

Exploiting Hot Spots in Heuristic Safety Analysis of Dynamic Access Control Models

Marius Schlegel^a and Winfried E. Kühnhauser
Technische Universität Ilmenau, Germany

Keywords: Security Engineering, Security Policies, Access Control, Dynamic Access Control Models, Heuristic Safety Analysis, Security Model Checking, Formal Methods.

Abstract: Model-based security engineering approaches frequently suffer from computational complexity of model analysis. As a consequence, a considerable amount of human expertise is involved in the analysis process, rendering model analysis an expensive approach applied mostly to sophisticated systems with challenging security requirements. This paper discusses algorithmic foundations for an automated safety analysis of dynamic access control models. The computational complexity is tackled by a heuristic-based approach, rendering the analysis algorithm scalable to large real-world access control systems.

1 INTRODUCTION

Security engineering approaches for IT systems with sophisticated security requirements increasingly apply security policies for defining and implementing security properties. A security policy precisely describes strategies that implement these requirements. The critical importance of policy correctness calls for a careful policy engineering process.

In order to achieve correctness guarantees, model-based security policy engineering (MSPE) approaches take advantage of formal models, allowing for rigorous analysis and proof of a policy's formalized *security properties* (Vimercati et al., 2005, Li and Tripunitara, 2006, Tripunitara and Li, 2007, Jha et al., 2008, Barker, 2009, Basin et al., 2011), some of which are tractable by static methods, while others require to reason about the dynamic evolution of a system. Beyond that, security models serve as formal specifications from which policy implementations are engineered.


In this paper, we focus on dynamic security properties of dynamic access control (AC) models. They allow to reason about authorizations to execute security-critical operations. Analyzing these properties has been termed *security analysis* (Tripunitara and Li, 2007), which can be subdivided in two classes of questions: Given some dynamic AC model at a *model protection state*, 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 operations which are, for example, demanded by confidentiality or integrity goals of a policy. For historical reasons, this second family of questions is called *safety properties*.

MSPE is not an easy approach. Especially the analysis of dynamic models suffers from its computational complexity and the tractability of large model state spaces, and in many cases essential model properties are intractable or even undecidable. As a consequence, MSPE involves challenging human expertise and thus is primarily applied in scenarios with quite critical security requirements.

This paper aims at paving the road to MSPE by developing foundations for automated model analysis tools. Focusing on model safety analysis of dynamic AC models, the paper tackles time and space complexity of heuristic safety analysis algorithms by shifting the effects of exponential time and space complexity to model state sizes significantly larger than those found in current real-world policies.

Specifically, we make the following contributions. (1.) We argue that the handling of large parameter spaces offers a great potential for optimizing heuristic safety analysis algorithms (Sec. 2). (2.) We introduce a novel approach for restricting large parameter spaces based on hot spots, enabling more effective and efficient state transitions, and we demonstrate our approach for dynamic HRU-type AC models (Sec. 3).

^a  <https://orcid.org/0000-0001-6596-2823>

(3.) We show that the implementation of our algorithm offers a significant runtime improvement over prior algorithms (Sec. 4).

2 MODEL SAFETY ANALYSIS

In dynamic AC models, model safety deals with the fundamental question whether permissions granted in a given model state may leak to different subjects in future model states. Safety analyses thus expose model properties that allow for the prediction of an AC system's dynamic behavior. Model safety in general is a non-decidable problem (Harrison et al., 1976). While safety-decidable AC models have been proposed (Harrison and Ruzzo, 1978, Sandhu, 1992, Lipton and Snyder, 1977) these models generally are targeted to particular application scenarios and have limitations in their ability to model complex real-world policies.

For more general models without such limitations safety analyses fall back to heuristic approaches that trade accuracy for tractability. Heuristic-based analysis algorithms exploit the fact that the safety problem is semi-decidable and try to prove that some given model state is *not* safe with respect to some right by finding a state sequence that leaks this right in some follow-up state. While they cannot always deliver analysis results, heuristic analysis algorithms often provide valuable insights in the reasons for right leakages. Nevertheless, while in general being capable of handling large and complex models heuristic-based approaches still severely suffer from computational complexity and the tractability of large model state spaces.

2.1 Dynamic Access Control Models

One of the earliest AC models for analyzing model safety is the HRU model (Harrison et al., 1975). While from today's perspective the HRU model uses quite low-level abstractions the model is still among the most powerful and general AC models. It provides a calculus for the rigorous specification of dynamic AC policies as well as a foundation for statements about the complexity of model safety analyses.

HRU models combine AC matrices (ACMs) with deterministic automata. Models in the HRU family share the idea of modeling the dynamic behavior of an AC system by state transition schemes (STSs) specifying the state transition function of their automata. Each state of an HRU model reflects a single protection state of an AC system; state transitions – triggered by inputs to the automaton – are effected by policy-specific operations in the STS that modify the state. Model safety properties now can be analyzed by

observing state transitions effected by input sequences; in particular, the boundaries of right proliferation can be explored by state reachability analyses.

State reachability is known to be a non-decidable problem. As a consequence, the HRU model has inspired numerous and more restricted offsprings (Goguen and Meseguer, 1982, Zhang et al., 2005, Mondal et al., 2009, Stoller et al., 2011, Sandhu et al., 1999, Sharifi and Tripunitara, 2013) that render model safety tractable by tailoring a model calculus to dedicated application scenarios, allowing for statements about worst-case execution times of safety analyses (Harrison and Ruzzo, 1978, Sandhu, 1992, Lipton and Snyder, 1977). As a consequence, the resulting safety analysis algorithms strongly depend on the restrictions of the respective model calculus and lack the generality needed for more universal model analysis tools aiming at a broad scope of HRU model variants.

