graph $\langle V, E \rangle$, E_v^{in} denotes the set of incoming edges of a node v, while E_v^{out} denotes its set of outgoing edges.

The Model Core. To provide a uniform formal basis for safety definition and dynamic model analysis, we use a generalization of the deterministic automaton introduced in HRU models. This automaton serves as an abstract pattern used to define specialized policies in a formally uniform calculus.

A core-based access control model is defined as an automaton, the *model core*:

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

where Q is a set of protection states, Σ is a set of inputs, $\delta : Q \times \Sigma \to Q$ is the state transition function, $\lambda : Q \times \Sigma \to \mathbb{B}$ is the output function, $q_0 \in Q$ is the initial protection state, and EXT is an arbitrary tuple of static model extensions.

The model core is tailored to a domain-specific security policy by defining Q (a system's dynamic protection state) and EXT (static portion of the policy). To describe protection state dynamics for a particular policy, the state transition function δ is defined via pre- and post-conditions of any possible state transition. This is done by comparing each input of the automaton with two formulas in second-order logic, PRE and POST. We then define δ by formally specifying the conditions that each pair of states q and q' has to satisfy w.r.t. an input $\sigma \in \Sigma$ for a state transition from q to q' to occur:

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

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 $\Sigma = \Sigma_C \times \Sigma_Y$, where Σ_C is a set of command identifiers and Σ_Y contains sequences of possible values (actual command parameters) for variables in PRE and POST. Corresponding to Σ_Y, Σ_X is a set of formal command parameters, i. e. variable names used in PRE and POST. An engineering-friendly, conventional notation of δ is then based on partial definitions for each input command: We call the set of such partial definitions

$\Delta = \{\langle cmd, x_{cmd}, \mathsf{PRE}(cmd), \mathsf{POST}(cmd)\rangle \,|\, cmd \in \Sigma_C; x_{cmd} \in \Sigma_X; \mathsf{PRE}(cmd), \mathsf{POST}(cmd) \in \mathbb{B}\}$ a model's *state transition scheme*, a specification of the behavior of δ.

In the core-based pattern, a HRU model is defined as $\langle Q, \Sigma, \delta, \lambda, q_0, \langle R\rangle\rangle$, with a protection state defined as $q = \langle S_q, O_q, acm_q\rangle \in Q$. We call R the set of access rights, S_q the subjects set, O_q the objects set, and $acm_q : S_q \times O_q \to 2^R$ the access control matrix of state q, the latter defining authorization rules in form of Lampson's ACM (Lampson, 1971). The notation of a simple example command *delegateRead* of an HRU model's state transition scheme is shown in Fig. 1.

▶ **delegateRead**(s_1, s_2, o) ::=
PRE: $read\_right \in acm_q(s_1, o)$;
POST: $acm_{q'}(s_2, o) = acm_q(s_2, o) \cup \{read\_right\}$

Figure 1: Exemplary command definition for a core-based HRU model: *delegateRead* with parameters $\langle s_1, s_2, o\rangle$ models delegation of *read_right* regarding o by s_1 to s_2. PRE(*delegateRead*) requires that s_1 possesses this right for a state transition by this command to occur, POST(*delegateRead*) requires s_2 to do so afterwards.

3 SAFETY ANALYSIS

In this section, we discuss the basic idea of heuristic safety analysis. We will outline our most analysis strategy, DEPSEARCH, along with the algorithm for HRU safety analysis. This will be the foundation for a subsequent discussion of algorithmic optimizations. We start by defining the notion of privilege proliferation introduced by HRU safety:

As formalized for the HRU model, 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 definition of this question, *(r)-simple-safety*, has been presented by Tripunitara and Li (Tripunitara and Li, 2013):

Definition 1. Given a core-based HRU model $\langle Q, \Sigma, \delta, \lambda, q_0, \langle R\rangle\rangle$, 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 $\exists q' = \langle S_{q'}, O_{q'}, acm_{q'}\rangle \in \{\delta^*(q,a) \,|\, a \in \Sigma^*\}$:

$$\exists s \in S_{q'}, \exists o \in O_{q'} : r \in acm_{q'}(s,o)$$
$$\wedge \quad \big(s \notin S_q \vee o \notin O_q \vee r \notin acm_q(s,o)\big)$$

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

As per this definition, *safety* 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*.

