Extending Access Control Models with Break-glass

Achim D. Brucker
SAP Research
Vincenz-Priessnitz-Str. 1
76131 Karlsruhe
Germany
achim.brucker@sap.com

Helmut Petritsch
SAP Research
Vincenz-Priessnitz-Str. 1
76131 Karlsruhe
Germany
helmut.petritsch@sap.com

ABSTRACT

Access control models are usually static, i. e., permissions are granted based on a policy that only changes seldom. Especially for scenarios in health care and disaster management, a more flexible support of access control, i. e., the underlying policy, is needed.

Break-glass is one approach for such a flexible support of policies which helps to prevent system stagnation that could harm lives or otherwise result in losses. Today, break-glass techniques are usually added on top of standard access control solutions in an ad-hoc manner and, therefore, lack an integration into the underlying access control paradigm and the systems' access control enforcement architecture.

We present an approach for integrating, in a fine-grained manner, break-glass strategies into standard access control models and their accompanying enforcement architecture. This integration provides means for specifying break-glass policies precisely and supporting model-driven development techniques based on such policies.

Categories and Subject Descriptors

D.4.6 [**Software**]: Operating Systems—*Security and Protection*

General Terms

security, languages

Keywords

disaster management, access-control, break-glass, model-driven security

1. INTRODUCTION

Today's IT systems comprise a fine-grained access control mechanism based on complex policies. The strict enforcement of these policies, at runtime, always contains the risk of hindering people in their regular work. Thus, writing security policies is always a trade-off between the risk of unintentionally revealing secured data or operations and the benefit gained by using them. This is especially true in exceptional cases where, by definition, seldom used (and thus not well tested) processes are executed.

Motivated by use cases from the disaster management domain, the *break-glass* principle [1] was introduced as one approach for resolving this conflict. Break-glass allows users to override access control decisions on demand. While originally introduced for applications in the disaster management [1] and health care domain [20], break-glass is becoming more and more important for general IT systems. The implementation of recent legitimate regulations, especially in the financial world, like Basel II [4] or Sarbanes-Oxley Act (SOX) [31], requires complicated dynamic access control policies [16]. This increase in policy complexity, together with the overall increase in complexity of IT systems and the increasing requirement to protect sensitive resources in a world wide connected network, also increases the risk of preventing important, business related, processes. To reduce this risk, break-glass concepts are implemented in major business software, e. g., Virsa Firefighter for SAP or Oracle's Role Manager.

Usually, break-glass solutions are implemented by issuing temporary accounts that comprise more powerful access rights (e. g., "root" accounts) on one hand and a more detailed logging on the other hand. While this approach provides a clear separation of the regular policy and the emergency mode, it complicates the a priori analysis of the security policy being effective. This also resembles the traditional software engineering practice where the development of a *design model* (business logic) and a *security model* are treated as different tasks. As a consequence, security features are often built into an existing system in an ad-hoc manner during the system administration phase. While the underlying motivation of this practice, to desire a separation of concerns, is understandable, the conflict between *security requirements* and *availability of services* cannot be systematically analyzed and reasonably balanced in this approach. Solving this problem requires both an integration of break-glass into access control models and the integration of access control models into the early stages of the software development process. Such an integration into one unified methodology is necessary, ranging from the modeling over the implementation to the deployment and the maintenance phase of a system.

To meet this challenge, Basin et al. [6] present a model-driven approach which is built upon the SecureUML lan-

.

guage. SecureUML provides a core security language for access control that can be easily combined with a system modeling language, e. g., UML class diagrams or UML statecharts. SecureUML allows to specify system models and security models within the same visual modeling tool. Moreover, SecureUML is supported by a security-aware model-driven development process, called Model-driven Security (MDS). We extend both SecureUML and MDS with support for break-glass strategies.

Our contributions are four-fold: first, we present a generic break-glass model. Second, we present a SecureUML extension supporting break-glass. Third, we present a security architecture supporting break-glass and, finally, a transformation from break-glass SecureUML policies to XACML.

The rest of the paper is structured as follows: after introducing the preliminaries of our work in Section 2, we present a generic break-glass model which can be integrated into a large class of access control models in Section 3. In the same section, we also present, as an example for such an integration, an extension for SecureUML supporting break-glass. We present a security architecture supporting break-glass in Section 4. This architecture is the target of the transformation of break-glass SecureUML policies to XACML which we present in Section 5. Finally report on related work in Section 6 and present our conclusions in Section 7.

2. BACKGROUND

In this section, we introduce the technical background of our work: the break-glass principle and SecureUML.

2.1 Break-glass

Introduced in [1], *break-glass*[1] refers to quick means for extending a person's access rights in exceptional cases. Usually, the usage of emergency access rights needs to be documented for later audits and reviews. Typically, a special audit trail is created to monitor such override access. Moreover, the regular access control policies should be established in a way minimizing the need for break-glass events.

Usually, break-glass solutions are based on pre-staged user accounts. On one hand, these user accounts need to be available in exceptional cases with reasonable administrative overhead and, on the other hand, misuse of these accounts should be prevented. Typically, a strategy for implementing the break-glass is comprised of the following steps [1]:

1. *Pre-staging break-glass accounts:* Emergency accounts are created in advance to allow careful thought about the access control policies and audit trails associated with them.
2. *Distributing pre-staged accounts needs to be carefully managed to provide timely access when needed:* Break-glass requires the emergency accounts be made available in an appropriate and reasonable manner. The account details may be provided on media such as a printed page, a magnetic-stripe card, a smart card or a token.
3. *Monitoring the use of break-glass accounts:* The use of emergency accounts needs to be carefully monitored. Audit mechanisms should be used and a procedure defined to examine the security audit trails on a regular basis to identify any use of the emergency accounts. In addition, systems can alert the security administrator in the event an emergency account is activated.
4. *Cleaning up after break-glass:* A procedure should be established to clean up after an emergency account has been used.

Traditional break-glass solutions store such emergency accounts either completely electronically or printed on paper and, e. g., stored in a glass cabinet. In contrast, the integration of break-glass into the access control model we are suggesting makes pre-staging accounts unnecessary. And as such, solves the problems of creating and distributing such accounts.

[1]The term "break-glass" is derived from fire alarms that require breaking a glass cover for triggering an alarm.