While in practice HRU models have long since been supplanted by models supporting more application-friendly abstractions such as roles (RBAC models) or attributes (ABAC models), their fundamental idea of modeling dynamic behavior of AC systems by deterministic automata has been adopted in many HRU offsprings (Sandhu et al., 1999, Kühnhauser and Pölck, 2011, Ranise et al., 2014, Amthor, 2015, Schlegel and Amthor, 2020). In order to be applicable to all these offsprings our approach aims at a general foundation for safety analyses of general, non-restricted HRU models, following the idea that insights in the most general forefather in a model family are also applicable to his descendants.

2.2 Heuristic Safety Analysis

Because the safety problem for general HRU models is not decidable, safety analysis algorithms use heuristics that exploit the semi-decidability of the problem: given any two states of an AC system with one state being a follow-up state of the other it is efficiently decidable whether in the follow-up state a right leakage can be observed. Thus, starting with some initial state, heuristic search algorithms guide a model through its state space by generating input sequences, feeding them into the automaton and checking for each reached state whether it renders the initial state unsafe.

The problem here is the large number of eligible inputs for each state transition. In any HRU model, an AC system's dynamic behavior is modeled by its STS, consisting of a set of commands where each command triggers a state transition of the model's automaton. As an example, an application system's function to delegate a read right to a subject's deputy can be modeled by an HRU command consisting of a

```

command delegateReadRight(
  subjectfrom, subjectto, object) =
  if read_right ∈ m(subjectfrom, object) then
    enter read_right into m(subjectto, object)
  fi,

```

(set of) condition(s) and a (set of) primitive(s) with the semantics that if the conditions are met by the current model state, the primitives define the corresponding state transition. An eligible input for triggering a state transition in an HRU model thus consists of a command name (in the above example *delegateReadRight*) and its parameters (values for *subject_{from}*, *subject_{to}*, *object*).

A simple sample calculation shows that if input sequences to explore the automaton's state space are chosen randomly then the probability of detecting a state exposing a leakage is extremely small. On a medium-sized file server with 100 users, 10^6 files, and an STS with just 10 commands, each having 3 parameters, we encounter $10 \cdot (10^6)^3 = 10^{19}$ different input combinations, leading to as many possible follow-up states. Thus, the probability of randomly hitting parameter values that result in a state transition ultimately leading to a leakage is extremely small. Moreover, for the second generation of follow-up states we already encounter 10^{38} possibilities. With respect to memory complexity, assuming just 1 byte of memory required for each cell of the ACM (of size 100 users $\cdot 10^6$ objects), the second generation would require 10^{46} bytes of memory for storing the corresponding state spaces. It is easy to see that a random search of the complete state space for states exposing right leakages is like searching for a needle in a haystack.

2.3 The DEPSEARCH Heuristics

Other than randomized approaches, successful heuristics generally are well-tailored to the specific problem to be solved. The challenge for safety analysis heuristics thus is to exploit model properties that have an impact on the probability of an input to contribute to a state sequence that leads from an initial state to some follow-up state that actually exposes a right leakage.

The DEPSEARCH heuristics (Amthor et al., 2013, Amthor et al., 2014, Amthor, 2017) exploits the (obvious) fact that for a right to be leaked into some ACM cell it is a necessary condition that a command is executed that includes a primitive to write this right into an ACM cell. This command, again, has conditions that have to be met for its execution, so it, again, depends on other commands that establish these conditions, leading to a chain of commands that successively establish necessary conditions for a right leakage. A static analysis of a model's STS discloses such com-

mand dependencies, resulting in a command dependency graph (CDG) $\langle V, E \rangle$ where vertices $c_i, c_j \in V$ represent commands and edges $\langle c_i, c_j \rangle$ signify that c_i has primitives that enter rights into the ACM that are required by c_j (see Fig. 1a and Fig. 1b for examples).

DEPSEARCH now chooses paths from the CDG as command sequences that are fed into the automaton, each command having a certain probability to establish necessary conditions for the next command and thus moving the automation's state "nearer" to a state exposing a right leakage.

Model analysis thus consists of simulating a model's automaton by choosing command paths from the CDG, selecting for each command a parameter value set (see Sec. 3) and feeding them into a model simulation engine. Starting with the model's initial state, the first path component creates a follow-up state, and the simulation of a complete command sequence successively adds further states to a simulation's state transition tree. Once a path has been completely processed, the simulation continues by choosing new command sequences from the CDG until a state is reached leaking the target right (Amthor et al., 2013).

As discussed in (Amthor et al., 2013), safety analysis based on the DEPSEARCH heuristic becomes tractable for scenarios with model state sizes of up to 10^4 ACM cells.

3 HEURISTIC COMMAND PARAMETER SELECTION

For real-world scenarios with states typically comprising over 10^4 ACM cells, heuristic safety analysis algorithms still face effects of an exponential runtime complexity. At this point, the idea is to shift the boundary where model states become untractable to ACM sizes significantly larger than 10^6 ACM cells.

During model simulation, for any state transition of the model's automaton an input is needed, consisting of a command to be executed as well as its parameter values. In the DEPSEARCH heuristic (Amthor et al., 2013), only commands are selected effectively and efficiently, using the CDG-based command sequence generation approach. As discussed in Sec. 2.2, in very large AC model instances there is a huge number of possible value assignments for command parameters prohibiting an exhaustive testing of all possible parameter values. To tackle the problem of finding "good" parameter values (*parameter selection problem*, PSP), heuristics become here necessary as well.

3.1 The Role of Working Sets

Our approach to dealing with the PSP lies in the observation that there is a strong affinity to a fundamental problem in the virtual memory management (VMM) of operating systems. Typically, virtual memory is much larger than physical memory, so that not every virtual memory page can always be resident in physical memory. If a page required by a process is not present in physical memory, a page fault occurs and a present page frame is cleared to accommodate the actual page needed. Optimally, the page whose next use is furthest in future should always be selected for removal; page replacement algorithms try to achieve an approximation to this optimum.

