WorSE: A Workbench for Model-based Security Engineering

Peter Amthor, Winfried E. Kühnhauser, Anja Pölck

Ilmenau University of Technology
PO Box 100565, 98684 Ilmenau, Germany
Phone: +49 (0) 3677 69 4163
Fax: +49 (0) 3677 69 4541

Abstract

IT systems with sophisticated security requirements increasingly apply problem-specific security policies for specifying, analyzing, and implementing security properties. Due to their key role for defining and enforcing strategic security concepts, security policies are extremely critical, and quality assets such as policy correctness or policy consistency are essential objectives in policy engineering.

This paper argues for a tool-supported policy engineering approach to increase the efficiency and quality of security policy making. The paper’s general topic is WorSE, a policy engineering workbench encompassing the automation of engineering steps, pre-built model patterns, integrated plausibility checks, and model analysis tools; the paper especially focuses on tools supporting model engineering and model analysis, and describes their theoretical foundations and practical application.

Keywords: security engineering, security policy engineering, model safety analysis, information flow analysis, security model checking

1. Introduction

In the last decade, advances in systems security have evolved into new system paradigms that support the design, specification and implementation of so-
phisticated 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 system kernels emerge that are capable of directly enforcing security policies (Loscocco and Smalley, 2001; Watson and Vance, 2003; Faden, 2006).

Due to their key role for defining, implementing and enforcing strategic security concepts, security polices are extremely critical software components, and quality assets such as policy correctness or policy 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 the analysis and proof of crucial quality attributes difficult. As an example, the current reference policy for the SELinux operating system (a basic policy not yet configured for a specific operational environment) has approximately 50,000 lines of pure policy code (PeBenito et al., 2006). Criticality of security policies on the one hand and their complexity on the other imply that it is imperative to apply policy engineering methods for quality assurance.

A promising approach to support quality aspects of security policies is to apply formal methods for precisely specifying policy semantics, putting policy specifications in reach of formal analysis methods that verify critical properties such as model safety, completeness, or consistency (Harrison and Ruzzo, 1978; Sandhu, 1988; Kleiner and Newcomb, 2007; Stoller et al., 2007; Naldurg and Raghaven-dra, 2011; Stoller et al., 2011). This paper discusses a model-based approach to security policy engineering and focuses on tool-supported policy specification and analysis. The approach exploits the analytical power of model-based policy specifications, using formal security models (Harrison et al., 1976; Bell and LaPadula, 1976; Sandhu et al., 1996; Zanin and Mancini, 2004; Zhang et al., 2005; Hicks et al., 2010) as a foundation for tool-supported analysis and verification of critical policy properties.

2. Model-based Security Policy Engineering

The work described here seeks to contribute to a systematic and tool-supported approach to security policy engineering using formal security models. In this section, we will describe our notion of an ideal engineering process that reflects this idea. This process model will highlight the most critical and error-prone steps, that call for a strong analytical methodology to ensure fundamental quality assets of a security policy implementation. A brief look on existing projects that
provide tool-support for security engineering tasks and their application on these engineering steps will motivate the importance of a general, tool-supported policy engineering methodology.

We will then introduce WorSE, a prototypical framework to implement this methodology, and give an overview of its principle use cases.

2.1. Process Model

Security policies in general can be described as systems of rules or algorithms written in formal languages, not different from regular software systems. Thus our understanding of model-based security policy engineering is inspired by an abstract software engineering process model (Fig. A.1).

In the first step of this process, informal security requirements are stated by a thorough requirements engineering. Since this problem is already familiar from software engineering, several projects of the last years have focused on specialized approaches for security requirements analysis (Felderer et al., 2011; Katt et al., 2010; Innerhofer-Oberperfler and Breu, 2006). In the next step, a set of operational rules is composed that describe the way an IT system has to meet these requirements, resulting in a security policy that suits the specific needs of the intended application domain (Strembeck, 2010; Giuri and Iglio, 1997; Martino et al., 2008). In order to get a sound foundation for its correct implementation, we may now rewrite this policy by formal means, either using a suitable, existing security model or creating a new one. We hence call this third step model engineering. Once formalized in a model, the policy is analyzed using methods and tools that operate on this specific model. During this step, the compliance of the formalized security policy (which we call an instance of its particular security model) with security properties is checked, such as inner consistency (Jackson, 2002; Jha et al., 2008), access right proliferation (Harrison and Ruzzo, 1978; Sandhu, 1992a; Amthor et al., 2011), information flow leakage (Denning, 1976; Volpano et al., 1996; Guttman et al., 2003; Shaffer et al., 2008), or implementation adequacy (Bell and LaPadula, 1976; Zanin and Mancini, 2004). These are derived from quality parameters of the application domain and are formally defined in terms of the security model. That way, a valid formal specification is provided for the subsequent implementation of the policy.

Since the term “security model” has been heterogeneously used in the literature, its meaning in the scope of this paper should be clarified. A security model is a set of abstractions to precisely describe the semantics of a security policy, such as object classification, role-based access rules or auditing and logging practices. Additionally, a security model defines analytical security properties that may be
verified on a model instance, i.e. a particular security policy described using that model’s abstractions.

As the above description suggests, the main difference to a software engineering process lies in the paramount criticality of the goal security. This goal is reflected in security properties such as consistency, control of access right proliferation, or completeness of rules that may be expressed and analyzed by formal means. Generally, just as in software verification, the analysis of these properties is formally difficult as well as expensive. For this reason, comprehensive tool support for the whole process is highly needed.

2.2. Existing Tools for Model-based Security Engineering

There is already a wide range of tools for modeling, analyzing, and specifying security policies. Goal of this section is to show their broad scope and high degree of specialization in particular domains. In the following, we briefly present a selection of existing work.

**SPARK Examiner** is a verification tool for the SPARK programming language (Barnes, 2003). It has been used to verify information-flow properties (Chapman and Hilton, 2004) in a way that extends the SPARK language to implement and analyze the classical BLP model.

**Alloy Analyzer** is a verification tool for the Alloy specification language (Jackson, 2002). It has been applied on a formal security domain model (Shaffer et al., 2008) to identify covert channels by tracing the execution flow of a program.

**SEAL** is a specification language focused on finding errors in dynamic multi-level security policies (Naldurg and Raghavendra, 2011). It has been used to analyze vulnerabilities in a Windows-style access control system.

**Mohawk** is a model-checking tool specialized on analyzing administrative role-based access control (ARBAC) policies (Jayaraman et al., 2011, 2013). It is tailored to find undesired user-role-attributions, design errors that are specific to this policy family.

**Apol** (Tresys Technology, 2013) is a specialized tool by Tresys Technology for the analysis of security policies for the SELinux operating system (Loscocco and Smalley, 2001). It uses the original model of the SELinux policy specification language to design and analyze a given policy.

**SLAT** (Guttman et al., 2003) is an information flow oriented rewriting of the SELinux policy model. It has been used to analyze information flow properties of a real-world security policy (Sarna-Starosta and Stoller, 2004). Other tools that support information-flow analysis of SELinux security policies are based on
colored Petri nets (Ahn et al., 2008) and logical programming (*PALMS*) (Hicks et al., 2010).

SecureMOVA is a tool for security-related software modeling using its own specification language based on UML (SecureUML) (Basin et al., 2009). As with *UMLsec* (Jürjens, 2002) and its analysis tools (e.g. Jürjens and Fox (2006)), SecureMOVA is primarily designed for software specification.