Heuristic State-Space Exploration. In order to handle the undecidability of HRU safety, we choose to trade accuracy for tractability: using a heuristic, 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_{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. A successful heuristic must therefore maximize the probability of an input to contribute to a path from q to q_{target}.

The DEPSEARCH Approach. We developed DS based on the insight that in the most difficult case, right leakages in a model appear only after specific state transition sequences where each command executed depends exactly on the execution of its predecessor. From this observation we derived the idea of a two-phases algorithm: First, a static analysis of inter-command dependencies is performed; a subsequent, dynamic analysis consists of state space exploration by simulating the automaton's behavior. We will discuss these phases on an informal basis, for an in-depth discussion and evaluation of DS for HRU see (Amthor et al., 2013; Amthor et al., 2014).

During the first phase, a **static analysis** of the HRU state transition scheme is performed. It yields a graph-based 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. As a result of the static analysis, these dependencies are modeled by a *command dependency graph* (CDG) $\langle V, E\rangle$ where nodes $c_1, c_2 \in V$ represent commands, and an edge $\langle c_1, c_2\rangle$ denotes that a post-condition of c_1 matches at least one pre-condition of c_2. An example of a CDG is depicted in Fig. 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 Δ. It is generated by encoding acm_q in $\mathsf{POST}(c_q)$. 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 $\mathsf{PRE}(c_{target})$.

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}, which in turn determine the direction of state space exploration. Paths in the CDG are generated based on a simple ant algorithm: with each edge traversed to generate an input sequence, the algorithm increases an edge weight "scent" which has repellent effect for the next iteration of the CDG traversal, thus effectively leading to a full path coverage and potentially maximal chances for actually generating a right leakage.

DS successively generates input sequences by traversing the CDG on every possible path and in turn parameterizing the emerging sequence of commands with values from Σ_Y. Each effected state transition is simulated by the algorithm, and once a CDG path is completed, the unsafety-criteria (cf. Def. 1) is checked.

4 *f*DS IMPROVEMENTS

Three major problems affecting both efficiency and effectivity of the original DS have become evident during its usage:

Efficient State-Space Exploration: DS uses an ant algorithm for CDG path generation, which ultimately determines efficiency and effectivity of state-space exploration. As a closer look on more complex state transition schemes reveals, its partially randomized strategy is not optimal, possibly resulting in high rates of unsatisfiable simulation paths for such CDGs.

Static Satisfiability Analysis: A mere dependency analysis does not exploit particular static properties of a state transition scheme leading to leakage-free policies, which we call *unsafety-unsatisfiable* (i. e. provable safe). An intuitive example indicating an unsafety-unsatisfiable state transition scheme is the existence of cycles in the CDG.

Parameter Dependency Analysis: The assignment of state-specific values to parameters in the state transition scheme is a highly policy-specific task. However, since the inference of command sequences from mutual execution dependency as one portion of the automaton's input proved a successful idea, we expect significant improvements during the dynamic analysis phase by restricting the domain of possible parameter values in a similar manner.

The original DS was restricted to HRU models. In (Amthor, 2017), we generalized the approach to *f*DS, which can be tailored to a wide range of core-based models using generic interfaces. We have shown this principle based on the SELinux model SELX. However, since *f*DS does not yet address the above efficiency problems, we will discuss them more detailed in the following and present solution proposals.

4.1 Efficient State-space Exploration

In *f*DS, state space exploration is driven by input sequences generated by the heuristic. These sequences are controlled by the CDG, which is successively traversed with the goal of maximum path diversity in mind; however, this strategy disregards the influence of path ordering. In a worst-case dependency situation, where each command requires execution of *all* predecessors (i. e., each incoming edge of each CDG node stands for a different dependency), this may lead to the traversal algorithm not generating a minimal sequence of paths, but preferring redundant or unsatisfiable paths over necessary ones. The reason for this is the initially equal chance for all in-edges of c_{target} to be chosen; the algorithm does not state any deterministic criteria for preferring one of them.

