Auditing with Incomplete Logs

Umbreen Sabir Mian[1], Jerry den Hartog[1], Sandro Etalle[1 2], and Nicola Zannone[1]

[1] Eindhoven University of Technology
[2] University of Twente
{u.s.mian, j.d.hartog, s.etalle, n.zannone}@tue.nl

Abstract. The protection of sensitive information is of utmost importance for organizations. The complexity and dynamism of modern businesses are forcing a re-think of traditional protection mechanisms. In particular, *a priori* policy enforcement mechanisms are often complemented with auditing mechanisms that rely on an *a posteriori* analysis of logs recording users' activities to prove conformity to policies and detect policy violations when a valid explanation of conformity does not exist. However, existing auditing solutions require that the information necessary to assess policy compliance is available for the analysis. This assumption is not realistic. Indeed, a good deal of users' activities may not be under the control of the IT system and thus they cannot be logged. In this paper we tackle the problem of accessing policy compliance in presence of incomplete logs. In particular, we present an auditing framework to assist analysts in finding a valid explanation for the events recorded in the logs and to pinpoint policy violations if such an explanation does not exist, when logs are incomplete. We also introduce two strategies for the refinement of plausible explanations of conformity to drive analysts along the auditing process. Our framework has been implemented on top of CIFF, an abductive proof procedure, and the efficiency and effectiveness of the refinement strategies evaluated.

Keywords: Abduction, Policy Compliance, Abductive Reasoning

1 Introduction

Policy compliance is a critical issue faced by organizations. Regulations like HIPAA, Sarbanes-Oxley Act and Basel II, define guidelines and best practices that organizations have to implement in order to assure a minimum level of protection for sensitive information and corporate assets as well as to regulate the assignment and execution of work. Organizations usually translate these guidelines and best practices into policies and procedures for their enforcement. An important class of policies organizations have to comply with, is formed by *usage policies*. Usage policies define rules and constraints on the access and use of sensitive information. Compliance to usage policies (which we will refer to as 'conformity') is of utmost importance. Indeed, failure to comply with these policies can result in serious data leakage, privacy violation and fraud which have significant legal and financial consequences to organizations [9, 23].

When it comes to usage control, conformity is usually addressed using preventive enforcement mechanisms. However, these mechanisms are too inflexible to deal with exceptions and unpredictable situations, which often arise in complex organizations.

Moreover, usage policies often include constraints that cannot be enforced a priori, e.g. future obligations [25] and purpose control [26]. Therefore, *a priori* policy enforcement mechanisms can be complemented with auditing mechanisms (still, mostly human-driven) that rely on an *a posteriori* analysis of logs recording users' activities. Here, users are not prevented from accessing sensitive information; rather, they are held accountable for their actions.

Some approaches [1, 6, 7, 26] have been proposed for an automated a posteriori compliance with usage policies. The basic idea underlying these approaches is to analyze user activities, recorded in logs, against the policies in order to construct a justification for such activities and pinpoint the causes of non-conformity when a policy is violated. To perform such an analysis, existing solutions assume that all information necessary to assess policy compliance is available for the analysis. This assumption, however, is very difficult to meet in practice.

The logs maintained by organizations are often *incomplete* [12]. There are several causes for this incompleteness. For instance, actions that are not supported by the IT system cannot be monitored by the IT system itself [4]. Moreover, verbal authorizations (i.e., authorizations which are not documented in a written form) are often acceptable, and sometimes needed to deal with emergency situations. Another source of incompleteness is given by the distributed nature of modern applications, business processes and underlying IT systems. User activities are often not confined to a single system but span several systems. Each system may be under the control of a different authority/organization, which might not be willing to disclose logs to other organizations as they can reveal confidential corporate information.

Thus, an auditor who has to assess the adherence of an organization to regulations and policies can often only rely on incomplete logs, which provide a partial view of what happened within the system. These considerations raise the main research question addressed in this paper: *How can we effectively assist auditors in verifying conformity to usage policies based on incomplete logs?*

This paper presents an auditing framework for a posteriori compliance with usage policies when logs are incomplete, i.e. when the logs available for the analysis only provide a partial knowledge of what happened within the system. Our auditing framework makes it possible to *(i)* find a valid explanation for the events recorded in the logs and *(ii)* pinpoint policy violations if such an explanation does not exist. We analyze and discuss both theoretical and practical underpinnings underlying the framework.

To find explanations of conformity, we rely on abductive reasoning and, in particular, on Abductive Logic Programming (ALP) [17]. Abduction is a tool for hypothetical reasoning with incomplete knowledge which aims to explain *observations* describing the actual system state from *hypotheses* that are considered possible, provided that they are consistent with the intended system behavior. (Hereafter we refer to the set of possible hypotheses explaining the observations as *plausible explanations* of conformity). ALP extends logic programming to support abductive reasoning by allowing some predicates to be incompletely defined. The advantage of adopting a formalism based on logic programming is that it makes it possible to reuse well-defined frameworks for the specification and evaluation of usage policies and, especially, access control and trust management policies (e.g., [13, 16, 22, 29]).

To assess policy compliance an auditor should determine what actually happened, i.e. find a *valid explanation* of conformity. However, observations can have (infinitely) many plausible explanations. Thus, the auditor should investigate the hypotheses forming the plausible explanations and determine which ones are valid. Unfortunately, existing abductive logic programming frameworks [10, 17] do not provide support to identify valid explanations of conformity; they only provide a list of plausible explanations. To assist an auditor in the validation of plausible explanations, we introduce a new form of abductive reasoning which allows the refinement of plausible explanations of conformity until a valid explanation is obtained. Intuitively, the auditing process supported by our framework is *iterative*, where an auditor selects some hypotheses forming plausible explanations, checks their validity, and new plausible explanations are obtained by considering the gathered information.

The selection of the hypotheses to be validated has a significant impact on the effectiveness and efficiency of the auditing process. To assist an auditor in the validation of plausible explanations, we propose two *refinement strategies* that drive an auditor along the auditing process by selecting the "best" hypothesis to be validated. In particular, these strategies aim to minimize the efforts, in terms of the number of hypotheses that have to be validated and the number of iterations, necessary to obtain a valid explanation of conformity (or evidence of non-conformity).

The refinement of plausible explanations may lead to situations in which a valid explanation for the observations does not exist. These situations provide an indication that some user did not behave as intended, i.e. that a *policy violation* occurred. To assist an auditor in the investigation of policy violations, our auditing framework allows the identification of the potential damage of violations (called *consequences*) as well as their *root causes* (i.e., the actions that were the initial cause of the violation).

We have implemented a prototype supporting the auditing framework with incomplete logs on top of the CIFF Proof procedure [28], an abductive proof procedure implemented in Prolog. Moreover, we performed experiments to evaluate and compare the effectiveness and efficiency of the proposed refinement strategies.

The remainder of the paper is structured as follows. Section 2 introduces preliminaries on logic programming and ALP. Section 3 provides an overview of the auditing process with incomplete logs. Section 4 presents an auditing framework based on abductive reasoning for policy compliance, and Section 5 a framework for analysis of policy violations. Section 6 describes the refinement strategies. Section 7 presents a prototype implementation of the auditing framework with incomplete logs, and Section 8 presents an evaluation of the refinement strategies. Finally, Section 9 discusses related work, and Section 10 concludes the paper providing directions for future work.

2 Preliminaries

This section provides an overview of logic programming and, then, shows how abductive reasoning can be instantiated to logic programming.