Each of these examples uses some kind of formal model to tackle specific security engineering problems. All of them are however highly specialized in (1) certain policy domains and, consequently, distinct classes of security models and (2) certain policy-specific analysis and design tasks. Such problem-driven tool support is of limited value for a general approach for efficient security policy engineering. Both model-involving steps of the process model described above, model engineering and model analysis, require methods for policy-independent design, instantiation, analysis and implementation of security models. This motivation has driven the development of WorSE (“Workbench for Model-based Security Engineering”), a policy engineering workbench that seeks to meet these requirements.

### 2.3. WorSE: An Overview

Our workbench has the goal to facilitate the development, analysis and implementation of security models in a security policy engineering process. This is achieved by

- supporting model engineering efficiency,
- supporting the quality of the resulting security model, and
- supporting the security policy engineer with ready-to-use expertise in model analysis methods.

These goals span the two most critical steps in this process: The step of *security model engineering*, where the formal model for a policy is designed in the first place, and the step of *security model analysis*, where the model has to be validated before passing on to its actual implementation. The user interface as well as the implementation technique of the workbench matches these two steps: WorSE features separate functional groups for (1) model engineering, including model instantiation, and (2) model analysis that may be independently run, consecutively passed through or repeated just as needed. Both model engineering as
well as model analysis tasks are implemented as independent and exchangeable software modules.

In the following, the practical workbench support for these two steps and their individual problems will be pointed out.

Security Model Engineering. In the past decades, a wide range of security models has evolved. While new security requirements have emerged from the rapidly growing scope of security-relevant applications and their increasing complexity, new models have been designed to cover their needs. However, the diversity of security models also complicates the model engineering process.

Today, a security engineer has to find a matching security model for a specific policy domain. Existing security models have been designed mostly demand-driven, thus new application domains most likely require new model calculi, tailored to express exactly the semantics of the corresponding security policy. In this case, custom security models and security properties have to be defined and validated by the developer.

This engineering step is error-prone and requires expert knowledge. To this end, our tool-based approach seeks to normalize the design of policy-specific security models by providing a uniform yet customizable model core, that may be used to auto-generate model components and perform plausibility checks in a consistent workflow. This methodology along with its workbench support features will be presented in detail in section 3.

Security Model Analysis. Once an instance of a security model has been compiled from a security policy, its application-specific security properties have to be validated. However, these properties may not necessarily yield a decidable problem, which is notably the case for the property of access right proliferation (Harrison and Ruzzo, 1978; Sandhu, 1992b; Kleiner and Newcomb, 2007). On top of this qualitative obstacle of model analysis, practical scenarios may impose quantitative problems associated with the sheer input size: For instance, the dimensions of an access control matrix or an information flow graph may even rule out decidable algorithms.

Two examples in section 4 show how WorSE provides modular model analysis features to tackle these problems. Firstly, decidability issues that constrain reachability analysis of dynamic models states (e.g. on proliferation of access rights, roles or attributes) can be handled by heuristic approaches. A simulation engine compatible with core-based models (as explained in section 3) allows for heuristic analysis of these problems (section 4.1). Secondly, a huge input space
of a real access control system may be handled by algorithms that condense an access matrix. We used this method to implement an information flow analysis tool as described in section 4.2.

Summary. At the bottom line, the methodology of security model engineering and analysis is essential to ensure the quality of the resulting model instance, which constitutes the foundation for correctly implementing a security policy. To this end, we integrated an approach to design policy-specific security models without semantic gaps along with methods for their analysis into a security engineering workbench, thus providing expert knowledge for a more streamlined development of secure IT systems.

In the next section, we start the detailed discussion of WorSE with the step of security model engineering.

3. Security Model Engineering

The goal of this section is to present model engineering principles that are the key enabler for tool-supported model design, instantiation, and analysis. Applying these principles to security model engineering results in models that are amenable to the engineering methods of WorSE. This section hence discusses these principles for a better understanding of our workbench’s design and methods.

To formalize an application-specific security policy, a model that matches the policy domain is required. At this point, there are two options. Either we apply an existing model that is to some extent compatible to the policy domain, or we design a novel model that is tailored to the domain. The advantage of the first option is the reuse of established model abstractions. However, since these abstractions were often developed for specific domains, they support other domains only to a limited extent. This often leads to an intricate and thus error-prone modeling of the policy. Additionally, the analytical properties of the existing models are only to a limited degree applicable to a new policy, too. Thus, the potential of model-based security engineering is not sufficiently exploited.

On the other hand, designing a model that is tailored to the policy domain has the advantage that it supports abstractions close to the policy domain. An informal policy thus can be represented by a formal model without loss of the policy’s notions. Moreover, tailored models offer domain-specific analytical properties that are derived from the application’s context. Result is that the quality of a policy’s formal specification in terms of correctness, analyzability, and validity is
much better then compared to reused models. On this account, numerous models tailored to various policy domains have been developed for more than 30 years.

From an engineering point of view, however, model diversity is an obstacle to engineering efficiency and tool support. This can be overcome by domain-independent engineering principles that allow for tool-supported model engineering. This section illustrates model engineering principles that we have integrated in WorSE with the goal to provide model-independent tool support for model engineering, instantiation, and analysis.

3.1. Domain-independent Model Engineering Principles

The important prerequisite for domain-independent tool support is that security models share domain-independent model abstractions. The latter can be exploited to develop domain-independent model engineering and analysis methods that are implemented by engineering tools such as WorSE.

Following this idea, we first introduce the required domain-independent model abstractions. Afterwards, we present the idea of our model engineering principles, which lead to security models that share these abstractions.

We have integrated the engineering principles in WorSE. The result is that the workbench now provides support for a range of design, instantiation, and analysis methods for all models that adhere to these principles. In order that security models can benefit from WorSE’s methods, they have to contain the following model abstractions:

1. A deterministic automaton models policy dynamics.
2. This automaton has a state space that represents all possible protection states of a policy, e.g., its right, role, or attribute assignments, depending on a policy’s domain.
3. The automaton’s state transition function describes the dynamic behavior of a policy, e.g., by changing right, role, or attribute assignments.

Governed by these abstractions, our model engineering principles encompass a domain-independent uniform model foundation, which we call security model core in the following. Domain-dependent models can be derived from the core by core specialization, just as subclasses can be derived from super classes in object-oriented software design. The result is a systematically engineered core-based security model with shared abstractions, i.e., the model core, which are amenable to WorSE’s engineering and analysis methods.
In the following, we present the model core along with the core specialization method. On top of that, we discuss how WorSE supports policy engineers to develop core-based security models.

3.2. Security Model Core and Core Specialization

The security model core is a deterministic automaton \((Q, \Sigma, \delta, q_0)\), consisting of a set of protection states \(Q\), a finite set of inputs \(\Sigma\), a state transition function \(\delta: Q \times \Sigma \rightarrow Q\), and an initial protection state \(q_0 \in Q\).

In terms of object orientation, the state set \(Q\) represents a virtual type for the dynamic model component and is generally infinite. A protection state \(q \in Q\) is an \(n\)-tuple that contains \(n\) structurally different components, e.g., sets, mappings, or relations, that can be modified by policy-specific inputs. Characteristical state components are for example a subject set, a role set, a role assignment function, or an access control matrix (ACM).

