Heuristic Safety Analysis of Access Control Models

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

Winfried E. Kühnhauser  
Ilmenau University of Technology  
Germany  
winfried.kuehnhauser@tu-ilmenau.de

Anja Pölck  
Ilmenau University of Technology  
Germany  
anja.poelck@tu-ilmenau.de

ABSTRACT

Model-based security engineering uses formal security models for specifying and analyzing access control systems. Tool-based model analysis encounters a fundamental difficulty here: on the one hand, real-world access control systems generally are quite large and complex and require models that have high expressive power. On the other hand, analysis of such models is often pestered by computational complexity or even non-decidability, making it difficult to devise algorithms for automated analysis tools.

One approach to this problem is to limiting the expressive power of the modeling calculus, resulting in restrictions to the spectrum of application scenarios that can be modeled. In this paper we propose a different approach: a heuristic-based method for analyzing the safety properties of access control models with full expressive power. Aiming at generality, the paper focuses on the lineage of HRU-style, automaton-based access control models that are fundamental for modeling the dynamic behavior of contemporary role-based or attribute-based access control systems.

The paper motivates a heuristics-based approach to model analysis, describes in detail a heuristic model safety analysis algorithm, and discusses its computational complexity. The algorithm is the core of a security model analysis tool within the context of a security policy engineering workbench; a formal description of major components of its heuristic-based symbolic model execution engine is given, and its capacity to analyze complex real-world access control systems is evaluated.

Categories and Subject Descriptors

D.4.6 [Operating Systems]: Security and Protection—Access controls; D.2.4 [Software Engineering]: Software/Program Verification—Formal methods, Model checking

Keywords

Security engineering, access control systems, access control models, model safety, symbolic model execution

1. INTRODUCTION

In the last decade, advances in systems security have evolved into new system paradigms that support the design, specification, and implementation of sophisticated security concepts. Systems with advanced security requirements increasingly apply problem-specific security policies for describing, analyzing, and implementing strategic security concepts, and policy-controlled operating systems emerge that are capable of directly supporting and enforcing security policies [LS01, WV03].

Due to their key role in defining, implementing, and enforcing strategic security concepts, security policies are extremely critical, and quality assets such as correctness or consistency are essential objectives in policy engineering. On the other hand, given the large amount of their responsibilities, experiences with policy-controlled systems point out that security policies usually are large and complex, rendering analyses and quality guarantees difficult.

In order to improve policy quality, model-based security engineering uses formal security models such as [BL73, HRU76, BN89, SCFY96, CS96, EK08] for analyzing and proving critical policy properties. In the domain of access control policies, a core objective is to study the proliferation of access rights. This problem was first formalized in HRU access control models in order to find proliferation boundaries and to prove that, for a given security model, these boundaries will never be crossed – a security property known as HRU safety. Unfortunately, safety analysis of automaton-based security models has always been constricted by its general non-decidability for a large class of powerful HRU-style access control models, rendering it difficult to devise algorithms for automated safety analysis tools. As a consequence, several safety-decidable fragments of the HRU calculus emerged [HR78, LS78, San92] that bought safety decidability by limiting the expressive power of the calculus. Unfortunately, these fragments generally have severe limitations in their ability to model the complex policies of contemporary real-world systems.

Our approach to model safety analysis fundamentally differs from these approaches. Instead of restricting model expressiveness, our safety analysis technique works on unrestricted access control models and aims at a practical method for dealing with the complexity of large real-world security models. We tackle the problem by a family of heuristic model analysis algorithms that exploit the fact that the safety problem is semi-decidable: once given two states of a model we may efficiently decide whether one state renders the other unsafe with respect to a given right. The core of the algorithm is to explore a model’s state space by symbolic model execution, cutting through complexity using heuristic algorithms that exploit structural properties of a model’s authorization scheme.

The paper motivates heuristic algorithms for safety analyses, describes the most efficient member of the family of heuristic algo-
2. MODEL SAFETY ANALYSIS

This section briefly recapitulates HRU security models and the HRU safety problem, discusses their role within this work, and motivates the use of heuristic algorithms for safety analysis.

2.1 HRU Security Models

HRU security models [HRU76] are among the most powerful and general access control security models to date. In order to model dynamic behavior of access control systems, HRU security models combine access control matrices [Lam74] with deterministic state machines. Each state of an HRU model reflects a single protection state of an access control system; state transitions are triggered by system-specific operations that modify this state. Security properties such as right proliferation now can be analyzed by observing state transitions caused by input sequences; in particular, the boundaries of right proliferation can be explored by state reachability analyses.

Right proliferation analysis focuses on a fundamental family of questions: Given some model state, is it possible that a specific subject ever may obtain a specific right with respect to a specific object? Or, in terms of model abstractions, given an access control matrix (ACM), is it ever possible that a specific right is written into a specific matrix cell? If this may happen, such a state is not matrix (ACM), is it ever possible that a specific right is written.

A fundamentally different approach to safety analysis are heuristics. Aiming at generality, the problem and especially the feasibility and computational complexity of the analysis itself. The model’s basic abstractions (subjects, objects, ACMs) are aligned to the intended application area of HRU models (access control systems of operating systems) and match the needs at the time the model was devised (1975). From today’s point of view, these abstractions are rather low-level, making it difficult to actually use the model for designing and managing contemporary real-world access control systems.

Contemporary access control models provide application-friendly abstractions such as roles (RBAC) or attributes (ABAC) and define access control rules based on classes of subjects and objects. The earlier of these models only provide a rather static view on an access control system (e.g. [SCFY96]), while more recent models continue the lineage of HRU models and use deterministic automata to model dynamic behavior [ZL05, MSA09, SYGR11].

From a fundamental point of view, while high-level abstractions introduced e.g. by RBAC or ABAC models improve model usability and manageability, any such model can still be represented in the basic abstractions of a larger and more complex HRU model [KP11], in the same way that programs in high-level languages still can be (and are) represented in low-level machine language. Especially, high-level abstractions have no impact on the basic computational complexity of model analysis.