2.1 Logic Programming

Logic programming syntax is defined using Horn clauses. An *atom* is an expression of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol and $t_1, \ldots, t_n$ are terms. A literal is an expression of the form A or not A where A is an atom. A literal is *ground* if it contains no variables. A Horn clauses is a formula of the form

$$H \leftarrow B_1, \ldots, B_n \quad (\text{with } n \geq 0)$$

where H is an *atom* called *head* and $B_1, \ldots, B_n$, called *body*, is a conjunction of *atoms*. The empty head is equivalent to *false*, whereas the empty body is equivalent to *true*. A clause with an empty body is called *fact* (the "$\leftarrow$" symbol is usually omitted in facts). A *logic program* P is a finite set of clauses.

A query Q is a conjunction of literals. A *valuation* σ is an assignment of values to variables. The semantics of $\models$ has the usual logic programming interpretation, together with negation as failure. Notice that our programs do not contain negation (negation may be present only in queries), so we can refer to the minimal Herbrand model of a program. Given a logic program P, and a ground atom A, we write

$$P \models A$$

if A is true in the minimal Herbrand model of P. For negated atoms we use the negation as (possibly infinite) failure rule and write

$$P \models \neg A$$

iff $P \not\models A$. This notation extends in the straightforward way to more complex formulas without quantifiers (e.g., conjunction, disjunction).

2.2 Abductive Logic Programming

In this work we use Abductive Logic Programming (ALP) for abductive reasoning [17]. Abduction is a tool for hypothetical reasoning with incomplete knowledge. It aims to explain some observations describing the actual system state by means of possible hypotheses that are considered possible, provided that they are consistent with the intended system behavior. ALP combines abduction with logic programming enriched by integrity constraints. An abductive problem in ALP can be seen as the task of finding a set of hypotheses which, together with a logic theory representing the system behavior, allows the inference of the observations. In the remainder of this section, we present the main concepts of ALP.

Definition 1. *An* abductive framework *is a tuple $\langle P, A, IC \rangle$ where P is a logic program referred to as 'abductive theory', A is a set of predicate symbols referred to as 'abducibles' and IC is a set of integrity constraints.*

The set of abducibles is a domain-specific predefined class of predicate symbols used to construct the possible hypotheses explaining the observations. Hereafter, we

denote by $\overline{A}$ the set of all ground terms that can be constructed over a set of predicate symbols A.

Not every combination of hypotheses constitutes a valid explanation. *Integrity constraints* are formulas which have to be "satisfied" by abductive explanations. Explanations that do not satisfy integrity constraints are not valid. Various characterizations of integrity constraints have been proposed in the literature. Based on [28], we represent integrity constraints using a general implicative form:

$$L_1 \wedge \cdots \wedge L_n \rightarrow H_1 \vee \cdots \vee H_m$$

where $L_1, \ldots, L_n$ are literals and $H_1, \ldots, H_m$ are atoms. The $\rightarrow$ symbol is used instead of the $\leftarrow$ symbol to distinguish between program clauses ($\leftarrow$) and integrity constraints ($\rightarrow$).

A diagnosis is a set of hypotheses that explains a set of observations, represented as a query, based on the system description.

Definition 2. *A* diagnosis *to a query Q with respect to an abductive framework $\langle P, A, IC\rangle$ is a pair $\langle \Delta, \sigma\rangle$, where $\Delta \subseteq \overline{A}$ is a finite set of ground abducible atoms and σ is a valuation for the free variables occurring in Q, such that $P \cup \Delta \models IC \wedge Q\sigma$.*

3 Auditing Process

In this section, we present our auditing process with incomplete logs. The flow chart of the process is given in Figure 1. In the remainder of this section, we introduce the basic concepts and provide an overview of the process.

User actions are typically recorded by the system in logs. The information contained in these logs corresponds to *observations*. Observations are analyzed by auditors to determine whether a user violated the security policies employed by the organization. However, not all user actions can be logged. For instance, actions that are not supported by the IT system usually cannot be monitored by the IT system itself [4]. Therefore, an auditor has to determine whether the actions performed by users were legitimate based on a partial knowledge of what happened. The auditor can make some *hypotheses* about what he deems possible, i.e. what may have happened without being logged. From these hypotheses the auditor will want to find and validate those that justify the observations made on the system. However, there may be several (possibly infinitely many) plausible explanations to justify the observations.

The auditing process aims to find and fully validate an explanation for the observations. The input to the process consists of a description of the system and the policies defining its intended behavior (the *abductive theory* and *integrity constraints*) as well as some events observed by the auditor representing the actions performed by the users (*observations*). From this input, the approach aims to find a valid explanation (a *diagnosis*) for the observations. (We will use a form of abductive reasoning to find diagnoses if any exists as described in the next section). Intuitively, a diagnosis consists of a set of hypotheses that have to be analyzed and verified to demonstrate compliance with usage policies. If a diagnosis does not exist, it means that a policy violation occurred. In this case, the approach aims to identify which policy violations occurred along with their consequences and root causes (Section 5).

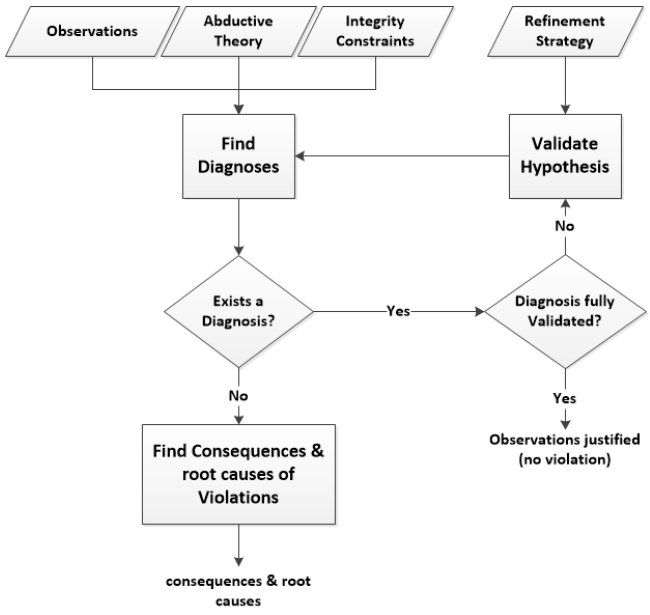

Fig. 1. Flowchart of the auditing process

If one plausible diagnosis (or more) does exist, the auditor should validate the hypotheses in plausible diagnoses by determining whether they correspond to events which have actually occurred. Based on the gathered information, the process is reiterated to refine the set of plausible diagnoses. The process terminates when one diagnosis, in which all hypotheses have been verified, is found or a plausible diagnosis does not exist. *Refinement strategies* can be employed to drive auditors in the selection of the "best" candidate hypotheses to be verified (Section 6).

Next, we introduce a scenario and an access control system with delegation inspired on Audit Logic [7], which are used in the remainder of the paper to demonstrate our auditing approach with incomplete logs.

Example 1. A firm is required to perform the analysis of the privacy practices of a hotel. The hotel is part of a worldwide chained-brand hotel. The chained-brand hotel is associated with several travel agencies which provide their customers the possibility of booking a complete traveling package including plane tickets, hotel reservation and car rental. The hotel also has an agreement with some airline companies to provide accommodation in case of flight delays and cancellations. The hotel records the access to customer data in logs. By analyzing these logs, the auditor observes that the hotel personnel had accessed the data of a customer who did not stay at the hotel and he wants to investigate this situation. However, the hotel log only provides a partial view of what actually occurred. Each partner of the hotel (e.g., chained-brand hotel, travel agencies, airline companies) has records of their customers along with access logs. These partners are not willing to give their logs to others, but they may answer some specific questions;

Abductive Theory:
R1 $\mathsf{own}(U, O) \leftarrow \mathsf{event}(U, O, create)$
R2 $\mathsf{has\_perm}(U, O, \_) \leftarrow \mathsf{own}(U, O)$
R3 $\mathsf{has\_perm}(U_1, O, R) \leftarrow \mathsf{event}(U_2, O, del(U_1, R))$
Integrity constraints:
IC1 $\mathsf{event}(U_1, O, create), \mathsf{event}(U_2, O, create), U_1 \neq U_2 \rightarrow \mathsf{false}$
IC2 $\mathsf{event}(U, O, R), not\ \mathsf{has\_perm}(U, O, R), R \neq create \rightarrow \mathsf{false}$

Fig. 2. Abductive theory and integrity constraints describing the system behavior

for example whether they have disclosed the data of a particular customer to a certain partner. The auditor thus has to contact the other partners with such questions in order to reconstruct the chain of delegations of permission from the customer to the hotel under investigation in order to verify whether the hotel personnel were allowed to access such data.