The input set \(\Sigma = C \times X\) is defined by the finite set \(C\) of policy-specific commands and an input vector set \(X\) of command parameters. Policy-specific commands are those commands whose execution is protected by a policy and which modify a protection state, e.g., \(chmod\) in a Unix/Linux-based system.

Model dynamics is specified by executing a policy-specific command \(c \in C\). Any command defines authorization rules by a set of conditions and a set of primitive actions. Basically, conditions check if executing the command is allowed and primitive actions allow for creating any protection state needed by a model. Thus, each command \(c\) is a tuple \((x, Cond, Prim)\) where \(x\) is the vector of formal input parameters, \(Cond\) is a set of conditions and \(Prim\) is a sequence of primitive actions. Executing a command \(c\) depends on the current model state and its input parameters, and may result in modifying the state so that the automaton enters a subsequent state \(q' = \delta(q, (c, x))\). The state transition function transitions a model’s protection state \(q\) to a subsequent state \(q'\) if all of the command-specific conditions are met. The subsequent state \(q'\) is computed by a command-specific sequence of primitive actions.

Core specialization tailors the core components to a specific domain-dependent model. The notion of core specialization goes back to generalization and specialization concepts of object-oriented software design. In object-oriented design, a subclass is derived from a super class by inheriting its attributes, operations, and constraints with the goal to reuse software. Subclasses are allowed to add new features, e.g., new attributes and operations, or to refine inherited operations to express individual properties. In the context of core specialization, the model core corresponds to a super class and any model corresponds to a subclass that can be
derived from the model core. Correspondingly, models inherit the core components \( Q, \Sigma, \delta, \) and \( q_0 \), which then must be refined.

Fig. A.2 shows an exemplary model specialization hierarchy, where two core-based models, i.e., the HRU model (Harrison et al., 1975, 1976) and a dynamic RBAC model, are derived from the model core. As can be seen, both models inherit the model core \((Q, \Sigma, \delta, q_0)\), and model specifics are represented by refining the core components. Just as the state space and its states are refined, all other model core components must also be specialized to express a model’s specific characteristics. For example, the HRU model refines the state space as \( Q = 2^S \times 2^O \times ACM \), where each state \( q = (S_q, O_q, acm_q) \) consists of a state-specific subject set \( S_q \in 2^S \), object set \( O_q \in 2^O \), and access control matrix \( acm_q \in ACM \) with \( acm_q : S_q \times O_q \to 2^R \) and \( R \) being a finite set of model-specific rights. On the other hand, the dynamic RBAC model refines the state space as \( Q = 2^U \times 2^S \times 2^{UA} \times USER \times ROLES \). Any state \( q = (U_q, S_q, UA_q, user_q, roles_q) \) consists of a state-specific user set \( U_q \in 2^U \), session set \( S_q \in 2^S \), user-to-role assignment relation \( UA_q \in 2^{UA} \), session-to-user mapping \( user_q \in USER \), and session-to-role mapping \( roles_q \in ROLES \). For detailed information about the components of RBAC models, refer to Ferraiolo and Kuhn (1992); Sandhu et al. (1996, 2000); Ferraiolo et al. (2007).

Following this approach, each core-based security model consists of a state automaton whose components \( Q, \Sigma, \delta, \) and \( q_0 \) are tailored to a model’s domain-specific requirements. So far, our experience shows that the model core along with the core specialization method can express a wide range of security models: traditional identity-based access control (IBAC), RBAC, and attribute-based access control (ABAC) models, information flow control models as well as trust management models like Li et al. (2002); Li and Winsborough (2003). The basis of the model core’s expressive power is a minimal modeling framework that does not provide domain-specific model abstractions.

3.3. Workbench Support

This section shows how domain-independent engineering principles are supported by engineering tools integrated in WorSE. The advantage of tool-supported model engineering is a significant reduction of the error rate of model engineering due to integrated plausibility checks, partial automation of engineering steps, and pre-built model patterns that a workbench user has to fill in.

WorSE supports the engineering principles by providing a graphical user interface (GUI) and a set of mechanisms that guide a policy engineer through the
process of core-based model design. WorSE provides the components of the common model core; the specialization of the core components must be accomplished by the policy engineer by refining the state space $Q$, the input set $\Sigma$, the state transition function $\delta$, and the initial state $q_0$. In doing so, WorSE supports a policy engineer by

- providing a range of standard mathematical types for state components, e.g., sets, $n$-ary tuples, relations, functions, matrices (as a special binary mapping), and lattices (as a special 2-tuple),
- finding and avoiding plausibility violations such as type violations within the domains and co-domains of primitive actions and conditions,
- avoiding inconsistencies between $q$ and $Q$ by automatically deriving the state space from the state refinement,
- avoiding redundancies of primitive actions and conditions, and
- enforcing pre-defined naming conventions.

Tool output is a core-based model. For further processing, WorSE allows to export the resulting model as an XML-file.

3.4. Summary

This section has presented domain-independent engineering principles that encompass a uniform security model core, from which core-based security models are derived by core specialization. The foundation are domain-independent model abstractions, based on which our workbench implements a range of engineering and analysis methods. All core-based models contain these abstractions and thus can be engineered and analyzed using WorSE.

4. Model Analysis

Considering the key role of security policies for defining and implementing security properties, quality assets like correctness, completeness and consistency are essential objectives in policy engineering. In order to improve policy quality towards these objectives, numerous formal security models have been developed that support different analysis goals (Harrison et al., 1975; Bell and LaPadula, 1976; Sandhu, 1988; Zhang et al., 2005; Kleiner and Newcomb, 2007; Stoller et al., 2007; Naldurg and Raghavendra, 2011; Stoller et al., 2011).
This section focuses on two of these analysis problems: right proliferation and covert information flow detection in access control policies. It describes corresponding analysis methods and tools integrated in WorSE and discusses their theoretical foundations as well as their practical application.

4.1. Safety Analysis

A core objective of access control models is to study the proliferation of access rights in access control systems. This problem was first formalized in HRU access control models (Harrison et al., 1975) 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.

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? If this may happen, such a model state is not considered safe with respect to that right. Model safety analyses provide fundamental insights into right proliferation in access control systems.

This section discusses the ideas and algorithms behind the tools in WorSE for analyzing safety properties of access control models; it focuses on core-based, HRU-style access control models as motivated and formalized in section 3. In order to provide an example for the operational principles in WorSE, one of WorSE’s model safety analysis algorithms is discussed in detail.

4.1.1. The Role of Heuristics

It is long since known that right proliferation properties of automaton-based access control models with infinite state spaces in general are not decidable (Harrison et al., 1975). In other words, there is no algorithm that, given some arbitrary HRU model, will always terminate and tell us whether a given model state is safe. Since core-based models are generalizations of HRU models, the same property holds for core-based models in WorSE, too.

In order to deal with this fundamental problem, the approach taken in WorSE uses heuristic analysis algorithms. Heuristic-based safety analysis trades accuracy for tractability: the analysis algorithm might not terminate, and model states that actually are not safe might be missed. The former, for example, holds for models where all the states of an infinite state space actually are safe. On the other hand, valuable hints on model correctness are obtained if unsafe states are found and policy engineers are pointed to input sequences that lead to such states.