2.2 SecureUML

SecureUML [6, 9] is a security modeling language based on a generalized Role-based Access Control (RBAC) [14, 30] model. SecureUML is defined using OMG's meta-modeling approach, i. e., the abstract syntax is given as MOF [22] compliant meta-model. Figure 1 illustrates the meta-model of SecureUML and its abstract syntax. SecureUML supports notions of users, roles and permissions, as well as assignments between them: Users can be assigned to roles, and roles are assigned to specific permissions. Users acquire permissions through the roles they are assigned to. Moreover, users are organized into a hierarchy of groups, and roles are organized into a role hierarchy. In addition to this RBAC model, permissions can be restricted by *Authorization Constraints* (expressed in a language similar to OCL [23]), which have to hold to allow access. Permissions specify which *Role* may perform which *Action* on which *Resource*. SecureUML is generic in that it does not specify the type of actions and resources itself. Instead, these are defined in the *design modeling language* which is then "plugged" into SecureUML as a SecureUML *dialect*. This dialect specifies exactly which elements of the design modeling language are protected resources and what actions are available on them. A dialect may also specify a hierarchy on these actions, so that actions, such as reading a class, can be expressed as lower-level actions, such as reading an attribute of the class or executing a side-effect-free method. Furthermore, a dialect specifies a *default policy*, i. e., whether access for a particular action is allowed or denied in the case that *no* permission is specified. Usually, and so did we in this paper, one specifies a default policy of *deny* to simplify the security specification.

Basin et al. [6] present two SecureUML dialects: One for a component-based design modeling language, and one for a state-machine based modeling language. Due to limitations of space, we will not address the issue of dialect definitions further in this paper, and refer to [6] for more details. Instead we will assume as given, without presenting in detail, a SecureUML dialect definition for UML class diagrams in the spirit of the ComponentUML dialect. This means that the dialect specifies classes, attributes and operations to be resources. The dialect also specifies, among others, the actions *create*, *read*, *update*, and *delete* on classes, *read* and *update* on attributes, and *execute* on operations.

Moreover, SecureUML is integrated into a Model-driven Engineering (MDE) toolchain [9] supporting the design of SecureUML policies in the context of UML [24] design models, the formal analysis of these models, and the transformation of these models into (executable) code and configuration for access control enforcement architectures.

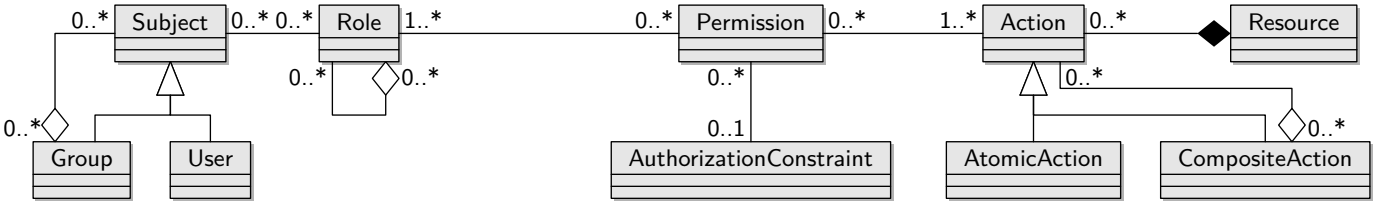

Figure 1: The SecureUML metamodel describes a language supporting users, roles and permissions, as well as assignments between them, e. g., Users can be assigned to roles, and roles are assigned to specific permission.

3. A GENERIC BREAK-GLASS MODEL

In this section, we present a generalized break-glass solution that can integrate various access control models. In more detail, we provide an approach of break-glass that is based on the notion of emergency levels which allow a fine-grained control of policies (rules) that can be overridden.

3.1 Emergency Levels

Compared with traditional break-glass approaches based on pre-staged accounts, our break-glass approach provides:

1. The ability to override access-control decisions on a per permission basis and not on a per role or per subject basis,
2. Several levels of emergency providing a classification on the kind of violation with respect to the regular policy. This classification can be used, for example, for informing users about their actions or for a fine-grained run-time configuration which kind of violations are currently tolerated.

In the following, we assume an access control model $\mathcal{A}$ in which an access control policy p is represented. A policy maps access control relevant information, e. g., subjects, resources, actions, and context information to an access control decision, e. g., deny or allow. In particular, we do neither restrict the expressiveness nor the structure of the policies.

As a prerequisite of introducing emergency levels, we need to introduce a notion of refinement of access control policies. Similar to trace refinement of CSP [29], we define:

Definition 1. A policy p *refines* a policy p' (written $p \sqsubseteq p'$) if and only if the set of system traces that are allowed under p is a subset of the system traces that are allowed under p'.

Informally, a policy p refines a policy p' if and only if p is at least as restrictive as p'. We write $p^\top$ for the policy that allows all actions and $p^\bot$ for the policy that allows no action. The relation $\_ \sqsubseteq \_$ defines a partial order on a set of policies where $p^\bot$ refines all policies and every policy is a refinement of $p^\top$. Therefore, $(P_\mathcal{A}, \sqsubseteq, p^\bot, p^\top)$ is a lattice, where $P_\mathcal{A}$ be the set of all policies of the access control model $\mathcal{A}$.

We refer to the *regular policy*, i. e., the policy that should be obeyed in normal operations, as p^{reg} and we refer to the set of policies that are refined by the regular policy, i. e.,

$$L_\mathcal{A} = \{p \mid p \in P_\mathcal{A} \wedge p^{\text{reg}} \sqsubseteq p \wedge p \neq p^{\text{reg}}\}$$

as *emergency levels* or *emergency policies* of the policy p^{reg}. We require that $(P_\mathcal{A} \backslash p^\bot, \sqsubseteq, p^{\text{reg}}, p^\top)$ is a lattice, i. e., $\inf(P_\mathcal{A} \backslash p^\bot) = p^{\text{reg}}$. At runtime, an emergency level can be *active* or *inactive* and only active emergency levels contribute to the access control decision. Obviously, the regular policy is always active.

An access that is only granted by an emergency policy $\ell \in L_\mathcal{A}$ (i. e., the regular policy evaluates to "deny" for this access) is called *override access*. Such an override access is granted if and only if there is an active policy $\ell \in L_\mathcal{A}$ allowing this access. As we will discuss later, an interactive *confirmation* for applying the override access can be required from the end user.