Below we present a logic programming characterization of the policies regulating the access to data. We first present the predicates used to describe the state of the system; then we present the clauses and integrity constraints describing the intended behavior of the system (i.e., the policies). Predicate $\mathsf{event}(u, o, r)$ is used to denote the action in which user u exercises right r on object o. A right r can be atomic (e.g., create, read, write) or a delegation. We use function symbol "del" to represent delegation rights where $del(u, r)$ denotes the right to delegate right r to user u. Binary predicate $\mathsf{own}(u, o)$ is used to denote the owner u of object o. Predicate $\mathsf{has\_perm}(u, o, r)$ is used to indicate that a user u has the permission to exercise right r on object o. The theory and integrity constraints underlying the access control system are presented in Figure 2. R1 states that the user who created an object is the owner of the object. R2 states that the owner of an object has full authority on the object, i.e. all rights. R3 states that if a user delegates a certain right over an object, then the delegatee has the right over the object. Integrity constraints are used to determine behavior that is not allowed by the system. IC1 states that an object cannot be created by two different users. IC2 allows a user to exercise a certain right only if he has such a right. IC2, however, does not apply to the creation of objects. Indeed, we assume that users do not need permission to create an object.

4 Checking Policy Compliance

To support auditors in the assessment of conformity to usage policies, we rely on abductive reasoning. Intuitively, we aim to find a diagnosis that explains why users were allowed to perform the actions observed by the auditor. To this end, we specify the problem of assessing conformity as an abductive diagnostic problem. Within an abductive framework, the *abductive theory* specifies the policies governing the system by defining the allowed behavior, while *integrity constraints* set boundaries on the system behavior by explicitly modeling which behavior is not allowed by the system. The effect of integrity constraints is to prune diagnoses which are not valid according to the policies.

Abducibles represent types of facts that are not logged within the system e.g. because they are not under the control of the system. *Observations* represent an ordered set of events logged by the system. Observations are used to construct the query which is evaluated against the abductive framework. It is worth noting that an observation may be explained by some observations that occurred previously. Thus, we will also use these observations to assess conformity to policies.

The abductive reasoning presented in Section 2.2 by itself is insufficient to assess conformity as it only aims to identify possible explanations for the observations. In contrast, auditors should prove conformity by establishing what actually happened not only what possibly happened. To support the auditing process presented in the previous section, we redefine the abductive diagnostic problem and introduce a new form of abductive reasoning. To this end, we first introduce the concepts of *proof obligation* and *validated hypothesis*.

To perform the actions observed by the auditor, users need to satisfy some properties (e.g., having the permission to execute the action). Inspired by [7] we introduce a proof obligation function *po* that gives for each observation a property that needs to be satisfied for the observations to be compliant. A *property* ϕ is an arbitrary formula built using connectives *not* (denoted $\neg$), *and* (denoted ",") and *or* (denoted ";"), representing the proof obligations underlying the observations to be proven. We denote Φ as the set of all properties. The semantics of $\models$ is extended to properties as follows. Let $\langle P, A, IC\rangle$ be an abductive framework and $\langle \Delta, \sigma\rangle$ a diagnosis, we say that $\langle P, A, IC\rangle$ and $\langle \Delta, \sigma\rangle$ *satisfy* property ϕ iff

$$P \cup \Delta \models \phi$$

Example 2. In the access control system of Example 1, proof obligations are represented using predicate has_perm. In particular, the proof obligation function *po* maps every event $\mathsf{event}(u, o, r)$ with $r \neq create$ to a property of the form $\mathsf{has\_perm}(u, o, r)$. The creation of an object, represented by an event $\mathsf{event}(u, o, create)$, is associated to an empty obligation proof (i.e., true). This corresponds to the assumption that users do not need permission to create an object.

To prove conformity of user actions to policies, auditors have to validate the hypotheses in the plausible diagnoses. In other words, auditors have to verify that those hypotheses correspond to actions that actually happened. In the remainder of the paper we call the hypotheses whose validity has been verified *validated hypotheses*. It is worth noting that observations, hypotheses, and validated hypotheses are events that the auditor has observed, deems possible, and has verified respectively. We stress that validated hypotheses differ from observations: observations are events logged by the system which have to be explained, while validated hypotheses are events which are believed to be true by the auditor. Validated hypotheses are used to verify whether the proof obligations underlying the observations hold.

We now have the machinery needed to define the abductive diagnostic problem for policy compliance.

Definition 3. *Given a set of events E, A* policy compliance problem *is a tuple $\langle AF, H, O, po\rangle$ where AF is an abductive framework, $H \subseteq E$ is the set of validated hypotheses. $O \subseteq E$*

is the set of observations *to be explained, and* $po : E \rightarrow \Phi$ *is a function from events to properties.*

The auditing process starts with policy compliance problem $\langle AF, \emptyset, O, po \rangle$, i.e. the set of validated hypotheses is initially empty. A diagnosis is a set of hypotheses that explain the proof obligations underlying observations based on the system description. Note that observations form an ordered sequence of events. Hereafter, we use $O[i]$ to denote the set of observations occurred before observation o_i, i.e. $O[i] = \{o_1, \ldots, o_{i-1}\}$. In the next definition recall that $\overline{A}$ denotes the set of all ground terms that can be constructed over a set of predicate symbols A.

Definition 4. *Let* $\mathcal{PCP} = \langle AF, H, O, po \rangle$ *be a policy compliance problem with abductive framework* $AF = \langle P, A, IC \rangle$. *A* diagnosis *for* $\mathcal{PCP}$ *is a set* $\Delta \subseteq \overline{A}$ *such that* $P \cup O[i] \cup \Delta \models po(o_i)$ *for all* $o_i \in O$ *and* $P \cup O \cup \Delta \cup H \models IC$. *A* validated diagnosis *is a diagnosis* Δ *such that* $\Delta \subseteq H$.

Besides explaining the observations (i.e., $P \cup O[i] \cup \Delta \models po(o)$) a diagnosis Δ should satisfy the integrity constraints when combined with the set of verified hypotheses H (i.e., $P \cup O \cup \Delta \cup H \models IC$). The end goal of the auditing process is to find a validated diagnosis, i.e. a diagnosis where all hypotheses are known facts ($\Delta \subseteq H$).

Example 3. Consider the scenario in Example 1. We assume that personal information of customers is stored by the hotel in customer profiles, which are maintained in its IT system. The IT system, however, does not keep track whether a profile was created by the customer, received from some partner, or created by a hotel employee with the explicit consent of the customer (for the sake of simplicity, we do not model customer consent in our formalization). Moreover, the log does not record the delegations of permission between the hotel employees. Indeed, an employee can informally give the permission to access customer profiles to other employees. Suppose that the auditor observes that a hotel employee (Bob) read a customer profile (obj_1). The top part of Figure 3 shows some diagnoses justifying this event where $\mathsf{has\_perm}(bob, obj_1, read)$ is the proof obligation for the event. For instance, Bob could have created obj_1 (with the user's consent); in this case he may read the profile for the purposes for which the profile was created. The second diagnosis shows that obj_1 could have been created by Alice (the customer) who delegated the permission to read it to Charlie (an employee at one of the partners of hotel); in turn Charlie granted the right to read the profile to Bob.