As a consequence and aiming at generality, the approach taken in this paper is to build the foundations of our work on HRU-style access control models the dynamic behavior of which is modeled by deterministic automata and then proceed to apply the results to the more application-friendly abstractions of contemporary access control models.

2.3 The Role of Heuristics

Safety properties of HRU security models in general are undecidable. In other words, there is no algorithm that, given some arbitrary HRU model, will always terminate and tell us whether a given state is safe, rendering it difficult to devise algorithms for automated safety analysis tools. Because safety-decidable fragments of the HRU calculus generally have considerable limitations in their ability to model complex real-world policies, safety analysis tools based on restricted calculi have only a limited applicability.

A fundamentally different approach to safety analysis are heuristic search algorithms. Heuristic search algorithms exploit the fact that HRU safety is semi-decidable and try to prove that some given
state $q$ is not safe with respect to a right $r$ by finding an input sequence that, starting in $q$ leaks $r$ into a matrix cell of some follow-up state $q_{target}$. To this end, each state reachable from $q$ by a finite input sequence is checked whether $r$ has leaked into a cell of its ACM. If such a target state is found, $q$ is proven to be unsafe with respect to $r$, and the state sequence from $q$ to $q_{target}$ reflects an input sequence violating the safety property. As long as no such target state is found, the search continues.

Heuristic-based approaches trade accuracy for tractability. Especially, for models with infinite state spaces where all states actually are safe with respect to all rights, the problem’s undecidability will render a heuristic search unsuccessful without the algorithm actually terminating with that result. On the other hand, valuable hints on model correctness are obtained if model analysts are pointed to unsafe states and input sequences that lead to in these states.

Starting with a state $q$ the safety of which is to be analyzed heuristic-based search algorithms symbolically execute a model and assemble all encountered states into a state transition tree – a digraph $(V,E)$, $V \subseteq Q$, $E \subseteq Q \times Q$, where each vertex $q_i$ represents a state from the model’s state space that is reachable from $q$ by $\delta'(q_i,a)$ (where $\delta'$ is the successive application of $\delta$ to an input sequence $a \in \Sigma^*$), and any direct successor of $q_i$ is a state $\delta(q_i, \alpha)$ where $\alpha$ is a single input from $\Sigma$ (Fig. 1).

The challenge now is to restrict the rapidly growing state transition tree by a proper heuristic that channels its growth. Adding a state $q_i$ to the tree corresponds to a state transition from $q_i$ to $q_i' = \delta(q_i, \alpha)$; a heuristic thus has to decide which vertex $q_i$ and which input $\alpha$ will be chosen for generating a follow-up vertex. An optimal choice of course will select $q_i$ and $\alpha$ such that $(q_i, q_i')$ is an edge on the path from $q$ to the target state $q_{target}$. Because $q_{target}$ is yet unknown, $q_i$ and $\alpha$ are heuristically chosen.

3. HEURISTIC

For heuristic approaches to be successful the heuristic should be well-tailored to the specific problem to be solved. When building the state transition tree step-by-step, the challenge is to find model properties that have an impact on the probability of a state or an input to contribute to a path from $q$ to $q_{target}$. As an example, for a right to be leaked into some matrix cell the conditions of a command entering this right must be satisfied. Because HRU authorization schemes do not check for the absence of rights, states with well-populated ACMs satisfy this condition with a higher probability and thus might be preferred.

In general, several model properties exist that provide such hints, and consequently, several heuristic metrics exist that combine into functions for selecting promising candidates for $q_i$ and $\alpha$, and many of them have been integrated in our security model engineering workbench WORS. This section focuses on our most efficient heuristic that has become the core of the DEPSEARCH safety analysis algorithm; the section motivates the heuristic, infers appropriate metrics, sketches their composition into a selection function for $q_i$ and $\alpha$, provides the formal definition of the DEPSEARCH algorithm, and discusses its computational complexity.

3.1 Heuristic Search Algorithms

Basically, the goal of each heuristic safety analysis algorithm is to find a command sequence that transits the model from a state $q$ to a state $q_{target}$. The idea behind the DEPSEARCH heuristic is that hints for a successful command sequence can be mined from a model’s authorization scheme. Because authorization schemes consist of a finite and usually manageable number of commands, this approach is fast and efficient.

The core of the DEPSEARCH heuristic is a recursive algorithm that gradually establishes necessary conditions for leaking some target right $r_{target}$. In the first step, the heuristic looks for commands in the authorization scheme that directly enter $r_{target}$ into a matrix cell: if $q$ is not safe with respect to $r_{target}$, at least one such command must exist, and the execution of at least one such command is a necessary condition for a leakage of $r_{target}$. Because a command in general is guarded by conditions, the presence of the rights of its condition in the ACM is necessary condition for its execution, and the process starts all over again with these rights added to the set of target rights, until we finally either encounter only commands that have no conditions at all, or we only have conditions that already are satisfied by the initial state $q$.