Overall, a system can have several regular sub-policies active at the same time (e. g., addressing different business divisions). In these cases, we require that the different policies are disjoint, i. e., for a given request the corresponding policy can be efficiently determined. For simplifying the presentation, we assume in the rest of this paper, that there is only one, uniquely defined, p^{reg}.

Finally, *obligations* can be attached to an (emergency) policy. Examples for such obligations are logging requirements that allow the a-posteriori audit of override accesses.

3.2 Implementing Break-glass

Implementing break-glass based on emergency levels instead of using special accounts allows a dynamic adaption of the current system policy and requires only little support from the underlying access control model. In more detail, we propose the following workflow:

- First, we derive the regular policy p^{reg} without taking break-glass situations into account,
- Second, we derive the set $L_\mathcal{A}$ of emergency policies together with a hierarchy of these policies from the domain requirements,
- Finally, a special policy is defined, describing the subjects allowed to activate and de-activate emergency policies during runtime.

Depending on the actual access control model and enforcement architecture, the policy describing the activation of emergency policies can be integrated into the regular policy p^{reg}. To ease analysis of the access control specification, we prefer to treat it, conceptually, as a separate policy.

As we understand the hierarchy of emergency policies as a requirement that is derived from the application domain, we enforce the desired refinement relation by a *policy-level combining algorithm*. For this, we sort all policies in topological order based on the refinement relations. The policy-level combining algorithm evaluates a given access request on all active policies. If all policies deny the request, the algorithm also denies the request. If at least one active policies allows the request, the policy combining algorithm returns the first allow (with respect to the topological sorting) together with the obligation attached to the corresponding policy. This construction obviously ensures the required refinement relation and prefers emergency policies with minimal distance to the regular policy and thus, avoid the need for special-

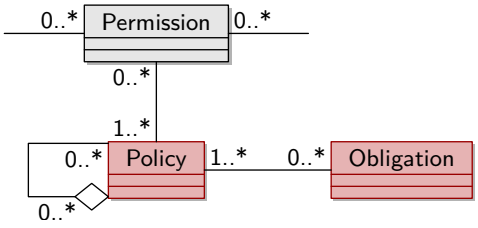

Figure 2: Extending SecureUML with support for break-glass support requires only means for expressing a hierarchy of policies.

ized algorithms for combining obligations, as, for example, developed by Alqatawna et al. [2].

As our refinement notion is only a partial order on the set of policies, the result of the topological sorting is not uniquely defined, i. e., if two active emergency policies allowing the actual request are equidistant from the regular policy, it depends on the implementation which one is chosen (and thus, which obligations are returned together with the policy evaluation result). This ambiguity arises, if two policies, equidistant from the regular policy, define rules for the same target, but require different obligations.

3.3 Break-glass for SecureUML

For supporting our notion of break-glass, SecureUML is mainly missing the notion of expressing a hierarchy of exception levels. As exception levels can be directly represented by policy sets, we extended the SecureUML metamodel with means for expressing hierarchy of policies (where a SecureUML policy is a set of permissions).

Figure 2 illustrates the extension with respect to the original SecureUML metamodel (see Figure 1). First, our extension introduces the concept of policies, i. e., a set of permissions. Second, policies can be organized into a hierarchy and every policy can be associated to obligations. Here, an obligation describes a requirement (e. g., log all upcoming actions with a certain level of detail) that the system must fulfill whenever a permission of the policy grants access.

SecureUML provides a concrete syntax based on UML class diagrams using a UML profile consisting of custom stereotypes. Users, Groups and Roles are represented by classes with stereotypes «secuml.user», «secuml.group», and «secuml.role» (Figure 3). Assignments between them are represented by ordinary UML associations, whereas the role hierarchy is represented by a generalization relationship. Permissions are represented as association classes with the stereotype «secureuml.permission» connecting the role and a *permission anchor*. The attributes of the association class specify which action (the attribute's type) on which resource (the attribute's name) is permitted by this permission. Authorization constraints are constraints attached to the association class. Attributes or operations on roles as well as operations on permission have no semantics in SecureUML and are therefore not allowed in the UML notation. We extend this notation with a *policy hierarchy*. Policies are represented by UML classes with stereotype «secuml.policy» and the hierarchy is represented as a generalization relationship, similar to the role hierarchy of SecureUML. Finally, we support *obligations* (represented by UML classes with stereotype «secuml.obligation») on a per policy basis, i. e., we can assign a set of obligations to every policy.

Figure 3 shows a small example of a system design in UML (i. e., ComponentUML) annotated with an access control policy with break-glass permissions. The design model describes the relation between a MedicalRecord and the Patient who owns the record. The role-based access control model consists of a role hierarchy and two permissions. In more detail, we have a role for regular users (UserRole) and another role for the system administrator AdministratorRole. The latter inherits all permissions of regular users. The regular, i. e., non-emergency, policy p^{reg} allows only the owner to update and delete his MedicalRecord. This requirement is expressed by an additional constraint attached to the permission OwnerMedicalRecord.

This regular SecureUML policy is extended by two exemplary emergency levels (LowEmergencyLevel and HighEmergencyLevel) and emergency permission EmergenyOwnerMedicalRecord allowing every user to read any MedicalRecord. For example, this emergency permission allows everyone to check the medication of a patient in an emergency situation.

For simplicity reasons, we omit the obligations attached to the emergency levels in this example. Such obligations are represented as classes that are associated to a policy.

4. A BREAK-GLASS ARCHITECTURE

In this section, we present an access control enforcement architecture supporting our notion of break-glass. First, we introduce an abstract view of the architecture and, second, we discuss an exemplary system architecture implementing break-glass access control.

4.1 Standard Enforcement Architectures

Figure 4 shows a common architecture for enforcing access control policies. Such an architecture is usually comprised of the following components:

PDP: The Policy Decision Point (PDP) manages the policies (e. g., written in XACML [21] or PERMIS [11]) and evaluates the policy for concrete access requests. Some implementations provide further services such as
- a *context provider* providing information about the environment to the PDP, e. g., the current system time or a list of currently logged in users.
- support for *obligations* that are specified in the policy and must be enforced by the Policy Enforcement Point (PEP). Examples for such obligations are raising log level or sending a notification to the administrator.

