A Framework for Managing and Analyzing Changes of Security Policies

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

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

***Abstract*—Modern enterprise systems need to comply to complex security policies. Due to legal regulations such as Basel II or HIPAA, the enforcement of these security policies needs to be carefully monitored and analyzed. The monitoring of complex and often dynamic access control requirements results in a vast amount of information that needs to be analyzed both in case of incidents and during regular audits.**

We present an extensible framework for managing and analyzing security policies during their whole life cycle. Our framework integrates versioning of policies and logfiles with policy animation, static analysis, and debugging techniques. For example, this combination allows for comparing different versions of security policies or the replaying and animation of system traces based on logfiles.

I. Introduction

Modern enterprise systems, e. g., for enterprise resource planning (ERP) or customer relationship management (CRM) need to enforce a variety of different and complex security policies. Monitoring the enforcement of the different security policies results in a large amount of log information (i. e., traced system events) that is, because of the sheer size, difficult to analyze. Moreover, today's enterprises are operating in a frequently changing environment (e. g., fluctuation of staff member, restructuring of business units, or changes in requirements and regulations) resulting in a significant number of deviations from the intended system behavior. For example, violations of the security policies caused, e. g., by wrongly configured systems, wrongly implemented system polices, systems or users still following old workflows, active policies that are not yet reflecting new organizational structures, or new staff members testing if their account is already enabled.

Additionally, businesses need to comply to regulations such as Basel II [2] or the Health Insurance Portability and Accountability Act (HIPAA) [16], which can only partially be enforced at runtime (see, e. g., [13]). In addition, recent trends, e. g., [4, 11], allowing users to override access restrictions in a controlled manner, result in an even larger amount of log information that needs to be analyzed. Both developments lead to a significant increase in the costs for manual system audits [15, 28]. Consequently, there is a large demand from, e. g., system administrator, auditors, or forensic experts, for tools supporting the analysis of the compliance to high-level regulations or the analysis of policy violations. We answer this demand by presenting an extensible framework for managing and analyzing policies and the generated logfiles during their whole life cycle. In particular, our framework allows to compare and analyze such policies and logfile with various plugins implementing different analysis techniques during audits and forensics. In more detail, our contributions are three-fold:

1) A generic framework for managing policies during their whole life cycle and, in particular, for comparing different versions of security policies and log information using both specialized algorithms and generally available external tools,
2) A generic framework for analyzing security policies that includes the animation and replaying, potentially based on a recorded system state, of access control requests and analyzing the access control decision for different policy versions, and
3) A policy debugger which allows to debug XACML policies by setting break-points on XACML elements, viewing the current state (e. g., the call stack, resolved attributes and their values), and manipulate the value of attributes.

Overall, our framework allows for developing and debugging security policies in isolation. This isolation helps to ensure that only well-tested configuration are deployed in the productive environment.

The rest of the paper is structured as follows: After introducing our running example in Section II, we present our policy management and analysis framework in Section III. In Section IV, we present several applications of our framework. Finally, we discuss related work and present our conclusions in Section V.

Table I
RECORDED MEDICAL RECORD ACCESSES FOR HEALTH RECORDS.

evalID	version	date	time	user	role	dept	pat.dept	patient	result
863	139	2010-06-30	18:11	alice	nurse	neurology	neurology	davis	permit
870	139	2010-06-30	18:13	carol	nurse	surgery	surgery	young	permit
875	139	2010-06-30	18:13	alice	nurse	neurology	surgery	young	deny
881	139	2010-06-30	18:17	dave	doctor	neurology	neurology	davis	permit
894	139	2010-06-30	18:18	alice	nurse	neurology	neurology	davis	permit
902	139	2010-06-30	18:23	bob	doctor	surgery	surgery	johnson	permit
914	139	2010-06-30	18:29	dave	doctor	neurology	neurology	earp	permit
923	139	2010-06-30	18:32	bob	doctor	surgery	neurology	davis	permit
...	...	...	...	...	...	...	...	...	...
1001	142	2010-07-01	17:54	bob	doctor	surgery	neurology	moore	permit
1012	142	2010-07-01	17:55	alice	nurse	neurology	neurology	moore	permit
1023	142	2010-07-01	17:57	carol	nurse	surgery	surgery	white	permit
1034	142	2010-07-01	17:59	marvin	nurse	dentistry	surgery	white	deny
1045	142	2010-07-01	18:02	dave	doctor	neurology	neurology	moore	permit
1067	142	2010-07-01	18:03	marvin	nurse	dentistry	surgery	white	deny
1078	142	2010-07-01	18:06	bob	doctor	surgery	surgery	young	permit
1089	142	2010-07-01	18:07	carol	nurse	surgery	surgery	miller	deny
1100	142	2010-07-01	18:08	carol	nurse	surgery	surgery	miller	deny
1110	142	2010-07-01	18:08	marvin	nurse	dentistry	surgery	miller	deny
1117	142	2010-07-01	18:08	alice	nurse	neurology	neurology	davis	deny
1128	142	2010-07-01	18:10	bob	doctor	surgery	neurology	moore	permit