In general, heuristic search algorithms exploit the fact that HRU safety is semi-decidable and try to prove that some given model state is not safe with respect
to a right $r$ by finding an input sequence that, starting at $q$, leaks $r$ into an ACM cell of some follow-up state $q_{\text{target}}$. 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 $q_{\text{target}}$ reflects an input sequence violating the safety property. As long as no such target state is found, the search continues.

In doing so, heuristic search algorithms assemble all reached states into a state transition tree (Fig. A.7) – a tree-structured digraph $(V, E), V \subseteq Q, E \subseteq Q \times Q$ where the root is a state $q$ whose safety is to be analyzed. Each vertex represents a state from the model’s state space that is reachable from $q$ by an input sequence $a \in \Sigma^*$ through $\delta^*(q, a)$, and each direct successor of a state $q_i$ is a state $\delta(q_i, \alpha)$ where $\alpha$ is a single input from $\Sigma$.

The challenge here is to restrict the rapidly growing state transition tree by a proper heuristic that channels its growth. Adding a new vertex $q_i'$ to the tree corresponds to a state transition from $q_i$ to $q_i' = \delta(q_i, \alpha)$. For generating a new vertex, a heuristic thus has to decide which vertex $q_i$ from the state transition tree and which input $\alpha$ is chosen. 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 a target state $q_{\text{target}}$ where the target right has leaked into a matrix cell. Because $q_{\text{target}}$ is yet unknown, the right proliferation analysis algorithms in WorSE choose $q_i$ and $\alpha$ heuristically.

### 4.1.2. Dependency Search Heuristics

Heuristic approaches are most successful if they are well-tailored to the specific problem to be solved, so the challenge when analyzing model safety is to exploit model properties that have an impact on the probability of a given state or command to contribute to a path from $q$ to $q_{\text{target}}$. As an example, for a state to not be safe with respect to a given right, in HRU-style security models this right has to be entered into some matrix cell by a command from the model’s authorization scheme. A static analysis of the authorization scheme will reveal such commands, and a heuristic for choosing $\alpha$ might give them some priority.

In general, several model properties exist that can be exploited to obtain such hints, and several heuristics have accordingly been developed and integrated in WorSE. This section focuses on the DEPSEARCH safety analysis heuristic for core-based access control models; it motivates the main ideas, describes its major algorithms, discusses its computational complexity and infers practical pragmatic restrictions. For an in-depth discussion and evaluation of the algorithm see Amthor et al. (2013).
The DEPSEARCH Heuristic. Basically, the goal of each heuristic safety analysis algorithm is to find a command sequence that transitions the analyzed model from a state $q$ to a state $q_{target}$ where a right has leaked into a matrix cell. The idea behind the DEPSEARCH heuristic is that hints for such a command sequence can be mined from a model’s authorization scheme. Because authorization schemes are static, finite and usually consist of a manageable number of commands, their analysis is fast and efficient.

The first step in the DEPSEARCH heuristic is to find commands that directly enter the target right $r$ into some matrix cell. Such commands usually have conditions that must be satisfied to execute the command’s primitives. Because conditions consist of tests for the presence of rights in the ACM, all tested rights thus become additional target rights. All following steps are identical to the first step but for the single exception that they have larger sets of target rights, until finally we either encounter commands that have no conditions at all, or we only have conditions that are already satisfied by $q$.

This recursive process assembles a command dependency graph whose vertices are commands, and an edge from command $c_1$ to command $c_2$ denotes that $c_1$ is capable to establish at least one condition for $c_2$. All paths from vertices having no incoming edges to vertices having no outgoing edges reflect command sequences where each command plays a role in establishing preconditions for reaching a target state $q_{target}$ from $q$.

Fig. A.8 shows a command dependency graph (CDG) of 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 $m_q$ or by the fact that $c_1$ simply has no conditions; the incoming edges to $c_2$ denote that $c_2$’s conditions can be satisfied by the execution of $c_1$ and/or $c_4$.

Formally, a command dependency graph is a connected attributed digraph $(V,E)$ where

$V = C \cup \{c_q, c_{target}\}$ where $C$ is the set of commands in the authorization scheme (see section 3.1), $c_q$ is a virtual command with no condition that just enters each right present in $m_q$, $c_{target}$ is a virtual command with no primitives that requires the target right as a condition, and

$E \subseteq V \times V$ is defined by a model-specific heuristic that is aware of model-dependent conditions for a command to be executed.
$c_q$ and $c_{\text{target}}$ are virtual commands that are added to a CDG for convenience of the command sequence generation algorithm (see Alg. 2, discussed below). The edges in $E$ depend on the type of conditions in the authorization scheme of a specific security model. As an example, in core-based HRU models it holds that

$$(c_i, c_j) \in E \iff (c_i.\text{Prim. Enter. Rights} \cap c_j.\text{Cond. Rights} \neq \emptyset)$$

where $c_i.\text{Prim. Enter. Rights}$ denotes the set of rights entered by the primitives of $c_i$, and $c_j.\text{Cond. Rights}$ denotes the rights needed to satisfy the conditions of $c_j$.

Alg. 1 shows the assembly of a CDG in its generic form. Starting with $c_{\text{target}}$, it recursively computes the predecessors of all vertices and terminates when all commands $c \in C$ are included in $V$ that have no predecessors, and for which a path to $c_{\text{target}}$ exists.

The assignment $P \leftarrow \text{predset<modeltype>}$ tailors the CDG assembly to a specific core specialization, allowing the predset function to be aware of the model-specific necessary conditions for a command to be executed. In Alg. 1 for example, predsetHRU tailors predecessor selection to HRU models where commands in the authorization scheme have conditions that check for rights being present in the ACM. Thus, for any given command $c_i$ in a HRU CDG, a command $c \in C$ is a predecessor of $c_i$ if and only if rights entered by $c$ occur in the conditions required by $c_i$, so $\text{predsetHRU}(c_i) = \{ c \in C | c.\text{Prim. Enter. Rights} \cap v.\text{Cond. Rights} \neq \emptyset \}$. As another example, in RBAC models predsetRBAC involves roles, sessions, and permissions.

In the DepSearch safety analysis algorithm the CDG is used to channel the growth of the state transition tree (Fig. A.7) by providing command sequences that establish necessary conditions for the model to transit from $q$ to $q_{\text{target}}$. For discussing the CDG’s role in the analysis algorithm, let us first make observations on some of its properties.

- The existence of a path from a vertex with no incoming edges 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 of a model’s authorization scheme, while only the parameters of the commands at runtime determine their precise effect.

- The vertex sequence in such a path is not yet identical to a command sequence that will result in a leakage of $r_{\text{target}}$. A path $c_q, c_1, c_2, \ldots, c_{\text{target}}$ only tells us that $c_q, c_1^*, c_2^*, \ldots, c_{\text{target}}$ has the potential to leak $r_{\text{target}}$. 