Consider the example in Fig. 2. Based on the particular order of choosing between equally-weighted edges, the minimal number of paths needed to execute c_6 varies from 2 to 5 (unsatisfiable commands in parentheses):

optimal:
$c_q \to c_1 \to c_2 \to c_4 \to (c_6 \to c_t)$
$c_q \to c_1 \to c_3 \to c_5 \to c_6 \to c_t$

worst case:
$c_q \to c_1 \to (c_4 \to c_6 \to c_t)$
$c_q \to c_1 \to (c_6 \to c_t)$
$c_q \to c_1 \to (c_5 \to c_6 \to c_t)$
$c_q \to c_1 \to c_2 \to c_4 \to (c_6 \to c_t)$
$c_q \to c_1 \to c_3 \to c_5 \to c_6 \to c_t$

In order to prioritize such paths more likely to be executed, nodes with low in-degree in the CDG should be traversed before such with higher in-degree. This leads to paths whose dependencies can be satisfied early on, and which thus produce more successful state transitions.

Solution. Our modified algorithms for path generation (Alg. 1) and CDG assembly (Alg. 2) adapt edge weights ("scents") according to the above idea:

Algorithm 1: *f*DS++::CDGPathGeneration

Input: $\langle V,E\rangle$ ... CDG as generated by Alg. 2
$d,\hat{d}$ ... edge weighting/increment function of $\langle V,E\rangle$
c_q ... CDG source node
Output: *path* ... path in the CDG
d ... modified edge weights

function lowestScent(**in** $E_v \subseteq E$)
  $e_{min} \leftarrow E_v.someMember$;
  $minw \leftarrow d(e_{min})$;
  for $e_v \in E_v$ **do**
    if $minw > d(e_v)$ **then**
      $e_{min} \leftarrow e_v$;
      $minw \leftarrow d(e_{min})$;
  return e_{min};

$v \leftarrow c_q$, $path \leftarrow v$;
repeat
  $\langle u,v,i\rangle \leftarrow$ lowestScent(E_v^{out});
  $path \leftarrow path \circ v$;
(1) $d(\langle u,v,i\rangle) \leftarrow d(\langle u,v,i\rangle) + \hat{d}(\langle u,v,i\rangle)$;
until $E_v^{out} = \emptyset$;
return $path, d$;

Algorithm 2: *f*DS++::CDGAssembly

Input: Δ ... model's state transition scheme
Output: $\langle V,E\rangle$ ... command dependency graph
$d,\hat{d} : E \to \mathbb{N}$ ... edge weighting/increment function
c_q ... CDG source node

procedure predecessors(**in** $v \in V$)
(3) (c) $P \leftarrow$ buildPredSet(Δ, v);
  for $\langle c,i\rangle \in P$ **do**
    if $c \notin V$ **then**
      $V \leftarrow V \cup \{c\}$;
      predecessors(c);
    $E \leftarrow E \cup \{\langle c,v,i\rangle\}$;
(1) $d(\langle c,v,i\rangle) \leftarrow |P|$;
(2) $\hat{d}(\langle c,v,i\rangle) \leftarrow |P|$;

(d) $c_q \leftarrow$ createCDGSource(q);
(e) $c_{target} \leftarrow$ createCDGSink(*target*);
$\Delta \leftarrow \Delta \cup \{c_q\}$;
$V \leftarrow \{c_{target}\}$;
$E \leftarrow \emptyset$;
predecessors(c_{target});
return $\langle V,E\rangle, d, \hat{d}, c_q$;

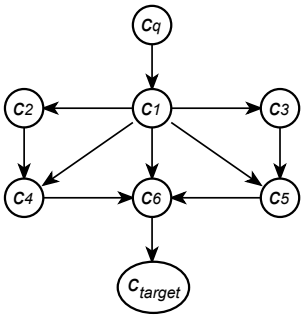


Figure 2: A CDG to illustrate the path generation problem.

The edge weighting function d is initialized on CDG assembly, where the cardinality of each node's set of predecessors determines the weight of all its incoming edges (see Alg. 2, line (1)). During path generation, edge weights serve as "scents" as usual, while an additional increment function $\hat{d}$ is used to modify them when passing an edge (Alg. 1, line (1)). Setting an edge increment to the respective edge's initial weight during CDG assembly (Alg. 2, line (2)) ensures that the intended traversal frequency is maintained for the rest of the dynamic analysis.