II. Running Example

In this section, we briefly introduce a running example that, on the one hand, illustrates an exemplary application of our framework and, on the other hand, introduces the technical background of our work.

In modern enterprise systems, the enforcement of security policies is usually based on a centrally managed and administered *Policy Decision Point* (PDP). The PDP stores security policies for all secured services and provides means for evaluating access control requests, i. e., requests asking if a certain users is allowed to access a certain resource. Policy languages, such as the *eXtensible Access Control Markup Language* (XACML) [24], allow for expressing whether a particular access should be allowed or not.

Listing 1 presents an excerpt of a security policy (in simplified XACML) for the management of health records. In our example, permissions are not only granted based on role assignments, but are also restricted by constraints: nurses (i. e., members of the role `Nurse`) are only allowed to read patient records during regular working hours (i. e., between 6 am and 8 pm) and if the patient belongs to the same department.

Table I illustrates a set of (simplified) logfile entries. During audit or forensics, such logfiles are usually examined manually for anomalies (e. g., deviations from the intended behavior). In the following, we assume that starting on July 1st, working hours are redefined to be between 6 am and 6 pm. Thus, the security policy needs to be updated (i. e., in Listing 1 line 10 is changed to `<value>18:00</value>`) and this change needs to be activated on July 1st.

```
    <PolicySet PolicyComb="first-applicable">
2    <Target><Resource>
       HealthRecord</Resource></Target>
4    <Policy RuleCombAlg="first-applicable">
      <Target><Role>Nurse</Role></Target>
6     <Rule Id="1" Effect="Deny">
       <Target/>
8      <Condition><Time-in-range>
        <current-time/>
10      <value>20:00</value>
        <value>06:00</value>
12     </Time-in-range></Condition>
      </Rule>
14    <Rule Id="2" Effect="Permit">
       <Target><Action>read</Action></Target>
16     <Condition><String-equal>
        <patient-department>
18      <subject-department>
       </Condition>
20   </Rule></Policy>
     <Policy RuleCombAlg="first-applicable">
22    <Target><Role>Doctor</Role></Target>
      <Rule Id="3" Effect="Permit">
24     <Target/></Rule>
     </Policy>
26   <Policy><Target/>
      <Rule Id="final" Effect="Deny"/>
28  </Policy></PolicySet>
```

Listing 1. An XACML policy: nurses are allowed to access health records during the day of patients located on their department.

Some time after the policy change, an administrator detects a suspicious large number of denies on July 1st, requiring an analysis, by a security expert, to rule out the possibility of an attack. By inspecting some of those requests manually, the security expert recognizes that some denied requests might also be caused by the change of the security policy at July 1st, i. e., nurses still following their old schedule. Using our framework, the security expert does not need to analyze every single request manually. Instead, the security expert can load the old policy, re-execute the recorded requests, eliminate requests permitted by the old policy, and analyze the remaining denied accesses. This allows the security specialist to focus on the most important requests, i. e., shielding those requests that are caused by nurses that are not yet used to the new working hours. The remaining denies are nearly all caused by user `marvin`, indicating a misbehaving user, leading to further, concrete investigations. As a next step the security specialist may chose select all (denied and permitted) access by user `marvin` around July 1st.

In the rest of the paper, we present a framework that allows to support such investigations by supporting the management and analysis of such changes in both the security policies and the corresponding logfiles.

III. Security Policy Analysis and Management Framework

In this section, we present a framework for managing and analyzing security policies together with the recorded system traces (e. g., logfiles). In particular, our framework supports the management of security policies during their whole life-cycle.

A. Architecture

Figure 1 illustrates our framework in an exemplary context of a system following the *service-oriented architecture* (SOA) principle (left-hand side of Figure 1).