PEP: The Policy Enforcement Point (PEP) is usually directly linked to the protected resource (e. g., as a library running within the same process). The PEP is responsible for querying the PDP and the enforcement of the returned access decisions. Moreover, if the answer from the PDP includes obligations, the PEP has to ensure that the system obeys them.

Protected Resources: Examples of protected resources are services, business processes, function calls, and files. All accesses to such resources must be executed via the PEP which enforces the decisions of the PDP.

UI: The User Interface (UI) is responsible for informing the user about access restrictions that concern his interactions with the system.

The workflow in Figure 4 resembles a client requesting access to a resource (1). The PEP queries the PDP (2), which evaluates the policies. The access decision is returned

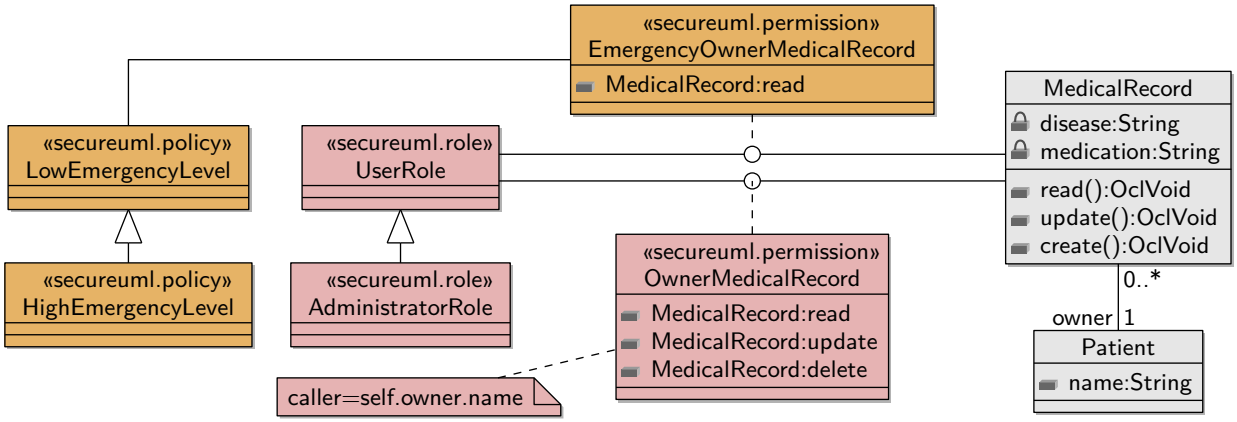

Figure 3: An example of a SecureUML policy for medical records utelizing our extension for supporting policy hierarchies. In particular, allowing every user to read patient data in case of an emergency.

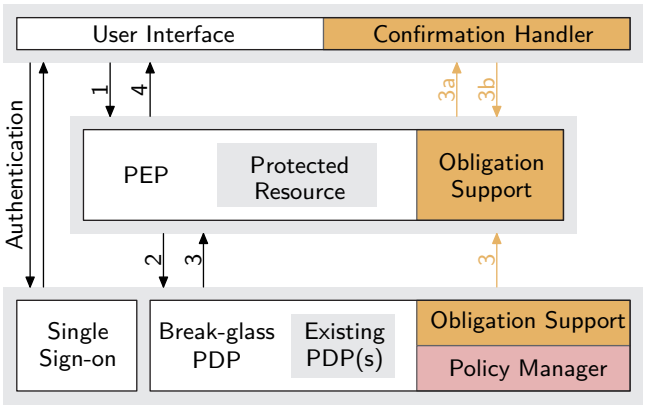

Figure 4: The Break-glass architecture and its message flow.

to the PEP (3), which enforces the result. Either the result of the executed action or an according permission denied message is returned to the client (4).

User authentication is an orthogonal problem which can be treated separately. Within our implementation, the client authenticates itself by passing a security token to the verifying PEP; whereas a single-sign-on engine provides a security token for the client and verifies this token for the PEP.

4.2 Break-Glass Enforcement Architecture

Figure 4 illustrates the required extension for supporting our notion of break-glass on top of a standard access control architecture. The optional, expanding extensions are used for obligations and user confirmation.

The (existing) PDPs are (without modifying them) encapsulated by a break-glass PDP. Such a break-glass PDP provides the interface to the PEPs, uses the emergency policy manager to evaluate the request on the active emergency policies, and executes the policy combination algorithm on the results.

The emergency policy manager is responsible for executing the policy-level combining algorithm based on the active emergency levels and providing an interface to activate and deactivate emergency levels. The means for activating or deactivating emergency is, usually, also subject to access control.

By integrating the policy-level combining algorithm and the emergency policy manager, we achieve two crucial properties of our approach. First, the policy refinement is given by construction, i. e., there is no need to formally analyze the emergency policies. As such, it is guaranteed that access rights are never "lost" when activating a higher emergency level. Second, the policies can be considered as black boxes, i. e., they can be used without modification. Although, in a concrete implementation, it is reasonable to exploit features of the used policy language.

This architecture can be extended stepwise with support for obligations and support for user confirmations.

Obligation Support.

Supporting obligations requires both the support of obligations in the (break-glass) PDP and in the PEP (Figure 4, step (3)). Moreover, break-glass accesses may need to be monitored for later evaluation, i. e., logged (e. g., with additional information such as emergency level used, justification provided by the user, etc.) to a secure medium in a format suitable for later evaluation and approval.

The obligations defined for an emergency level are, as part of the access decision, returned to the PEP. The PEP has to ensure that all obligations are fulfilled. If the compliance to all obligations is not possible, the access decision has to be treated as a deny. Thus, access control frameworks implementing break-glass techniques with support for, e. g., extended logging have to support obligations (or a comparable mechanism which ensures the execution of access restrictions by the PEP).

User Confirmation.

Implementations of break-glass may require the user to confirm an override access, for example, in cases where only the user can decide if an access is legitimate or not. Usually, even for experienced users it is not self-evident which actions are justifiable and which are not. Therefore, the system should give him or her the information required to decide what exactly he or she is not permitted to do, why he or she is not permitted to do so, and what the possible consequences (risks) of overriding the access control are.