Algorithm 1: Dependency Graph Assembly, tailored to an HRU security model

Input:
- a model’s authorization scheme $C$
- the model’s state $q$ the safety of which is analyzed
- a target right $r_{target}$

Output:
- a command dependency graph $(V, E)$
- the starting point for the command sequence generation (Alg. 2) $c_{target}$

procedure predecessors(in $c_i \in V$)
  
  \begin{enumerate}
    \item $P \leftarrow \text{predsetHRU}(c_i)$;
    \item for $c \in P$ do
      \begin{enumerate}
        \item if $c \notin V$ then
          \begin{enumerate}
            \item $V \leftarrow V \cup \{c\}$;
            \item predecessors($c$);
            \item $E \leftarrow E \cup \{(c, c_i)\}$;
          \end{enumerate}
      \end{enumerate}
  \end{enumerate}

assemble virtual vertices $c_q$ and $c_{target}$:
$C \leftarrow C \cup \{c_q\}$;
$V \leftarrow \{c_{target}\}$;
$E \leftarrow \emptyset$;
predecessors($c_{target}$);

return $(V, E), c_{target}$;
• Paths may contain cycles; these must not be removed, because multiple runs
of the same cycle may result in rights being entered into different matrix
cells; e.g. the CDG in Fig. A.8 contains a cycle $c_2, c_4$, and we only know
that a successful command sequence has the pattern $c_1^*, \{ c_2^*, c_4^* \}^+, c_5^*$.

While the goal of the DepSEARCH heuristic is to channel the growth of the
state transition tree, the CDG only provides command sequences $c_1...c_n$ that con-
tribute to establishing necessary conditions for a right leakage. The DepSEARCH
safety analysis algorithm uses such sequences to grow the state transition tree by
executing a sequence of model transitions

$$\delta(q_1, x_1) = q_2, \delta(q_2, x_2) = q_3, ..., \delta(q_{n-1}, x_n) = q_n,$$

registering each state $q_i$ as a new branch in the state transition tree (we will discuss
the command parameters $x_i$ below). If in doing so a target state where $r_{target}$ has
leaked into a matrix cell is found, we proved that $q$ is not safe with respect to
$r_{target}$. If not, we proceed with a new command sequence by choosing a new path
in the CDG.

Repeated path generation in the CDG results in different paths. This is achieved
by a “scent” attribute of each edge reflecting the amount an edge has already been
used. Generating paths then follows the idea of ant algorithms, although in a vari-
ation where the scent of an edge acts repelling. Thus paths generated from a CDG
are always minimal with respect to the accumulated scent of its edges. Scent min-
immality 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). Thus we
achieve that paths not yet tried are used with higher probability, and each run of
a cycle within a path decreases the probability of a second run of the same cycle.
The algorithm for path generation is outlined in Alg. 2.

4.1.3. The DepSEARCH Algorithm

Starting at $q$, the DepSEARCH safety analysis algorithm (Alg. 3) explores the
state space of a model in order to find a state where $r_{target}$ has leaked into some
matrix cell. The growth of the state transition tree is channeled by executing
command sequences that gradually establish necessary conditions for reaching
$q_{target}$. Command sequences repeatedly are generated according to the heuristic
discussed in section 4.1.2 by Alg. 2.

Two problems remain to be addressed: termination conditions and the selec-
tion of a parameter vector for each command execution.
Algorithm 2: CDG Path Generation

Input:
- a CDG \((V, E)\) as generated by Alg. 1
- the starting point for a command sequence generation \(c_{\text{target}}\)

Output:
- a sequence of vertices representing a path in \(G\)
- a modified CDG where all edges used for this path have a stronger scent

Variables:
- \(\text{currentVertex}\): a vertex in the CDG
- \(\text{currentEdge}\): an edge in the CDG
- \(\text{currentPath}\): a sequence of vertexes in the CDG

\[
\begin{align*}
\text{currentVertex} & \leftarrow c_{\text{target}}; \\
\text{currentPath} & \leftarrow \text{currentVertex}; \\
\text{repeat} & \quad \begin{align*}
\text{currentEdge} & \leftarrow \text{lowestScent} (\text{currentVertex}.\text{incomingEdges}); \\
\text{currentVertex} & \leftarrow \text{currentEdge}.\text{origin}; \\
\text{currentPath} & \leftarrow \text{currentVertex} \circ \text{currentPath}; \\
\text{currentEdge}.\text{scent} & \leftarrow \text{currentEdge}.\text{scent} + 1;
\end{align*} \\
\text{until} & \quad \text{currentVertex}.\text{incomingEdges} = \emptyset; \\
\text{return} & \quad \text{currentPath}, (V, E);
\end{align*}
\]
Algorithm 3: The DEPSEARCH Algorithm

**Input:**
- a model’s authorization scheme \( C \)
- a model state \( q \) the safety of which is to be analyzed
- a target right \( r_{\text{target}} \)

**Output:**
- a state transition sequence leaking \( r_{\text{target}} \)

**Variables:**
- \( \text{CDG} \): a CDG generated from \( C \)
- \( \text{STS} \): a state transition sequence leaking \( r_{\text{target}} \)

\[
\begin{align*}
(CDG, c_{\text{target}}) & \leftarrow \text{DependencyGraphAssembly}(C, q, r_{\text{target}}); \\
\text{STS} & \leftarrow q;
\end{align*}
\]

repeat
  \[ path \leftarrow \text{CDGPathGeneration}(CDG, c_{\text{target}}); \]
  while \( c \leftarrow \text{path.nextCommand} \) do
    \[ q' \leftarrow \delta(q, c, \text{selectParamvector}(q, c)); \]
    \[ \text{STS} \leftarrow \text{STS} + q'; \]
    \[ q \leftarrow q'; \]
  until \( r_{\text{target}} \) was written into a cell of \( m_{q'} \);
return \( \text{STS} \);
With respect to termination, for models with infinite state spaces the algorithm of course will not terminate if \( q \) is safe with respect to \( r_{\text{target}} \). In general, the termination issue with DEPSEARCH is part of the trade-off between accuracy and generality, and no precise termination conditions are known. As a consequence, for relating analysis runtime to the probability of actually finding a leak we have to resort to pragmatic approaches based on experience, which currently is based on a too small set of test cases for authoritative prognoses. The summary of this section (4.1.4) discusses some experimental results.

The problem of selecting a command’s parameter vector is currently solved by a randomized approach (Amthor et al., 2013). However, ongoing work reconsiders this problem as a constraint satisfaction problem (CSP). In general, a CSP (Lauriere, 1978; Jaffar and Maher, 1994) is a tuple \( (V, D, P) \) with a set of variables \( V = (v_1, \ldots, v_k) \) each of which is instantiated in a particular domain \( D \in D = (D_1, \ldots, D_k) \) where \( D_i \) defines the range of valued for \( v_i \), and \( P \) is a set of predicates defining constraints that the values of the variables must simultaneously satisfy. The problem of finding “good” parameters for a command then can be modeled 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. Whether parameters are “good” or “bad” is decided by a heuristic metric that reflects the probability of a parameter vector to promote right leakages. Then, any one of the well known CSP solving algorithms (e.g. AC-3 (Mackworth, 1977)) can be applied. An evaluation of the CSP approach is currently ongoing work.

4.1.4. DEPSEARCH Summary

Concluding, we have presented the background of a heuristic safety analysis algorithm for core-based security models integrated in the security model analysis workbench WorSE. 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 (Bell and LaPadula, 1976).