This recursive process assembles a command dependency graph (CDG, see Fig. 2) whose vertices are commands, and an edge from vertex $c_1$ to $c_2$ denotes that $c_1$ establishes at least one condition necessary for executing $c_2$. Each path to $q_{target}$ reflects a command sequence where each command has a role in establishing necessary conditions for reaching $q_{target}$. Fig. 2 shows an example CDG generated from a small authorization scheme consisting of 5 commands $c_1…c_5$. $c_5$ is a command that enters the target right into a matrix cell, and $c_1$’s conditions are already satisfied, either by $q$ or by the fact that $c_1$ simply has no conditions; command $c_2$ for example has conditions that can be satisfied by the execution of either $c_1$ or $c_4$ or both.1

![Figure 2: A Command Dependency Graph](image)

Formally, a command dependency graph of an HRU access control model is a connected digraph $CDG = (V,E)$ where

$$V = C \cup \{c_q, c_{target}\}$$

$C$ is the set of commands in the model’s authorization scheme $c_q$ is a virtual command without any conditions that enters each right present in $m_q$

1Note that a CDG is the result of a static analysis and thus only can provide necessary conditions; whether a command also establishes sufficient conditions depends on the actual state of the ACM as well as on the commands parameters when a model is symbolically executed.
\( c_{\text{target}} \) is a virtual command without any primitives that requires the target right as a condition
\[
E \subseteq V \times V \text{ where } (c_i, c_j) \in E \Leftrightarrow\\
c_i.\text{Prim. Enter. Rights} \cap c_j.\text{Cond. Rights} \neq \emptyset\text{ where}\\
c_i.\text{Prim. Enter. Rights} \text{ denotes the set of rights entered by the}\\
primitives of } c_i, \text{ and } c_j.\text{Cond. Rights} \text{ denotes the rights needed}\\
to satisfy the conditions of } c_j.
\]
\( c_q \) and \( c_{\text{target}} \) are virtual commands that are added to a CDG for convenience of the command sequence generation algorithm (Alg. 2) discussed below.

Alg. 1 builds a CDG from the authorization scheme of a model. Starting with \( c_{\text{target}} \), it recursively computes the predecessors of all vertices. The algorithm terminates when all those commands without conditions (vertices without incoming edges) are included in \( V \) for which a path to \( c_{\text{target}} \) exists.

\[\text{Algorithm 1: Dependency Graph Assembly}\]
\[
\begin{align*}
\text{Input:} & \quad \text{a model’s authorization scheme } C, \text{ a model state } q \text{ (so that we can assemble } c_q), \text{ a target right } r_{\text{target}} \\
\text{Output:} & \quad \text{a command dependency graph } CDG = (V, E) \\
\text{procedure} & \quad \text{predecessors}(v \in V) \\
& \quad \text{\{ } c \in C, c.\text{Prim. Enter. Rights} \cap v.\text{Cond. Rights} \neq \emptyset \} \\
& \quad \text{\{ } c \in P \text{ do} \\
& \quad \text{\{ } c \notin V \text{ then} \\
& \quad \quad V \leftarrow V \cup \{ c \} \\
& \quad \quad \text{predecessors}(c) \\
& \quad \quad E \leftarrow E \cup \{ (c, v) \} \\
\text{\{ } \text{assemble virtual vertices } c_q \text{ and } c_{\text{target}}; \\
& \quad C \leftarrow C \cup \{ c_q \}; \\
& \quad V \leftarrow \{ c_{\text{target}} \}; \\
& \quad E \leftarrow \emptyset; \\
& \quad \text{predecessors}(c_{\text{target}}); \\
\end{align*}
\]

Before addressing the CDG’s role in channeling the growth of the state transition tree let us first look at some of its properties.

- The existence of a path from a vertex with no incoming edges (e.g. \( c_1 \) in Fig. 2) to \( c_{\text{target}} \) is a necessary condition for \( q \) to not be safe. It is by no means sufficient, because the CDG is the result of a static analysis, and only the parameters of a command at runtime determine its actual effect. A path \( c_q, c_1, c_2, \ldots, c_{\text{target}} \) only tells us that a command sequence \( c_q, c_1^+, c_2^+, \ldots, c_{\text{target}} \) (where \( c_i^+ \) denotes the at-least-once execution of \( c_i \)) has the potential to leak \( r_{\text{target}} \).
- Paths may contain cycles; these must not be ignored, because multiple runs of the same cycle may result in different rights being entered into different matrix cells; e.g. in Fig. 2 with cycles \( c_2, c_4, c_2 \) and \( c_2, c_3, c_4, c_2 \) we only know that a successful command sequence has the pattern \( c_1^+, [c_2^+, [c_4^+, [c_3^+, c_4^+]^+]^+] \).\n- If a command sequence is found that actually leaks \( r_{\text{target}} \) and this sequence matches the shortest path in the CDG from \( c_q \) to \( c_{\text{target}} \), then this is the shortest existing sequence.

Based on these properties, the \textsc{DepSearch} heuristic makes three fundamental assumptions.

1st Assumption. Command sequences that have the property of establishing necessary conditions for a right leakage are more promising in comparison to arbitrary sequences without this property. Consequently, the \textsc{DepSearch} heuristic chooses command sequences that are paths in the CDG, beginning at a node without incoming edges (meaning that all conditions of this command are already met) and ending at a node that has an edge to \( c_{\text{target}} \). Such a path \( c_1 \ldots c_n \) is then fed into the symbolic execution engine that updates the state transition tree by executing model transitions \( \delta(q_1, c_1, x_1) = q_1, \delta(q_1, c_2, x_2) = q_2, \ldots, \delta(q_{n-1}, c_n, x_n) = q_n \), registering all states \( q_i \) as a single branch in the state transition tree (we will discuss the command parameters \( x_i \) below).

If \( q_{\text{target}} \) (any state where \( r_{\text{target}} \) has leaked into a matrix cell) is found, a command sequence was found that leaked \( r_{\text{target}} \), and \( q \) is proved to not be safe with respect to \( r_{\text{target}} \). Otherwise, we proceed with a new command sequence by generating a new path from the CDG.

2nd Assumption. If a command sequence was used once, the probability that a second execution of the same sequence will establish additional necessary condition decreases. Consequently, each path generation in the CDG decreases the probability that the same path will be generated again. Especially, each run of a cycle decreases the probability of a second run.

Generating different paths follows the idea of ant algorithms, although in a variation where the scent of an edge acts repelling. Scent is implemented by an edge attribute (\textit{flavor} in Alg. 2) where the flavor of an edge is incremented whenever it is used in a path. Thus paths are always minimal with respect to the accumulated scent of its edges. Scent minimality enforces path diversity, dealing with the situation that commands can have conditions that can only be satisfied by prior execution of two or more different commands (advertised by more than one incoming edges in the CDG).