As shown in the example above more than one diagnosis may exist to explain the observations. The auditor should determine which of these diagnoses corresponds to what actually happened. To this end, the auditor should select some hypotheses, verify their validity, and then reiterate the auditing process considering the gathered information.

Let $\Delta_1, \ldots, \Delta_n$ be the diagnoses of a policy compliance problem $\langle AF, H, O, po \rangle$. Suppose that an auditor selects a hypothesis h from a diagnosis Δ_i with $1 \leq i \leq n$.

- If h is true (i.e., the event happened), h is added to the set of validated hypotheses (i.e., $H' = H \cup \{h\}$), and the diagnoses for the policy compliance problem $\langle AF, H', O, po \rangle$ are computed.

Proof obligations: has_perm$(bob, obj_1, read)$
Validated hypotheses:

Diagnosis: event$(bob, obj_1, create)$
Diagnosis: event$(alice, obj_1, create)$ event$(alice, obj_1, del(charlie, del(bob, read)))$
event$(charlie, obj_1, del(bob, read))$
Diagnosis: event$(alice, obj_1, create)$ event$(alice, obj_1, del(bob, read))$
Diagnosis: event$(charlie, obj_1, create)$ event$(charlie, obj_1, del(dave, del(bob, read)))$
event$(dave, obj_1, del(bob, read))$
Diagnosis: event$(charlie, obj_1, create)$ event$(charlie, obj_1, del(bob, read))$

Proof obligations: has_perm$(bob, obj_1, read)$
Validated hypotheses: event$(charlie, obj_1, create)$

Diagnosis: event$(charlie, obj_1, create)$ event$(charlie, obj_1, del(dave, del(bob, read)))$
event$(dave, obj_1, del(bob, read))$
Diagnosis: event$(charlie, obj_1, create)$ event$(charlie, obj_1, del(bob, read))$

Fig. 3. Diagnoses

– If h is false (i.e., the event did not happen), not h is added to the set of validated hypotheses (i.e., $H' = H \cup \{\text{not } h\}$), and the diagnoses for the policy compliance problem $\langle AF, H', O, po\rangle$ are computed.

The analysis is reiterated until one diagnosis for which all hypotheses have been verified is found. We refer to Section 6 for strategies for selecting the hypotheses to be verified by the auditor. Below we present an iterative step for the scenario in Example 3.

Example 4. Consider the scenario in Examples 3. Suppose that the auditor has verified that profile obj_1 was created by Charlie. A new iteration of the analysis taking into account this additional information returns the diagnoses in the bottom part of Figure 3. It is worth noting that these diagnoses are a subset of the ones in the top part of Figure 3.

The auditing process may lead to the case where the policy compliance problem does not have any solution. This corresponds to the situation in which a policy violation occurred. In the next section we present an approach to investigate the consequences and root causes of policy violations.

5 Checking Policy Violations

In the previous section we have presented an approach for determining a diagnosis explaining the observations. However, it can be the case that no plausible diagnosis exists because the hypotheses identified by the auditor are not sufficient to explain the observations or because any (sub)set of hypotheses explaining the observations violates the integrity constraints of the system. These situations provide an indication that some user did not behave as intended. Hereafter, we refer to these situations as *policy violations*.

Definition 5. *Let $\mathcal{PCP} = \langle AF, H, O, po\rangle$ be a policy compliance problem with abductive framework $AF = \langle P, A, IC\rangle$. We say that a* policy violation *occurs iff $\nexists \Delta$ such that $P \cup O[i] \cup \Delta \models po(o_i)$ and $P \cup O[i] \cup Delta \cup H \models IC$ for all $o_i \in O$.*

Auditors have to investigate policy violations by assessing the potential damage of violations (i.e., the consequences) as well as by identifying their root causes for accountability purposes. To this end, the auditor should determine which actions performed by the users do not have a justification. In the remainder of this section, we first formally define consequences and then root causes of policy violations.

Consequences The consequences of a policy violation are the observed user actions that are not compliant with the policies. In other words, the observations that cannot be justified based on the abductive framework capturing the system behaviour. Note that the auditor may observe new actions during the audit process (the validated hypotheses in H) and could start a new query to also consider the compliance of these actions.

There are two possible reasons for not being able to justify an observation o_i; either the proof obligation $po(o_i)$ cannot be proven from the available facts in $O[i]$ and H or one of the integrity constraints ic is violated. In the later case the violation could be caused by actions that have 'nothing to do with' the observation in question in which case this observation, even though it cannot be justified, should not be seen as a consequence. Consider, for example, in the setting of Example 1, a scenario in which the auditor is justifying reading file obj_1 by Bob and another file obj_2 by Charlie, $O = \{event(bob, obj_1, read), event(charlie, obj_2, read)\}$, and found out that both Alice and Bob created the same file obj_1 (violating integrity constraint IC1) and Charlie created the file obj_2; $H = \{event(alice, obj_1, create), event(bob, obj_1, create), event(charlie, obj_2, create)\}$. While $event(bob, obj_1, read)$ is a consequence of the violation, the reading of obj_2 by Charlie $event(charlie, obj_2, read)$ is not related to this violation and would seem to be allowed. To distinguish these cases we introduce the notion of relevant facts and allow ignoring non-relevant facts when justifying an observation. A set of facts F is considered a *minimal assumption for* ϕ if $F \models \phi$ and $F' \not\models \phi$ for any strict subset F' of F. A fact is considered *relevant* to an event o if it is part of some minimal assumption for $po(o)$. A ϕ-relevant subset of F is any subset of F that retains all facts relevant to ϕ.

Definition 6. *Let $\langle AF, H, O, po\rangle$ be a policy compliance problem with abductive framework $AF = \langle P, A, IC\rangle$. A* consequence *of policy violation is an observation $o_i \in O$ such that $P \cup O[i] \cup H \not\models po(o_i)$ or $P \cup F \not\models IC$ for any $po(o_i)$-relevant subset F of $O[i] \cup H$.*

Although the definition is general, we are interested in the consequences identified at the end of the auditing process. At this point, if there is a validated diagnosis, there are clearly no consequences. However, note that there could be a violation without any consequences.

Root Causes The consequences of a policy violation tell the auditor which actions were not allowed from a global perspective, representing the damage caused by the violation. This, however, does not explain which actions initially caused the violation and who is responsible. Yet, this information is needed to deal with misconduct. In terms of policy compliance problems, policy violations can be led back to the fact that any potential diagnosis explaining the observations violates the integrity constraints of the system

or to the fact that the hypotheses are not sufficient to explain the observations. If a consequence is caused by violation of an integrity constraint it may be the consequence of another action rather than a root cause itself. We thus distinguish between root causes and derived consequences.

Definition 7. *Let $\langle AF, H, O, po\rangle$ be a policy compliance problem with abductive framework $AF = \langle P, A, IC\rangle$ where a policy violation occurred. We say that a consequence o_i is a* derivative consequence *of the policy violation if there is some minimal assumption for $po(o_i)$, subset of $O[i] \cup H$ and containing a consequence. Any other consequence is called a* root cause *of policy violation.*

Root causes are consequences that cannot be derived from other consequences. This could be because no proof can be made for the root cause action at all or because the facts needed to make such a proof introduce a 'new' violation of some integrity constraint.

Example 5. Consider the following observations in the system of Example 1; a profile obj_1 was created by Alice, $\mathsf{event}(alice, obj_1, create)$, read right to obj_1 was delegated to Dave by Charlie, $\mathsf{event}(charlie, obj_1, del(dave, read))$, and Dave read the object, $\mathsf{event}(dave, obj_1, read)$. Assuming no other hypothesis are validated ($H = \emptyset$), there is a policy violation and both the delegation by Charlie and the reading by Dave are consequences. The reading by Dave is a derivative consequence, as its proof obligation $\mathsf{has\_perm}(dave, obj_1, read)$ can be proven from the consequence $\mathsf{event}(charlie, obj_1, del(dave, read))$ using rule R3. The delegation by Charlie is a root cause as its proof obligation cannot be proven.