Contemporary page replacement algorithms, e. g. WSClock (Carr and Hennessy, 1981), initially assume that, due to the locality of page references, memory access patterns of processes have hot spots in which almost all accesses take place. Pages belonging to hot spots are called its *working set* (i. e. the set of pages a process is currently using) (Denning, 1968). Consequently, accesses from a process to pages that belong to its working set are much more likely than accesses to pages that do not belong to it. The working set of a process is determined by the pages last used in a fixed time interval; pages that do not belong to any process' current working set are therefore candidates for removal from physical memory.

In a nutshell, working sets heuristically limit a large number of elements to a small number of the most interesting elements. This offers an interesting perspective on the PSP: If we can find a subset of ACM cells that are more likely to be required for a sequence of commands leading to a right leakage, then the heuristic search algorithm can ignore a large part of the ACM. In order to make the parameter selection both efficient and effective, the large number of ACM cells can be restricted to only those cells which offer the greatest potential for generating a right leakage.

However, the PSP and the page replacement problem have also a fundamental difference. Page replacement algorithms can observe and exploit a process' reference behavior to form a working set. In contrast, the situation in heuristic safety analysis is totally different: The dynamic model behavior is completely simulated under heuristic control, so that hot spot patterns can and have to be *generated strategically*. Therefore, the heuristic for the PSP lies in how the parameter search space is strategically constrained to enable both an effective and efficient parameter selection.

3.2 Parameter Working Sets

Generally, our approach for restricting the command parameter value space adopts the idea of VMM working sets by limiting the large number of ACM cells to only those that have the greatest potential for generating a right leakage. Here, a working set is a subset of ACM cells, where each cell is defined by its subject and object identifiers used as command parameter values. We refer to it as *parameter working set* (PWS).

In VMM, working sets depend on the execution behavior of individual processes rendering them process-individual. In the context of heuristic safety analyses, command sequences (paths generated from the CDG) are executed. And in the same way, PWSs depend on the execution of individual paths rendering them path-individual. Analogous to VMM working sets, a PWS limits the value space for parameter selection and thus restricts the ACM to the most filled cells that contain as many rights as possible to satisfy the conditions of the current path's commands.

In order that the execution of a command also leads to the entering of rights into ACM cells and even additionally satisfiable conditions, these rights must not yet be present in the cells addressed in the *enter right* primitives of that command. In general, this may apply to many cells. Therefore, to increase the potential that entered rights lead to additionally satisfiable conditions of subsequently executed commands (and not just fill any ACM cells randomly), cells are selected as parameter values for command bodies (and *enter right* primitives) that are already included in the PWS.

When creating a PWS for a specific path, all rights occurring in the conditions of that path's commands have to be considered. The generation of a subsequent path then most likely requires to consider different or additional rights and regenerating the PWS for that new path. Since we strive to fill already well-filled cells even more to potentially unlock commands which were not executable before, it would not be sensible to regenerate the PWS completely from scratch. Therefore, we add new cells required by a path to the PWS without removing the previously added cells (knowledge of the past, cf. WSClock (Carr and Hennessy, 1981)). Then, newly added cells can be used to satisfy command conditions, but contained, well-filled cells can still be filled with rights by command primitives.

However, it is often observable, especially as an analysis progresses, that an existing PWS is completely valid for a subsequent path and already meets the criterion of path specificity, i. e. conditions of that path's commands are satisfiable by the cells present in the PWS. Thus, reusing an existing PWS and gentle fostering is to be preferred over a regeneration. Conse-

quently, in case that not yet all cells belonging to the PWS are already saturated and commands can still be executed effectively resulting in new states, a PWS can be left the same even across the execution of different paths. In case that neither a right leakage was evoked, nor new rights were entered, the PWS seemingly is inadequate and must be fostered. Therefore, a PWS is used for subsequent paths only until the execution of such a command sequence was completely ineffective and did not result in a single new state.

3.3 The WSDEPSEARCH Algorithm

In this section, the described parameter space approach based on working sets is specified as the *working-set-based dependency search* (WSDEPSEARCH) algorithm building upon previous works (Amthor et al., 2013, Amthor et al., 2014, Amthor, 2017).

In the first phase, a static analysis of the HRU STS is performed to construct the CDG (see (Amthor, 2017, p. 5, Alg. 2)). It yields a graph-based description of inter-command dependencies, constituted by entering (as part of primitives) and requiring (as part of conditions) the same right in two different commands. The CDG is recursively 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 (by *createCDGSource* and *createCDGSink*): c_q is the source of all paths in the CDG, since it mimics the state q to be analyzed, represented by a virtual command specification added to the set C of all commands. It is generated by encoding the ACM m_q in the body of 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 c_{target} in some cell of m_q . In order to determine predecessor-edges between a command c and other commands that establish necessary conditions for it to be executed (*predecessors*), *buildPredSet* returns a set $P = \{c \in C \mid c.Prim.Enter.Rights \cap v.Cond.Rights \neq \emptyset\}$ where $c.Prim.Enter.Rights$ denotes the set of rights entered by primitives of c and $v.Cond.Rights$ denotes the rights required to satisfy conditions of command v .