In SOAs, the security policy is usually enforced by several Policy Enforcement Points (PEPs) which use a central Policy Decision Point (PDP) for evaluating the security policy. In contrast, most information, e. g., properties of resources, are only available through the service which is managing that specific resource. Thus, if such a property is required during the evaluation of the policy, the PDP requests its resolution by the *context information service*, also called Policy Information Point (PIP). Our framework extends, respectively, adds the following components (right-hand side of Figure 1):

Versioning Policy Storage: This component provides a versioning store for policies. Thus, all changes of the policy are recorded. In the following, we assume that the security policy also contains the mapping from users to roles.[1] Overall, the versioning policy storage allows for managing, selecting and comparing different versions of a policy. For example, this allows for creating and testing a new policy version before it is actually deployed in a productive system.

Versioning Logfile Storage: This component stores all access control requests, including the version of the active policy,[2] the access control decision, obligations, and context attributes with their actual values. The policy and logfile storage provide an interface to search for and load elements by multiple selection criteria.

Policy Decision Point: In our framework, the PDP needs to be able to load the active policy from the versioning policy store and, moreover, it needs to provide the relevant information to the versioning logfile storage. On the implementation level, the required functionality can be fully implemented by a wrapper, i. e., existing PDPs can be integrated.

Workbench: The *Security Policies Analysis and Management Workbench* bundles the core components for the analysis and management of versioned polices and provides, on a conceptual level, two user interfaces: the first one (Administrator/Management User Interface) is dedicated to the management of policies along their life-cycle. The second one (Analysis/Audit User Interface) focuses on the analysis part. While we expect the former one to be used by regular system administrators, the latter one will be mainly used by security experts. In more detail, the workbench comprises the following modules:

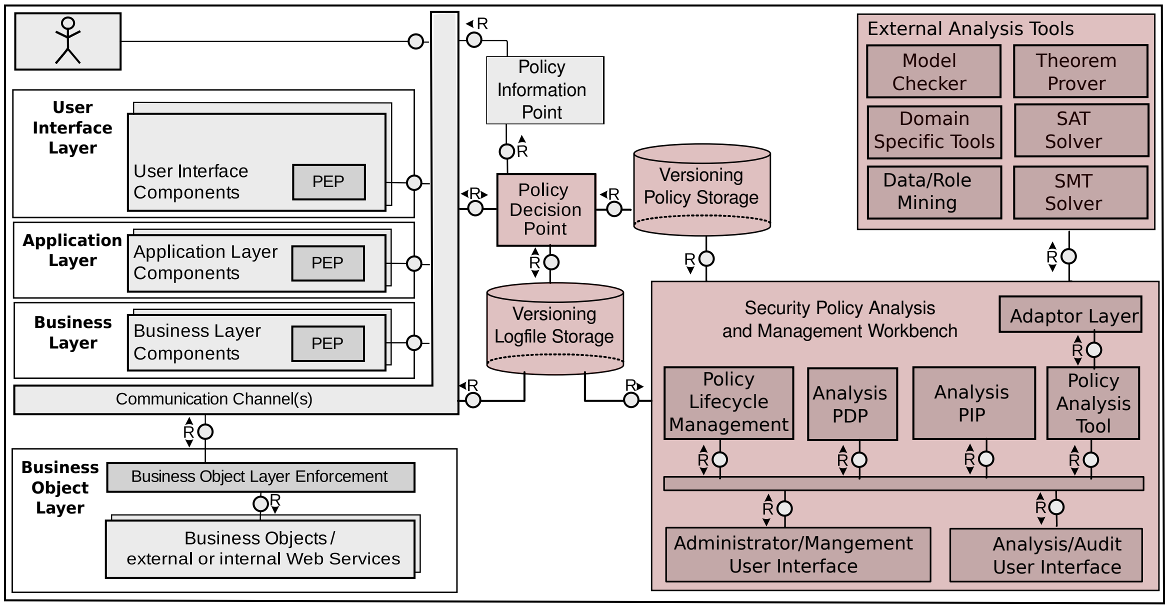
Policy Management : The Policy Management Component (consisting out of a back-end service and a dedicated user interface) allows for the administration of policies. For example, policies can be modified (i. e., new versions of policies can be created) and be activated (i. e., the currently active policy version can be set) whereon the productive PDP reloads the currently active policy. For most parts, existing solutions can be adopted and reused.