Note that, as with consequences, the scope of the root causes is also restricted to observations. Thus, if the auditor had found an additional delegation $H = \mathsf{event}(bob, obj_1, del(charlie, del(dave, read)))$, the identified root cause would still be the *observation* $\mathsf{event}(charlie, obj_1, del(dave, read))$. However, the auditor can formulate an updated policy compliance problem by adding relevant validated hypotheses, $\mathsf{event}(bob, obj_1, del(charlie, del(dave, read)))$ in this case, to the observations. In the updated policy compliance problem, the delegation by Bob is a root cause while the delegation by Charlie and the reading by Dave are derivative consequences.

In addition to finding the actions causing the violation, one may also want to establish the parties responsible for the violation. Beyond identifying the user(s) involved in an action one can also introduce a "local perspective" by extending the proof obligation function to exactly identify what the responsibilities of each involved actor are. A *localized proof obligation* function po_l takes an observation and a user and returns a property expressing the requirements that have to be fulfilled from that user's perspective. We do not address this further in this paper but only give the intuition using the example below.

Example 6. A "global" proof obligation such as in Example 1 can be combined with different trust/responsibility models to obtain a local view. For example:

$$po_l(event(Alice, O, del(Bob, R)), U) = \begin{cases} has\_perm(Alice, O, del(Bob, R)) & \text{if } U = Alice \\ true & otherwise \end{cases}$$

captures that Alice has to establish right to delegate while B may trust any delegation

given to him; he does not need to check anything to be allowed to use it. Reversing responsibility

$$po_l(event(Alice, O, del(Bob, R)), U) = \begin{cases} has\_perm(Alice, O, del(Bob, R)) & \text{if } U = Bob \\ true & \text{otherwise} \end{cases}$$

means Alice may delegate as she likes but delegatee Bob must establish the delegation is permitted before using it. We could also require both to check validity of the delegation:

$$po_l(event(Alice, O, del(Bob, R)), U) = \begin{cases} has\_perm(Alice, O, del(Bob, R)) & \text{if } U \in \{Alice, Bob\} \\ true & \text{otherwise} \end{cases}$$

here both Alice and Bob must check that the delegation is permitted. Each of these localized proof obligations preserve *consistency* with the global view, i.e. $po(o)$ holds exactly when $po_l(o, u)$ holds for all u. Other combinations are possible, for example, in the case Alice must establish the right to delegate and Bob checks that Alice is "trusted" according to some criteria. Note that such combinations may result in local requirements stronger than the global model used above. They should, however, maintain the weaker property of *safety*, i.e. if $po_l(o, u)$ holds for all u then this implies $po(o)$ holds; as long as the users adhere to their local requirements no violations will occur in the system as a whole.

6 Refinement Strategies

The policy compliance problem presented in Section 4 can have more than one diagnosis explaining the observations. The auditor should verify the validity of the hypotheses forming such diagnoses in order to determine what actually happened. This, however, is a time consuming and costly task. Indeed, there can be a large number of diagnoses; also, each diagnosis may contain a large number of hypotheses.

To make the auditing process efficient, the auditor should select a hypothesis that minimizes the number of iterations needed to obtain a validated diagnosis. Several criteria can be used to select a hypothesis to be verified, e.g. the most recurring hypothesis, the hypothesis that is least expensive to verify, etc. Below we discuss two refinement strategies that consider the multiplicity of hypotheses. In this work we focus on the efficiency of the auditing process. A complementary problem is the study of the completeness and correctness of the auditing process using a given refinement strategy. We will informally discuss this problem in Section 8 and leave its formal analysis for future work.

6.1 Multiplicity

The diagnoses satisfying a policy compliance problem can have common hypotheses. To minimize the efforts for policy compliance, an auditor can select the hypothesis that occurs more frequently in the diagnoses of the policy compliance problem. We refer to the number of diagnoses in which a hypothesis occurs as the *multiplicity* of the hypothesis.

Definition 8. *Let $\mathcal{PCP}$ be a policy compliance problem and $\Delta_1, \ldots, \Delta_n$ the diagnoses for $\mathcal{PCP}$ and H the set of validated hypothesis in $\mathcal{PCP}$. The* multiplicity *of a hypothesis h is $\#(h) = \mid \{\Delta_i : h \in \Delta_i \setminus H\} \mid$.*

A simple refinement strategy based on the multiplicity of hypotheses is to select the hypothesis with the greatest multiplicity that has not yet been validated. This strategy can be implemented within the auditing procedure presented in Section 4 by keeping the hypotheses in the diagnoses in a priority queue: higher priority is given to hypotheses with greater multiplicity.

6.2 Abstraction & Delay Declaration

The idea of using multiplicity for the selection of the hypothesis to be validated is to minimize the efforts needed for policy compliance in terms of the number of iterations of the auditing process. However, the strategy presented in the previous section makes it possible to validate only one hypothesis per iteration. Diagnoses may contain "similar" hypotheses that can be validated in a single iteration. For instance, the hypotheses show that different users could have delegated the permission to a given user; the auditor can validate all these hypotheses by asking the delegatee which user granted him the permission.

Based on this observation, we present a refinement strategy based on multiplicity, which employs the combination of two well-known techniques: abstract interpretation and delay declarations. Intuitively, abstract interpretation is used to group similar hypotheses, i.e. hypotheses that can be validated in a single iteration; delay declarations together with multiplicity are used for the selection of the hypothesis to be verified.

Abstract interpretation [8] has been proposed to prove behavioral properties of the program without performing all calculations. The idea underlying abstract interpretation is to analyze a system by an approximation of its formal semantics using abstract values. Based on this idea we introduce the notion of *most specific abstraction*.

Definition 9. *Let t and s be two terms. The* most specific abstraction for terms *($msaT$) is*

$$msaT(t,s) = \begin{cases} t & \text{if } t = s \text{ (modulo variable renaming)} \\ f(msaT(t_1,s_1),\ldots,msaT(t_n,s_n)) & \text{if } t = f(t_1,\ldots,t_n) \wedge s = f(s_1,\ldots,s_n) \\ x & \text{otherwise} \end{cases}$$

where f is an n-ary function symbol, t_i, s_j terms with $i,j \in \{1,\ldots,n\}$, and x is a fresh variable.

Definition 10. *Let $p(t_1,\ldots,t_n)$ and $q(s_1,\ldots,s_m)$ be two atoms where p is an n-ary predicate symbol, q is an m-ary predicate symbol, and t_i, s_j (with $i \in \{1,\ldots,n\}$ and $j \in \{1,\ldots,m\}$) are terms. The* most specific abstraction *(msa) is*

$$msa(p(t_1,\ldots,t_n), q(s_1,\ldots,s_m)) = \begin{cases} p(msaT(t_1,s_1),\ldots,msaT(t_n,s_n)) & \text{if } p = q \\ fail & \text{otherwise} \end{cases}$$

Intuitively, msa does not exist if the predicate symbols of the considered atoms are different (i.e., $fail$). Otherwise, it recursively determines the most specific abstraction of the (sub)terms where terms are replaced by a fresh variable when they differ.

Given a set of hypotheses H, msa can be used to define an abstraction for H. Notice that the computation of msa is independent from the order in which atoms are

considered. Therefore, it can be safely extended to determine the most specific abstraction of a set of atoms. Hereafter, we use notation $msa(H)$ to denote the msa of a set of hypotheses H.

An abstract hypothesis can represent a number of hypotheses which can occur in one or more diagnoses. To this end, we extend the notion of multiplicity for abstract hypotheses.