3rd Assumption. If the same path is used more than once, then a modification of the command’s parameters improves the probability that the same path will establish new necessary conditions. Consequently, the \textsc{DepSearch} algorithm outlined in Alg. 3 modifies the command parameters in each run.

These three assumptions have been combined into a path generation algorithm outlined in Alg. 2.

\[\text{Algorithm 2: CDG Path Generation}\]
\[
\begin{align*}
\text{Input:} & \quad \text{a CDG as generated by Alg. 1, } c_{\text{target}} \\
\text{Output:} & \quad \text{a path in the CDG, represented as a sequence of} \\
& \quad \text{vertices; a modified CDG where all edges of this path} \\
& \quad \text{have a stronger flavor} \\
\text{currentVertex} \leftarrow c_{\text{target}}; \\
\text{path} \leftarrow \text{currentVertex}; \\
\text{repeat} & \quad \text{edge} \leftarrow \text{lowestFlavor(currentVertex.incomingEdges)}; \\
& \quad \text{currentVertex} \leftarrow \text{edge.origin}; \\
& \quad \text{path} \leftarrow \text{path} + \text{currentVertex}; \\
& \quad \text{edge.flavor} \leftarrow \text{edge.flavor} + 1; \\
\text{until} & \quad \text{currentVertex.incomingEdges} = \emptyset; \\
\end{align*}
\]

3.2 The \textsc{DepSearch} Algorithm

Starting at \( q \), the \textsc{DepSearch} safety analysis algorithm explores the state space of a model in order to find a state where \( r_{\text{target}} \) has leaked into some matrix cell. The growth of the state transition tree is channeled by generating command sequences that gradually establish necessary conditions for the model to reach \( q_{\text{target}} \). Command sequences repeatedly are generated according to the heuristic discussed in section 3.1 by Alg. 2.
Algorithm 3: DepSearch Algorithm

Input: a model's authorization scheme $C$, a model state $q$, a target right $r_{target}$
Output: a state transition sequence $STS$ that leaks $r_{target}$

$CDG \leftarrow$ DependencyGraphAssembly($C, q, r_{target}$);
$STS \leftarrow q$;
repeat
    $path \leftarrow$ CDGPathGeneration($CDG, r_{target}$);
    while $c \leftarrow path.next$ do
        $q \leftarrow \delta(q, c, s_{param}(q, c))$;
        $STS \leftarrow STS + q$;
    until $q$ contains right leakage;

Two problems remain to be addressed: termination conditions and the selection of a parameter vector for each state transition. With respect to termination, the algorithm of course does not terminate if $q$ is safe with respect to $r_{target}$. This problem will be addressed in the evaluation section 4.1. Selection of a parameter vector is discussed in the next section.

3.3 Command Parameter Selection

The problem of selecting a command’s parameter vector is basically a constraint satisfaction problem [Lau78, JM94]. A constraint satisfaction problem (CSP) is a tuple $(V, D, P)$ with a set of variables $V = \{v_1, ..., v_k\}$ each of which is instantiated in a particular domain $D = \{D_1, ..., D_k\}$ where $D_i$ defines the range of value of $v_i$. $P$ is a set of predicates defining constraints that the values of the variables must simultaneously satisfy. A solution of a CSP is an assignment of values to the variables which satisfies all constraints.

The problem of finding “good” parameters for a command can be modeled as a CSP by perceiving the set of formal parameters as the set $V$, the domains being either $S_q$ or $O_q$ (depending on the formal parameter being a subject or object), and the constraints defining restrictions on the variables such that only “good” values solve the CSP. Then, any one of the several well known CSP solving algorithms (e.g. AC-3 [Mac77]) can be applied.

Whether parameters are “good” or “bad” is decided by a heuristic metric that reflects the probability of a parameter vector to promote right leakages. As already argued in the introduction of this section, states with large numbers of rights generally provide better chances for right leakages. Given a state $q$, a command $c$, and a parameter vector $x$, this can easily be captured by a simple heuristic metric $h_{param}$ comparing the number of rights in the matrices of $q$ and $q'$ where $q'$ is the follow-up state of $q$ obtained by $\delta(q, c, x)$. Thus $h_{param} : Q \times C \times (S \cup O)^k \rightarrow \mathbb{Z}$ is computed by

$$h_{param}(q, c, x) \Rightarrow \sum_{(s,x) \in S_q \times O_q} |m_q'(s, o)| - \sum_{(s,x) \in S_q \times O_q} |m_q(s, o)| .$$

Parameter selection then is captured by a function $s_{param} : Q \times C \rightarrow (S \cup O)^k$ that is computed by a CSP solving algorithm where

$$s_{param}(q, c) \Rightarrow x \in (S_q \cup O_q)^k \text{ such that }$$

$$h_{param}(q, c, x) = \max_{z \in (S_q \cup O_q)^k} h_{param}(q, c, z) .$$

3.4 DepSearch Summary

Concluding, we have presented a heuristic safety analysis algorithm for security models of the HRU automaton-based lineage. The heuristic is based on static analysis of a model’s authorization scheme, similar to the idea that led to the Bell/LaPadula basic security theorem for proving BLP model security [BL76].

Computational complexity of assembling the CDG is $O(n^2)$, where $n$ is the number of commands in a model’s authorization scheme. Generating a path in the CDG runs in $O(n)$, because due to the minimum scent of each path, eventual cycles occur at most once in a path. For selecting the parameters for a command execution we currently use a brute force approach running in $O(max(|S_q|, |O_q|)^p)$, where $S_q$ and $O_q$ are the subject and object sets in $q$, and $p$ denotes the maximum number of parameters in a model’s authorization scheme.