Analysis PDP: These PDPs within the workbench can be used with different policy configurations, e. g., different versions of the system policy. The Analysis PDPs operate in a simulated environment, i. e., not being connected to a real back-end system.

Analysis PIP: The Analysis PIP acts as context information service for the Analysis PDPs. As they do not have a current application context at analysis time, the Analysis PIP either retrieves context information from the log storage (i. e., as is has been available at runtime) or from the policy analysis tool.

[1]If this is not the case, our framework can easily be extended with a versioning user and role storage.

[2]Alternatively, the version can be inferred from the timestamp.



External Analysis Tools
Model Checker
Theorem Prover
Domain Specific Tools
SAT Solver
Data/Role Mining
SMT Solver
User Interface Layer
User Interface Components
PEP
Application Layer
Application Layer Components
PEP
Business Layer
Business Layer Components
PEP
Communication Channel(s)
Policy Information Point
Policy Decision Point
Versioning Policy Storage
Versioning Logfile Storage
R
Business Object Layer
Business Object Layer Enforcement
Business Objects / external or internal Web Services
Security Policy Analysis and Management Workbench
Adaptor Layer
Policy Lifecycle Management
Analysis PDP
Analysis PIP
Policy Analysis Tool
Administrator/Mangement User Interface
Analysis/Audit User Interface

Figure 1. Extended authentication and authorization infrastructure with analysis and forensics workbench.

Policy Analysis: The Policy Analysis (consisting out of a Policy Analysis Tool and a dedicated user interface) allows to analyze policies together with the corresponding logfiles, i. e., access information from the versioning storage, load Analysis PDPs and execute (recorded) requests against those Analysis PDPs.

Moreover, various external analysis tools can be integrated, ranging from generic to domain specific tools and algorithms (e. g., [3]). Generic tools such as model checkers, SMT and SAT solvers, theorem provers, or data mining tools may be used to solve complex tasks needed for an analysis method implemented by the policy analysis tool. An Adaptor Layer provides an uniform interface to the analysis tool and for abstracting from the concrete data formats of the different externals tools.

The analysis framework can also be used in isolation, i. e., providing an analysis workbench that is independent from the business system.

B. Implementation

Our prototype is based on Sun's XACML implementation (http://sunxacml.sourceforge.net/) as a core PDP implementation and Subversion, or svn for short, (http://subversion.apache.org/) as underlying versioning system. We extended the class holding the configuration information (`PDPConfig`) using a svn-enabled policy finder module. This extensions allows the PDP to load arbitrary policy versions from the versioned policy store.

Replaying access control requests requires to store the system state at runtime for later evaluation, i. e., all information used by the XACML engine has to be logged. For efficiency reasons, we implement the logging functionality within the PDP, using *aspectJ* (http://www.eclipse.org/aspectj/) to set a pointcut on all methods resolving attributes. This has two advantages. First, we do not need to parse the XACML input and output as a wrapper would have to. Second, third party libraries providing non-standard attributes or implementing the access of the PIP can be monitored without additional effort. At runtime, the attributes are collected by a class managing the current evaluation context (`EvaluationCtx`), whereas the log entries are processed by another thread, wherewith the logging functionality creates only minimal overhead. We allow the usage of non-standard attributes and functions, as long as, first, the attributes are resolved using an implementation of the abstract `AttributeFinderModule` class and, second, functions do not have any side-effects and do not retrieve external information.

For the Analysis PDP we implemented a class `AttributeFinderModule` which retrieves attributes required for evaluation from the workbench. Thus, the workbench can decide if the attribute is retrieved from the logstore, or if the user or an (external) analysis tool is queried for a value. We also enhanced the core of the XACML engine. For this, we introduced a `RuntimeInfo` class (see Listing 2), responsible for the management of the source location (i. e., source file and line number) and runtime information required for constructing the call stack. Most XACML elements have a direct representing Java object, e. g., for the

```
package com.sun.xacml.debug;
interface Locatable{
 RuntimeInfo getRuntimeInfo();
}
interface IndirectLocatable
                    extends Locatable {
 void setRuntimeInfo(RuntimeInfo src);
 void unsetRuntimeInfo(RuntimeInfo src);
}
class RuntimeInfo {
 int getLineNumber();
 String getFileName();
 void setCalledFrom(Object o);
 Object getCalledFrom();
 RuntimeInfo getIndirectSourceLocator();
}
```

Listing 2. Java classes added to the XACML core.

policy given in Listing 1 there are four Java objects of type `Rule`. Classes representing such elements have to implement the `Locatable` interface and retrieve a `RuntimeInfo` object during policy loading, which remains the same at runtime.