Definition 11. *Let $\mathcal{PCP}$ be a policy compliance problem, $\Delta_1, \ldots, \Delta_n$ the diagnoses for $\mathcal{PCP}$ and H the set of validated hypothesis in $\mathcal{PCP}$. The* multiplicity *of an abstract hypothesis ah* with respect to a diagnosis Δ_i*, denoted as $\#_{\Delta_i}(ah)$ is the number of hypotheses $h' \in \Delta_i \setminus H$ such that $msa(ah, h') = ah$. The* multiplicity *$\#(ah)$ of ah is $\sum_{i=1}^{n} \#_{\Delta_i}(ah)$.*

Note that, for ground hypotheses, this definition coincides with Definition 8; a ground hypothesis can only occur once in a diagnosis so its multiplicity is given by the number of diagnoses in which it occurs.

Example 7. Consider the diagnoses in the top part of Figure 3 and abstract hypotheses $\mathsf{event}(alice, obj_1, del(Y, read))$ and $\mathsf{event}(X, obj_1, del(Y, read))$. The multiplicity of these abstract hypotheses is $\#(\mathsf{event}(alice, obj_1, del(Y, read))) = 2$ and $\#(\mathsf{event}(X, obj_1, del(Y, read))) = 6$.

The obvious solution would be to select the (abstract) hypothesis with greatest multiplicity. However, although an abstraction can significantly reduce the number of hypotheses to be verified, the resulting abstract hypotheses may require additional efforts for their verification.

Example 8. As shown in Example 7, abstract hypothesis $\mathsf{event}(X, obj_1, del(Y, read))$ has a multiplicity greater than abstract hypothesis $\mathsf{event}(alice, obj_1, del(Y, read))$. Accordingly, the former hypothesis may appear to be preferable. However, hypothesis $\mathsf{event}(X, obj_1, del(Y, read))$ requires to verify all delegations of permission for reading profile obj_1, while hypothesis $\mathsf{event}(alice, obj_1, del(Y, read))$ would only require to verify whether Alice has delegated the permission to read obj_1.

To assist the auditor in the selection of the (abstract) hypothesis to be verified, we used an approach based on delay declarations. Delay declarations have been proposed in [24] to allow a dynamic control of the selection of atoms in Prolog. The idea underlying delay declarations is to delay the evaluation of atoms until they become sufficiently instantiated.

Definition 12. *Let p be a predicate of arity n. A delay declaration is a rule of the form:*

$$DELAY \;\; p(x_1, \ldots, x_n) \;\; UNTIL \;\; Cond(x_1, \ldots, x_n)$$

where $x_1, \ldots, x_n$ represent the arguments of p, and $Cond(x_1, \ldots, x_n)$ is a condition over $x_1, \ldots, x_n$ in some assertion language.

Intuitively, $p(x_1, \ldots, x_n)$ is considered a candidate hypothesis for the analysis if and only if $Cond(x_1, \ldots, x_n)$ is **true**.

Example 9. In our scenario we can consider the following delay declarations

$DELAY\ \ \mathsf{event}(X, O, create)\ \ UNTIL\ \ \mathsf{ground}(O)$
$DELAY\ \ \mathsf{event}(X, O, del(Y, R))\ \ UNTIL\ \ \mathsf{ground}(O) \wedge \mathsf{ground}(R) \wedge (\mathsf{ground}(X) \vee \mathsf{ground}(Y))$

where predicate $\mathsf{ground}(x)$ is used to verify whether the term x is instantiated. Intuitively, the first rule allows the selection of a hypothesis $\mathsf{event}(X, O, create)$ only if the object is known. The second rule requires that both the object and the delegated right is known in order for a hypothesis $\mathsf{event}(X, O, del(Y, R))$ to be selected. In addition, it requires that either the delegator or the delegatee is known. Based on these rules, the verification of $\mathsf{event}(X, obj, del(Y, read))$ is delayed, and $\mathsf{event}(alice, obj, del(Y, read))$ can be considered as a candidate hypothesis for verification.

The strategy works as follows: Given a set of diagnoses $\Delta_1, \ldots, \Delta_n$ and validated hypothesis VH, the set of abstract hypotheses $\hat{H}$ is given by

$$\hat{H} = \{msa(H) : H \subseteq (\Delta_1 \cup \cdots \cup \Delta_n) \setminus VH\}$$

The set of candidate abstract hypothesis C contains those that are not delayed:

$$C = \{ah \in \hat{H} : delay(ah) = false\}$$

From C we select the candidate with the greatest multiplicity.

The auditing process in Section 4 should be revised to take into account the fact that an abstract hypothesis can represent more than one hypothesis. In particular, the validation of an abstract hypothesis ah gives for each of the hypotheses represented by the abstract hypothesis whether it is true or not. We add $\#(ah)$ elements to the set of validated hypotheses; for each represented hypothesis h we either add h if it is true or $\mathsf{not}\ h$ if it is not. Then, the auditing process again follows the steps described in Section 4.

7 Prototype Implementation

We have implemented a tool to compute a validated diagnosis for a policy compliance problem based on the approach presented in this paper. The tool takes as input an abductive framework (i.e., an abductive theory and integrity constraints) and a set of observations representing events recorded in the logs. The set of observations is used to construct the query to be evaluated with respect to the abductive framework. The tool iteratively finds the plausible diagnoses to the query. At each iteration, the tool selects a hypothesis from the plausible diagnoses and requires an auditor to validate it. If the hypothesis is valid, the tool adds such a hypothesis to the query; otherwise, its negation is added to the query (see Section 4). The process is iterated until a validated diagnosis is obtained or no plausible diagnoses exist. The tool returns a valid diagnosis (if it exists) or an error message saying that a policy violation has occurred.

To find the diagnoses of a policy compliance problem, the tool relies on the CIFF Proof procedure [28] for abductive reasoning and SWI Prolog as the underlying reasoning engine. The CIFF Proof procedure is an abductive proof procedure implemented in

Prolog, which extends the IFF procedure [11] by integrating abductive reasoning with constraint solving. This procedure takes a theory, a set of integrity constraints and a query as input, and returns a set of plausible diagnoses for the query if the procedure succeeds or the procedure fails indicating that there are no plausible diagnoses. We refer to [28] for detail on CIFF.

CIFF allows a query to contain more than one literal. However, CIFF treats each literal in the query as an individual query. In particular, it computes the plausible diagnoses for each literal in the query independently from the other literals. The diagnoses to the query are obtained by the union of the diagnoses to all literals in the query. This poses the problem of redundant information as it can result in diagnoses which are subset of other diagnoses. To address this, our tool filters the diagnoses computed by CIFF by removing the ones having redundant information (i.e., a diagnosis Δ' is removed if there exist a diagnosis Δ'' such that $\Delta'' \subset \Delta'$).

For the selection of the hypothesis to be validated, the tool supports three refinement strategies. In particular, the tool implements the Multiplicity and Abstraction & Delay strategies described in Section 6. In addition to them, we consider a strategy that selects the hypothesis to be validated randomly from the list of plausible hypothesis. We refer to this strategy as the Random Selection strategy.

8 Experiments

We have performed a number of experiments to evaluate the auditing process with incomplete logs along with the refinement strategies. This section presents a number of metrics for the evaluation of the refinement strategies. Then, it presents the setting for the experiments and discusses the results.

Evaluation framework The purpose of the experiments is to evaluate and compare the performance of the refinement strategies presented In Section 6. In particular we study the effectiveness and efficiency of a strategy in obtaining a validated diagnosis. In addition, we assess the ability of a strategy to detect policy violations. For the analysis of these aspects, we employ the following metrics:

- M_1: # iterations to reduce the set of possible diagnoses until only one diagnosis remains.
- M_2: # iterations to validate the diagnosis or to detect a policy violation.
- M_3: outcome of the policy compliance problem.