The DepSearch heuristic originally was developed as a consequence of failures of earlier heuristics in analyzing atypical models where right leakages were well hidden and appeared only after long command sequences where each command depended exactly on the execution of its predecessor. For such models, DepSearch has turned out to be quite successful, and its application to more regular models also demonstrated its superiority to earlier approaches.

4. EVALUATION

This section is an evaluation of the heuristic analysis approach and addresses two major points: (i) practical feasibility in terms of runtime performance for analyzing real-world access control models and (ii) relative quality of the heuristic. Subject of the evaluation is the DepSearch heuristic algorithm.

4.1 Evaluation Method

Practical feasibility is evaluated using two different model types: Role-based Access Control (RBAC) models demonstrate the analysis of a real-world scenario, and synthetic High-Dependency Models that maliciously are designed to stress-test the analysis with well-hidden right leaks. The RBAC models are based on a health care security policy and used to demonstrate the performance of the DepSearch algorithm on a realistic authorization scheme. High-Dependency Models on the other hand will serve as a benchmark-test that imposes maximal calculation effort.

The relative quality of the DepSearch algorithm is evaluated by comparing it with a randomized heuristic, a simple heuristic that does not exploit any specific model knowledge and that serves as a baseline. The same real-world and synthetic model types as mentioned above are used as test cases.

The basic test procedure is executing our DepSearch implementation to find a specific right leakage. Therefore, randomly initialized ACMs have been generated for each model, featuring an increasing number of matrix cells. This way, we have covered the two main indicators of analysis performance in practice: general matrix size and authorization scheme complexity in terms of command dependencies.

The remainder of this section addresses the evaluation method: it describes the two model types and the RANDOM heuristic (4.1.1), the execution environment “WorSE” (4.1.2), and finally test conditions such as execution parameters and evaluation metrics (4.1.3).

4.1.1 Test Cases

RBAC Models.

The RBAC models used in this paper are generally based on a security policy of a real-world Health Information System (HIS) for an aged-care facility introduced by [EB04]. This policy was extended and rendered more precisely by [GRSY09, SYGR09, SYRGO7] to develop a formal security model. We have enhanced this model by a state automaton along with an authorization scheme.
for modeling dynamic behavior of the policy. The result is an RBAC model based on [SFK00] with 20 roles, 15 commands in the authorization scheme, a role hierarchy, and separation-of-duty focusing on role exclusion.

When analyzing the dynamic behavior of RBAC models, two main questions are significant. (i) Given some model state, is it possible that a specific user is ever assigned a specific role? In terms of model abstractions, we have to analyze the behavior of the user-to-role assignment (UA) relation, which is a many-to-many relation between users and roles. More precisely, given a user-to-role assignment relation, we have to monitor whether a tuple \((u, r)\) with a specific user \(u\) and a specific role \(r\) can under a given authorization scheme become an element in this relation. If this may happen, such a state is not considered \textit{safe} with respect to that role. Analogously, the second question deals with the permission-to-role assignment (PA) relation, which is a many-to-many relation between roles and permissions: (ii) Given some model state, is it possible that a specific role is assigned a specific permission, i.e. a specific operation on a specific object? In this case, we call such a state not \textit{safe} with respect to that operation. Both of these safety flavors are quite relevant in practice.

Since the \textsc{DepSearch} algorithm focuses on HRU-style model abstractions, we rewrote the RBAC model in HRU style. In analogy to [SYRGO7, SYSR06], we omit sessions because the above addressed safety properties are independent of sessions. Additionally, due to demonstration purposes, we adopt a simplified model where object attributes for mapping users to their objects, e.g. the electronic health record of a specific patient, are also omitted. This is reasonable because on the one hand it promotes unfogged evaluation results, and on the other it is no restriction to generality because object attributes may be reproduced by simply using the product set of objects and attributes. Result of this model rewriting are two HRU-type models \textit{HIS I} and \textit{HIS II}, each of which focusing on certain parts of the original RBAC model.

The \textit{HIS I} model focuses on the user-to-role assignment and separation-of-duty based on role exclusion. Its state \(q = (S_q, O_q, m_q)\) 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 \(m_q : S_q \times O_q \rightarrow 2^{\text{roles}}\) which maps a user-object-pair to a set of roles. Any command of the authorization scheme is then guarded by at least one condition of the following type: a user has to own a specific role for a specific object (cf. authorization scheme specificiation, section 2.1). The \textit{HIS I} model implements role exclusion by negative roles: since conditions in HRU-like models cannot test the absence of rights in the ACM, we add one negative role for each role, that simulates its absence. Finally, for modeling the health care policy, our authorization scheme contains \(7\) commands that modify the user-to-role assignment of the original RBAC model under consideration of its role exclusion relation. Analyzing the \textit{HIS I} with respect to HRU safety is then equivalent to analyzing the original model regarding RBAC safety flavor (i).

The \textit{HIS II} model implements the permission-to-role assignment and the role hierarchy of the original model. As in \textit{HIS I}, its state \(q = (S_q, O_q, m_q)\) includes the object set of the original RBAC model \(O_q\). \(S_q\) is now defined as the role set of the original RBAC model and the ACM \(m_q : S_q \times O_q \rightarrow 2^{\text{operations}}\) maps a role-object-pair to a set of operations. The ACM hence exactly represents the PA relation of the original model, omitting the indirection level of permissions. The authorization scheme incorporates the role hierarchy relation by solving the level of indirections of PA, resulting in a HRU-like RBAC model with 7 commands. HRU safety analyses of the \textit{HIS II} model equals the analysis of the original RBAC model regarding safety flavor (ii).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig3.png}
\caption{CDG of High-Dependency Models}
\end{figure}

\textbf{High-Dependency Models.}

As mentioned earlier, the motivation for the \textsc{DepSearch} approach was to analyze a special case of access control models, featuring sequences of causally dependent commands that require each predecessor’s execution to satisfy their conditions. Such models have proven to be a worst-case scenario for previous heuristic analysis approaches; they highlight a type of complexity that is effectively part of almost any real-world security policy. For this reason, two of these models will serve as an artificial benchmark-test to scale up this dimension of authorization scheme complexity. They are used to judge both feasibility in terms of absolute runtime as well as relative quality of \textsc{DepSearch} compared to \textsc{Random}.