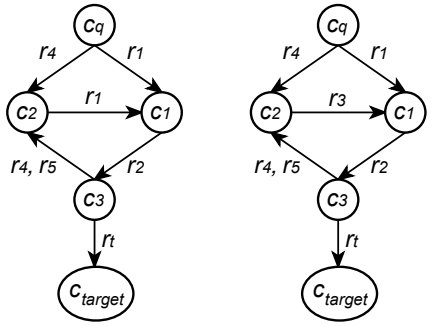
In order to satisfy as many dependencies as possible as early as possible, it is advantageous for Alg. 1 to traverse the CDG in direction of its edges, starting at c_q (instead of in reverse, starting with c_{target}).

4.2 Static Satisfiability Analysis

For policies with a leakage-free state transition scheme, safety might be statically inferred from reachability properties of the CDG (e. g. indicating cyclic dependencies); however, the current definition of the CDG does not allow this: Semantics of an edge $\langle c_1,c_2\rangle$ only denote that "c_1 establishes at least one condition necessary for executing c_2" (cf. (Amthor et al., 2013)) – but not how many and which particular conditions. Fig. 3(b) shows an HRU-based example of such a graph, with the specific rights that establish the respective dependency added to each edge.

As indicated above, unsatisfiable dependencies can be statically identified by refining edge semantics in the CDG. For this we use an edge-colored multigraph $\langle V,E\rangle, E = V \times V \times \mathbb{N}$, where $\langle u,v,i\rangle \in E \Leftrightarrow u$ establishes a precondition of v whose color is i. In HRU for example, $1 \le i \le |R|$ would cover all possible precondition colors (matching HRU rights). In SELX, assuming *(t)-safety* (Amthor, 2017), colors $1 \le i \le |T|$ match SELinux types.

We will now define *potential satisfiability* of a CDG as a goal of static satisfiability analysis: Using this definition, a costly, heuristic state-space exploration can be avoided for such state transition schemes



(a) potentially satisfiable (b) unsatisfiable

Figure 3: Fine-grained edges semantics for CDG satisfiability analysis: In case (a), conditions of c_1 may be satisfied by c_q, while in case (b), $c_{1..3}$ all depend on each other's prior execution in a cycle.

Algorithm 3: *f*DS++::unsatisfiable

Input: $CDG = \langle V, E \rangle$
Output: $\top$ if CDG is unsatisfiable; $\bot$ if CDG is potentially satisfiable

```
function isPSat(in v ∈ V)
    if v = c_q then return ⊤ ;
    if v ∈ H then return ⊥ ;
    H ← H ∪ {v};
    for ⟨u, v, i⟩ ∈ E_v^in do
        for ⟨u', v, j⟩ ∈ E_v^in do
            if i = j then
                return isPSat(u) ∨ isPSat(u')
            else
                return isPSat(u) ∧ isPSat(u')
H ← ∅;
return ¬ isPSat(c_target);
```

which do not allow for c_{target} to be reachable under any dynamic conditions.

Definition 2. Potential Satisfiability of a CDG:

1. A CDG node v is potentially satisfiable iff $\forall u, u' \in V, \{\langle u, v, i\rangle, \langle u', v, j\rangle\} \subseteq E_v^{in}$:
   $i = j \Rightarrow u$ is potentially satisfiable $\vee\, u'$ is potentially satisfiable
   $i \neq j \Rightarrow u$ is potentially satisfiable $\wedge\, u'$ is potentially satisfiable.
2. c_q is potentially satisfiable.
3. A CDG is potentially satisfiable iff c_{target} is potentially satisfiable.

Solution. To implement our extended CDG definition, we had to modify CDG assembly: the function *buildPredSet* (Alg. 2, line (3)) now returns a set of command-color-pairs, allowed to include multiple pairs for the same predecessor command. For a core-based model with variables set *Var*, it is defined as

$$buildPredSet_{\mathsf{SELX}}(\Delta, c_{succ}) = \{\langle c_{pred} \in \Delta, i \in \mathbb{N}\rangle \mid \exists x_i \in Var : \mathsf{PRE}(c_{succ}) \text{ and } \mathsf{POST}(c_{pred}) \text{ depend on the assignment of } x_i\}$$

where *Var* is defined as the set of variables in the state transition scheme, which are used as formal parameters of some command. It holds $\Sigma_X = Var^*$.