Computational runtime complexity of assembling the CDG is \( O(n^2) \), where \( n \) is the number of commands in a model’s authorization scheme. Generating a command sequence from the CDG runs in \( O(n) \), because due to the minimum scent of each path, eventual cycles occur at most once in a path. While the current randomized scheme to select command parameters runs in \( O(1) \), the quality of the selection is far from satisfying, and there is work in progress to apply CSP solutions here.
The DepSEARCH heuristic 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 in the sequence depends exactly on the execution of its predecessor. For these models, the DepSEARCH algorithm has turned out to be quite successful, and its application to more regular models also demonstrated its universal applicability. Among others, the DepSEARCH algorithm was applied to a core-based model of a real-world RBAC$_3$ Health Information System policy for an age-care facility (Evered and Bögeholz, 2004; Stoller et al., 2011) that was fitted with an authorization scheme to model dynamic behavior, resulting in a dynamic RBAC$_3$ model with 20 roles, 15 commands in the authorization scheme, a role hierarchy, and separation-of-duty focusing on role exclusion. Two types of model safety were analyzed, focusing on user-to-role assignment (role-safety) and on permission-to-role assignment (permission-safety). The DepSEARCH algorithm precisely found the optimal effective state transitions that rendered a state unsafe, requiring a mean value of approximately 100 s on a standard desktop PC (Intel Core i7, 3.4 GHz CPU, 16 GB RAM) for a model encompassing 500 objects. As another example, the artificially tailored stress-test model mentioned above had a well-hidden permission leak appearing only after executing precisely a sequence of 7 commands, each one with a precise parameter set. The leak was found after 15 effective state transitions (the optimal solution would have been 7 steps), and the analysis took approximately 1000 s for a model size of 500 objects. In comparison, earlier baseline heuristics randomly choosing state transitions needed 143 effective steps for the Health care model, and totally failed for the stress test model after having executed $10^7$ steps without finding the leak. A detailed evaluation of the DepSEARCH algorithm encompassing the evaluation goals, methods, targets, and results is included in Amthor et al. (2013).

In the future, the practical impact of this work will be exploited by furnishing the DepSEARCH heuristic with sophisticated knowledge about role and attribute model paradigms, allowing for still more efficient safety analyses of dynamic RBAC and ABAC models as well as analyses of models written in policy specification languages such as used for example in the SELinux operating system (Zanin and Mancini, 2004).

With respect to a policy engineer’s support, once a model state has been proven to not be safe, WorSE provides access to all information collected during the analysis. The CDG is displayed, the critical command sequences that caused the leakage are highlighted, and the policy engineer may zoom in on path segments to inspect command conditions and primitives in detail. The state transition tree
highlights the state sequence that caused the right leakage, and details on command parameters, command effects and model states can be inspected. Based on this information, the cause of a model state’s unsafety can easily be pinpointed, and the model can appropriately be revised.

4.2. Information Flow Analysis

While the analysis of right proliferation properties in sophisticated, policy-controlled access control systems is not an easy task, correctly configuring off-the-shelf access control systems yet can be complex and error-prone. As an application example for the information flow analysis tools in WorSE, this section discusses an information-flow based analysis tool to detect covert information flows in large access control systems and illustrates its application to a Linux OS access control system.

4.2.1. Contemporary Access Control Systems

Today’s IT systems implement access control by access control lists (ACLs) and capability lists. The set of all ACLs, respectively capability lists, along with the user management reflects a system’s ACM (Fig. A.3), representing its entire protection state. Even if almost all read/write rights were set according to the access control policy, analyses of real-world server ACMs, however, have exposed indirect information flow channels that can be exploited by insiders to gain information, which they are actually not allowed to have.

Example. Consider an excerpt from a company’s IT system with two departments, “Research & Development (R&D)” and “Sales”. In the example, Ann is working in R&D as a project manager, Bob belongs to her project staff, and Chris is working in Sales. Ann is the owner of a confidential file ProjectXFiles and another file ProjectXBoard; she has read and write access to both, resulting in information being able to flow from both files to Ann and vice versa. Bob is a member of the group StaffX and may read the project’s internal ProjectXBoard. Moreover, Bob manages the communication with Sales by a separate SalesBoard where he is the owner and has read and write permissions. Chris is currently working on a new sales brochure SalesFlyer and has read and write access to this document. Additionally, he is a member of the group Sales that is allowed to read Bob’s SalesBoard. That way, Chris is able to read the project’s announcements for promotional use. Even though the resulting ACM (Fig. A.3) is directly derived from this access control policy, it does not sufficiently enforce it: Fig. A.4 shows that
there is an indirect information flow (illustrated by a dashed arrow) from the confidential ProjectXFiles to the SalesFlyer exploiting legal direct information flows (solid arrows) via intermediary subjects and objects. In practice, because Bob and Chris are colleagues this can easily be achieved e.g. by contaminated application software planted by social engineering. As a result, confidential project information may indirectly flow from ProjectXFiles to SalesFlyer, where Chris can take advantage of them. Thus, by exploiting vulnerabilities of today’s access control systems, i.e. transitivity of information flow on the one hand, discretionary access control mechanisms on the other hand, insiders do not even need to bother bypassing the access control systems. They just need to reveal covert information flow potentialities and take advantage of intermediary subjects and objects.

Covert information flow potentialities are not easy to avoid. Reasons are huge search spaces along with high runtime complexity of search algorithms. That is, ACMs of real-world server systems usually contain billions of cells leading to search spaces that are too large to be searched manually. On the other hand, algorithms that simply analyze the direct rights of an ACM already have a basic runtime complexity of $O(|\text{subjects}| \times |\text{objects}|)$, and runtime complexity of searching information flow potentialities via intermediary entities are even higher.

In the following, a workbench-supported search method is discussed that allows for analyzing ACMs of real-world systems. Questions to be answered are for example: Is it possible that a subject may gain information from a specific directory? Where does the information of a document flow to? Which information may flow to a particular user? The problem’s complexity is tackled by rewriting a system’s ACM into an equivalent information flow graph (IFG) (Denning, 1976), which allows for several types of graph reduction. On this account, we discuss the rewriting of ACMs into IFGs and illustrate the analysis of covert information flow potentialities. Afterwards we present the implementation of this search method in WorSE, allowing a user to extract the ACM from a Unix-based system and analyze it regarding covert information flow potentialities.

4.2.2. Rewriting Access Control Matrices

Information flow graphs are digraphs $IFG = (V, E)$. The rewriting of ACMs with a two-element right set $R = \{\text{read, write}\}$ into IFGs is based on a relation that can be described as

$$\forall v_1, v_2 \in V : (v_1, v_2) \in E \leftrightarrow (\text{read} \in \text{acm}(v_2, v_1) \lor \text{write} \in \text{acm}(v_1, v_2)).$$

By this means, subjects and objects are mapped to vertices of the IFG, and any read and write right is mapped to a directed edge. Because of this relation, the
results of analyzing an IFG can be applied to its ACM. On the other hand, the revised ACM can be translated back to the ACLs for reconfiguring the access control system.

For example, rewriting the ACM of Fig. A.3 results in an IFG as shown in Fig. A.5 where graph reduction methods can now be applied to reduce the complexity of the analysis problem. As an example, IFGs allow for analyzing the confidentiality and integrity levels of entire sets of vertices by building the transitive closure. If a vertex belongs to a specific set, this vertex has exactly the same confidentiality and integrity level as every other vertex of this set. We call these sets informational equivalence classes, describing vertices with the same information potential. On top of that, by subsuming vertices of one equivalence class to a single vertex, we can reduce the number of vertices of a IFG and hence the runtime of graph algorithms.