The high-dependency models, referred to as \textit{High-Dep I} and \textit{High-Dep II}, are traditional HRU models, solely designed to incorporate many different command dependencies. Their initial state and authorization scheme implement the dependencies shown in Fig. 3: In \textit{High-Dep I}, the target right \(r_5\) is entered by command \(c_4\). Each label of an edge \((c_i, c_j)\) denotes a right that is entered by \(c_i\) and required by \(c_j\); the initial state \(q_0\) is chosen so that \(\forall (s, o) \in S_q \times O_q : \{r_2, \ldots, r_5\} \notin m(s, o)\) (apart from this, all matrix cells are initialized randomly based on a generic right set \(R = \{r_1, \ldots, r_{10}\}\). No command requires or enters more than one right out of \(\{r_2, \ldots, r_5\}\). \textit{High-Dep II} is analogously constructed (target right \(r_{13}\), leaked by \(c_{10}\), just with the more complex dependencies shown by Fig. 3(b). Note that in case of our test 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 matrix. Therefore, a minimum number of state transitions is required to discover a right leakage.

\textbf{Random Heuristic.}

The \textsc{Random} heuristic serves as a baseline to judge the performance of \textsc{DepSearch}, since it uses a poor algorithm that requires zero knowledge about any part of the model. Its basic mode of operation is simple: it selects (i) a random state \(q\) from the state transition tree, (ii) a random command from the authorization scheme, and (iii) a random parameter vector \(x \in (S_q \cup O_q)^k\). Each of these selections runs in \(O(1)\), so each input set generation by \textsc{Random} takes constant time.

Yet the runtime behavior of \textsc{Random} is not uniformly distributed. While for a small initial matrix the brute force approach tends to be quite fast, for larger matrices it becomes increasingly
unlikely to find a viable parameter vector that actually results in a state transition. However, since no heuristic knowledge about the model is used, there is of course no guarantee that even frequent state transitions gain progress towards finding the desired right leakage, so the overall success of this heuristic is largely random.

### 4.1.2 Execution Environment

The experimental evaluation of the DepSEARCH heuristic is performed in the security policy engineering framework “WorSE”. WorSE enables automated symbolic execution of access control models employing variable heuristic algorithms. This section outlines the WorSE implementation of the DepSEARCH algorithm and its data structures.

The overall architecture of WorSE is module-based, featuring tools for different security engineering and analysis problems. The DepSEARCH algorithm has been implemented as a module of the model safety analysis tool. At present, WorSE focuses on symbolic execution of HRU-type access control models. However, since our goal is to provide method and tool support for analyzing more up-to-date security models like role-based or attribute-based access control models, a symbolic model execution engine class (SMEE class) based on a generic deterministic state machine is implemented here. For evaluating the DepSEARCH heuristic, we used a HRU automaton class derived from the SMEE class. The implementation of the algorithm itself is likewise derived from a heuristic base class\(^2\) that allows for using and comparing different heuristics independent of the actual model. The model itself is a shared data structure accessible from both the SMEE and the heuristic, which basically contains the state transition tree and the authorization scheme.

Both the SMEE and the heuristic interact via a tuple space, a flexible, anonymous communication architecture which allows for an easy integration of new SMEEs (such as for role-based models) and heuristics into WorSE. The interaction basically follows a simple pattern: Whenever the current heuristic is executed, it selects a particular state from the state transition tree, a command from the authorization scheme, and a parameter vector. This information is sent to the SMEE, eventually triggering a state transition, depending on the given combination of state, command, and parameters. Possible reasons for a failing state transition may be that conditions of the given command cannot be satisfied using the given state and parameter vector, or that primitives of the given command do not affect the given state at all (if it only enters rights that are already present in the matrix and/or only deletes rights that are not). If, on the other hand, a successful state transition has occurred, the follow-up state is inserted into the state transition tree (see section 2.3).

The result of the transition attempt is sent back to the heuristic that is then re-executed on the (possibly enhanced) state tree. Each of these iterations is called a step. Since a heuristic step does not necessarily trigger a transition to a new state, we distinguish those steps that actually do by calling them effective in the following.

WorSE is under ongoing development. Current work focuses on enhancing its analysis tool to support automaton-based access control models beyond HRU, such as dynamic RBAC models.

**4.1.3 Test Conditions**

All test models used for evaluation are not safe by construction, i.e. we will only analyze known-positive cases of the safety problem. However, due to its semi-decidable nature, our algorithm implementation has to decide when to terminate without result based on a certain criterion. For this purpose, the overall number of heuristic steps is limited to a certain upper bound, called step limit, which is left to the user for the following reason: The concrete number of necessary steps heavily depends on the operation principle of a certain heuristic. With respect to a particular access control model, only a “model administrator” (who knows about the context and impact of the safety property of this model) can decide, how many steps are feasible to make a justified statement about safety using the given heuristic. For the purpose of this evaluation, we will define the step limit as part of an input configuration (see below). Due to the generally large number of heuristic steps performed by the Random algorithm, all runs using this heuristic are assigned a step limit one order of magnitude greater than the corresponding run using DepSEARCH.

**Test Parameters.**

The following parameters determine an input configuration of our test runs: Initial subject and object count \(|S_0|\) and \(|O_0|\), ACM initialization \(|m_0|\), target right, and step limit. Here, the object count of the initial state and its respective matrix contents are varied, whereas the initial subject count is kept constant (since only the number of cells in the matrix impacts the behavior of both heuristics, not their layout). We used matrix dimensions starting from 20x20 and ending with 20x500, increasing in steps of 20 objects (i.e. 400 cells); the initial matrix contents are randomized using a fixed, model-specific right set. Moreover, the target right (that safety is analyzed for) and the step limit are fixed, but also model-specific. In case of the step limit, values of 1,000,000 and 10,000,000 are used.