As per CDG construction, there is a path from each node to c_{target}. This implies we can check the conditions in Def. 2 in a backwards-depth-first search, starting from c_{target}. Based on this strategy, our static satisfiability analysis is implemented in Alg. 3 and called right before the dynamic analysis phase in Alg. 5, line (1).

4.3 Parameter Dependency Analysis

In order to assist a model-specific parameter selection heuristic, domains of possible values for parameter variables should be as small as possible. This can also be achieved by a more fine-grained static dependency analysis: Parameter selection can be mapped to a constraint satisfaction problem (CSP), and thus tackled by a CSP solver algorithm. If we take a basic arc consistency algorithm such as AC-3 (Mackworth, 1977), an additional data structure is needed to represent variables, domains, and constraints of the CSP: the parameter constraints network (PCN).

The PCN is defined as an undirected, bipartite, and node-labeled graph $PCN = \langle V \cup Cons, E \subseteq V \times Cons, DOM\rangle$ where $V \subseteq Var$ is a set of variables, *Cons* is a set of constraints (logical formulas), and $DOM = \{dom^k : Var^k \rightarrow 2^{Val^k} \mid 1 \leq k \leq K\}$ is a set of domain-labeling functions for each variable. Here, *Val* is the set of all possible values that could be assigned to a parameter, corresponding to *Var*. It also constitutes $\Sigma_Y = Val^*$ in a core-base model; for HRU e. g. $Val = S \cup O$, for SELX $Val = E \cup C \cup P \cup U \cup R \cup T$. K denotes the number of value domains, which is derived from the number of basic sets for values: for HRU it holds $K = 2$, for SELX $K = 6$.

Solution. The PCN is assembled by subdividing edges in the CDG, als specified in Alg. 4. We initialize our PCN with empty sets, and iteratively extend it by transforming each edge $\langle c_1, c_2, i\rangle$ in the CDG into a number of constraint nodes (line (1)): one for each precondition in c_2 that matches CDG edge color i. Each such constraint is a disjunctive formula including one clause for each potentially i-satisfying state

Algorithm 4: fDS++::PCNAssembly

Input: $CDG = \langle V_{CDG}, E_{CDG} \rangle$
Output: $PCN = \langle V \cup Cons, E, DOM \rangle$

$V \leftarrow \emptyset$, $Cons \leftarrow \emptyset$, $E \leftarrow \emptyset$, $DOM \leftarrow \emptyset$;
(6) **for** $1 \leq k \leq K$ **do**
(1) **for** $\langle c_1, c_2, i \rangle \in E_{CDG}$ **do**
$P^k_{src} \leftarrow \{x \in Var^k \mid$ POST(c_1) depends on x and $i\}$;
$P^k_{sink} \leftarrow \{x \in Var^k \mid$ PRE(c_2) depends on x and $i\}$;
for $x \in P^k_{sink}$ **do**
if $x \notin V$ **then**
$dom^k(x) \leftarrow Val^k$;
(3) $V \leftarrow V \cup \{x\}$;
$exp \leftarrow [\![\bot]\!]$;
for $x' \in P^k_{src}$ **do**
(2) $exp \leftarrow [\![\underline{exp} \vee (\underline{x} = \underline{x'})]\!]$;
(4) $E \leftarrow E \cup \{\langle x', exp \rangle\}$;
(5) $E \leftarrow E \cup \{\langle x, exp \rangle\}$;
$Cons \leftarrow Cons \cup \{exp\}$;
$DOM \leftarrow DOM \cup \{dom^k\}$;
return $\langle V \cup Cons, E, DOM \rangle$;

manipulation in POST(c_1), which is iteratively assembled in line (2). We then add all variables in these formulas as nodes to the PCN (line (3)), each connected with the previously created constraint nodes (lines (4), (5)) according to arc consistency semantics. To clearly separate and thus minimize the different domains of variables, the whole process is repeated for each domain (line (6)).

During dynamic analysis, right before a command sequence is attempted to be executed, we run the CSP solver on the PCN, yielding a (possibly pruned) domain for each variable x which in turn modifies the initial PCN for the next iteration. We then select an assignment for each variable based on a policy-specific heuristic. We have abstracted this policy-specific parameter assignment heuristic, which is assumed to call a CSP solver as a subroutine, through the interface *assignParams* (Alg. 5, line (2)).