Finding informational equivalence classes is equivalent to searching cycles in the graph. The example IFG in Fig. A.6 contains three cycles and thus three informational equivalence classes: ProjectXFiles, Ann, and ProjectXBoard are members of equivalence class 1, Bob and his SalesBoard belong to the second equivalence class, and Chris and the SalesFlyer share equivalence class three. This small example already shows that by subsuming vertices with shared confidentiality and integrity levels, the size of an IFG can be significantly reduced allowing for decreasing the runtime of IFG analyses.

4.2.3. Information Flow Analysis in WorSE

This section describes the operational principle behind WorSE’s IFG analysis tool to expose unintended information flows. The method consists of three steps: Protection state extraction, model rewriting, and IFG analysis. We discuss details of each step along with examples of a Unix file server in the following.

1. Extracting the Protection State. The objective of the first step is to extract a system’s protection state, including its subjects, objects, and right assignments by rebuilding its ACM. Typically, subjects are the users registered in a system, objects are files and directories, and right assignments are the access control or capability lists of a system. From these informations the access control system’s global ACM is reconstructed contains all user rights on the system’s objects at the time of extraction.

For example in Unix-like access control systems, access control information is scattered among the inodes in the file systems and a few configuration files, e.g. /etc/passwd and /etc/group. From an abstract point of view, the users compose the
subject set, and files, directories, sockets etc. compose the object set of the system’s ACM that precisely reflects the system’s protection state. For reconstructing this ACM, we have to examine the read/write/execute right tuples of each object with respect to the read and write rights for any subject. While root generally owns both rights for any object, for all other users we have to determine the rights with respect to their owner, their group, and their all-other status. To get insightful analysis results, the origin of user rights, i.e. whether they are owner, group, or all-others rights, must be maintained.

2. **Rewriting the ACM.** This step rewrites the reconstructed ACM as an IFG by using the relation discussed in section 4.2.2.

3. **Information Flow Analysis.** The third step analyzes the IFG regarding covert information flow paths. Here, informational equivalence classes are built to reduce the size of the graph. Analyzing information flow potentialities depends on the system’s access control policy. At this point, a security engineer has to define specific analysis goals and questions that are derived from the policy. Once these are identified, graph algorithms searching for paths and cycles can be applied.

   In case we find information flow paths, step three also analyzes their possible causes. For that purpose we have to know whether the rights are owner, group, or all-other rights, since these rights account for the paths between vertices or the vertices’ memberships to equivalence classes. We are able to exactly identify the reasons for covert information flow potentialities by determining the ACM’s rights that are responsible for specific edges in the IFG. As an example, running the analysis on a Linux file server of our institute revealed several covert information flows in its ACM, where typical reasons were the membership of a user to a group, incorrect group rights, and incorrect rights of all-others.

4.2.4. **Workbench Support**

Covert information flow analysis is supported in WorSE by a tool encompassing ACM extraction, the transformation of an ACM into an isomorphic IFG, and IFG analysis algorithms. The tool is capable of dealing with real-world-size access control systems and was, for example, applied to one of our institute’s Linux file servers with an ACM of approximately one billion cells.

1. **Extracting the Protection State.** For extracting a protection state, WorSE provides a file system crawler that browses the file systems’ inodes and extracts its ACLs, and accesses the system’s user management files to extract user names and group memberships. Result is an ACM that represents the current protection state...
of a system. The crawler either uses a life file system or a frozen file system taken from a snapshot of a virtual machine monitor. At present, the file system crawler is tailored to extract data of Unix-based access control systems; however, the crawler is general enough to support any hierarchical file system. The tool can be configured to the amount of information to be extracted. For example, the omnipotent subject root as well as non-persistent directories such as /proc are usually disregarded in security analyses. After having configured the input, the crawler extracts the required data and composes an ACM containing the rights of the selected subjects and objects as well as the subject and object identifiers. Subject identifiers are the user names registered in the system; object identifiers are the absolute paths along with the object names. The user can export and import the extracted ACM as XML files.

2. **Rewriting the ACM.** After the ACM has been created it is automatically transformed into an IFG. At the GUI level, the IFG is either represented graphically or textually, where the latter is often more convenient if an IFG is too large to allow for a manageable graphic representation.

3. **Information Flow Analysis.** In this step, IFG cycles are automatically detected and can optionally be collapsed to informational equivalence classes, resulting in a minimal IFG regarding information flow potentialities. An input dialog allows the user to search for specific paths. In doing so, the user has to name both the start node and the destination node by subject and object identifiers. WorSE then runs path analysis algorithms that output all existing paths between start and destination node, making it easy for a user to decide which paths are intended by the system’s security policy and which are not. In a second step, objectionable paths can be revised by identifying their causes.

Automated identification of the causes of covert information flow channels is not yet supported. Currently, users have to evaluate the origin of rights, i.e. owner, group, or all-other rights, manually by re-mapping information flow paths to their causes in the ACM. A future implementation of this particular feature is straightforward, since the necessary information is already contained in the XML files representing an ACM.

5. **WorSE Implementation**

The goal of this section is to show how implementation techniques of functional and non-functional features support the goals of WorSE. Therefore, we will describe how the model design and analysis methods presented in sections 3 and
4 are integrated into the workbench. After discussing their respective functionality, we will address non-functional features that complement them to fulfill these goals and finish with an overview of the current development status.

5.1. Functional Features

In order to support the model-based security policy engineering process worked out in section 2, the principal user interaction with the workbench tools aims at the two critical steps, model engineering and model analysis. Consequently, a security policy engineer may use the workbench to perform individual tasks (analysis of an existing model or engineering of a policy-specific model) or combine both functions starting a new project. A security policy analyst, on the other hand, will most probably use the analysis tools for an existing model, possibly after its initialization with a real-world protection state. These operation modes allow for both steps of the process model to be combined or executed independently and iteratively (cf. Fig. A.9).

WorSE includes three functional components for the activities within both operation modes:

The **Model Engineering** component provides a GUI-based tool to design a security model by specializing the generic model core (cf. section 3.3). The idea of core specialization is also reflected in the object-oriented implementation of the tool, where individual security models designed by the user are derived from a common model core base class. This enables WorSE to support flexible data structures that are capable to represent every core-based security model.

The **Model Instantiation** component provides an easy way to instantiate a security model with an initial protection state, initial values for static model components and state transition rules. The user has several ways to do this: She may manually pick a value (e.g. a string identifier) for each component or import a previously created model instance using an XML-based exchange format. A third way is to extract the protection state from a real system (e.g. a Unix file system), using tools such as the file system crawler described in section 4.2.4.

The third and most comprehensive **Model Analysis** component provides analysis tools for core-based security models. Again, the construction idea of the model core is leveraged here to achieve flexibility for policy-independent tools, that can operate on any security model derived from the basic core data structures. Two such tools are currently implemented: Heuristic Safety Analysis as per section 4.1 and Information Flow Analysis as per section 4.2. Since the information flow analysis tool deals with static ACMs, it just needs to operate on a model-independent
matrix class for IFG-transformation and analysis. In contrast, the safety analysis tool has to simulate model dynamics, thus a model simulator compatible to the common model core is implemented here. As with the model class itself, the policy-specific features of the simulator data structures can be implemented in a uniform and straight-forward way by subclassing a base class that generally implements the state automaton behavior. Correspondingly, a tailored heuristic for each particular core-based model is derived from a heuristics base class, in a way matching their specialized simulator. Currently, a simulator for HRU- and core-based RBAC-models and several heuristics (such as DEPSEARCH, cf. section 4.1.2) are implemented in WorSE.