In contrast, XACML functions and combining algorithms have to implement the `IndirectLocatable` interface as they do not have a direct representing Java object (e. g., for Listing 1, there is only one Java object representing the first-applicable combining algorithms). Thus, instead of creating a `RuntimeInfo` at policy loading time, this object is created at runtime (using the `getIndirectSourceLocator()` method) depending on the context they are executed in. As recursive calls are permitted, for `IndirectLocatable` elements both set and unset method have to be called.

Creating the call stack at runtime requires three modifications of the core XACML engine: First, the used policy finder module has to write the source file and line information into the `RuntimeInfo` object. Second, a `RuntimeInfo` object has to be created whenever a `IndirectLocatable` object is evaluated. Third, we use *aspectJ* pointcuts to retrieve all evaluation events. This is used to call the `setCalledFrom(Object o)` method and therefore to create the call stack and to provide the workbench with notifications (event) before and after each evaluation of an XACML element.

IV. Application

After a brief introduction of the necessary preliminaries, we discuss several analysis techniques that are supported by our framework.

A. Preliminaries

In the following, we describe the content of the versioning logfile storage (recall the exemplary logfile presented in Table I), i. e., a single log entry le as follows as an n-tupel:

$$le = (id_{eval}, id_{policy}, date, time, model)$$

where

- id_{eval} describes the unique evaluation id,
- id_{policy} describes the version of the policy that was active during the creation of this entry. In the following, we assume that there exists a partial order $\_ \leq_p \_$ on policy versions.
- $date$ describes the date on which the entry was created. Naturally, there is a total order $\_ \leq \_$ on dates values.
- $time$ describes the time on which the entry was created. Naturally, there is a total order $\_ \leq \_$ on time values.
- $model$ describes the security model specific entries of the log-entry, i. e., in our example,

$$model = (user, role, dep_{\text{staff}}, dep_{\text{patient}}, patient, result)\,.$$

Moreover, we define selection functions that allow for accessing specific elements of such an n-tupel. For example,

$$\text{getId}_{\text{policy}}\,(id_{eval}, id_{policy}, date, time, model) = id_{policy}\,.$$

This allows us to describe the versioning logfile storage LS. as a set of log entries, e. g., $le_i \in LS$.

B. Selecting Ranges of Log-entries

One basic step for analysis is the selection of ranges of log-entries along multiple selection criteria such as recorded within a certain period of time, accessing a specific resource, evaluating to a specific result, containing a specific context attribute value, etc. For example,

$$\big\{le \in PS \mid (\text{getDate}\ \ le = \text{2010-06-30} \vee \text{getDate}\ \ le = \text{2010-07-01}) \wedge \text{getPatient}\ \ le = \text{white}) \wedge \text{getResult}\ \ le = \text{permit})\big\}$$

selects all successful access attempts to the record of the patient white between June 30th and July 1st.

C. Interactive Exploration of Policies

A very important step during the development of technical security policies is the validation that the technical security policy implements the high-level security goals. Interactively exploring the policy is a first approach that helps to convince oneself that certain security goals are fulfilled. Our framework allows to execute an access control request against a Analysis PDP running, for example, the test version of a new policy. During the evaluation of such an explorative request, the user is prompted for any missing attribute value (e. g., the system time or the department of the patient). Thus, the user can directly simulate access control requests with respect to a specific system configuration.

D. Replaying Access Control Requests

Access control requests can be evaluated in a simulated environment, i. e., the requests are evaluated in an environment which simulates the system state at recording time. But, our workbench also allows to modify the environment as needed for an analysis, i. e., simulating an environment for testing the behavior under a different situation. Thus, our replaying facility allows not only to reproduce an access control decision that was taken in a given situation, it also allows to simulate arbitrary environments. This allows, for example, to re-evaluate a recorded request under a different policy, e. g., based on an log entry

$$\begin{aligned} le = (&1089, 142, \text{2010-07-01}, \text{18:07}, \\ &\text{carol}, \text{nurse}, \text{surgery}, \text{surgery}, \text{miller}, \text{deny}) \end{aligned}$$

we can construct the access control requests

$$\begin{aligned} (&1089, 139, \text{2010-07-01}, \text{18:07}, \\ &\text{carol}, \text{nurse}, \text{surgery}, \text{surgery}, \text{miller}) \end{aligned}$$

and

$$\begin{aligned} (&1089, 142, \text{2010-07-01}, \text{18:07}, \\ &\text{carol}, \text{nurse}, \text{surgery}, \text{surgery}, \text{miller}) \end{aligned}$$

that will help us to understand if, for this specific request, the policy versions 139 and 142 result in a different access control decision.