For each of these configurations, at least 10 successful runs per model have been performed (plus those aborted without result), with one exception: For the High-Dep II model, the Random algorithm did not terminate within the steps limit for any input (including those featuring the smallest, 20x20 matrix). To this end, we did not include this model in our comparison of heuristics quality.

The WorSE framework is implemented in C++ under Ubuntu Linux 12.04. All runs were performed on contemporary desktop hardware (Intel Core i7-2600 3.4 GHz CPU, 16 GB RAM).

**Metrics.**

Three different measures are used in the following: Effective step count (ESC), effective step time (EST), and total runtime of a heuristic. The ESC measure is the total count of effective steps required to leak a target right, while EST is the average time required by a heuristic to make an effective step (including any ineffective steps within that time). For the feasibility criteria, we performed a straight-forward runtime comparison of the DepSEARCH algorithm for both model types, that allows for educated performance estimations in a real-world analysis session regarding both average and worst-case runtime. The quality comparison with the Random algorithm turns out to be slightly more difficult due to the fundamentally different operation of both heuristics. In addition to total runtime, we use the measures of ESC and EST here.

### 4.2 Evaluation Results

We will now discuss the results of the experimental evaluation with respect to feasible runtime performance and heuristic quality.

#### 4.2.1 Feasible Runtime Performance

We compared the total runtime of the DepSEARCH algorithm for all four access control models with matrix sizes from 20x20 to 20x500 cells. The results are shown in Fig. 4.
Figure 4: Runtime performance of the DepSearch heuristic (error bars show standard deviation)

While still bound by the $O(n^4)$ complexity of parameter selection (cf. section 3.4), we argue that the absolute runtime behavior on the used, common hardware suggests reasonable feasibility in practice: Even for a worst-case in model complexity, illustrated by the High-Dep II model, the absolute runtime (up to several thousands of seconds) does not exceed the scale of a real-world analysis session. Also, it is observable that on both real-world models (HIS I and HIS II), DepSearch does not show a significant difference in runtime behavior due to the common number of command dependencies (cf. “Minimal ESC” in Tab. 1).

However, one should bear in mind that we base the assessment of these numbers on two assumptions: (1) ACMs of several thousand cells are a realistic quantitative bound for most contemporary access control systems; (2) the complexity of real-world access control policies does not exceed the degree of dependencies imposed by our High-Dep models (independent from application domain). Assumption 1 may be too strong in a traditional IBAC context, e.g. when analyzing a file system on Unix-permission level. However, according to our belief, it should be suitable in context of contemporary access control policies that are usually written on a much higher level of abstraction and thus also analyzed on that level. An example are policies based on an RBAC model, such as modeled by HIS I and HIS II. Assumption 2 appears legitimate, since the design of our High-Dep models (strict single-permission dependencies over several state-modifying operations in succession) is largely removed from realistic policy design goals and thus provides a reasonable upper bound for the degree of dependency imposed by current real-world access control policies.

Finally, it should be noted that our present implementation still leaves room for improvement in runtime performance. Besides this, current work focuses on an advanced heuristic parameter selection algorithm that may break this major drawback in runtime: One approach here is to weight the rights that are entered, exploiting already gathered information such as search path length or previously used matrix cells.

4.2.2 Heuristic Quality
For quality evaluation, we compared the DepSearch performance with the RANDOM algorithm as a baseline.

Effective Step Quality.
Table 1 shows the mean ESC of both algorithms for each model, along with its standard deviation and the percentage of aborted runs. These values apply to all successful runs, since matrix size is not relevant here: The ESC value of both algorithms only depends on the model’s authorization scheme. To judge heuristic quality based on ESC, the ratio of minimal to mean ESC is given, called effective step quality ($Q_{ESC}$).

<table>
<thead>
<tr>
<th></th>
<th>HIS I</th>
<th>HIS II</th>
<th>High-Dep I</th>
<th>High-Dep II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimal ESC</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>Mean ESC</td>
<td>136.3</td>
<td>143.6</td>
<td>10,758.9</td>
<td>–</td>
</tr>
<tr>
<td>Std. Dev.</td>
<td>134.4</td>
<td>138.8</td>
<td>9,570.3</td>
<td>–</td>
</tr>
<tr>
<td>$Q_{ESC}$</td>
<td>0.015</td>
<td>0.014</td>
<td>$&lt; 0.01$</td>
<td>$&lt; 0.01$</td>
</tr>
<tr>
<td>Abort rate</td>
<td>0%</td>
<td>0%</td>
<td>21%</td>
<td>100%</td>
</tr>
</tbody>
</table>

**Table 1: Mean ESC and effective step quality**

As expected, the target-oriented DepSearch approach only requires a fraction of the state transitions needed by RANDOM. More important, $Q_{ESC}$ is nearly optimal for DepSearch on all models but the most complex High-Dep II. This means that, judged by pure path length in the state transition tree, DepSearch outperforms the brute-force approach even in the low-dependency models HIS I and HIS II by about two orders of magnitude.

With respect to the high-dependency models, it becomes clear that a zero-knowledge approach as employed by RANDOM is futile: As soon as authorization scheme complexity rises only slightly in terms of dependency, the ESC of this heuristic increases exponentially and, consequently, so do total step count and runtime. This led to a 100% abort rate in case of the High-Dep II model, even under a step limit of 10,000,000.

Runtime Quality.
To evaluate the heuristic quality based on runtime performance, both EST and total runtime of DepSearch and RANDOM were compared for each model. As mentioned above, the High-Dep II model could not be analyzed with the RANDOM heuristic under the given step limit and is thus excluded here.