In the second phase, the dynamic analysis, the CDG is used to guide state transitions by generating input sequences to the automaton. The commands in each sequence are chosen according to different paths from c_q to c_{target} (see (Amthor, 2017, p. 5, Alg. 1)). Paths are generated based on a modified ant algorithm: with each edge traversed to generate an input sequence, the algorithm increases an edge weight ("scent" d increased by \hat{d}) which has a repellent effect for the next

```

In: –  $W \subseteq S \times O$ : current PWS
      –  $R_{path}$ : set of rights required in the conditions of the
         path's commands
      –  $q = \langle S, O, m \rangle$ : current model state
Out: –  $W$ : PWS for path

procedure addParameters()
     $\langle s_c, o_c \rangle \leftarrow CW.someMember$ ;
     $maxr \leftarrow |m(s_c, o_c) \cap R_{path}|$ ;
    for  $\langle s, o \rangle \in CW$  do
        if  $|m(s, o) \cap R_{path}| > maxr$  then
             $s_c \leftarrow s$ ;  $o_c \leftarrow o$ ;
             $maxr \leftarrow |m(s_c, o_c) \cap R_{path}|$ ;

    if  $maxr > 0$  then
         $W \leftarrow W \cup \{ \langle s_c, o_c \rangle \}$ ;
         $CW \leftarrow CW \cap \{ \langle s_c, o_c \rangle \}$ ;
         $R_{path} \leftarrow R_{path} \cap m(s_c, o_c)$ ;
        if  $R_{path} \neq \emptyset$  and  $CW \neq \emptyset$  then
             $\text{addParameters}()$ ;

     $CW \leftarrow \emptyset$ ;
    for  $\langle s, o \rangle \in m$  do
        if  $\langle s, o \rangle \notin W$  and  $m(s, o) \cap R_{path} \neq \emptyset$  then
             $CW \leftarrow CW \cup \{ \langle s, o \rangle \}$ ;

    if  $CW \neq \emptyset$  and  $W \neq \emptyset$  then
         $\text{addParameters}()$ ;
    else
         $s \leftarrow S.someMember$ ;  $o \leftarrow O.someMember$ ;
         $W \leftarrow W \cup \{ \langle s, o \rangle \}$ ;

    return  $W$ ;
```

Algorithm 1: PWSGeneration.

iteration of the CDG traversal. Since the path generation always selects edges with lower scents before such with higher scents (*lowestScent*), a uniform coverage of the CDG is achieved and, thus, the threat of running and staying in blind alleys is countered. As a prerequisite for the PWS generation, with each edge added to a path, the set R_{path} of all rights required in the conditions of that path's commands is filled.

Based on the PWS approach, Alg. 1 outlines their generation and fostering. First, ACM cells are collected in the set of cells CW that qualify as PWS member candidates. For this, cells have to satisfy at least one condition of a current path's command, i. e. hold at least one right of R_{path} , and may not yet be element of the PWS. Subsequently, the actual PWS members are determined (*addParameters*). In each recursive step, a new member cell is added to the PWS. For reasons of efficiency, the algorithm always selects as few cells as possible, but adds those with the greatest potential for contributing to a right leakage. Therefore, a member candidate to be added to the PWS is required to contain most of the rights not yet considered compared to all other candidates (such that $|m(s, o) \cap R_{path}| = \max_{\langle s', o' \rangle \in CW} |m(s', o') \cap R_{path}|$), where at least one

In: – C : the model’s STS
– q_0 : state the safety of which is to be analyzed
– r_{target} : target right

Out: – $stateSeq$: state sequence leaking r_{target}

function `isLeaked`(**in** $q = \langle S, O, m \rangle \in Q$, **in** $q' = \langle S', O', m' \rangle \in Q$)

```

  for  $s \in S \cap S'$  do
    for  $o \in O \cap O'$  do
      if  $r_{target} \notin m(s, o) \wedge r_{target} \in m'(s, o)$  then
        return true;
    return false;

```

$q \leftarrow q_0$; $stateSeq \leftarrow q$;
 $\langle V, E \rangle, d, \hat{d}, c_q \leftarrow \text{CDGAssembly}(C, q, r_{target})$;
 $W \leftarrow \emptyset$; $effCount \leftarrow 0$;

repeat

```

   $path, d, R_{path} \leftarrow \text{CDGPathGeneration}(\langle V, E \rangle, d, \hat{d}, c_q)$ ;
  if  $effCount = 0$  then
     $W \leftarrow \text{PWSGeneration}(W, R_{path}, q)$ ;
   $effCount \leftarrow 0$ ;
  while  $c \leftarrow path.nextCommand$  do
     $q' \leftarrow \delta(q, c, \text{selectParameters}(W, c))$ ;
    if  $q' \notin stateSeq$  then
       $effCount \leftarrow effCount + 1$ ;
     $stateSeq \leftarrow stateSeq \circ q'$ ;
     $q \leftarrow q'$ ;

```

until `isLeaked`(q_0, q', r_{target});
return $stateSeq$;

Algorithm 2: WSDEPSEARCH.

new right must be present in that cell (i. e. $m(s, o) \cap R_{path} \neq \emptyset$). If one of the candidate cells meets those requirements, it is added to the PWS W , removed from CW , and the cell’s rights are removed from R_{path} . This procedure is called recursively until R_{path} is empty (i. e. all required rights have been considered) or none of the remaining candidates meets the requirements.

Finally, we have specified the WSDEPSEARCH algorithm in Alg. 2 using (Amthor, 2017, Alg. 2), (Amthor, 2017, Alg. 1) and Alg. 1. WSDEPSEARCH successively generates input sequences by traversing the CDG on every possible path and, depending on the success of the previous path in terms of resulting effective steps, modifies the PWS. With parameter values selected from the PWS (*selectParameters*), the execution of each command is simulated by the algorithm, and once a CDG path is completed, the violation of the HRU safety criteria is checked (*isLeaked*).

Runtime and Space Complexity. The CDG assembly runs in $O(n^2)$, where n is the number of commands in the model’s STS. The CDG requires a space complexity of $O(|V| + |E|)$ using an adjacency list representation, where a quadratic worst case complexity occurs only in the untypical case of a complete CDG.