Overall, break-glass should be part of the (normal) work process: if a user tries to access a resource he or she is, under normal circumstances (i. e., under policy p^{reg}), not permitted to, but an (active) emergency level ($\ell \in L_{\mathcal{A}}$) could grant access, he should be informed about the denied permission and the possibility to override access. If the user is willing to take the risk and is sure his access is legitimate, he has to confirm the break-glass access. The intuition of a break-glass policy requiring a user confirmation is: "the user is permitted to access the resource, if he is willing to accept that he is overriding his normal competences and willing to accept the responsibilities in case of misuse."

In cases not requiring a user confirmation (e. g., a crises management system being more permissible during a major disaster), no such obligation is attached to the corresponding emergency level.

For supporting user confirmation, the PEP requires access to the UI. If the PEP receives a user conformation obligation in step (3), the PEP accesses the user interface in step (3) and informs the user about the denied permission and the possibility to override access. If the user confirms that the access is justifiable, the UI delivers a confirmation as part of the response in step (3b). This message may contain further (obligated) information, e. g., a justification message from the user. If the user confirms the override access and the PEP is able to enforce all other obligations, in step 4 the result of the access is returned to the client. If the user does not confirm the override, the PEP returns a "permission denied" message back to the client.

5. MODEL TO CODE TRANSFORMATION

In this section, we first present an encoding of break-glass policies in XACML and, second, we present a code-generation for both the security model and the design model given as SecureUML model.

For our implementation, we restrict ourselves to a two-valued OCL-like language for specifying SecureUML permission constraint, i. e., we only support

- calls to public attributes and the result of calling side-effect-free public operation of the current resource (object),
- the special SecureUML keyword `caller` referring to the subject executing an action,
- a limited set of functions for comparing values, e. g., `_ = _`, `_ <= _`, `_ < _`, and
- the Boolean operations `_ and _`, `_ or _`, and `_ not _`.

In particular, we do not support recursive OCL expressions and iterator-based expressions like `->forAll( _ | _ )`.

5.1 Encoding Break-glass in XACML

XACML [21] is a widely used access control language that is supported by freely available frameworks, e. g., an open source XACML PDP is provided by Sun (`http://sunxacml.sourceforge.net/`). In particular, XACML has a built-in support for obligations and context providers, which simplifies the implementation of our break-glass architecture.

For mapping break-glass SecureUML to XACML we need to implement (partially) the Emergency Policy Manager and provide a mapping of SecureUML elements (e. g., permissions, roles) to XACML. In more detail:

- We need to implement an Emergency Policy Manager:
  - implementing a persistence layer that allows for storing and updating the set of active emergency levels and provides the set of active emergency levels to the XACML PDP
  - implementing an externally accessible interface to activate and deactivate emergency levels, protected by an internal policy

  As XACML supports sets of policies and the (user-configurable) combination thereof as built-in, this minimal infrastructure is sufficient.
- For mapping the core SecureUML constructs to XACML and Java, we follow the presentation in [6]. In particular, we need to map OCL-like formulae to XACML (the required attributes for the formulae evaluation are provided in the XACML request from the PEP) and to expand the role hierarchy similar to the approach taken in [6] for the EJB platform.
- Permissions from the SecureUML model are mapped to XACML permit rules. As our mapping only uses permit rules within one policy (i. e., in each emergency policy and in the regular policy), we can use the "first-applicable combining algorithm" for joining the rule sets into one policy.
- Obligations are, as defined by the model, assigned to policies.
- In case of XACML, it is not necessary to implement a custom policy level combining algorithm. Instead, we can re-use the XACML infrastructure and resolve the ordering during the model-to-code transformation. In particular, for each class of the design model, we generate a policy set (XACML element `PolicySet`) containing the permissions (`Rule`) assigned to emergency policies (`Policy`) for this class ordered with respect to the topological sorting of the emergency policies. Thus, all permissions belonging to the regular policy will be mapped on the "top" policy of the policy set, followed by the rules belonging to the first emergency level. Therefore, the policy combining algorithm "first-applicable," which is provided by standard XACML can be used for evaluating the policy.

We generate policies for an arbitrary XACML PDP (we only require the corresponding context provider). The XACML policies generated by our framework are only based on the standard XACML tool set, i. e., no extensions of the language are required. The policy, defining the access control for the Emergency Policy Manager, can be specified in SecureUML itself. Thus, the access controls required for updating the set of active emergency levels is also generated automatically.

5.2 Code Generation

5.2.1 *Generating XACML Policies*

Listing 1 shows the policy for the MedicalRecord from Figure 3, using a reduced form of XACML: for simplifying the presentation, we removed namespaces, shortened attribute names and values, eliminated empty target, subject (role), resource, and action declarations, and simplified condition and obligation definitions (e. g., the `Target` definition in line 2 does not contain a subject or action definition and is therefore reduced to the `Resource` definition).

The `PolicySet` generated for every UML class is saved in a separate policy file. If the permissions of one class are altered, only the file for this class has to be updated. The `Target` definition of the policy set matches the UML class (line 2–4). All permissions defined for this class are con-

```
<PolicySet PolicyCombAlg="first-applicable">
 <Target><Resource>
   MedicalRecord
 </Resource></Target>
 <Policy RuleCombAlg="first-applicable">
  <Rule Effect="Permit">
   <Target>
    <Role>UserRole</Role>
    <Action>update</Action>
   </Target>
   <Condition FunctionId="string-equal">
    <Attribute>subject</Attribute>
    <Attribute>owner.name</Attribute>
   </Condition>
  </Rule>
  <Rule Effect="Permit">
   <Target>
    <Role>UserRole</Role>
    <Action>write</Action>
   </Target>
   <Condition FunctionId="string-equal">
    <Attribute>subject</Attribute>
    <Attribute>owner.name</Attribute>
   </Condition>
 </Rule><Policy>
 <Policy RuleCombAlg="first-applicable">
  <Rule Effect="Permit">
   <Target>
    <Role>UserRole</Role>
    <Action>read</Action>
   </Target>
   <Condition>LowEmergencyLevel</Condition>
  </Rule>
  <Obligation Id="log" FulfillOn="Permit">
   <LogLevel>DEBUG</LogLevel></Obligation>
  <Obligation Id="confirm" FulfillOn="Permit">
   <EmergencyLevel>LowEmergencyLevel
       </EmergencyLevel>
  </Obligation>
 </Policy>
 <Policy>
  <Rule Effect="Deny"/>
<Policy></PolicySet>
```

Listing 1: A break-glass PolicySet: default policy, policy for the LowEmergencyLevel with confirmation and log obligation, and final deny policy

tained in this policy set, so the target matches to `AnySubject` and `AnyAction`.