But, not only the policy version may be changed, also the availability and values of recorded attributes, i. e., everything which defines the environment of the evaluation. For example, one may test if request 1089 would have been permitted an hour earlier, i. e., if

$$\begin{aligned} le = (&1089, 142, \text{2010-07-01}, \text{17:07}, \\ &\text{carol}, \text{nurse}, \text{surgery}, \text{surgery}, \text{miller}) \end{aligned}$$

would have resulted in a permit decision.

E. Debugging

Debugging tools are a well known and widely used technique for investigating program errors, helping the user to understand the execution flow of a program to find and eliminate bugs.

Similar to this, we provide a debugger for XACML, which allows to define breakpoints for the evaluation of every XACML element. When the system is halted at such a break-point, one can browse the current state of the evaluation, i. e., the call stack, look for resolved attributes or even manipulate their values, etc. This allows, for example, to figure out, if a specific rules is evaluated or why a specific target match does not match. Of course, the interactive exploration of policies and the replaying of access control requests can be combined with the policy debugger.

F. Policy Animation

In the following we show how symbolic input-partitioning [5, 10] (i. e., the computation of equivalence classes), can be used for providing support for policy animations. With policy animation, we refer to techniques that allow to abstractly evaluate access control requests, i. e., requests are evaluated with abstract attribute representations instead of concrete values. This allows for evaluating access control requests without the need of resolving attributes from an actual system context or a recorded system state.

We use HOL-TESTGEN [6] as an external analysis tool for implementing the policy animation. In more detail, we integrate the input partitioning of HOL-TESTGEN as follows: we extended the combining algorithms used within the Analysis PIP in such a way that they support the partial evaluation of access control requests, i. e., the evaluation of rules and policies for which not all attributes can be resolved. Using those extended combining algorithms, we generate a constraint describing the rule being evaluated (including the indented effect of that rule). This constraint (called *test specification* in HOL-TESTGEN) is transferred to HOL-TESTGEN which is used for computing the partitions (equivalence classes) for each attribute. In case of XACML, the test specification needs to consider the different rule and policy combining algorithms.

The formalization of security policies in HOL-TESTGEN (see [8, 9] for details) allows for handling a large set of attribute types and functions defined over them—without the need of converting those datatypes into bit-vectors. For example, recall the `time-in-range` function from Listing 1, which takes three input parameters (time: x, m, n) and checks, if the first parameter x is within the second and the third (i. e., $m \leq x \leq n$, whereas time ranges may span over midnight, e. g., $x =$ 23:00 is within the time range $m =$ 20:00 to $n =$ 06:00). As HOL-TESTGEN is extensible using higher-order logic, which is quite similar to a functional programming language, the translation into a test-specification is straight-forward:

$$\text{PUT}\,\text{x} \;=\; \text{timeInRange 20 6 t} \longrightarrow \text{deny}$$

where PUT represents the policy under test and where the security policy formalization of HOL-TESTGEN defines the function timeInRange as follows:

```
definition
  timeInRange :: "int ⇒ int ⇒ int ⇒ bool"
  where
  "timeInRange frm to time
  =  (({frm, to, time} ⊆ {0 .. 24})"
       ∧ ( if frm < to
         then (frm ≤ time) ∧ (time ≤ to)
         else ((frm ≤ time) ∧ (time ≤ 24)
                ∨ (0 ≤ time) ∧ (time ≤ to )))) "
```

We also need to consider the rule, respectively, policy combining algorithms. Thus, the complete test-specification looks as follows:

```
PUT x  =
 firstApplicable  [ firstApplicable
 [timeInRange 20 6 t ⟶ deny, λ x. permit ],
                    λ x. deny]
```

Which results in the following three equivalence classes:

1) $6 < \text{time} < 20$ results in a permit
2) $0 \leq \text{time} \leq 6$ results in a deny
3) $20 \leq \text{time} \leq 24$ results in a deny

For a fully automated solution, HOL-TestGen can also generate concrete test-data, i. e., select a representative ground value for each equivalence class.