The first two metrics measures the performance of a strategy. M_1 assesses the ability of a strategy to prune non-plausible diagnoses by measuring how many iterations are needed to obtain a single diagnosis. M_2 assesses the efficiency of a strategy to compute a validated diagnosis or identify a policy violation. Note that we only consider M_1 when a policy violation is not detected. Indeed, the number of iterations needed to detect a policy violations is already accounted in M_2. Thus, we report symbol '-' (dash) for M_1 when a validated diagnosis is not found. Finally, M_3 reports whether a plausible justification for the observations is found (A) or a policy violation is detected (V). This metric is used to assess the ability of a strategy to detect policy violations.

	Random Selection			Multiplicity			Abstraction & Delay		
Scenario	M_1	M_2	M_3	M_1	M_2	M_3	M_1	M_2	M_3
1 (A)	16.2	20	A	10	11	A	6	7	A
2 (A)	13.2	16.2	A	11	13	A	7	8	A
3 (A)	17.4	20.6	A	10	11	A	6	7	A
4 (V)	14.6	16.8	A(3) V(2)	10	11	A	-	2	V
5 (V)	-	21	V	-	10	V	-	5	V

Table 1. Results of the experiments.

Experiment Settings To evaluate the refinement strategies, we used a policy describing the intended system behavior (i.e., an abductive theory and integrity constraints) which extends the abductive framework presented in Figure 2. In particular, we extended such an abductive framework by introducing clauses and integrity constraints tailored to restrict the allowed delegations of permission among entities based on their role within the system. For instance, we defined constraints to prevent delegation chains forming loops, i.e. situations in which a user delegates the permissions to another user that (possibly indirectly) has delegated the permission to him. We believe that these constraints are reasonable in real situations. From a technical perspective, these constraints are necessary to prevent the existence of an infinite number of plausible explanations for the observations. For the experiments, we considered queries that initially consist of one event.

The auditing process requires the interaction with the auditor for the validation of hypotheses. For the experiments we have automated this step by providing the sequence of events which actually occurred (hereafter called *scenario*) as input to the tool. Intuitively, a scenario is used to determine whether a hypothesis is valid or not. We have evaluated the auditing process against five scenarios. The first three scenarios contain only legitimate events, while the other two scenarios contain some events which violate the defined policy. In particular, scenario 4 has two create events for the same object performed by two different users (violation of integrity constraint IC1 in Figure 2). In scenario 5, we assumed that an entity delegated access rights to another entity without the proper permission (violation of integrity constraint IC2 in Figure 2). Each experiment corresponds to the execution of the auditing process using a different refinement strategy over a different scenario. For the Abstraction & Delay strategy we used the delay declarations defined in Example 9. We repeated each experiment five times.

Results Table 1 presents the results of our experiments where every entry reports the average over the five runs of the experiments. For each scenario, the table reports metrics M_1, M_2 and M_3 when the three strategies were used to find a validated diagnosis for the policy compliance problem. In the table each scenario is annotated either with a label (A) to represent that no policy violations occurred in the scenario or with a label (V) to represent that a policy violation occurred in the scenario.

The results show that the Abstraction & Delay strategy is more effective in pruning non-plausible diagnoses than the Multiplicity and Random Selection strategies (M_1). Moreover, we can observe that for the Multiplicity and Abstraction & Delay strategies, the difference between M_2 and M_1 is usually one iteration. The reason for this is

that we usually obtain one diagnosis only when there is either one or no non-validated hypothesis left in the diagnosis.

Comparing the label associated to the scenarios (next to the scenario identifier in Table 1) with M_3 we can observe that the Abstraction & Delay strategy was always able to detect policy violations. On the other hand, the Random Selection and Multiplicity strategies were not able to detect the policy violation in scenario 4. The violation in this scenario can only be detected if both create events are detected. However, the Random Selection and Multiplicity strategies are not able to identify all such events. Indeed, because of integrity constraint IC1 diagnoses can only contain one create event; thus, when a create event is validated, the diagnoses returned in the next iterations are only the ones that contain such an event, leaving other create events undetected. It is worth noting that the Random Selection strategy in some cases is able to detect that a policy violation occurred. However, when a policy violation is detected, the strategy finds a violation of integrity constraint IC2 (instead of IC1). This can be explained by the scenario representing the events that occurred. In particular, the scenario comprises two create events together with the events justifying the read event starting from only one of the two create events. Thus, the create event that is selected by the strategy determines whether the policy violation is detected or not.

The difference in detecting policy violations is mainly due to the different type of "questions" that an auditor should ask in order to validate the hypothesis selected using different refinement strategies. In particular, the Random Selection and Multiplicity strategies lead to *closed questions* that aim to verify whether a specific event occurred or not. In contrast, the Abstraction & Delay strategy leads to *open questions* which allow an auditor to retrieve all events related to a given (abstract) hypothesis. In our tool an abstract hypothesis is instantiated by checking all events in the scenario. One may argue that the validation of an abstract hypothesis may be too costly and so not feasible in practice. To this end, we have combined abstraction with delay declarations which restrict the questions that an auditor can ask. For example, the second delay declaration in Example 9 would only allow an auditor to ask an entity from whom he received a certain permission or to whom he delegated the permission.

9 Related Work

The protection of sensitive information is often achieved using preventive policy enforcement mechanisms. These mechanisms, however, are not suitable to deal with dynamic and unpredictable domains (e.g., healthcare) or to enforce certain classes of policies (e.g., future obligation, purpose control). This has led to the development of approaches for a posteriori policy compliance and root causes analysis [2, 7, 20, 26, 27, 15] in which the problem of preventing unauthorized behavior is shifted to an accountability problem. However, most existing proposals do not address the problem of auditing when audit logs are incomplete (i.e., they do not contain the information necessary to determine whether a policy is violated).

Only few proposals [4, 12] deal with policy compliance when audit logs are incomplete. Garg et al. [12] propose an algorithm to check audit logs for compliance with privacy and security policies when audit logs are incomplete. The proposed algorithm

iteratively reduces policies specified in first-order logic based on the information available to the auditor. This work differs from our work in several ways. First, the work in [12] requires auditors to have an extensive logic background to be able to interpret the results of the analysis. In contrast, our approach provides auditors with a list of facts to be verified. In addition, the work in [12] does not provide auditing strategies that assist an auditor to select which portion of a formula should be analyzed. Moreover, this work requires specifying in advance how each predicate should be verified. In contrast, our approach is independent from how predicates should be verified and thus is more flexible to deal with possible sources of incompleteness in audit logs. Bertoli et al. [4] propose an approach to reconstruct partial execution traces of a process model using deductive techniques. In particular, their approach deduces all the logic models that subsume the knowledge about the execution and that are compliant with a process model and domain knowledge. Similarly to our approach, the non-existence of such a model denotes the non-compliance of the partial trace. Our proposal makes a number of steps further compared to the work in [4]. First, our approach proposes strategies to support the investigation of what actually happened rather than just determining all possible execution traces that could have been occurred. In addition, in case of a deviation occurred, our approach determines the consequences and root causes rather than merely detect non-compliant partial traces.

Our work is not the first that uses abductive reasoning in access control and trust management. Becker et al. [3] use abductive reasoning to explain access denials and automate delegation. Gupta et al. [14] propose an abductive approach for the analysis of administrative policies in rule-based access control. Other proposals [5, 19] use abduction analysis to calculate the credentials that a client has to provide in order to get access. The approaches above focus on issues that are complementary to the goal of our work. In particular, they use abductive reasoning to find plausible sets of facts, e.g. representing credentials, which satisfy a given policy. These sets can be obtained using existing abductive reasoning tools [11, 21, 18]. On the other hand, our goal is to identify a valid explanation of conformity for which requires to refine plausible diagnosis until all hypothesis forming a diagnosis have been validated.