The policy set contains the default policy (line 5–25), the emergency policies ordered by the hierarchy of emergency levels (line 26–40), and the final deny policy (line 41–43). If an emergency level does not define any permission for the UML class defined by the policy set, the complete policy describing this emergency level can be left out (e. g., no `Policy` for HighEmergencyLevel in Listing 1 as no permission is assigned to this emergency level in Figure 3). The policies match the same resource as the policy set, thus, we do not need to restrict the target of the policies.

Attached to the emergency policy is the log obligation (line 34–35) and the user confirmation obligation (line 36–39). The definition of the emergency level is attached to the rule (line 27–33) as a condition (line 32).

Additionally to the policy set for every UML class, the policy for the emergency policy manager is generated (which has to be modeled in SecureUML with OCL constraints).

```
public interface UserContext {
  // returns authentication information
  AuthnInfo getAuthn();
}
public interface ObligationContext {
  ObligResult fulfill(Obligation obligation);
}
public interface UIContext extends
                          ObligationContext {
  // confirm override-access by the user
  ConfirmationResult confirm(
      Obligation confirmObligation);
}
```

Listing 2: PEP Context Interfaces: supplying the PEP with environment related authentication information and obligation implementations

The generated files are imported into a prepared PDP, which provides the emergency policy manager, protected by the generated policies. Either a jar file is generated, which can be integrated in any environment, or a war file, which can be deployed in a servlet container (providing the interfaces as web service).

5.2.2 *Generating Java*

In our presentation of the code generation, we focus on the break-glass concepts effecting the PEP and the communication with the PDP. The dependencies of the PEP to the environment are represented by context interfaces (Listing 2): the `UserContext` interface helps to abstract from the authentication mechanism (i. e., the authentication technique used in the target application must be encapsulated by a class implementing this interface). The `ObligationContext` interface is used to provide the implementation of obligations for the PEP. For example, the `UIContext` interface helps to acquire the confirmation from the user (i. e., this interface must be implemented by a UI class if user confirmation obligations are used).

For our implementation, we use a simple Java based user interface which implements the `UIContext` interface and authenticates the user at the startup of the application. As the source files of our model are generated, the code accessing the PEP is directly integrated into the Java source files of the entities (Listing 3). The PEP itself is a Java API which is attached as a jar file to the compiled class files. Of course, the PEP API can be used for any (manual, non-generated) implementation.

The `PEP` is the representation of the Policy Enforcement Point which is initialized at application startup and available for every class by singleton. It is part of an API and therefore not generated. During the initialization of the `pep`, the dependencies to the environment must be resolved, depending on the used obligations, e. g., a valid `UIContext` object for user confirmation obligations. For further obligations (e. g., logging), a key-value pair of `ObligationId` and implementing `ObligationContext` interface must be provided (supporting the obligation, identified by an `ObligationId`).

The `PEP` is responsible for resolving the required attributes for the evaluation of the OCL formulae associated with the evaluated permission (e. g., the "owner name" from `this`, passed as first parameter to the `PEP.update()` function). As both the resolving code and the policies for the PDP

```
public class MedicalRecord {
  private static PEP pep;
  // model related class-variables
  public void update(MedicalRecord record) {
    AuthzResult authzResult = pep.authorize(
        this, userContext,
        "MedicalRecord", "update");
    if (authzResult.getDecision() ==
        AuthzResult.PERMIT ) {
      //execute action
    } else {
      throw new PermissionDeniedException(
          authzResult);
    }
  }
}
```

Listing 3: A generated Java class, using a PEP

are generated from the same source (i. e., the SecureUML model), only the required attributes are passed to the PDP.

6. RELATED WORK

The problem of access control unable to handle exceptional situations has been known for at least ten years [7]. One solution is to *pre-stage accounts*, permitted to access more sensitive resources. These accounts (e. g., username/-password pairs) are kept in sealed covers or protected by glass panels, which have to be broken in case of emergency [1]. Similar approaches are implemented by commercial GRC solutions, e. g., Virsa Firefighter for SAP or Oracle's Role Manager.

Optimistic security [26] and a posteriori compliance control [13] delay the access control after access by providing an infrastructure which allows securely auditing and rolling back in case of a denied access, focusing on risks of *not* granting privileges [25]. Similarly, risk-based access control models, e. g., [12], are based on risk or trust models. This allows for integrating the risk of granting access into the access control decision. While this usually results in a more flexible access control decision, we see these techniques as a concept being orthogonal to break-glass. As such, both concepts may benefit from each other. For example, estimating the risk of different emergency levels could, by allowing different risk classes depending on the current emergency level, minimize the need for user confirmations within emergency situations.

Existing work using the term *break-glass* [1] implements break-glass in an ad-hoc, application specific manner on top of the underlying access control mechanism. Ferreira et al. [15] implement the break-glass mechanism in the source code of the business logic, i. e., the decision is not defined in a policy but directly in the accessed entity. Longstaff et al. [20] focus on a specific health care application with an application specific authorization mechanism.

Stevens et al. [32] distinguish in the point of time when permissions are defined: *ex-ante* (before access), *uno-tempore* (during access), and *ex-post* (after access). Rissanen et al. [27] combine this with Access Control Spaces [18] and propose a model which has, additional to *permit* and *deny*, a further access decision: *possibility-with-override*, allowing the user to override the access restrictions. Furthermore, Rissanen et al. [28] describe a technique called Authority Resolution (finding, with a given override and the XACML policy, a person who is in position to approve the override).

Based on the work of Rissanen et al. [27, 28], Alqatawna et al. [2] present an approach for overriding access control in XACML with obligations. They introduce a specific type of obligation, *override-obligation*, and, in addition to the effects-combining algorithms in standard XACML, an *obligations-combining algorithm* to be able to distinguish between normal and override obligations. Thus, if a permit rule is available, no *possible-with-override* rule is used.

A more universal approach, related to break-glass, is to make the decision process more flexible, thus expanding the expressiveness of rules and its describing language (such as Generalized Temporal RBAC [19], context aware RBAC [33, 17]). For example, XACML allows to take the environment into concern (such as system time, or physician-injured ratio). Alam et al. [**?**] model pre-defined conditions which have to be true before break-glass access is permitted, e. g., during an emergency visit. These concepts help to define more accurate policies, but also raise complexity—*"a wide variety of access control models have the expressive power to represent almost arbitrary policies, few are ever used by others due to their complexity"* [18].