The CDG path generation runs in $O(n)$, because due to the minimum scent of each path, eventual cycles occur at most once in a path (Amthor et al., 2013). The PWS generation (Alg. 1) has a computational runtime complexity of $O(|S_q \times O_q|^{R_{path}})$, where S_q and O_q are the subject and object sets in model state q . In worst case, each ACM cell is added to the PWS candidate set and checked for the presence of a particular right of R_{path} (each call of *addParameters*). A PWS can require a worst case space complexity of $O(|S_q \times O_q|)$ (ACM dimensions), although the PWS size in the evaluation was never larger than 1 % of the ACM’s size. The actual parameter selection (*selectParameters* in Alg. 2) using a brute force scheme runs in $O(\max(|S_q|, |O_q|)^p)$, where S_q and O_q as above, and p denotes the maximum number of command parameters. Finally, due to the semi-decidable nature of the safety problem, we can tell nothing about the total number of steps taken by the heuristic (Alg. 2).

4 EVALUATION

This section presents the evaluation of the WSDEPSEARCH heuristic. Two major goals are addressed: (1.) the practical feasibility regarding runtime performance of analyses of real-world sized AC model instances, and (2.) the relative heuristic quality. Evaluation subject is the WSDEPSEARCH algorithm (Alg. 2).

4.1 Evaluation Method

The evaluation goals are examined by measuring analysis runtimes. To obtain meaningful results, the models used must cover two key performance indicators of analyses in practice: (1.) STS scheme complexity in terms of command dependencies and (2.) model state size in terms of ACM size.

For the practical feasibility evaluation, we consider two different types of models: hybrid RBAC/HRU (role-based AC) models of a health care information system (HIS) security policy serve as realistic scenarios, and synthetic high-dependency HRU models with STSs designed to contain well-hidden right leaks serve as stress-tests (with a big calculation effort). By using randomly initialized ACMs with up to $2 \cdot 10^7$ cells, we cover model state sizes of practical AC systems.

The relative quality is evaluated by comparing WSDEPSEARCH with the classical DEPSEARCH using a brute-force parameter search in the whole ACM. The same realistic and synthetic models are used as test cases, but due to longer simulation times lower limits on the number of ACM cells are used.

Basically, the test procedure is as follows: Whenever the heuristic is executed, it selects a particular model state from the state transition tree (STT), a command from the STS, and a parameter vector. These inputs are passed to a simulation engine implementing the deterministic state machine of an HRU model, which eventually triggers a state transition. A possible reason for a failing transition may be that conditions of a command cannot be satisfied, or that primitives do not modify the given state at all. If a transition was successful and led to a new follow-up state, this state is inserted into the STT. The result of a transition is passed to the heuristic which then selects a new input. Each of these iterations is called a *step*. Since a heuristic does not necessarily trigger a transition to a new state, we call those steps that actually do *effective*.

To judge the results, three different measures are used: effective step count (ESC), effective step time (EST), and total runtime of a heuristic. The ESC is the total count of effective steps required to leak a target right, while the EST is the average time required to make an effective step (including any ineffective steps within that time). For the feasibility evaluation, we performed a runtime comparison of the WSDEP-SEARCH algorithm for all test models, that allows for performance estimations regarding both average and worst-case runtime. The qualitative comparison with DEPSEARCH is done based on the total runtime and the measures of ESC and EST.

The remainder of this section addresses the evaluation method in detail: it describes the used models, the selection of parameter values, the test input configurations and the execution environment.

4.1.1 Test Cases

HIS Models. The realistic models are based on a security policy of a real-world HIS for an aged-care facility introduced by (Evered and Bögeholz, 2004) and rendered more precisely by (Gofman et al., 2009, Stoller et al., 2011) to develop a formal *static* RBAC model (Sandhu et al., 1996) with 20 roles, a role hierarchy, and separation-of-duty focusing on role exclusion. To describe *dynamic* behavior, (Amthor et al., 2013) enhanced the model by a state automaton along with a STS with 16 commands. The result are two hybrid RBAC₃/HRU models, HIS I and HIS II, each focusing on certain parts of the original RBAC model¹ to allow for analyses of relevant aspects.

Regarding the dynamic behavior of RBAC/HRU models, two analysis questions are practically relevant. (1.) Given some state, is it possible that a user u is ever

¹Note that due to simplicity, sessions and object attributes were omitted.

assigned a role r ? In terms of model abstractions, the dynamics of the user-to-role assignment UA are examined: Given an STS, if a tuple $\langle u, r \rangle$ can ever become an element of UA , the corresponding state is considered *not safe* with respect to that role (cf. user-role reachability in (Sasturkar et al., 2011)). Analogously, the second question deals with the permission-to-role assignment PA : (2.) Given some state, is it possible that a role r is assigned a permission $\langle o, op \rangle$, i. e. an operation on an object? In this case, we call such a state *not safe* with respect to that operation (cf. permission-role reachability in (Sasturkar et al., 2011)).

The HIS I model focuses on user-to-role assignment and separation-of-duty based on role exclusion. Its state $q = \langle S_q, O_q, m_q \rangle$ is described by the user set of the original RBAC model as the subject set S_q , the RBAC object set as O_q , and an ACM $m_q : S_q \times O_q \rightarrow 2^{roles}$ which maps a user-object-pair to a set of roles. Any STS command is guarded by at least one condition of the type “a user has to own a specific role for a specific object”. The HIS I model implements role exclusion by negative roles: Since conditions in HRU-like models cannot test the absence of rights in the ACM, a negative role is added for each role simulating its absence. The STS contains 7 commands that modify the RBAC user-to-role assignment considering role exclusions. Analyzing the HIS I model with respect to HRU safety is then equivalent to analyzing the original RBAC model regarding RBAC safety flavor (1).