10 Conclusions

In this paper, we have presented an auditing framework based on abductive reasoning to assist auditors in verifying the conformity of users to usage policies in presence of incomplete log. The framework makes it possible to find plausible explanations for the observations recorded in logs and pinpoint the consequences and root causes of policy violation if a valid explanation of conformity does not exist. A policy compliance problem can have many plausible explanations. To identify a valid explanation, our auditing framework allows an auditor to select some hypotheses, verify their validity, and then reiterate the auditing process considering the gathered information. To assist auditors during the auditing process, we have analyzed two refinement strategies that aim to determine the "best" hypothesis to be validated, i.e. the hypothesis that minimize the efforts required to find a valid explanation of conformity. We have implemented a prototype supporting the proposed framework and conducted experiments to compare

the efficiency and effectiveness of refinement strategies. Experiment results show that the Abstraction & Delay Declaration strategy is more efficient compared to the other refinement strategies. Moreover, due to the nature of the questions this strategy makes it possible to identify policy violations when they occur.

The work presented in this paper suggests some interesting directions for future work. The refinement strategies considered in this work are based on the number of occurrence of hypotheses in plausible diagnoses. However, other criteria can be relevant for the selection of the hypotheses to be verified. For instance, different hypotheses may require different amount of effort for their verification based on particular users (e.g., alice vs. bob) or actions (create vs. delegate). Therefore, we plan to investigate other refinement strategies, e.g. based on the cost of validating hypotheses, to optimize the auditing process. However, as shown in Section 8, a strategy may drive an auditor to the wrong conclusions, leaving policy violations undetected. To this end, we will study the completeness and correctness of the auditing process when using different refinement strategies. Moreover, the ideas underlying this work can be applied for other types of decision analysis based on incomplete knowledge. An example is the alignment of network monitoring with access control for intrusion detection. Access control policies are usually defined at the application layer, while network monitoring relies on the analysis of messages and packages transmitted at the network layer. Our approach can be used to bridge the two layers, allowing a seamless analysis of network logs and access control policies.

Acknowledgments This work has been partially funded by the EU FP7 project AU2EU, the ITEA2 project FedSS, and the Dutch national program COMMIT under the THeCS project.

References

1. Adriansyah, A., van Dongen, B.F., Zannone, N.: Controlling break-the-glass through alignment. In: Proceedings of International Conference on Social Computing. pp. 606–611. IEEE (2013)
2. Adriansyah, A., van Dongen, B.F., Zannone, N.: Privacy analysis of user behavior using alignments. it - Information Technology 55(6), 255–260 (2013)
3. Becker, M.Y., Nanz, S.: The role of abduction in declarative authorization policies. In: Proceedings of 10th International Conference on Practical Aspects of Declarative Languages. pp. 84–99. Springer (2008)
4. Bertoli, P., Di Francescomarino, C., Dragoni, M., Ghidini, C.: Reasoning-based techniques for dealing with incomplete business process execution traces. In: AI*IA 2013: Advances in Artificial Intelligence. pp. 469–480. LNCS 8249, Springer (2013)
5. Bistarelli, S., Martinelli, F., Santini, F.: A formal framework for trust policy negotiation in autonomic systems: Abduction with soft constraints. In: Proceedings of the 7th International Conference on Autonomic and Trusted Computing. pp. 268–282. Springer (2010)
6. Butin, D., Le Mtayer, D.: Log analysis for data protection accountability. In: Formal Methods, pp. 163–178. LNCS 8442, Springer (2014)
7. Cederquist, J., Corin, R., Dekker, M., Etalle, S., Hartog, J., Lenzini, G.: Audit-based compliance control. International Journal of Information Security 6(2-3), 133–151 (2007)

8. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 238–252. ACM (1977)
9. Crampton, J., Huth, M.: Detecting and Countering Insider Threats: Can Policy-Based Access Control Help? In: Proceedings of 5th International Workshop on Security and Trust Management (2009)
10. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: The Diagnosis Frontend of the DLV System. AI Commun. 12(1-2), 99–111 (1999)
11. Fung, T.H., Kowalski, R.: The {IFF} proof procedure for abductive logic programming. The Journal of Logic Programming 33(2), 151–165 (1997)
12. Garg, D., Jia, L., Datta, A.: Policy auditing over incomplete logs: Theory, implementation and applications. In: Proceedings of the 18th ACM Conference on Computer and Communications Security. pp. 151–162. ACM (2011)
13. Giorgini, P., Massacci, F., Mylopoulos, J., Zannone, N.: Requirements engineering for trust management: model, methodology, and reasoning. Int. J. Inf. Sec. 5(4), 257–274 (2006)
14. Gupta, P., Stoller, S., Xu, Z.: Abductive analysis of administrative policies in rule-based access control. IEEE Transactions on Dependable and Secure Computing 11(5), 412–424 (2014)
15. Jagadeesan, R., Jeffrey, A., Pitcher, C., Riely, J.: Towards a theory of accountability and audit. In: Proc. European Symp. Research in Computer Security. Lecture Notes in Computer Science, vol. 5789, pp. 152–167. Springer-Verlag (2009)
16. Jajodia, S., Samarati, P., Sapino, M.L., Subrahmanian, V.S.: Flexible support for multiple access control policies. ACM Trans. Database Syst. 26(2), 214–260 (2001)
17. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive logic programming. Journal of Logic and Computation 2(6), 719–770 (1992)
18. Kakas, A.C., Nuffelen, B.V., Denecker, M.: A-system: Problem solving through abduction. In: Proceedings of 17th International Joint Conference on Artificial Intelligence. pp. 591–596. Morgan Kaufmann Publishers (2001)
19. Koshutanski, H., Massacci, F.: Interactive access control for autonomic systems: From theory to implementation. ACM Trans. Auton. Adapt. Syst. 3(3), 9:1–9:31 (2008), http://doi.acm.org/10.1145/1380422.1380424
20. Kveler, K., Bock, K., Colombo, P., Domany, T., Ferrari, E., Hartman, A.: Conceptual framework and architecture for privacy audit. In: Privacy Technologies and Policy. pp. 17–40. LNCS 8319, Springer (2014)
21. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Trans. Comput. Logic 7(3), 499–562 (2006)
22. Li, N., Mitchell, J.C.: DATALOG with Constraints: A Foundation for Trust Management Languages. In: Practical Aspects of Declarative Languages. pp. 58–73. LNCS 2562, Springer (2003)
23. Massacci, F., Zannone, N.: Detecting Conflicts between Functional and Security Requirements with Secure Tropos: John Rusnak and the Allied Irish Bank. In: Social Modeling for Requirements Engineering, pp. 337–362. MIT Press (2011)
24. Naish, L.: An introduction to MU-Prolog. Tech. Rep. 82/2, Dept. of Computer Science, University of Melbourne (1982)
25. OASIS XACML Technical Committee: eXtensible Access Control Markup Language (XACML) Version 3.0. Oasis standard, OASIS (2013)
26. Petkovic, M., Prandi, D., Zannone, N.: Purpose Control: Did You Process the Data for the Intended Purpose? In: Secure Data Management. pp. 145–168. LNCS 6933, Springer (2011)

27. Ruebsamen, T., Reich, C.: Supporting Cloud Accountability by Collecting Evidence Using Audit Agents. In: Proceedings of 5th International Conference on Cloud Computing Technology and Science. pp. 185–190. IEEE (2013)
28. Terreni, G.: The CIFF Proof Procedure for Abductive Logic Programming with Constraints: Definition, Implementation and a Web Application. Ph.D. thesis, Università di Pisa (2008)
29. Trivellato, D., Zannone, N., Etalle, S.: GEM: A distributed goal evaluation algorithm for trust management. TPLP 14(3), 293–337 (2014)