4.4 fDS++ Heuristic

In Alg. 5, we have specified the improved fDS++ algorithm, using Algs. 2, 3, 4, and 1. Arabic labels in the algorithms denote references throughout this chapter, while latin labels indicate abstract interfaces that must be implemented based on the particular core-based/EL model to analyze: *assignParams* (a) implements a parameter assignment heuristic, which is based on a momentary state, a command sequence

Algorithm 5: fDS++

Input: δ ... model's state transition function
Δ ... model's state transition scheme, specifying δ
q_0 ... model state to be analyzed
target ... leakage target
Output: *seq* ... states sequence leaking *target*

$q \leftarrow q_0$, $seq \leftarrow \varepsilon$;
$\langle CDG, d, \hat{d}, c_q \rangle \leftarrow$ CDGAssembly$(\Delta, q, target)$;
if unsatisfiable(CDG) **then**
return *seq*;
else
$seq \leftarrow q$;
$PCN \leftarrow$ PCNAssembly(CDG);
repeat
$\langle path, d \rangle \leftarrow$ CDGPathGeneration$(CDG, d, \hat{d}, c_q)$;
(2) (a) $\langle assignment, PCN \rangle \leftarrow$ assignParams$(q, path, PCN)$;
while $c \leftarrow path.nextNode$ **do**
$q' \leftarrow \delta(q, c, assignment(c))$;
$seq \leftarrow seq \circ q'$;
$q \leftarrow q'$;
until (b) isLeaked$(q_0, q', target)$;
return *seq*;

to be executed, and a PCN defining domains of possible values for each variable used as a parameter in the input command sequence. It returns an assignment function for these parameters and a modified PCN with updated (and thus minimized) domains. *isLeaked* (b) implements a safety definition (e. g. Def. 1), which is evaluated to a boolean value based on some initial state, some reached state to check for a leakage, and *target*. *buildPredSet* (b) checks model-specific dependencies in Δ to return the set of predecessor nodes, along with their respective dependency color, based on some given node in the CDG. *createCDGSource* (d) creates a definition of c_q based on q_0. *createCDGSink* (e) creates a definition of c_{target} based on *target*.

(Amthor, 2017) shows how to implement these interfaces based on the example of SELX.

5 ONGOING EVALUATION

This section describes ongoing and future work regarding an experimental evaluation of fDS++. Using SELX-implementations of the interfaces listed in Sec. 4.4, we have implemented Alg. 5 and performed an exemplary *(t)-safety* analysis of an SELinux policy excerpt as a first proof-of-concept application.

One major prerequisite for applying the heuristic is model instantiation. Here, we have reused existing tools and methods for extracting both an initial state space and static policy rules. These methods are described in detail in (Amthor, 2016) and include (1) a file system crawler that extracts protection state information from a file system of a virtual machine snapshot, (2) a policy scanner, which parses the SELinux policy specification language and produces a SELX model implementation, (3) a simulation runtime environment (*WorSE*, cf. (Amthor et al., 2014)) that implements the automaton's state transition scheme.

We specified our state transition scheme using SELX basic commands, as described in (Amthor, 2016), which led to a expectable, trivial CDG consisting solely of *create* and *relabel*. Composing these basic commands and taking into account other than only type-dependencies leads to a more complex graph. Note that in this specific analysis scenario, inter-command dependencies are solely type variables – leading to a runtime-specific CDG, whose dynamic reanalysis has not been subject of our heuristic yet (as already pointed out in Sec. 4.2).

Moreover, focusing on rather complex policy semantics as in SELinux emphasizes the impact of multiple dependency classes: in SELX these are, for example, type-, role-, class-, or user-dependency, all of which must be represented in the CDG. For the sake of clearly implementing and presenting the algorithmic concepts, we have so far opted for a one-dimensional solution, taking into account solely type-dependencies. Since our approach is approximative, this delivers valid results, yet open to further optimization in terms of fine-grained policy semantics.

Both problems, dynamic CDGs and complex dependencies, indicate the direction of our future research in heuristic safety analysis. The immediate next step, an experimental, comparative evaluation of the improvements of *f*DS++ in terms of heuristic steps count and heuristic runtime, is subject ongoing work regarding practical systems analyses.