The HIS II model implements the permission-to-role assignment and the role hierarchy of the original RBAC model. As in HIS I, its state $q = \langle S_q, O_q, m_q \rangle$ includes the RBAC object set O_q . S_q is now defined as the RBAC role set and the ACM $m_q : S_q \times O_q \rightarrow 2^{ops}$ maps a role-object-pair to a set of operations. The ACM hence exactly represents the RBAC PA relation, omitting the indirection level of permissions. The STS incorporates the role hierarchy relation by solving the level of indirections of PA , resulting in a HRU-like RBAC model with 6 commands. Analyzing the HRU

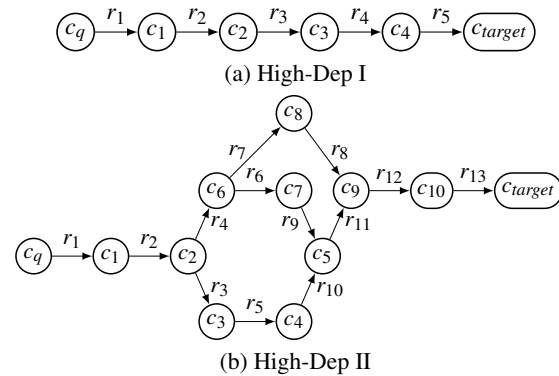


Figure 1: CDGs of High-Dependency Models.

safety of the HIS II model then equals analyzing the original RBAC model regarding safety flavor (2).

High-dependency Models. The DEPSEARCH approach originally was developed because earlier heuristics failed in analyzing atypical models where right leakages are well hidden and appeared only after long command sequences where each command depends exactly on the execution of its predecessor – a type of complexity which is part of many real-world security policies. For this reason, two artificially tailored stress-test models from (Amthor et al., 2013) will serve as a kind of benchmark for this type of STS complexity.

The high-dependency models, High-Dep I and High-Dep II, are traditional HRU models where the STS and initial state reflect the dependencies shown in the corresponding CDGs (see Fig. 1): In High-Dep I, the target right r_5 is entered by command c_4 . Each label of an edge $\langle c_i, c_j \rangle$ denotes a right that is entered by c_i and required by c_j ; the initial state q_0 is chosen such that $\forall (s, o) \in S_{q_0} \times O_{q_0} : \{r_2, \dots, r_5\} \notin m_{q_0}(s, o)$ (apart from this, all ACM cells are initialized randomly based on a generic right set $R = \{r_1, \dots, r_{20}\}$). No command requires or enters more than one right out of $\{r_2, \dots, r_5\}$. High-Dep II is analogously constructed (target right r_{13} , leaked by c_{10}), comprising more complex dependencies. In case of both models, each node of the CDG has to be visited at least once because none of the rights that impose the dependencies are present in the initial ACM. Therefore, a minimum number of state transitions is required to discover a right leakage.

Brute-force Parameter Selection. For the sake of efficiency regarding ESTs, we use a brute-force parameter selection approach running in $O(|W|^{p/2})$ for WSDEPSEARCH and in $O(\max(|S_q|, |O_q|)^p)$ for DEPSEARCH, where W is the current PWS, S_q and O_q are state q 's subject and object sets, and p denotes the maximum number of parameters in any STS commands.

4.1.2 Test Input Configurations

The following input configurations were used for the test runs: initial subject and object count ($|S_{q_0}|$ and $|O_{q_0}|$), ACM initialization (m_{q_0}), and target right. For each model, the initial state's object count and ACM contents are varied, whereas the subject count is kept constant (since only the number of ACM cells impacts the heuristic behavior, not their layout).

For the feasibility evaluation, we cover ACM dimensions starting from $20 \times (5 \cdot 10^x)$ and $20 \times (10 \cdot 10^x)$, increasing x from 0 stepwise by 1 up to 5 (i.e. 20,000,000 cells). For comparing WSDEPSEARCH and DEPSEARCH, we used dimensions starting from

20×20 and ending with 20×500 , increasing in steps of 20 objects (i.e. 400 cells). The initial ACM contents are randomized using a fixed, model-individual right set R , such that the resulting model is not safe by construction. Moreover, the target right (that *safety* is analyzed for) is also fixed and model-specific. For each configuration, 10 runs have been performed.

4.1.3 Execution Environment

The evaluation is performed in our security policy engineering framework WorSE (Amthor et al., 2014). We have integrated WSDEPSEARCH into a reimplementa-tion of the model safety analysis tool for dynamic AC models. Each model simulation engine is based on a generic deterministic state machine implementation. Similarly, the implementation of WSDEPSEARCH is derived from a heuristic base class. The model itself is a data structure shared by the simulation engine and heuristics, and includes implementations of the STT and the STS. All measurements were performed on contemporary desktop hardware with an Intel Core i7-7700K CPU at 4.2 GHz and 32 GiB DDR4 RAM at 2,400 MHz under Ubuntu Linux 18.04 LTS.

4.2 Evaluation Results

We will now discuss the results of the experimental evaluation. In all figures the error bars show the 95 % confidence intervals (CIs).

4.2.1 Feasible Runtime Performance

We measured the total runtime of WSDEPSEARCH for each of the 4 models covering ACM sizes from 10^2 to $2 \cdot 10^7$ cells. The results shown in Fig. 2 suggest a quadratic growth of the runtime. For model state sizes of up to 20,000,000 ACM cells, the absolute analysis runtime for both HIS models is at most just over 10 seconds and for the High-Dep stress models with a highly complex STS at most just over 100 seconds. Hence, it can be argued that the absolute runtime does not nearly exceed the scale of real-world analysis sessions.