Fig. 5 depicts a per-model comparison of DepSearch and Random heuristics regarding average EST. It can be noticed that the huge impact of DepSearch on effective step quality does not equally reflect in per-step runtime: In case of both HIS models, DepSearch EST largely exceeds that of Random, which raises by magnitudes slower. Even in case of High-Dep I, where Random features a $Q_{ESC}$ several orders of magnitudes worse, the EST is on a comparable level here. This striking discrepancy is explainable with the parameter selection performance, that features $O(n^4)$ runtime (for a matrix size $n$) for DepSearch, while the Random approach requires constant time. This result emphasizes the importance to eliminate this algorithmic bottleneck in order to take full advantage of the efficient dependency-based path generation algorithm of the DepSearch heuristic.

The average total runtime of both algorithms is compared in Fig. 6. Here, DepSearch generally outperforms Random. This
becomes especially clear in case of HIS I and High-Dep I, were the idea of a dependency-based path to approach a leaking state pays off. The brute force approach, in contrast, does not use any additional information to narrow down the state space; consequently, a heavily varying runtime with large standard deviations is observable. In accordance to the ESC evaluation, the High-Dep model shows again that a slight rise in authorization scheme complexity yields dramatic performance drops for the RANDOM approach.

As a side note, Fig. 6(b) illustrates a difference in model complexity between HIS I and HIS II that is not directly related to command dependencies: since HIS II commands include maximal 2 conditions (as opposed to 6 in HIS I), a notable improvement of EST is possible here for the probabilistic parameter selection of the RANDOM heuristic (cf. Fig. 5(b)). The DEPSearch heuristic on the other hand tries to minimize the number of steps by always selecting parameters that satisfy all conditions of the current command; consequently, average EST does not improve in the same extent.

A comparison of Figs. 5 and 6 emphasizes that pure throughput in triggering state transitions is not a crucial parameter for overall performance of the safety-analysis, while exploiting model-specific knowledge about structural properties of the authorization scheme and state components can lead to a significant increase in analysis quality.

5. RELATED WORK

Extensive research work has been done to develop methods that analyze specific properties of security policies. This work can be classified in several ways; in terms of analysis goals we distinguish between static and dynamic policy analyses.

Much work has focused on static analysis, which analyses a specific policy state with respect to some security requirements. For example, [NRN07] develops a deterministic heuristic using a flow logic approach. The authors have demonstrated their work by means of the Bell-LaPadula model, where the heuristic analyzes entity classifications against a policy’s mandatory part. In case of policy violation, the heuristic suggests actions to reclassify specific entities such that the classifications become legitimate. On the other hand, [JZ03] has developed and implemented an analysis algorithm for access control models, which system administrators can apply for analyzing real-world policies, e.g. an SELinux policy, regarding incomplete or conflicting right assignments.

The approach in this paper focuses on dynamic policy analysis like [ZLN05, KN07, LT06, SYGR11, JGT+11, MSA09, AAR11]. The latter have in common that their analysis methods are developed for specific security models. For example, while [ZLN05] develops an HRU-style ABAC model and analyses the decidability of its safety problem, [LT06] deals with safety analysis for RBAC models. Following, others such as [SYGR11, JGT+11, MSA09, AAR11] have researched analysis methods for specialized variants of RBAC models. In [JGT+11], an iterative abstraction-refinement approach is used to perform symbolic execution and leakage search in administrative RBAC policies. As with our method, the algorithm cannot terminate with a “no leakage” result due to the semi-decidability of the reachability problem.

Most closely related to our DEPSearch heuristic is [SYGR11]. Here, the authors have developed an algorithm for analyzing user-
role-reachability for parameterized administrative RBAC models. Just like DepSEARCH, their algorithm contains two stages: The first stage performs a backward search from the goal to the initial state; the second stage runs a forward search from the initial state by limiting the search based on the results of the first stage [SYGR11]. Analogously, DepSEARCH first generates the CDG, which is then used to limit the search by command sequences. The significant difference is that our heuristics have not been developed for a specific security model, but to analyze any automaton-based model and their respective safety problems. This is also the reason why DepSEARCH in general cannot prove the absence of a right leakage (policy safety). It should be noted that the algorithm in [SYGR11] achieves decidability under some conditions by exploiting specific properties of a model calculus more restrictive than general HRU. Thus, the termination issue with DepSEARCH is part of the trade-off between accuracy and generality. However, our approach still leaves room for model-dependent tuning of the heuristic based on a more constrained authorization scheme, e.g. involving role hierarchies or separation-of-duty constraints.

6. CONCLUSIONS

This paper addresses the computational complexity problem of access control model safety analysis. The idea is to use heuristic analysis algorithms that exploit model-specific properties gained from a static analysis of a model’s authorization scheme, resulting in necessary conditions for safety violations. These conditions then are used for restricting model input sequences that are fed into a symbolic model execution engine and guide the model to an unsafe state.

The paper focuses on the lineage of HRU-style, automaton-based access control models with sufficient power to model the dynamic behavior of real-world access control systems. The results provide the foundations for heuristic-based safety analysis of contemporary role-based or attribute-based access control models that integrate automata to model their dynamic behavior.

There is a huge potential in static model-specific properties that still remains to be exploited. The effectiveness of the necessary conditions restricting the search paths will be fueled by models with constraints in the authorization scheme such as RBAC2. For boosting the success of a search path, the command parameter selection algorithm can be furnished with a scheme that exploits hot spots in the ACM similar to the concept of working sets in virtual memory management algorithms. Then, in role-based or attribute-based models the focus of analysis goals can be narrowed down, distinguishing e.g. in RBAC models between role-safety, permission-safety, or session-safety, or in ABAC models between the safety of different attributes.

An implementation of the analysis heuristic was integrated into the security policy engineering framework “WorSE”. Its evaluation points out that the approach allows for safety analysis in many practical cases.

7. REFERENCES


[KN07] Eldgar Kleiner and Tom Newcomb. On the