A further approach to temporarily increase access rights is delegation [3, 34]. There are major analogies: different delegation approaches have to solve the question which rights are delegated; i. e., the rights to access a function or the rights to access the resources, this function will work on. As for delegation the delegate should only delegate a task one time (and not be asked again, if an additionally required permission should be granted), even in the case of break-glass policies the user should break the glass only once.

7. DISCUSSION AND FUTURE WORK

We presented an access control model agnostic approach for implementing break-glass policies. As an example for the integration of our approach into an existing access control model, we extended SecureUML with our notion of break-glass. This extension is the basis for a model-driven development approach supporting role-based access control policies with break-glass. In particular, our approach supports the generation of break-glass SecureUML policies for a concrete security architecture based on Java and XACML.

Compared with existing solutions, we avoid the need for special accounts which cause additional administrative overhead and are also an additional security threat (e. g., due to loss or theft of the pre-staged accounts). Moreover, our approach integrates means for monitoring and logging the usage of emergency rights using obligations.

In this paper, we assume that in emergencies the rights of subjects are extended (i. e., they have at least their normal rights); in fact, our notion of break-glass guarantees this property by construction (instead of requiring a formal proof). While, for example, this is a strict requirement in the disaster management or health-care domain, there are applications where the opposite behavior is required. For these situation, extend our model by the set $L_{\mathcal{A}}^{\text{deny}}$ of policies that refine the regular policy

$$L_{\mathcal{A}}^{\text{deny}} = \{p \mid p \in P_{\mathcal{A}} \wedge p \sqsubseteq p^{\text{reg}} \wedge p \neq p^{\text{reg}}\}$$

and adapt our construction accordingly, i. e., an access is only granted, if it is granted by the regular policy *and not denied* by any active emergency polices.

Although we implemented our approach using XACML (and exploiting its support for obligations and generic policy combining algorithms), our approach can be transferred to a large class of policy languages. First, by using an external emergency policy manager and extending, if necessary, the PEP with support for obligations, our notion of break-glass can be added on top of arbitrary policy languages and frameworks. Optionally, users can be asked for confirmations. For supporting such confirmations, the PEP needs to be integrated in the user interface. Notably, the policy combination algorithms and obligation support can be encapsulated in the emergency policy manager. Second, the core idea of our notion of break class, i. e., a hierarchy of policies with a partial order can be encoded directly in many policy languages. Recall our running example (see Figure 3); for example for Prolog-like languages like Datalog, we can write

```
readMedicalRecord(P, medicalRecord)
  :- role(P, userRole), owner(P).
readMedicalRecord(P, medicalRecord)
  :- role(P, userRole), emergencyLevel(low).
```

for modeling that in emergency cases not only the owner of a medical record is allowed to read it. Similarly, by extending XACML with an additional "emergency level" attribute for `Policy` elements, the ordering of the emergency levels can be done by an additional component within the PDP.

Our approach provides means for specifying a fine-grained hierarchy of emergency policies together with a policy restricting the activation and deactivation of the emergency policies. A fine-grained policy hierarchy is a prerequisite for both a careful monitoring and for providing detailed feedback to the user. Nevertheless, for practical reasons the policy restricting the activation and deactivation of emergency policy should be coarse-grained, i. e., most emergency levels should be active by default. Otherwise, the set of subjects that are able to activate the emergency levels are becoming a bottle-neck. Such activation of emergency levels can also be implemented without user intervention, based on context information such as time, monitoring information, or sensor values. For example, a hospital could, automatically, increase the set of activated emergency levels on weekends where only a reduced number of doctors is on duty.

We see several lines of future research motivated by the integration of break-glass support into traditional access control models. On the practical side, we see the need for better means of informing users about the effect of overriding regular policies.

On the analytical side, we plan to extend existing analysis methods for SecureUML [5, 10] to our extension of SecureUML. The (formal) access control specification with different emergency levels should also allow for a more detailed and automated post-mortem analysis techniques and information flow across emergency levels.

While our approach for supporting break-glass is, in principle, access control model agnostic, extending systems based on data labeling, e. g., Bell-LaPadula, with break-glass needs to be investigated in more detail. Moreover, a classification of long-living vs. short living data (or different abstraction level of data) could simplify the audit, i. e., the post-mortem analysis. This could result in combinations of role-based access control and approaches based on data-labeling.

Finally, support for dynamic (i. e., additional checks at runtime are necessary) access control specifications like separation of duty or binding of duty has to be developed. Integrating such dynamic requirements into a break-glass framework makes a controlled transition from an exception level to normal behavior particular challenging. Of course, static (i. e., properties that can be ensured statically by the policy set) separation of duty or binding of duty constraints are supported by our framework.

The concrete (UML-based) syntax of SecureUML is quite lengthy. For working with large models, an integration of SecureUML concepts into a CASE tool is available [8]. This extension allows for directly specifying access control within the GUI of ArgoUML. This extension needs to be extended for supporting our SecureUML extension, i. e., a notion of policies and obligations.

8. ACKNOWLEDGMENTS

We would like to thank Adam J. Lee and the anonymous referees for helpful comments on the paper.

This work has been supported by the German "Federal Ministry of Education and Research" in the context of the project "SoKNOS." The authors are responsible for the content of this publication.

9. REFERENCES

[1] Break-glass: An approach to granting emergency access to healthcare systems. White paper, Joint NEMA/COCIR/JIRA Security and Privacy Committee (SPC), 2004.

[2] J. Alqatawna, E. Rissanen, and B. Sadighi. Overriding of access control in XACML. In *Proceedings of the Eighth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 87–95, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[3] E. Barka and R. Sandhu. Framework for role-based delegation models. In *Proceedings of the 16th Annual Computer Security Applications Conference*, pages 168–176, Los Alamitos, CA, USA, 2000. IEEE Computer Society.

[4] Basel Committee on Banking Supervision. Basel II: International convergence of capital measurement and capital standards. Technical report, Bank for International Settlements, Basel, Switzerland, 2004.