However, Fig. 2 also reveals a peculiarity: In order to generate a right leakage when performing the safety analysis of HIS I model, a minimal ESC of 2 steps would have been necessary at least. Nevertheless, in the first 4 configurations 4 steps were actually needed resulting in a total runtime higher than needed for the analysis of High-Dep I model. The reason for this is the randomized initialization of ACM cells and the PWS initialization: Since after the first path generation a cell was added to the PWS, where only one of the two rights necessary for the execution of a command (*assignReferredDoctorRole*) was present. Then, the

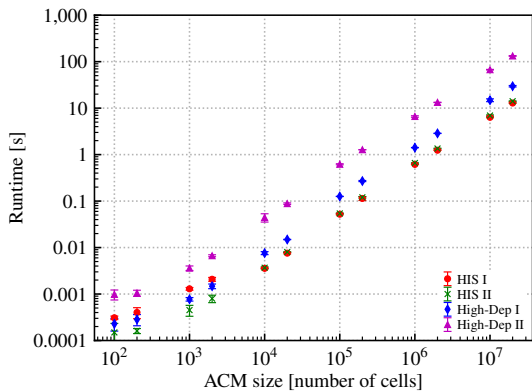


Figure 2: WSDepSEARCH runtimes.

right leakage occurred only after the second path generation, the enlargement of the PWS by an additional cell and the execution of that path.

4.2.2 Heuristic Quality

For evaluating the relative quality, we compared the performance of WSDepSEARCH with the classical DepSEARCH using a randomized brute-force parameter selection algorithm for both heuristics.

Effective Step Quality. For all models and runs, Table 1 shows the mean ESCs of both heuristic algorithms along with the corresponding 95 % CIs. Additionally to judge the heuristic quality based on ESC, the ratio of minimal ESC to mean ESC is given, called *effective step quality* (Q_{ESC}).

Considering the ESC and Q_{ESC} values, both algorithms are nearly equally effective due to their common foundation. However, there are two recognizable differences: For HIS I model, the WSDepSEARCH’s mean ESC is slightly worse than that of DepSEARCH. We explain this, as in Sec. 4.2.1, with the initialization of the ACM cells in small model states and the PWS initialization. For High-Dep II model, WSDepSEARCH achieves a noticeable improvement due to the

Table 1: Mean ESC and Q_{ESC} .

	HIS I	HIS II	H.-Dep I	H.-Dep II
Min. ESC	2	2	4	10
DepSEARCH heuristic				
Mean ESC	2.0	2.0	4.0	15.0
95 % CI	0.0	0.0	0.0	0.0
Q_{ESC}	1.0	1.0	1.0	0.667
WSDepSEARCH heuristic				
Mean ESC	2.08	2.0	4.0	10.0
95 % CI	3.196	0.0	0.0	0.0
Q_{ESC}	0.961	1.0	1.0	1.0

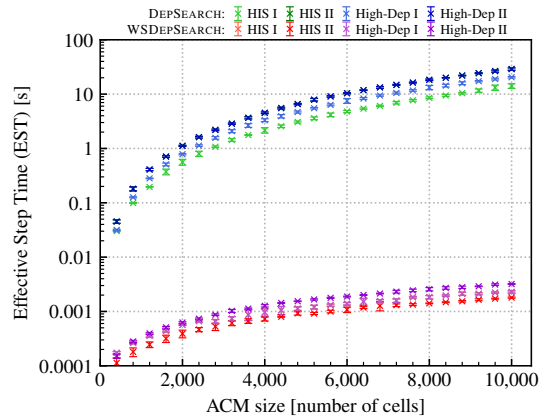


Figure 3: EST comparison.

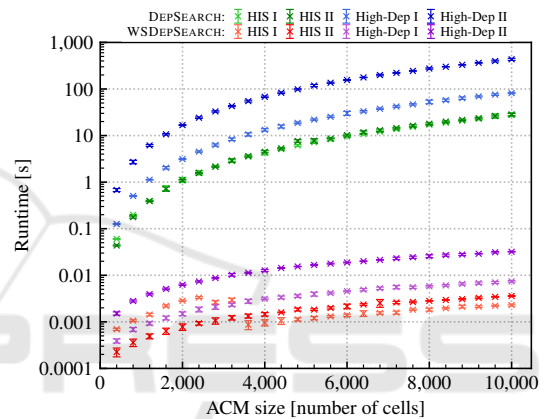


Figure 4: Runtime comparison.

strategically restricted parameter space. In result, averaging over all ESC and Q_{ESC} values, WSDepSEARCH beats DepSEARCH.

Runtime Quality. In order to evaluate the heuristic quality based on runtime performance, both ESTs and total runtimes of WSDepSEARCH and DepSEARCH were compared for each model.

Fig. 3 depicts a comparison regarding average ESTs. Overall, it can be clearly seen that, for all models, the WSDepSEARCH heuristic made effective steps in a much shorter time compared to the DepSEARCH heuristic. Even though both heuristics have a comparable Q_{ESC} values for models HIS I, HIS II and High-Dep I, the restriction of the parameter search space by means of PWSs in WSDepSEARCH shows a clear EST improvement of nearly 4 orders of magnitude over DepSEARCH for all models.

The average total runtime of both algorithms is compared in Fig. 4. Here again, WSDepSEARCH clearly outperforms DepSEARCH. A final comparison of both Figs. 3 and 4 concludes that even under the usage of a simple brute-force parameter selection

algorithm the working-set-based approach of WSDep-SEARCH leads to significantly better EST values and thus, under consideration of the ESC values, to a similar significant improvement of the overall runtime.

5 CONCLUSIONS

This paper addresses the exponential time complexity problem of heuristic algorithms for the safety analysis of dynamic AC models with large parameter spaces. While former heuristics identify well-hidden command sequences in which each command establishes the preconditions of its subsequent command until a target right is leaked, the selection of corresponding command parameter values is neither effective nor efficient resulting in over 90 % of the total analysis runtime.

To drastically reduce this weakness, we present WSDepSEARCH which adapts the idea of working sets known from page replacement algorithms by strictly limiting the parameter value space to only those cells of the ACM that offer a high potential for a command sequence to contribute to a right leakage. The implementation of the heuristic analysis algorithm was integrated into our security policy engineering framework WorSE. The evaluation shows a runtime improvement up to 4 orders of magnitude which makes analyses of automaton-based AC models with realistic state transition schemes and model state sizes encompassing more than 10^7 ACM cells tractable.