G. Discussion

The discussed analysis techniques cannot only be used in isolation. For example, when replaying an access control request, attributes may be missing (i. e., they are not resolvable from the logstore). In such a situation, combining the replay with other analysis techniques may help to analyze this situation in more detail: First, using the interactive policy exploration allows for querying the user for a concrete value of the missing attribute. Second, the policy animation allows for providing a set of equivalence classes of missing attributes to the user and, third, missing attributes may serve as a natural break-point for the policy debugging.

The basic techniques used for implementing our framework allow for the implementation of further functionalities. For example, the pointcuts for retrieving all evaluation events (see Section III-B) can also be used for collecting detailed runtime information about the policy evaluation, which, e. g., may be used for optimizing the performance of the policy evaluation (e. g., similar to [22]).

V. Related Work and Conclusion

A. Related Work

There is a large body of literature presenting static analysis [1, 3, 12, 18, 20, 21], dynamic analysis [14] analysis, or specification-based test case generation techniques [7, 14] of security policies. Many of those techniques can be integrated into our framework and, moreover, can profit from the detailed versioning of policies and logfiles our framework provides. For example, integrating the Alloy Analyzer [19] into our framework seems to be very attractive as there are several analysis methods for security policies that are based on Alloy, e. g., for the development of conflict-free role-based access control policies [27], or conformance testing [17].

There are several analysis tools for XACML-policies, e. g., [12, 23]. Although only supporting a small subset of XACML, Magrave [12] is an interesting candidate for providing additional functionality for analyzing differences between various versions of a policy.

B. Conclusion

We presented an extensible framework for managing and analyzing security policies that helps to answer the challenges of modern enterprise systems which have to, in an ever-changing environment, comply to complex security policies and compliance regulations. As a unique feature, our framework allows to load and compare different versions of policies as well as access control requests executed within the system (including requested runtime information and results), to analyze them with various plug-ins implementing different analysis techniques during audits and forensics.

The flexibility of our frameworks allows for the continuous integrating of both dynamic and static analysis and, thus, combining the strengths of both techniques. Thus, we see the integration of further techniques for analyzing and transforming of security policies as a main line for future work. Besides extending the current portfolio of analysis tools, e. g., with access control approaches that rely on post-hoc audits such as break-glass [4, 26] or optimistic security [25], we see the integration of tools for comparing and optimizing policies, e. g., [1, 7, 21], as particularly promising.

Acknowledgments

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant no. 257930.

References

[1] M. Backes, G. Karjoth, W. Bagga, and M. Schunter. Efficient comparison of enterprise privacy policies. In H. Haddad, A. Omicini, R. L. Wainwright, and L. M. Liebrock, editors, *SAC*, pages 375–382, New York, NY USA, 2004. ACM Press. doi: 10.1145/967900.967983.

[2] Basel Committee on Banking Supervision. Basel II: International convergence of capital measurement and capital standards. Technical report, Bank for International Settlements, Basel, Switzerland, 2004. URL http://www.bis.org/publ/bcbsca.htm.

[3] D. Basin, M. Clavel, J. Doser, and M. Egea. Automated analysis of security-design models. *Information and*

Software Technology, 51(5):815–831, 2009. ISSN 0950-5849. doi: 10.1016/j.infsof.2008.05.011. Special Issue on Model-Driven Development for Secure Information Systems.

[4] A. D. Brucker and H. Petritsch. Extending access control models with break-glass. In B. Carminati and J. Joshi, editors, *ACM symposium on access control models and technologies (SACMAT)*, pages 197–206. ACM Press, 2009. doi: 10.1145/1542207.1542239.

[5] A. D. Brucker and B. Wolff. Symbolic test case generation for primitive recursive functions. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Testing of Software*, number 3395 in Lecture Notes in Computer Science, pages 16–32. Springer-Verlag, 2004. doi: 10.1007/b106767.

[6] A. D. Brucker and B. Wolff. HOL-TESTGEN: An interactive test-case generation framework. In M. Chechik and M. Wirsing, editors, *Fundamental Approaches to Software Engineering (FASE09)*, number 5503 in Lecture Notes in Computer Science, pages 417–420. Springer-Verlag, 2009. doi: 10.1007/978-3-642-00593-0_28.