5.2. Non-Functional Features

To exploit the full flexibility of the model core, WorSE is designed and implemented in a modular way. This means that in practice, analysis tools for various models may be integrated into the workbench, new model simulators and heuristics may be integrated into existing tools, and auxiliary tools (e.g. another file system crawler, see section 4.2) may be plugged in dynamically.

To enable this, non-functional features of WorSE are based on two major design principles:

1. **Extensibility**: Integration of additional models, analysis tools and heuristics, additional model engineering and instantiation tools.

2. **Distributed Computing**: Spatial distribution of software functionality for remote access and parallelization of computations.

To implement these principles, the functional components described in section 5.1 communicate anonymously via a tuple space (Gelernter, 1989). The motivation here is to logically isolate the individual tools as possible: The tuple space defines a standard communication interface that may be accessed uniformly by any tool. Besides exchanging data, analysis or model engineering tools may also register their user interfaces at the top-level GUI or use auxiliary tools provided by the workbench framework, such as XML importers and exporters. Since direct communication involves only the tuple space component, a tool may be indifferent to its particular communication partner.

This anonymous communication supports a flexible extension of the workbench to support arbitrary core-based models. Moreover, this architecture enables WorSE to be run on a distributed hardware. Thus, load scalability may be improved by parallelizing time-consuming computation (such as safety analysis of a real-world access control system). On top, a distributed implementation allows for
remote access to individual tools that run on dedicated hardware but communicate their input and results to a tuple space client running remotely.

Fig. A.10 depicts a sample scenario for communication between different components during a safety analysis session. Messages are communicated as tuples with an element type that identifies appropriate message recipients. Messages stay in the tuple space until they are read and detached by any tool processing messages of the respective type. The sample communication scenario in Fig. A.10 contains the components Simulator, TopLevelGui, and Heuristic (that exemplarily stands for any model-specific heuristic). During an analysis session, these classes communicate as follows (cf. section 4.1):

1. TopLevelGui sends a message to Heuristic to start the analysis, containing optional configuration values for the used heuristic (such as the target right, maximum number of simulation steps, etc.).
2. Heuristic sends a message containing an input to perform a state transition of this particular model’s automaton to Simulator.
3. Simulator sends a message containing an identifier of the new state and, optionally, additional information (“flags”) about this particular transition (such as “conditions failed” or “transition to same state”) to Heuristic.
4. Heuristic sends a message
   (a) to Simulator as in 2., continuing from there (e.g. if a safety decision cannot yet be made) or
   (b) to TopLevelGui containing structured information about the analysis result (e.g. paths to \(q_{target}\)).

In the figure, the respective tuple for each of the given messages is listed. Since their actual delivery is carried out by the tuple space component, the tools do not need to specify (nor know) the concrete recipient for their messages as long as an appropriate tuple type has been chosen (for instance, the simulator class will only receive and detach tuples that control the simulation behavior). In the example, the TopLevelGui in step 1 simply attaches a tuple of type START_ANALYSIS to the tuple space. The concrete heuristic that will receive and handle this message may vary – depending on the implemented heuristic classes for the analysis of a particular security model. Identifier for tuple types of course may be arbitrarily fine-grained, enabling the addressing of one specific out of several heuristics or simulators running parallel.

In practice, there may be cases that require a compromise towards more traditional communication mechanisms in support of performance. One example
for this is the safety analysis tool: Here, the complete data of a model’s protection state would have to be copied via the tuple space in messages 2 and 3 of the example above. Instead, to facilitate high-performance analysis, shared memory communication is used: The class for the model instance, holding full information about any state ever reached, is implemented as a monitor to allow for synchronized, memory-based access from either the simulator or the heuristic. This way, both classes may read and write protection states, while tuple space messages only contain references. In this particular case, the assumption is made that the analysis tool (including simulators and heuristics) and the model instance representation are located on common hardware, what should be the normal case: even in a parallelized analysis, distinct model fragments would have to be processed locally on distributed hardware.

5.3. Development Status

WorSE is under ongoing development. It is written in C++, using free standard libraries for user interaction (Qt, log4cxx), XML parsing (expat, xerces) and graph manipulation (boost).

Current work focuses on the implementation of heuristic algorithms for safety-analysis of core-based dynamic RBAC models (based on the idea of DEPSEARCH, cf. section 4.1.2) as well as simulation engines for core-based ABAC and an SELinux access control model. Also, parallelization of safety analysis is an ongoing implementation issue, related to parallel analysis of separate branches of the state tree as well as to parallel analysis by multiple heuristics.

6. Conclusions

This paper argues for a tool-supported approach to model-based security policy engineering with the objective to increase the efficiency and quality of security policy design and implementation. We substantiate our arguments by discussing WorSE, a security engineering workbench promoting a systematic approach to security policy engineering by providing methods and tools for a model-based design, analysis and implementation of security policies.

The policy engineering principles in WorSE exploit the analytical power of formal security models. All security policies engineered with WorSE are represented by domain-specific security models that share a common model core. The common model core basically is a deterministic state machine that provides a formal framework for modeling the dynamic behavior of a policy; core specialization is used to tailor the generic model core to domain-specific model properties. The
engineering of security models based on these principles then provides the foundation for generic, domain-independent model design and analysis tools.

To support the feasibility of this approach, the paper discusses two well-known analysis problems in access control policies, right proliferation and covert information flows, and describes corresponding analysis methods and tools that have been integrated in WorSE. The application of these tools to real-world scenarios for example revealed several unsafe states in a health care information system policy and detected covert information flows in an access control policy for a large server system.

WorSE is implemented as a distributed software architecture consisting of components implementing a graphical user interface, the model core, a model design tool supporting core specialization, a database of domain-specific off-the-shelf security models, and several generic analysis tools. All WorSE components communicate by a tuple space defining a standardized component interface that can be accessed remotely, providing openness and expandability and making it easy to integrate new model design tools, new domain-specific models, or new analysis tools.

Acknowledgments

This work was partially supported by Carl-Zeiss-Stiftung.

References


Appendix A. Figures

Figure A.1: Security Policy Engineering Process Overview

Figure A.2: Model Core Specification Hierarchy
Figure A.3: Access Control Matrix

Figure A.4: Covert Information Flow Potentialities

Figure A.5: Information Flow Graph
Figure A.6: Reduced Information Flow Graph

Figure A.7: State Transition Tree

Figure A.8: A Command Dependency Graph
Figure A.9: Workbench Use Cases
Tuple Space

1. (type: START_ANALYSIS, values: (v0, ..., vi))
2. (type: SIMULATION_STEP, state: q, command: c, params: (p0, ..., pn))
3. (type: SIMULATION_RESULT, state: q', flags: (f0, ..., fm))
4. (a) (type: SIMULATION_STEP, state: q, command: c, params: (p0, ..., pn))
   (b) (type: ANALYSIS_RESULT, values: (v0, ..., vj))

Figure A.10: Communication Model, exemplary for the Model Analysis Component