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 Secur. and Crypt.*, pages 88–99.
- Amthor, P. (2017). Efficient Heuristic Safety Analysis of Core-based Security Policies. In *Proc. 14th Int. Conf. on Secur. and Crypt.*, pages 384–392.
- Amthor, P., Kühnhauser, W. E., and Pölck, A. (2013). Heuristic Safety Analysis of Access Control Models. In *Proc. 18th ACM Symp. on Access Control Models and Technol.*, pages 137–148.
- Amthor, P., Kühnhauser, W. E., and Pölck, A. (2014). WorSE: A Workbench for Model-based Security Engineering. *Comp. & Secur.*, 42:40–55.
- Barker, S. (2009). The Next 700 Access Control Models or a Unifying Meta-Model? In *Proc. 14th ACM Symp. on Access Control Models and Technol.*, pages 187–196.
- Basin, D., Clavel, M., and Egea, M. (2011). A Decade of Model-Driven Security. In *Proc. 16th ACM Symp. on Access Control Models and Technol.*, pages 1–10.
- Carr, R. W. and Hennessy, J. L. (1981). WSClock – A Simple and Effective Algorithm for Virtual Memory Management. In *Proc. 8th Symp. on Operating Syst. Principles*, pages 87–95.
- Denning, P. J. (1968). The Working Set Model for Program Behaviour. *Comm. of the ACM*, 11(5):323–333.
- Evered, M. and Bögeholz, S. (2004). A Case Study in Access Control Requirements for a Health Information System. In *Proc. Austral. Inf. Secur. Workshop*, pages 53–61.
- Gofman, M. I., Ramakrishnan, C. R., Stoller, S. D., et al. (2009). Parameterized PRBAC and ARBAC Policies for a Small Health Care Facility.
- Goguen, J. A. and Meseguer, J. (1982). Security Policies and Security Models. In *Proc. IEEE Symp. on Secur. and Priv.*, pages 11–20.
- Harrison, M. A. and Ruzzo, W. L. (1978). Monotonic Protection Systems. In *Found. of Sec. Comp.*, pages 337–365.
- Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. (1975). On Protection in Operating Systems. *Oper. Syst. Rev.*, 9(5):14–24.
- Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. (1976). Protection in Operating Systems. *Comm. of the ACM*, 19(8):461–471.
- Jha, S., Li, N., Tripunitara, M. V., Wang, Q., et al. (2008). Towards Formal Verification of Role-Based Access Control Policies. *IEEE Trans. on Dep. and Sec. Comp.*, 5(4):242–255.
- Kühnhauser, W. E. and Pölck, A. (2011). Towards Access Control Model Engineering. In *Proc. 7th Int. Conf. on Inf. Syst. Secur.*, volume 7093, pages 379–382.
- Li, N. and Tripunitara, M. V. (2006). Security Analysis in Role-Based Access Control. *ACM Trans. on Inf. and Syst. Secur.*, 9(4):391–420.
- Lipton, R. J. and Snyder, L. (1977). A Linear Time Algorithm for Deciding Subject Security. *Journal of the ACM*, 24(3):455–464.
- Mondal, S., Sural, S., and Atluri, V. (2009). Towards Formal Security Analysis of GTRBAC Using Timed Automata. In *Proc. 14th ACM Symp. on Access Control Models and Technol.*, pages 33–42.
- Ranise, S., Truong, A., and Armando, A. (2014). Scalable and Precise Automated Analysis of Administrative Temporal Role-based Access Control. In *Proc. 19th ACM Symp. on Access Control Models and Technol.*, pages 103–114.
- Sandhu, R., Bhamidipati, V., and Munawer, Q. (1999). The ARBAC97 Model for Role-based Administration of Roles. *ACM Trans. on Inf. and Syst. Secur.*, 2(1):105–135.
- Sandhu, R. S. (1992). The Typed Access Matrix Model. In *Proc. IEEE Symp. on Research in Secur. and Priv.*, pages 122–136.
- Sandhu, R. S., Coyne, E. J., Feinstein, H. L., et al. (1996). Role-Based Access Control Models. *IEEE Comp.*, 29(2):38–47.
- Sasturkar, A., Yang, P., Stoller, S. D., et al. (2011). Policy Analysis for Administrative Role-Based Access Control. *Theor. Comp. Sci.*, 412(44):6208–6234.
- Schlegel, M. and Amthor, P. (2020). Beyond Administration: A Modeling Scheme Supporting the Dynamic Analysis of Role-based Access Control Policies. In *Proc. 17th Int. Conf. on Secur. and Crypt.* (to appear).

- Sharifi, A. and Tripunitara, M. V. (2013). Least-restrictive Enforcement of the Chinese Wall Security Policy. In *Proc. 18th ACM Symp. on Access Control Models and Technol.*, pages 61–72.
- Stoller, S. D., Yang, P., Gofman, M. I., et al. (2011). Symbolic Reachability Analysis for Parameterized Administrative Role-Based Access Control. *Comp. & Secur.*, 30(2–3):148–164.
- Tripunitara, M. V. and Li, N. (2007). A Theory for Comparing the Expressive Power of Access Control Models. *Journal of Comp. Secur.*, 15(2):231–272.
- Vimercati, S. D. C. d., Samarati, P., and Jajodia, S. (2005). Policies, Models, and Languages for Access Control. In *Proc. 4th Int. Workshop on Databases in Netw. Inf. Syst.*, volume 3433/2005 of *LNCS*, pages 225–237.
- Zhang, X., Li, Y., and Nalla, D. (2005). An Attribute-based Access Matrix Model. In *Proc. 2005 ACM Symp. on Applied Comp.*, pages 359–363.