[5] D. Basin, M. Clavel, J. Doser, and M. Egea. Automated analysis of security-design models. *Information and Software Technology*, 51(5):815–831, 2009. Special Issue on Model-Driven Development for Secure Information Systems.

[6] D. A. Basin, J. Doser, and T. Lodderstedt. Model driven security: From uml models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology*, 15(1):39–91, 2006.

[7] K. Beznosov. Requirements for access control: US healthcare domain. In *Proceedings of the third ACM workshop on Role-based access control (RBAC)*, page 43, New York, NY USA, 1998. ACM Press.

[8] A. D. Brucker and J. Doser. Metamodel-based UML notations for domain-specific languages. In J. M. Favre, D. Gasevic, R. Lämmel, and A. Winter, editors, *4th International Workshop on Software Language Engineering (ATEM 2007)*. Oct. 2007.

[9] A. D. Brucker, J. Doser, and B. Wolff. An MDA framework supporting OCL. *Electronic Communications of the EASST*, 5, 2006.

[10] A. D. Brucker, J. Doser, and B. Wolff. A model transformation semantics and analysis methodology for SecureUML. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS 2006: Model Driven Engineering Languages and Systems*, number 4199 in Lecture Notes in Computer Science, pages 306–320. Springer-Verlag, 2006. An extended version of this paper is available as ETH Technical Report, no. 524.

[11] D. W. Chadwick and A. Otenko. The PERMIS X.509 role based privilege management infrastructure. In *Proceedings of the seventh ACM symposium on Access control models and technologies (SACMAT)*, pages 135–140, New York, NY USA, 2002. ACM Press.

[12] N. Dimmock, A. Belokosztolszki, D. Eyers, J. Bacon, and K. Moody. Using trust and risk in role-based access control policies. In *Proceedings of the ninth ACM symposium on Access control models and technologies (SACMAT)*, pages 156–162, New York, NY USA, 2004. ACM Press.

[13] S. Etalle and W. H. Winsborough. A posteriori compliance control. In *Proceedings of the 12th ACM symposium on Access control models and technologies (SACMAT)*, pages 11–20, New York, NY USA, 2007. ACM Press.

[14] D. F. Ferraiolo, R. S. Sandhu, S. I. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[15] A. Ferreira, R. Cruz-Correia, L. Antunes, P. Farinha, E. Oliveira-Palhares, D. Chadwick, and A. Costa-Pereira. How to break access control in a controlled manner. In *Proceedings of the 19th IEEE International Symposium on Computer-Based Medical Systems (CBMS)*, pages 847–854, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[16] C. Fox and P. Zonneveld. *IT Control Objectives for Sarbanes-Oxley: The Role of IT in the Design and Implementation of Internal Control Over Financial Reporting*. IT Governance Institute, Rolling Meadows, IL, USA, 2nd edition, Sept. 2006.

[17] J. Hu and A. C. Weaver. Dynamic, context-aware access control for distributed healthcare applications. In *Proceedings of the First Workshop on Pervasive Security, Privacy and Trust (PSPT)*, 2004.

[18] T. Jaeger, A. Edwards, and X. Zhang. Managing access control policies using access control spaces. In *Proceedings of the seventh ACM symposium on Access control models and technologies (SACMAT)*, pages 3–12, New York, NY USA, 2002. ACM Press.

[19] J. B. Joshi, E. Bertino, U. Latif, and A. Ghafoor. A generalized temporal role-based access control model. *IEEE Transaction on Knowledge and Data Engineering*, 17(1):4–23, 2005.

[20] J. Logstaff, M. Lockyer, and M. Thick. A model of accountability, confidentiality and override for healthcare and other applications. In *Proceedings of the fifth ACM workshop on Role-based access control*, pages 71–76, New York, NY USA, 2000. ACM Press.

[21] eXtensible Access Control Markup Language (XACML), version 2.0, 2005.

[22] OMG XML metadata interchange (XMI) specification (version 1.1), Nov. 2000. Available as OMG document formal/00-11-02.

[23] UML 2.0 OCL specification, Oct. 2003. Available as OMG document ptc/03-10-14.

[24] UML 2.0 superstructure specification, July 2005. Available as OMG document formal/05-07-04.

[25] D. Povey. Enforcing well-formed and partially-formed transactions for Unix. In *Proceedings of the 8th conference on USENIX Security Symposium*, volume 8, pages 5–5. USENIX Association, 1999.

[26] D. Povey. Optimistic security: A new access control paradigm. In *Proceedings of the 1999 workshop on New security paradigms*, pages 40–45, New York, NY USA, 1999. ACM Press.

[27] E. Rissanen. Towards a mechanism for discretionary overriding of access control (transcript of discussion). In B. Christianson, B. Crispo, J. A. Malcolm, and M. Roe, editors, *Proceedings of the 12th International Workshop on Security Protocols*, volume 3957 of *Lecture Notes in Computer Science*, pages 320–323, Heidelberg, Mar. 2004. Springer-Verlag.

[28] E. Rissanen, B. S. Firozabadi, and M. J. Sergot. Discretionary overriding of access control in the privilege calculus. In T. Dimitrakos and F. Martinelli, editors, *Proceedings of the Workshop on Formal Aspects Security and Trust (FAST)*, volume 173, pages 219–232, Heidelberg, 2004. Springer-Verlag.

[29] A. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1998.

[30] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[31] P. Sarbanes, G. Oxley, et al. Sarbanes-Oxley Act of 2002. 107th Congress Report, House of Representatives, 2nd Session, 107–610, 2002.

[32] G. Stevens and V. Wulf. A new dimension in access control: studying maintenance engineering across organizational boundaries. In *Proceedings of the ACM conference on Computer supported cooperative work (CSCW)*, pages 196–205, New York, NY USA, 2002. ACM Press.

[33] M. Wilikens, S. Feriti, A. Sanna, and M. Masera. A context-related authorization and access control method based on RBAC: A case study from the health care domain. In *Proceedings of the seventh ACM symposium on Access control models and technologies (SACMAT)*, pages 117–124, New York, NY USA, 2002. ACM Press.

[34] L. Zhang, G.-J. Ahn, and B.-T. Chu. A role-based delegation framework for healthcare information systems. In *Proceedings of the seventh ACM symposium on Access control models and technologies (SACMAT)*, pages 125–134, New York, NY USA, 2002. ACM Press.