[7] A. D. Brucker, L. Brügger, P. Kearney, and B. Wolff. Verified firewall policy transformations for test-case generation. In *Third International Conference on Software Testing, Verification, and Validation (ICST)*, pages 345–354. 2010. doi: 10.1109/ICST.2010.50.

[8] A. D. Brucker, L. Brügger, M. P. Krieger, and B. Wolff. HOL-TESTGEN 1.5.0 user guide. Technical Report 670, ETH Zurich, Apr. 2010.

[9] A. D. Brucker, L. Brügger, P. Kearney, and B. Wolff. An approach to modular and testable security models of real-world health-care applications. In *ACM symposium on access control models and technologies (SACMAT)*. ACM Press, 2011.

[10] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J. Woodcock and P. Larsen, editors, *Formal Methods Europe 93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284, Heidelberg, Apr. 1993. Springer-Verlag.

[11] 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 IEEE International Symposium on Computer-Based Medical Systems (CBMS)*, pages 847–854, 2006. doi: 10.1109/CBMS.2006.95.

[12] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *ICSE*, pages 196–205, New York, NY USA, 2005. ACM Press. doi: 10.1145/1062455.1062502.

[13] 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.

[14] S. K. Ghai, P. Nigam, and P. Kumaraguru. Cue: a framework for generating meaningful feedback in XACML. In *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*, SafeConfig '10, pages 9–16, New York, NY USA, 2010. ACM Press. doi: 10.1145/1866898.1866901.

[15] A. Ghose. Information disclosure and regulatory compliance: Economic issues and research directions. http://ssrn.com/abstract=921770, July 2006.

[16] HIPAA. Health Insurance Portability and Accountability Act of 1996. http://www.cms.hhs.gov/HIPAAGenInfo/, 1996.

[17] H. Hu and G.-J. Ahn. Enabling verification and conformance testing for access control model. In *ACM symposium on Access control models and technologies (SACMAT)*, pages 195–204, New York, NY USA, 2008. ACM Press. doi: 10.1145/1377836.1377867.

[18] G. Hughes and T. Bultan. Automated verification of access control policies using a sat solver. *International Journal on Software Tools for Technology*, 10:503–520, October 2008. ISSN 1433-2779. doi: 10.1007/s10009-008-0087-9.

[19] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002. ISSN 1049-331X. doi: 10.1145/505145.505149.

[20] V. Kolovski, J. Hendler, and B. Parsia. Analyzing web access control policies. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 677–686, New York, NY USA, 2007. ACM Press. doi: 10.1145/1242572.1242664.

[21] D. Lin, P. Rao, E. Bertino, and J. Lobo. An approach to evaluate policy similarity. In V. Lotz and B. M. Thuraisingham, editors, *SACMAT*, pages 1–10, New York, NY USA, 2007. ACM Press. doi: 10.1145/1266840.1266842.

[22] A. X. Liu, F. Chen, J. Hwang, and T. Xie. XEngine: A fast and scalable XACML policy evaluation engine. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (Sigmetrics)*, Annapolis, Maryland, June 2008.

[23] M. Mankai and L. Logrippo. Access control policies: Modeling and validation. In *Conférence Internationale sur les Nouvelles Technologies de la Repartition (NOTERE)*, pages 85–91, 2005.

[24] OASIS. eXtensible Access Control Markup Language (XACML), version 2.0, 2005. URL http://docs.oasis-open.org/xacml/2.0/XACML-2.0-OS-NORMATIVE.zip.

[25] 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. doi: 10.1145/335169.335188.

[26] L. Rostad and O. Edsberg. A study of access control requirements for healthcare systems based on audit trails from access logs. In *Annual Computer Security Applications Conference (ACSAC)*, pages 175–186, Los Alamitos, CA, USA, 2006. IEEE Computer Society. doi: 10.1109/ACSAC.2006.8.

[27] A. Schaad and J. D. Moffett. A lightweight approach to specification and analysis of role-based access control extensions. In *Proceedings of the seventh ACM symposium on Access control models and technologies*, SACMAT '02, pages 13–22. ACM Press, 2002. doi: 10.1145/507711.507714.

[28] L. Sneller and H. Langendijk. Sarbanes Oxley Section 404 Costs of Compliance: a case study. *Corporate Governance: An International Review*, 15(2):101–111, 2007. URL http://econpapers.repec.org/RePEc:bla:corgov:v:15:y:2007:i:2:p:101-111.