6 CONCLUSIONS

In this paper, we have presented solutions to three open problems with heuristic safety analysis: First, we identified an efficiency-enhanced path generation scheme for state-space exploration, which prefers such commands with less dependencies required to be executed, and derived an edge-weighting scheme for its implementation. Second, we defined the unsafety-unsatisfiability property of a state transition scheme in terms of command reachability (which also means potential executability). We then specified an algorithm for a static analysis that may identify policies satisfying this property as safe, eliminating the need for a heuristic simulation in such cases. Third, a static support algorithm for model-specific parameter assignment heuristics was described. The idea is basically identical to that of inter-command dependencies, which is implemented leveraging a constraint satisfaction problem solver (an arc consistency algorithm was proposed for this).

As a proof of concept, we have presented practical insights given by an analysis of an SELinux security policy based on SELX, which pave the way for current and future research focusing multiple dependency classes in complex policies and dynamic command dependency graphs.

REFERENCES

Amthor, P. (2015). A Uniform Modeling Pattern for Operating Systems Access Control Policies with an Application to SELinux. In *Proc. 12th Int. Conf. on Security and Cryptography*, SECRYPT 2015, pp. 88–99.

Amthor, P. (2016). *E-Business and Telecommunications: 12th Int. Joint Conf., ICETE 2015, Colmar, France, July 20–22, 2015, Revised Selected Papers*, chapter The Entity Labeling Pattern for Modeling Operating Systems Access Control, pp. 270–292. Springer.

Amthor, P. (2017). Towards a Uniform Framework for Dynamic Analysis of Access Control Models. In *Proc. 10th International Symposium on Foundations & Practice of Security*, FPS 2017. (under review)

Amthor, P., Kühnhauser, W. E., and Pölck, A. (2013). Heuristic Safety Analysis of Access Control Models. In *Proc. 18th ACM Symposium on Access Control Models and Technologies*, SACMAT '13, pp. 137–148. ACM.

Amthor, P., Kühnhauser, W. E., and Pölck, A. (2014). WorSE: A Workbench for Model-based Security Engineering. *Computers & Security*, 42(0):40–55.

Ferrara, A. L., Madhusudan, P., and Parlato, G. (2013). Policy Analysis for Self-administrated Role-Based Access Control. In Piterman, N. and Smolka, S. A., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 7795 of *LNCS*, pp. 432–447. Springer.

Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. (1976). Protection in Operating Systems. *Communications of the ACM*, 19(8):461–471.

Kleiner, E. and Newcomb, T. (2007). On the Decidability of the Safety Problem for Access Control Policies. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 185:107–120.

Kühnhauser, W. E. and Pölck, A. (2011). Towards Access Control Model Engineering. In *Proc. 7th Int. Conf. on Information Systems Security*, ICISS'11, pp. 379–382.

Lampson, B. W. (1971). Protection. In *5th Ann. Princeton Conf. on Information Sciences and Systems*, pp. 437–443. Reprinted Jan, 1974: Protection. In *Operating Systems Review*, 8(1), pp. 18–24.

Mackworth, A. K. (1977). Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118.

Pölck, A. (2014). *Small TCBs of Policy-controlled Operating Systems*. Universitätsverlag Ilmenau.

Ranise, S., Truong, A., and Armando, A. (2014). Scalable and precise automated analysis of administrative temporal role-based access control. In *Proc. 19th ACM Symposium on Access Control Models and Technologies*, SACMAT '14, pp. 103–114. ACM.

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

Sandhu, R. S. (1992). The Typed Access Matrix Model. In *Proc. 1992 IEEE Symposium on Security and Privacy*, SP '92, pp. 122–136. IEEE.

Shahen, J., Niu, J., and Tripunitara, M. (2015). Mohawk+T: Efficient Analysis of Administrative Temporal Role-Based Access Control (ATRBAC) Policies. In *Proc. 20th ACM Symposium on Access Control Models and Technologies*, SACMAT '15, pp. 15–26. ACM.

Stoller, S. D., Yang, P., Gofman, M., and Ramakrishnan, C. R. (2011). Symbolic Reachability Analysis for Parameterized Administrative Role Based Access Control. *Computers & Security*, 30(2-3):148–164.

Tripunitara, M. V. and Li, N. (2013). The Foundational Work of Harrison-Ruzzo-Ullman Revisited. *IEEE Trans. Dependable Secur. Comput.*, 10(1):28–39.